Essentials of Theoretical Computer Science

F. D. Lewis
University of Kentucky

How to Navigate This Text

Table of Contents

CONTENTS

AUTOMATA

LANGUAGES

COMPUTABILITY

Before examining the intrinsic nature of computation we must have a precise idea of what computation means. In other words, we need to know what we're talking about! To do this, we shall begin with intuitive notions of terms such as *calculation*, *computing procedure*, and *algorithm*. Then we shall be able to develop a precise, formal characterization of computation which captures all of the modern aspects and concepts of this important activity.

Part of this definitional process shall involve developing models of computation. They will be presented with emphasis upon their finite nature and their computational techiques, that is, their methods of transforming inputs into outputs. In closing, we shall compare our various models and discuss their relative power.

The sections are entitled:

The *NICE* Programming Language

As computer scientists, we tend to believe that computation takes place inside computers. Or maybe computations are the results from operating computing machinery. So, when pressed to describe the limits of computation we might, in the spirit of Archimedes and his lever, reply that anything can be computed given a large enough computer! When pressed further as to how this is done we probably would end up by saying that all we need in order to perform all possible computations is the ability to execute programs which are written in some marvelous, nonrestrictive programming language. A *nice* one!

Since we are going to *study* computation rather than engage in it, an actual computer is not required, just the programs. These shall form our model of computation. Thus an ideal programming language for computation seems necessary. Let us call this the *NICE* language and define it without further delay. Then we can go on to describing what is meant by computation.

We first need the raw materials of computation. Several familiar items come immediately to mind. *Numbers* (integers as well as real or floating point) and *Boolean constants* (*true* and *false*) will obviously be needed. Our programs shall then employ *variables* that take these constants as values. And since it is a good idea to tell folks exactly what we're up to at all times, we shall declare these variables before we use them. For example:

```
var  x, y, z: integer;
     p, q: Boolean;
     a, b, c: real;
```

Here several variables have been introduced and defined as to *type*. Since the world is often nonscalar we always seem to want some data structures. *Arrays* fill that need. They may be multidimensional and are declared as to type and dimension. Here is an array declaration.

```
var  d, e: array[ , ] of integer;
     s: array[ ] of real;
     h: array[ , , , ] of integer;
```

We note that s is a one-dimensional array while h has four dimensions. As is usual in computer science, elements in arrays are referred to by their position. Thus s[3] is the third element of the array named s.

So far, so good. We have placed syntactic restrictions upon variables and specified our constants in a rather rigid (precise?) manner. *But we have not placed any bounds on the magnitude of numbers or array dimensions.* This is fine since we did not specify a particular computer for running the programs. In fact, we are dealing with ideal, gigantic computers that can handle *very large* values. So why limit ourselves? To enumerate:

a) Numbers can be of any magnitude.
b) Arrays may be declared to have as many dimensions as we wish.
c) Arrays have no limit on the number of elements they contain.

Our only restriction will be that at any instant during a computation *everything must be finite.* That means no numbers or arrays of infinite length. Huge - yes, but not infinite! In particular, this means that the infinite decimal expansion 0.333... for one third is not allowed yet several trillion 3's following a decimal point is quite acceptable. We should also note that even though we have a number type named *real,* these are not real numbers in the mathematical sense, but floating point numbers.

On to the next step - *expressions.* They are built from variables, constants, operators, and parentheses. *Arithmetic expressions* such as these:

```
x + y*(z + 17)
a[6] - (z*b[k, m+2])/3
```

may contain the operators for addition, subtraction, multiplication, and division. *Boolean expressions* are formed from arithmetic expressions and *relational operators.* For example:

```
x + 3 = z/y -17
   a[n] > 23
```

Compound Boolean expressions also contain the *logical connectives and, or,* and *not.* They look like this:

```
x - y > 3 and b[7] = z and v
(x = 3 or x = 5) and not z = 6
```

These expressions may be evaluated in any familiar manner (such as operator precedence or merely from left to right). We do not care how they are evaluated, as long as we maintain consistency throughout.

In every programming language computational directives appear as *statements.* Our *NICE* language contains these also. To make things a little less wordy we shall introduce some notation. Here is the master list:

E	**arbitrary expressions**
AE	**arithmetic expressions**
BE	**Boolean expressions**
V	**variables**
S	**arbitrary statements**
N	**numbers**

Variables, statements, and numbers may be numbered (*V6, S1, N9*) in the descriptions of some of the statements used in *NICE* programs that follow.

a) **Assignment.** Values of expressions are assigned to variables in statements of the form: *V = E.*

b) **Transfer.** This statement takes the form *goto N* where *N* is an integer which is used as a *label.* Labels precede statements. For example: 10: *S.*

c) **Conditional.** The syntax is: *if BE then S1 else S2* where the *else* clause is optional.

d) **Blocks.** These are groups of statements, separated by semicolons and bracketed by *begin* and *end.* For example: *begin S1; S2; ... ; Sn end*

Figure 1 contains a fragment of code that utilizes every statement defined so far. After executing the block, z has taken on the value of x factorial.

```
begin
    z = 1;
10: z = z*x;
    x = x - 1;
    if not x = 0 then goto 10
end
```

Figure 1- Factorial Computation

e) **Repetition.** The *while* and *for* statements cause repetition of other statements and have the form:

while BE do S
for V = AE to AE do S

Steps in a *for* statement are assumed to be one unless *downto* (instead of *to*) is employed. Then the steps are minus one as then we decrement rather than increment. It is no surprise that repetition provides us with structured ways to compute factorials. Two additional methods appear in figure 2.

```
begin
   z = 1;
   for n = 2 to x
       do z = z*n
end
```

```
begin
   z = 1;
   n = 1;
   while n < x do
      begin
         n = n + 1;
         z = z*n
      end
end
```

Figure 2 - Structured Programming Factorial Computation

f) **Computation by cases.** The *case* statement is a multiway, generalized *if* statement and is written:

case AE of N1: S1; N2: S2; ... ; Nk: Sk endcase

where the *Nk* are numerical constants. It works in a rather straightforward manner. The expression is evaluated and if its value is one of the *Nk*, then the corresponding statement *Sk* is executed. A simple table lookup is provided in figure 3. (Note that the cases need not be in order nor must they include all possible cases.)

```
case x - y/4 of:
   15: z = y + 3;
    0: z = w*6;
   72: begin
         x = 7; z = -2*z
       end;
    6: w = 4
endcase
```

Figure 3 - Table Lookup

g) **Termination.** A *halt(V)* statement brings the program to a stop with the value of *V* as output. *V* may be a simple variable or an array.

Now that we know all about statements and their components it is time to define programs. We shall say that a *program* consists of a *heading*, a *declaration section* and a *statement* (which is usually a block). The heading looks like:

program name(V1, V2, ... , Vn)

and contains the name of the program as well as its *input parameters.* These parameters may be variables or arrays. Then come all of the declarations followed by a statement. Figure 4 contains a complete program.

```
program expo(x, y)
var n, x, y, z: integer;
begin
   z = 1;
   for n = 1 to y do z = z*x;
   halt(z)
end
```

Figure 4 - Exponentiation Program

The only thing remaining is to come to an agreement about exactly what programs do. Let's accomplish this by examining several. It should be rather obvious that the program of figure 4 raises x to the y-th power and outputs this value. So we shall say that *programs compute functions.*

Our next program, in figure 5, is slightly different in that it does not return a numerical value. Examine it.

```
program square(x)
var x, y: integer;
begin
   y = 0;
   while y*y < x do y = y + 1;
   if y*y = x then halt(true) else halt(false)
end
```

Figure 5 - Boolean Function

This program does return an answer, so it does compute a function. But it is a *Boolean function* since it returns either *true* or *false*. We depict this one as:

square(x) = *true* if x is a perfect square and *false* otherwise.

Or, we could say that the program named 'square' decides whether or not an integer is a perfect square. In fact, we state that this program *decides membership* for the set of squares.

Let us sum up all of the tasks we have determined that programs accomplish when we execute them. We have found that they do the following two things.

a) compute functions
b) decide membership in sets

And, we noted that (b) is merely a special form of (a). That is all we know so far.

So far, so good. But, shouldn't there be more? That was rather simple. And, also, if we look closely at our definition of what a program is, we find that we can write some strange stuff. Consider the following rather silly program.

```
program nada(x)
var x, y: integer;
x = 6
```

Is it a program? Well, it has a heading, all of the variables are declared, and it ends with a statement. So, it must be a program since it looks exactly like one. But, it has no *halt* statement and thus can have no output. So, what does it do? Well, not much that we can detect when we run it!

Let's try another in the same vein. Consider the well-known and elegant:

```
program loop(x)
var x: integer;
while x = x do x = 17
```

which does something, but alas, nothing too useful. In fact, programs which either do not execute a *halt* or even contain a *halt* statement are programs, but accomplish very little that is evident to an observer. We shall say that *these compute functions which are* ***undefined*** (one might say that f(x) = ?) since we do not know how else to precisely describe the results attained by running them.

Let us examine one that is sort of a cross between the two kinds of programs we have seen thus far. This, our last strange example, sometimes halts, sometimes loops and is included as figure 6.

```
program fu(x)
var n, x: integer;
begin
   n = 0;
   while not x = n do n = n - 2;
   halt(x)
end
```

Figure 6 - A Partially Defined Function

This halts only for even, positive integers and computes the function described as:

fu(x) = x if x is even and positive, otherwise undefined

When a program does not halt, we shall say it diverges. The function fu could also be defined using mathematical notation as follows.

$$fu(x) = \begin{cases} x \text{ if } x \text{ is even and positive} \\ \text{diverge otherwise} \end{cases}$$

Since it halts at times (and thus is defined on the even, positive integers) we will compromise and maintain that it is *partially defined*.

To recap, we have agreed that computation takes place whenever programs are run on some ideal machine and that programs are written according to the rules of our *NICE* language.

An important note is needed here. We have depended heavily upon our computing background for the definition of exactly what occurs when computation takes place rather than dwell upon the semantics of the *NICE* language. We could go into some detail of how statements in our language modify the values of variables and so forth, but have agreed not to at this time.

So, given that we all understand program execution, we can state the following two assertions as our definition of computation.

- *programs compute functions*
- *any computable function can be computed by some program.*

The functions we enjoy computing possess values for all of their input sets and are called *defined*, but some functions are different. Those functions that never have outputs are known as *undefined functions.* And finally, the functions that

possess output values for some inputs and none for others are the *partially defined functions.*

At last we have fairly carefully defined computation. It is merely the process of running *NICE* programs.

Turing Machines

Computation has been around a very long time. Computer programs, after all, are a rather recent creation. So, we shall take what seems like a brief detour back in time in order to examine another system or model of computation. We shall not wander too far back though, merely to the mid 1930's. After all, one could go back to Aristotle, who was possibly the first Western person to develop formal computational systems and write about them.

Well before the advent of modern computing machinery, a British logician named A. M. Turing (who later became a famous World War II codebreaker) developed a computing system. In the 1930's, a little before the construction of the first electrical computer, he and several other mathematicians (including Church, Markov, Post, and Turing) independently considered the problem of specifying a system in which computation could be defined and studied.

Turing focused upon human computation and thought about the way that people compute things by hand. With this examination of human computation he designed a system in which computation could be expressed and carried out. He claimed that any nontrivial computation required:

- a simple sequence of computing instructions,
- scratch paper,
- an implement for writing and erasing,
- a reading device, and
- the ability to remember which instruction is being carried out.

Turing then developed a mathematical description of a device possessing all of the above attributes. Today, we would recognize the device that he defined as a special purpose computer. In his honor it has been named the *Turing machine*.

This heart of this machine is a *finite control box* which is wired to execute a specific *list of instructions* and thus is precisely a special purpose computer or computer chip. The device records information on a *scratch tape* during computation and has a *two-way head* that *reads* and *writes* on the tape as it moves along. Such a machine might look like that pictured in figure 1.

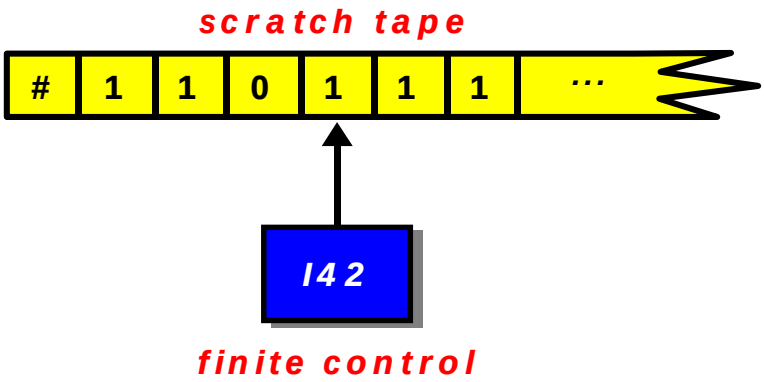


Figure 1 - A Turing Machine

A *finite control* is a simple memory device that remembers which *instruction* should be executed next. The *tape*, divided into *squares* (each of which may hold a *symbol),* is provided so that the machine may record results and refer to them during the computation. In order to have enough space to perform computation, we shall say that the tape is *arbitrarily long.* By this we mean that a machine never runs out of tape or reaches the right end of its tape. This does *NOT* mean that the tape is infinite - just long enough to do what is needed. A *tape head* that can *move* to the left and right as well as *read* and *write* connects the finite control with the tape.

If we again examine figure 1, it is evident that the machine is about to execute instruction *I42* and is reading a 1 from the tape square that is fifth from the left end of the tape. Note that we only show the portion of the tape that contains non-blank symbols and use three dots (. . .) at the right end of our tapes to indicate that the remainder is blank.

That is fine. But, what runs the machine? What exactly are these instructions which govern its every move? A *Turing machine instruction* commands the machine to perform the sequence of several simple steps indicated below.

a) read the tape square under the tape head,
b) write a symbol on the tape in that square,
c) move its tape head to the left or right, and
d) proceed to a new instruction

Steps (b) through (d) depend upon what symbol appeared on the tape square being scanned before the instruction was executed.

An instruction shall be presented in a chart that enumerates outcomes for all of the possible input combinations. Here is an example of an instruction for a machine which uses the symbols 0, 1, #, and blank.

	symbol read	*symbol written*	*head move*	*next instruction*
I93	**0**	**1**	**left**	**next**
	1	**1**	**right**	**I17**
	b	**0**	**halt**	
	#	**#**	**right**	**same**

This instruction (I93) directs a machine to perform the actions described in the fragment of *NICE* language code provided below.

```
case (symbol read) of:
      0: begin
           print a 1;
           move one tape square left;
           goto the next instruction (I94)
         end;
      1: begin
           print a 1;
           move right one square;
           goto instruction I17
         end;
  blank: begin print a 0; halt end;
      #: begin
           print #;
           move to the right;
           goto this instruction (I93)
         end
endcase
```

Now that we know about instructions, we need some conventions concerning machine operation. *Input strings* are written on the tape prior to computation and will always consist of the symbols 0, 1, and blank. Thus we may speak of inputs as binary numbers when we wish. This may seem arbitrary, and it is. But the reason for this is so that we can describe Turing machines more easily later on. Besides, we shall discuss other input symbol alphabets in a later section.

When several binary numbers are given to a machine they will be separated by blanks (denoted as *b).* A sharp sign (#) always marks the left end of the tape at the beginning of a computation. Usually a machine is never allowed to change this marker. This is so that it can always tell when it is at the left end of its tape and thus not fall off the tape unless it wishes to do so. Here is an input tape with the triple <5, 14, 22> written upon it.

In order to depict this tape as a string we write: #101*b*1110*b*10110 and obviously omit the blank fill on the right.

Like programs, Turing machines are designed by coding sequences of instructions. So, let us design and examine an entire machine. The sequence of instructions in figure 2 describes a Turing machine that receives a binary number as input, adds one to it and then halts. Our strategy will be to begin at the lowest order bit (on the right end of the tape) and travel left changing ones to zeros until we reach a zero. This is then changed into a one.

One small problem arises. If the endmarker (#) is reached before a zero, then we have an input of the form 111...11 (the number $2^n - 1$) and must change it to 1000...00 (or 2^n).

sweep right to end of input

	read	*write*	*move*	*goto*
I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	***b***	**left**	**next**

change 1's to 0's on left sweep, then change 0 to 1

I2	**0**	**1**	**halt**	
	1	**0**	**left**	**same**
	#	**#**	**right**	**next**

input = 11...1, so sweep right printing 1000...0 (print leading 1, add 0 to end)

I3	**0**	**1**	**right**	**next**

I4	**0**	**0**	**right**	**same**
	b	**0**	**halt**	

Figure 2 - Successor Machine

In order to understand this computational process better, let us examine, or in elementary programming terms, trace, a computation carried out by this Turing machine. First, we provide it with the input 1011 on its tape, place its head on the left endmarker (the #), and turn it loose.

Have a peek at figure 3. It is a sequence of snapshots of the machine in action. One should note that in the last snapshot (step 9) the machine is *not* about to execute an instruction. This is because it has halted.

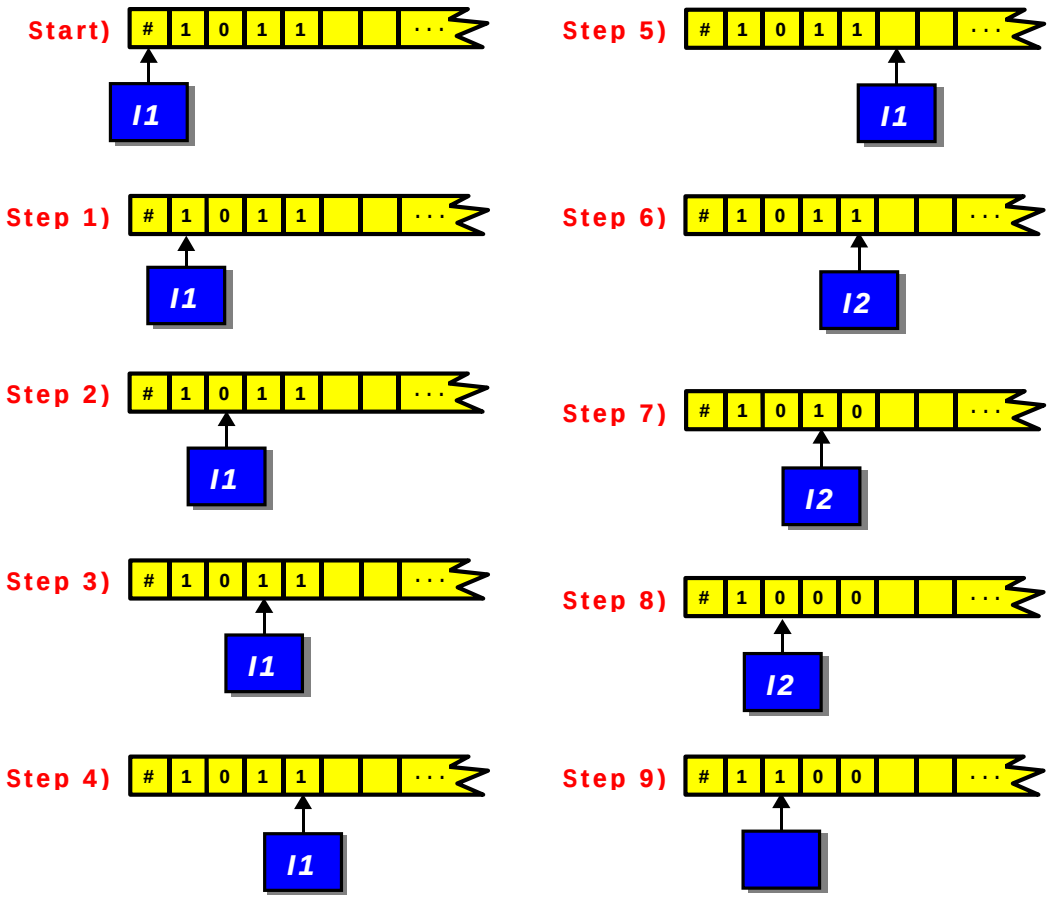


Figure 3 - Turing Machine Computation

Now that we have seen a Turing machine in action let us note some features, or properties of this class of computational devices.

a) There are no space or time constraints.
b) They may use numbers (or strings) of any size.
c) Their operation is quite simple - they read, write, and move.

In fact, Turing machines are *merely programs written in a very simple language.* Everything is finite and in general rather uncomplicated So, there is not too much to learn if we wish to use them as a computing device. Well, maybe we should wait a bit before believing that!

For a moment we shall return to the previous machine and discuss its efficiency. If it receives an input consisting only of ones (for example: 111111111), it must:

1) Go to the right end of the input,
2) Return to the left end marker, and
3) Go back to the right end of the input.

This means that *it runs for a number of steps more than three times the length of its input.* While one might complain that this is fairly slow, the machine does do the job! One might ask if a more efficient machine could be designed to accomplish this task? Try that as an amusing exercise.

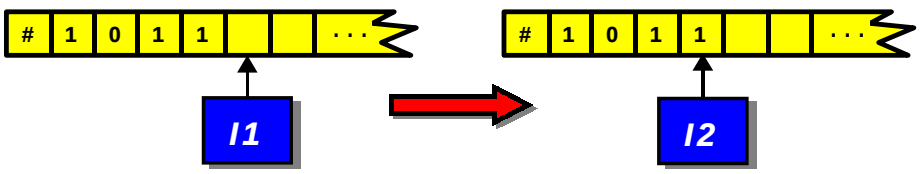
Another thing we should note is that when we present the machine with a blank tape it runs for a few steps and gets stuck on instruction I3 where no action is indicated for the configuration it is in since it is reading a blank instead of a zero. Thus it cannot figure out what to do. We say that this in an *undefined computation* and we shall examine situations such as this quite carefully later.

Up to this point, our discussion of Turing machines has been quite intuitive and informal. This is fine, but if we wish to study them as a model of computation and perhaps even prove a few theorems about them we must be a bit more precise and specify exactly what it is we are discussing. Let us begin.

A *Turing machine instruction* (we shall just call it an *instruction*) is a box containing *read-write-move-next* quadruples. A *labeled instruction* is an instruction with a label (such as I46) attached to it. Here is the entire machine.

> **Definition.** *A **Turing machine** is a finite sequence of labeled instructions with labels numbered sequentially from I1.*

Now we know precisely what Turing machines are. But we have yet to define what they do. Let's begin with pictures and then describe them in our definitions. Steps five and six of our previous computational example (figure 3) were the *machine configurations:*



If we translate this picture into a string, we can discuss what is happening in prose. We must do so if we wish to define precisely what Turing machines accomplish. So, place the instruction to be executed next to the symbol being read and we have an encoding of this change of configurations that looks like:

$$\#1011(I1)b... \rightarrow \#101(I2)1b...$$

This provides the same information as the picture. It is almost as if we took a snapshot of the machine during its computation. Omitting trailing blanks from the description we now have the following computational step

$$\#1011(I1) \rightarrow \#101(I2)1$$

Note that we shall always assume that there is an arbitrarily long sequence of blanks to the right of any Turing machine configuration.

> **Definition.** *A **Turing machine configuration** is a string of the form x(In)y or x where n is an integer and both x and y are (possibly empty) strings of symbols used by the machine.*

So far, so good. Now we need to describe how a machine goes from one configuration to another. This is done, as we all know by applying the instruction mentioned in a configuration to that configuration thus producing another configuration. An example should clear up any problems with the above verbosity. Consider the following instruction.

I17

0	**1**	**right**	**next**
1	***b***	**right**	**I3**
b	**1**	**left**	**same**
#	**#**	**halt**	

Now, observe how it transforms the following configurations.

a) #1101(I17)01 → #11011(I18)1
b) #110(I17)101 → #110*b*(I3)01
c) #110100(I17) → #11010(I17)01
d) (I17)#110100 → #110100

Especially note what took place in (c) and (d). Case (c) finds the machine at the beginning of the blank fill at the right end of the tape. So, it jots down a 1 and moves to the left. In (d) the machine reads the endmarker and halts. This is why the instruction disappeared from the configuration.

> **Definition.** *For Turing machine configurations C_i and C_j, C_i **yields** C_j (written $C_i \rightarrow C_j$) if and only if applying the instruction in C_i produces C_j .*

In order to be able to discuss a sequence of computational steps or an entire computation at once, we need additional notation.

Definition. *If C_i and C_j are Turing machine configurations then C_i* ***eventually yields*** *C_j (written $C_i \Rightarrow C_j$) if and only if there is a finite sequence of configurations $C_1, C_2, \ldots, C_k$ such that:*

$$C_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k = C_j .$$

At the moment we should be fairly at ease with Turing machines and their operation. The concept of computation taking place when a machine goes through a sequence of configurations should also be comfortable.

Let us turn to something quite different. What about configurations which do not yield other configurations? They deserve our attention also. These are called *terminal configurations* because they terminate a computation). For example, given the instruction:

I3				
	0	**1**	**halt**	
	1	***b***	**right**	**next**
	#	**#**	**left**	**same**

what happens when the machine gets into the following configurations?

a) (I3)#01101
b) #1001(I3)*b*10
c) #100110(I3)
d) #101011

Nothing happens - right? If we examine the configurations and the instruction we find that the machine cannot continue for the following reasons (one for each configuration).

a) The machine moves left and falls off of the tape.
b) The machine does not know what to do.
c) Same thing. A trailing blank is being scanned.
d) Our machine has halted.

Thus none of those configurations lead to others. Furthermore, any computation or sequence of configurations containing configurations like them must terminate immediately.

By the way, configuration (d) is a favored configuration called a *halting configuration* because it was reached when the machine wished to halt. For example, if our machine was in the configuration #10(*I3*)0011 then the next configuration would be #101011 and no other configuration could follow. These halting configurations will pop up later and be of very great interest to us.

We name individual machines so that we know exactly which machine we are discussing at any time. We will often refer to them as M_1, M_2, M_3, or M_i and M_k. The notation $M_i(x)$ means that Turing machine M_i has been presented with x as its input. We shall use the name of a machine as the function it computes.

> *If the Turing machine M_i is presented with x as its input and eventually halts (after computing for a while) with z written on its tape, we think of M_i as a function whose value is z for the input x.*

Let us now examine a machine that expects the integers x and y separated by a blank as input. It should have an initial configuration resembling #x*b*y.

erase x, find first symbol of y

I1	**#**	**#**	**right**	**same**
	0	***b***	**right**	**same**
	1	***b***	**right**	**same**
	b	***b***	**right**	**next**

get next symbol of y - mark place

I2	**0**	*****	**left**	**next**
	1	*****	**left**	**I5**
	b	***b***	**halt**	

find right edge of output - write 0

I3	**b**	**b**	**left**	**same**
	#	**#**	**right**	**next**
	0	**0**	**right**	**next**
	1	**1**	**right**	**next**
I4	**b**	**0**	**right**	**I7**

find right edge of output - write 1

I5	**b**	**b**	**left**	**same**
	#	**#**	**right**	**next**
	0	**0**	**right**	**next**
	1	**1**	**right**	**next**
I6	**b**	**1**	**right**	**next**

find the * and resume copying

I7	**b**	**b**	**right**	**same**
	*	**b**	**right**	**I2**

Figure 4 - Selection Machine

The Turing machine in figure 4 is what we call a *selection* machine. These receive several numbers (or strings) as input and select one of them as their output. This one computes the function: M(x, y) = y and selects the second of its inputs. This of course generalizes to any number of inputs, but let us not get too carried away.

Looking carefully at this machine, it should be obvious that it:

1) erases x, and
2) copies y next to the endmarker (#).

But, what might happen if either x or y happens to be blank? Figure it out! Also determine exactly how many steps this machine takes to erase x and copy y. (The answer is about n^2 steps if x and y are each n bits in length.)

Here is another Turing machine.

find the right end of the input

I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	**b**	**left**	**next**

is low order bit is 0 or 1?

I2	**0**	**b**	**left**	**next**
	1	**b**	**left**	**I5**
	#	**#**	**right**	**I6**

erase input and print 1

I3	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I4	**b**	**1**	**halt**	

erase input and print 0

I5	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I6	**b**	**0**	**halt**	

Figure 5 - Even Integer Acceptor

It comes from a very important family of functions, one which contains functions that compute relations (or predicates) and membership in sets. These are known as *characteristic functions,* or *0-1 functions* because they only take values of zero and one which denote false and true.

An example is the characteristic function for the set of even integers computed by the Turing machine of figure 5. It may be described:

$$\text{even}(x) = \begin{cases} 1 \text{ if } x \text{ is even} \\ 0 \text{ otherwise} \end{cases}$$

This machine leaves a one upon its tape if the input ended with a zero (thus an even number) and halts with a zero on its tape otherwise (for a blank or odd integers). It should not be difficult to figure out how many steps it takes for an input of length n.

Now for a quick recap and a few more formal definitions. We know that Turing machines *compute functions.* Also we have agreed that if a machine receives x as an input and halts with z written on its tape, or in our notation:

$$(I1)\#x \Rightarrow \#z$$

then we say that M(x) = z. When machines never halt (that is: run forever or reach a non-halting terminal configuration) for some input x we claim that the value of M(x) is *undefined* just as we did with programs. Since output and halting are linked together, we shall precisely define halting.

> **Definition.** *A Turing machine **halts** if and only if it encounters a halt instruction during computation and **diverges** otherwise.*

So, we have machines that always provide output and some that do upon occasion. Those that always halt compute what we shall denote the ***total functions*** while the others merely compute ***partial functions***.

We now relate functions with sets by discussing how Turing machines may characterize the set by deciding which inputs are members of the set and which are not.

> **Definition.** *The Turing machine M **decides membership** in the set A if and only if for every x, if $x \in A$ then M(x) = 1, otherwise M(x) = 0.*

There just happens to be another method of computing membership in sets. Suppose you only wanted to know about members in some set and did not care

at all about elements that were not in the set. Then you could build a machine which halted when given a member of the set and *diverged* (ran forever or entered a non-halting terminal configuration) otherwise. This is called *accepting* the set.

> **Definition.** *The Turing machine M **accepts** the set A if and only if for all x, M(x) halts for x in A and diverges otherwise.*

This concept of acceptance may seem a trifle bizarre but it will turn out to be of surprising importance in later chapters.

A Smaller Programming Language

At this point two rather different models or systems of computation have been presented and discussed. One, programs written in the *NICE* programming language, has a definite computer science flavor, while the other, Turing machines, comes from mathematical logic. Several questions surface.

- *which system is better?*
- *is one system more powerful than the other?*

The programming language is of course more comfortable for us to work with and we as computer scientists tend to believe that programs written in similar languages can accomplish any computational task one might wish to perform. Turing machine programs are rather awkward to design and there could be a real question about whether they have the power and flexibility of a modern programming language.

In fact, many questions about Turing machines and their power arise. Can they deal with real numbers? arrays? Can they execute *while* statements? In order to discover the answers to our questions we shall take what may seem like a rather strange detour and examine the *NICE* programming language in some detail. We will find that many of the features we hold dear in programming languages are not necessary (convenient, yes, but not necessary) when our aim is only to examine the power of a computing system.

To begin, what about numbers? Do we really need all of the numbers we have in the *NICE* language? Maybe we could discard half of them.

Negative numbers could be represented as positive numbers in the following manner. If we represent numbers using *sign plus absolute value* notation, then with companion variables recording the signs of each of our original variables we can keep track of all values that are used in computation. For example, if the variable x is used, we shall introduce another named signx that will have the value 1 if x is positive and 0 if x is negative. For example:

value	x	signx
19	19	1
-239	239	0

Representing numbers in this fashion means that we need not deal with negative numbers any longer. But, we shall need to exercise some caution while doing arithmetic. Employing our new convention for negative numbers, multiplication and division remain much the same although we need to be aware of signs, but addition and subtraction become a bit more complicated. For example, the assignment statement z = x + y becomes the following.

```
if signx = signy then
   begin z = x + y; signz = signx end
else if x > y
   then begin z = x - y; signz = signx end
   else begin z = y - x; signz = signy end
```

This may seem a bit barbaric, but it does get the job done. Furthermore, it allows us to state that we need only *nonnegative numbers.*

[NB. An interesting side effect of the above algorithm is that we now have two different zeros. Zero can be positive or negative, exactly like some second-generation computers. But this will not effect arithmetic as we shall see.]

Now let us rid ourselves of *real* or ***floating point numbers***. The standard computer science method is to represent the number as an integer and specify where the decimal point falls. Another companion for each variable (which we shall call pointx) is now needed to specify how many digits lie behind the decimal point. Here are three examples.

value	x	signx	pointx
537	537	1	0
0.0025	25	1	4
-6.723	6723	0	3

Multiplication remains rather straightforward, but if we wish to divide, add, or subtract these numbers we need a *scaling* routine that will match the decimal points. In order to do this for x and y, we must know which is the greater number. If pointx is greater than pointy we scale y with the following code:

```
while pointy < pointx do
   begin
     y = y*10;
     pointy = pointy + 1
   end
```

and then go through the addition routine. Subtraction (x - y) can be accomplished by changing the sign (of y) and adding.

As mentioned above, multiplication is rather simple because it is merely:

```
z = x*y;
pointz = pointx + pointy;
if signx = signy then signz = 1
                 else signz = 0;
```

After scaling, we can formulate division in a similar manner.

Since numbers are never negative, a new sort of subtraction may be introduced. It is called *proper subtraction* and it is defined as:

$$x - y = \text{maximum}(0, x - y).$$

Note that the result never goes below zero. This will be useful later.

A quick recap is in order. None of our arithmetic operations lead below zero and our only numbers are the *nonnegative integers.* If we wish to use negative or real (floating point) numbers, we must now do what folks do at the machine language level; fake them!

Now let's continue with our mutilation of the *NICE* language and destroy expressions! ***Boolean expressions*** are easy to compute in other ways if we think about it. We do not need $E_1 > E_2$ since it is the same as:

$$E_1 \geq E_2 \text{ and not } E_1 = E_2$$

Likewise for $E_1 < E_2$. With proper subtraction, the remaining simple Boolean arithmetic expressions can be formulated arithmetically. Here is a table of substitutions. *Be sure to remember that we have changed to proper subtraction and so a small number minus a large one is zero.*

$E_1 \leq E_2$	$E_1 - E_2 = 0$
$E_1 \geq E_2$	$E_2 - E_1 = 0$
$E_1 = E_2$	$(E_1 - E_2) + (E_2 - E_1) = 0$

This makes the Boolean expressions found in *while* and *if* statements less complex. We no longer need to use relational operators since we can we assign

these expressions to variables as above and then use those variables in the *while* or *if* statements. Only the following two Boolean expressions are needed.

x = 0
not x = 0

Whenever a variable such as z takes on a value greater than zero, the (proper subtraction) expression 1 - z turns its value into zero. Thus Boolean expressions which employ *logical connectives* may be restated arithmetically. For example, instead of asking if x is not equal to 0 (i.e. *not* x = 0), we just set z to 1 - x and check to see if z is zero. Thee transformations necessary are included in the chart below and are followed by checking z for zero.

`not x = 0`	`z = 1 - x`
`x = 0 and y = 0`	`z = x + y`
`x = 0 or y = 0`	`z = x*y`

Using these conversions, Boolean expressions other than those of the form x = 0 are no longer found in our programs.

Compound arithmetic expressions are not necessary either. We shall just break them into sequences of statements that possess one operator per statement. Now *we no longer need compound expressions of any kind!*

What next? Well, for our finale, let's remove all of the wonderful features from the *NICE* language that took language designers years and years to develop.

a) ***Arrays***. We merely encode the elements of an array into a simple variable and use this. (This transformation appears as an exercise!)

b) Although ***while*** statements are among the most important features of structured programming, a statement such as:

while x = 0 do S

(recall that only x = 0 exists as a Boolean expression now) is just the same computationally as:

```
10: z = 1 - x;
    if z = 0 then goto 20;
    S;
    goto 10
20: (* next statement *)
```

c) The ***case*** statement is easily translated into a barbaric sequence of tests and transfers. For example, consider the statement:

$$case\ E\ of:\ N_1: S_1;\ \ N_2: S_2;\ \ N_3: S_3\ endcase$$

Suppose we have done some computation and set x, y, and z such that the following statements hold true.

$$\text{if } x = 0 \text{ then } E = N_1$$
$$\text{if } y = 0 \text{ then } E = N_2$$
$$\text{if } z = 0 \text{ then } E = N_3$$

Now the following sequence is equivalent to the original *case*.

```
    if x = 0 then goto 10;
    if y = 0 then goto 20;
    if z = 0 then goto 30;
    goto 40;
10: begin S1; goto 40 end;
20: begin S2; goto 40 end;
20: begin S3; goto 40 end;
40: (* next statement *)
```

d) ***if-then-else*** and ***goto*** statements can be simplified in a manner quite similar to our previous deletion of the *case* statement. Unconditional transfers (such as *goto* 10) shall now be a simple *if* statement with a little preprocessing. For example:

```
z = 0;
if z = 0 then goto 10;
```

And, with a little bit of organization we can remove any Boolean expressions except x = 0 from if statements. Also, the *else* clauses may be discarded after careful substitution.

e) ***Arithmetic.*** Let's savage it almost completely! Who needs *multiplication* when we can compute z = x*y iteratively with:

```
z = 0;
for n = 1 to x do z = z + y;
```

Likewise *addition* can be discarded. The statement z = x + y can be replaced by the following.

```
z = x;
for n = 1 to y do z = z + 1;
```

The removal of *division* and *subtraction* proceeds in much the same way. All that remains of arithmetic is *successor* (x + 1) and *predecessor* (x - 1).

While we're at it let us drop simple assignments such as x = y by substituting:

```
x = 0;
for i = 1 to y do x = x + 1;
```

f) The ***for*** statement. Two steps are necessary to remove this last vestige of civilization from our previously *NICE* language. In order to compute:

for i = m to n do S

we must initially figure out just how many times S is to be executed. We would like to say that;

$$t = n - m + 1$$

but we cannot because we removed subtraction. We must resort to:

```
    z = 0;
    t = n;
    t = t + 1;
    k = m;
10: if k = 0 then goto 20;
    t = t - 1;
    k = k - 1;
    if z = 0 then go to 10;
20: i = m;
```

(Yes, yes, we cheated by using k=m and i=m! But nobody wanted to see the loops that set k and i to m. OK?) Now all we need do is to repeat *S* over and over again t times. Here's how:

```
30: if t = 0 then goto 40;
    S;
    t = t - 1;
    i = i + 1;
    if z = 0 then go to 30;
40: (* next statement *);
```

Loops involving *downto* are similar.

Now it is time to pause and summarize what we have done. We have removed most of the structured commands from the *NICE* language. Our deletion strategy is recorded the table of figure 1. Note that statements and structures used in removing features are not themselves destroyed until later.

Category	*Item Deleted*	*Items Used*
Constants	**negative numbers** **floating point numbers**	**extra variables** ***case*** **extra variables** ***while***
Boolean	**arithmetic operations** **logical connectives**	**arithmetic**
Arrays	**arrays**	**arithmetic**
Repetition	***while***	***goto, if-then-else***
Selection	***case, else***	***if-then***
Transfer	**unconditional *goto***	***if-then***
Arithmetic	**multiplication** **addition** **division** **subtraction** **simple assignment**	**addition, *for*** **successor, *for*** **subtraction, *for*** **predecessor, *for*** **successor, *for, if-then***
Iteration	***for***	***If-then*, successor, predecessor**

Figure 1 - Language Destruction

We have built a smaller programming language that seems equivalent to our original *NICE* language. Let us call it the *SMALL* programming language and now precisely define it.

In fact, we shall start from scratch. A ***variable*** is a string of lower case Roman letters and if x is an arbitrary variable then an (*unlabeled)* ***statement*** takes one of the following forms.

```
x = 0
x = x + 1
x = x - 1
if x = 0 then goto 10
halt(x)
```

In order to stifle individuality, we mandate that statements *must have* ***labels*** that are just integers followed by colons and are attached to the left-hand sides of statements. A ***title*** is again the original input statement and program heading. As before, it looks like the following.

```
program name(x, y, z)
```

> **Definition.** *A* ***program*** *consists of a title followed by a sequence of consecutively labeled instructions separated by semicolons.*

An example of a program in our new, unimproved *SMALL* programming language is the following bit of code. We know that *it is a program* because it conforms to the syntax definitions outlined above. (It does addition, but we do not know this yet since the semantics of our language have not been defined.)

```
program add(x, y);
1: z = 0
2: if y = 0 then goto 6;
3: x = x + 1;
4: y = y - 1;
5: if z = 0 then go to 2;
6: halt(x)
```

On to semantics! We must now describe computation or the execution of *SMALL* programs in the same way that we did for Turing machines. This shall be carried out in an informal manner, but the formal definitions are quite similar to those presented for Turing machine operations in the last section.

Computation, or running *SMALL* language programs causes the value of variables to change throughout execution. In fact, this is all computation entails. So, during computation we must show what happens to variables and their values. A variable and its value can be represented by the pair:

$$\langle x_i, v_i \rangle$$

If at every point during the execution of a program we know the environment, or the contents of memory, we can easily depict a computation. Thus knowing what instruction we are about to execute and the values of all the variables used in the program tells us all we need know at any particular time about the program currently executing.

Very nearly as we did for Turing machines, we define a *configuration* to be the string such as:

$$k \langle x_1, v_1 \rangle \langle x_2, v_2 \rangle \ldots \langle x_n, v_n \rangle$$

where k is an instruction number (of the instruction about to be executed), and the variable-value pairs show the current values of all variables in the program.

The manner in which one configuration *yields* another should be rather obvious. One merely applies the instruction mentioned in the configuration to the proper part of the configuration, that is, the variable in the instruction. The only minor bit of defining we need to do is for the halt instruction. As an example, let instruction five be *halt*(z). Then if x, y, and z are all of the variables in the program, we say that:

$$5 \langle x, 54 \rangle \langle y, 23 \rangle \langle z, 7 \rangle \rightarrow 7$$

Note that a configuration may be either an integer followed by a sequence of variable-value pairs or an integer. Also think about why a configuration is an integer. This happens *if and only if a program has halted*. We may now reuse the Turing machine system definition for *eventually yielding* and computation has almost been completely defined.

Initially the following takes place when a SMALL program is executed.

a) input variables are set to their input values
b) all other variables are set to zero
c) execution begins with instruction number one

From this we know what an *initial configuration* looks like.

Halting configurations were defined above to be merely numbers. *Terminal configurations* are defined in a manner almost exactly the same as for Turing machines. We recall that this indicates that terminal configurations might involve undefined variables and non-existent instructions.

Since this is the stuff detected by compilers, here is a point to ponder. Are there any more things that might pop up and stop a program?

We will now claim that *programs compute functions* and that all of the remaining definitions are merely those we used in the section about Turing machines. The formal statements of this is left as an exercise.

At this point we should believe that any program written in the *NICE* language can be rewritten as a *SMALL* program. After all, we went through a lot of work to produce the *SMALL* language! This leads to a characterization of the computable functions.

> **Definition.** *The **computable functions** are exactly those computed by programs written in the SMALL programming language.*

Equivalence of the Models.

Our discussion of what comprises computation and how exactly it takes place spawned three models of computation. There were two programming languages (the *NICE* language and the *SMALL* language) and Turing Machines. These came from the areas of mathematical logic and computer programming.

It might be very interesting to know if any relationships between three systems of computation exist and if so, what exactly they are. An obvious first question to ask is whether they allow us to compute the same things. If so, then we can use any of our three systems when demonstrating properties of computation and know that the results hold for the other two. This would be rather helpful.

First though, we must define exactly what we mean by *equivalent programs* and *equivalent models of computation*. We recall that both machines and programs compute functions and then state the following.

> **Definition.** *Two **programs** (or **machines**) are **equivalent** if and only if they compute exactly the same function.*
>
> **Definition.** *Two **models of computation** are **equivalent** if and only if the same exact groups of functions can be computed in both systems.*

Let us look a bit more at these rather official and precise statements. How do we show that two systems permit computation of exactly the same functions? If we were to show that Turing Machines are equivalent to *NICE* programs, we should have to demonstrate:

- For each *NICE* program there is an equivalent Turing machine
- For each Turing machine there is an equivalent *NICE* program

This means that we must prove that for each machine M there is a program P such that for all inputs x: M(x) = P(x) and *vice versa*.

An fairly straightforward equivalence occurs as a consequence of the language destruction work we performed in painful detail earlier. We claim that our two languages compute exactly the same functions and shall provide an argument for this claim in the proof of theorem 1.

(In the sequel, we shall use short names for our classes of functions for the sake of brevity. The three classes mentioned above shall be *TM*, *NICE*, and *SMALL*.)

Theorem 1. *The following classes of functions are equivalent:*
a) the computable functions,
b) functions computable by NICE programs, and
c) functions computable by SMALL programs.

Informal Proof. We know that the classes of computable functions and those computed by *SMALL* programs are identical because we defined them to be the same. Thus by definition, we know that:

computable = *SMALL*.

The next part is almost as easy. If we take a *SMALL* program and place *begin* and *end* block delimiters around it, we have a *NICE* program since all *SMALL* instructions are *NICE* too (in technical terms). This new program still computes exactly the same function in exactly the same manner. This allows us to state that:

computable = *SMALL* ⊂ *NICE*.

Our last task is not so trivial. We must show that for every *NICE* program, there is an equivalent *SMALL* program. This will be done in an informal but hopefully believable manner based upon the section on language destruction.

Suppose we had some arbitrary *NICE* program and went through the step-by-step transformations upon the statements of this program that turn it into a *SMALL* program. If we have faith in our constructions, the new *SMALL* program computes exactly same function as the original *NICE* program. Thus we have shown that

computable = *SMALL* = *NICE*

and this completes the proof.

That was really not so bad. Our next step will be a little more involved. We must now show that Turing machines are equivalent to programs. The strategy will be to show that *SMALL* programs can be converted into equivalent Turing machines and that Turing machines in turn can be transformed into equivalent *NICE* programs. That will give us the relationship:

***SMALL* ⊂ *TM* ⊂ *NICE*.**

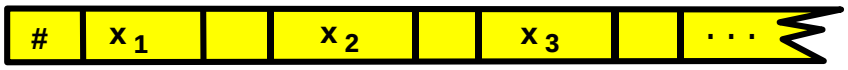
This relationship completes the equivalence we wish to show when put together with the equivalence of *NICE* and *SMALL* programs shown in the last theorem. Let us begin by transforming *SMALL* programs to Turing machines.

Taking an arbitrary *SMALL* program, we first reorganize it by *renaming the variables.* The new variables will be named x_1, x_2, ... with the input variables leading the list. An example of this is provided in figure 1.

```
program example(x, y)          program example(x1, x2)
1: w = 0;                      1: x3 = 0;
2: x = x + 1;                  2: x1 = x1 + 1;
3: y = y - 1;                  3: x2 = x2 - 1;
4: if y = 0 then goto 6;       4: if x2 = 0 then goto 6;
5: if w = 0 then goto 2;       5: if x3 = 0 then goto 2;
6: halt(x)                     6: halt(x1)
```

Figure 1 - Variable Renaming

Now we need to design a Turing machine that is equivalent to the *SMALL* program. The variables used in the program are stored on segments of the machine's tape. For the above example with three variables, the machine should have a tape that looks like the one shown below.



Note that each variable occupies a sequence of squares and that variables are separated by blank squares. If x_1 = 1101 and x_2 = 101 at the start of computation, then the machine needs to set x_3 to zero and create a tape like:

Now what remains is to design a Turing machine which will mimic the steps taken by the program and thus compute exactly the same function as the program in as close to the same manner as possible.

For this machine design we shall move to a general framework and consider what happens when we transform any *SMALL* program into a Turing machine.

We first set up the tape. Then all of the instructions in the *SMALL* program are translated into Turing machine instructions. A general schema for a Turing machine equivalent to a *SMALL* program with m instructions follows.

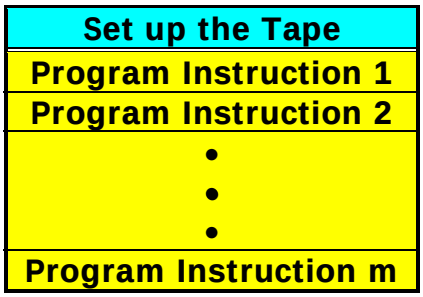


Figure 2 - *SMALL* Program Simulator

Each of the m+1 sections of the Turing machine in figure 2 contains several Turing machine instructions. Let us examine these sections.

Setting up the tape is not difficult. If the program uses x_1, ... x_n as variables and the first k are input parameters, then the tape arrives with the values of the first k variables written upon it in the proper format. Now space for the remaining variables (x_{k+1} through x_n) must be added to the end of the input section. To begin, we must go one tape square past the end of x_k. Since two adjacent blanks appear at the end of the input, the following instruction pair finds the square where x_{k+1} should be written.

#	#	right	same
0	0	right	same
1	1	right	same
b	*b*	right	next

0	0	right	previous
1	1	right	previous
b	*b*	left	next

Now that the tape head is on the blank following x_k we need to initialize the remaining variables (x_{k+1}, ... , x_n). This is easily done by n-k instruction pairs exactly like the following.

b	0	right	next

b	*b*	right	next

The *Program Instruction* sections contain sequences of Turing machine instructions that perform each corresponding *SMALL* program instruction. They are merely translations of each program instruction into a chain of Turing machine instructions.

Here is the general format for executing a *SMALL* program instruction.

a) Find the variable used in the program instruction
b) Modify it according to the program instruction
c) Prepare to execute the next instruction

In the following translation examples, we note that there is only one variable in each instruction. We shall assume that the instruction we are translating contains the variable named x_i.

To locate the variable x_i, we first move the tape head to the first character of x_1 by going all the way to the left endmarker and moving one tape square to the right with the instruction:

0	**0**	**left**	**same**
1	**1**	**left**	**same**
b	***b***	**left**	**same**
#	**#**	**right**	**next**

At this point we use i-1 instructions of the form:

0	**0**	**right**	**same**
1	**1**	**right**	**same**
b	***b***	**right**	**next**

to move right past the variables x_1, ... , x_{i-1} and place the tape head on the first character of x_i. Now we are ready to actually execute the instruction.

We shall now examine the instructions of the *SMALL* language one at a time and show how to execute them on a Turing machine. Recall that we begin execution with the tape head on the first character of the variable (x_i) mentioned in the program instruction.

a) The Turing machine executes **$x_i = 0$** by changing the characters of x_i to zero with the instruction:

0	**0**	**right**	**same**
1	**0**	**right**	**same**
b	**b**	**right**	**next**

b) To execute **$x_i = x_i - 1$** the Turing machine must change all lower order zeros to ones and then change the lowest one to zero if x_i is indeed greater than zero. (That is, convert 101100 to 101011.)

If x_i is zero then nothing must be changed since in the *SMALL* language there are no numbers less than zero. Recall that we use *proper* subtraction. One way to prevent this is to first modify the program so that we only subtract when x_i is greater than zero. Whenever we find subtraction, just insert a conditional instruction like this:

```
74: if xi = 0 then goto 76;
75: xi = xi - 1;
76:
```

Here is the pair of Turing machine instructions that accomplish proper predecessor .

move to the right end of x_i

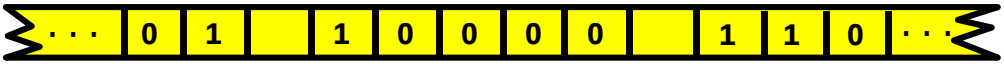
0	0	right	same
1	1	right	same
b	b	left	next

go back, flipping the bits

0	1	left	same
1	0	left	next

c) We designed a machine for **$x_i = x_i + 1$** as our first Turing machine example. This machine will not work properly if x_i is composed totally of ones though since we need to expand the space for x_i one square in that case. For example if the portion of tape containing x_i resembles this:

then adding one changes it to:



Thus we need to move the variables x_{i+1}, ... , x_n to the right one square before adding one to x_i. This is left as an exercise.

d) A check for zero followed by a transfer executes program instructions of the form **if x_i = 0 then goto 10**. If x_i is zero then the machine must transfer to the beginning of the section where program instruction 10 is executed.

Otherwise control must be transferred to the next program instruction. If I64 is the Turing machine instruction that begins the section for the execution of program instruction 10 then the correct actions are accomplished by the following instruction.

0	**0**	**right**	**same**
1	**1**	**right**	**next**
b	**b**	**right**	**I64**

e) The selector machine seen earlier computed the function f(x, y) = y. The **halt(x_i)** instruction is merely the more general $f(x_1, \ldots, x_n) = x_i$.

We have defined a construction method that will change a *SMALL* program into a Turing machine. We need now to show that the program and the machine are indeed equivalent, that is, they compute exactly the same functions for all possible inputs. We shall do this by comparing configurations in another informal argument.

Recalling our definitions of Turing machine configurations and program configurations, we now claim that if a program enters a configuration of the form:

$$k \langle x_1, v_1\rangle \langle x_2, v_2\rangle \ldots \langle x_n, v_n\rangle$$

(where the k-th instruction in the program refers to the variable x_i) then the Turing machine built from the program by the above construction will eventually enter an equivalent configuration of the form:

$$\#v_1 b v_2 b \ldots (\mathrm{Im}) v_i b \ldots b v_n$$

Where Im is the Turing machine instruction at the beginning of the set of instructions that perform program instruction k. In other words, the machine and the program go through equivalent configurations. And, if the program's entire computation may be represented as:

$$1 \langle x_1, v_1\rangle \ \ldots \ \langle x_k, v_k\rangle \langle x_{k+1}, 0\rangle \ \ldots \ \langle x_k, 0\rangle \Rightarrow z$$

then the Turing machine will perform the computation:

$$(\mathrm{I1})\#v_1 b \ldots b v_k \Rightarrow \#z$$

On the strength of this informal argument we now state the relationship between Turing machines and *SMALL* programs as our second theorem.

Theorem 2. *For every SMALL program there is a Turing machine which computes exactly the same function.*

An interesting question about our translation of programs into Turing machines involves *complexity*. The previous transformation changed a *SMALL* program into a larger Turing machine and it is obvious that the machine runs for a longer period of time than the program. An intellectually curious reader might work out just what the relationship between the execution times might be.

The last step in establishing the equivalence of our three models of computation is to show that *NICE* programs are at least as powerful as Turing machines. This also will be carried out by a construction, in this case, one that transforms Turing machines into programs.

First we need a data structure. The Turing machine's tape will be represented as an array of integers. Each element of the array (named *tape* of course) will have the contents of a tape square as its value. Thus tape[54] will contain the symbol from the Turing machine's alphabet that is written on the tape square which is 54th from the left end of the tape. In order to cover Turing machine alphabets of any size we shall encode machine symbols. The following chart provides an encoding for our standard binary alphabet, but it extends easily to any size alphabet.

TM symbol	b	0	1	#
tape[i]	0	1	2	3

If a Turing machine's tape contains:

then the program's array named *tape* would contain:

tape = <3, 2, 2, 1, 2, 0, 0, ... >

Thinking ahead just a little bit, what we are about to do is formulate a general method for translating Turing machines into programs that simulate Turing machines. Two indexing variables shall prove useful during our simulation. One is named *head* and its value denotes the tape square being read by the machine. The other is called *instr* and its value provides the number of the Turing machine instruction about to be executed.

The simulation proceeds by manipulating these variables. To move, we change the variable named *head* and to go to another instruction, the variable named *instr* must be modified. In this manner, the Turing machine instruction components translate into program instructions very simply as follows.

move	
left	**head = head - 1**
right	**head = head + 1**

go to	
I43	**instr = 43**
same	
next	**instr = instr + 1**

It should be almost obvious how we shall program a Turing machine instruction. For example, the instruction:

0	**1**	**left**	**next**
1	**1**	**left**	**same**
b	**0**	**halt**	
#	**#**	**right**	**I32**

is programmed quite easily and elegantly in figure 3.

```
case tape[head] of
  1: begin                      a zero is read
       tape[head] = 2;            print 1
       head = head - 1;           move left
       instr = instr + 1          goto next
     end;
  2: head = head -1;            a one is read
  0: begin                      a blank is read
       tape[head] = 1;            print 0
       halt(tape)                 output = tape[]
     end;
  3: begin                      a # is read
       head = head + 1;           move right
       instr = 32                 goto I32
     end
endcase
```

Figure 3 - Translating a Turing Machine Instruction

The next step is to unite all of the machine's instructions or actually the *case* statements they have now become. To do this we merely place the *case* statements for all of the Turing machine instructions in order and surround sequence by a *case* on the instruction number of the Turing machine. The general schema is provided as figure 4.

```
case instr of
   1: case tape[head] of
   2: case tape[head] of
            •
            •
            •
   n: case tape[head] of
endcase
```

Figure 4 - All of a Turing Machine's Instructions

Since a Turing machine starts operation on its first square and begins execution with its first instruction, we must set the variables named head and instr to one at the beginning of computation. Then we just run the machine until it encounters a *halt*, tries to leave the left end of the tape, or attempts to do something that is not defined. Thus a complete Turing machine simulation program is the instruction sequence surrounded by a *while* that states:

'*while the Turing machine keeps reading tape squares do …*'

and resembles that of figure 5.

```
program name(tape);
var instr, head: integer;
    tape[]: array of integer;

begin
  instr = 1; head = 1
  while head > 0 do
        case instr of
            •
            •
            •
        endcase
end
```

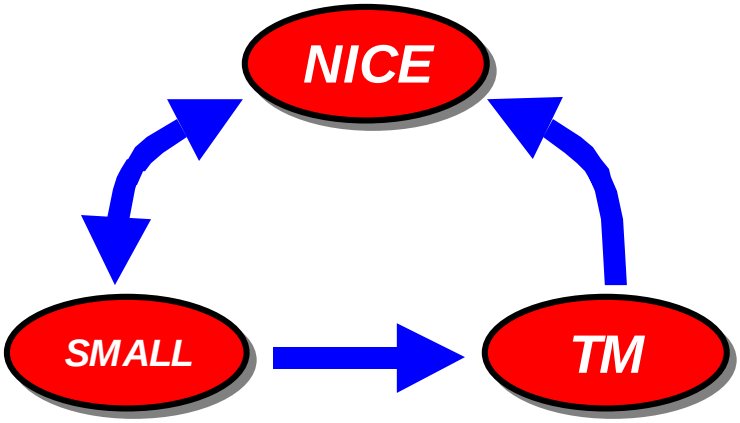
Figure 5 - Simulation of a Turing Machine

This completes the transformation.

If we note that the *NICE* program exactly simulates the Turing machine step for step, and that the termination conditions are identical for each, then we may state the following theorem.

Theorem 3. *Every function that is Turing machine computable can also be computed by a NICE program.*

What we have done in the last two sections is show that we can transform programs and machines. The diagram below illustrates these transformations.



The circularity of these transformations shows us that we can go from any computational model to any other. This now provides the proof for our major result on system equivalence.

Theorem 4. *The class of computable functions is exactly the class of functions computed by either Turing machines, SMALL programs, or NICE programs*

This result is very nice for two major reasons. First, it reveals that two very different concepts about computability turn out to be identical in a sense. This is possibly surprising. And secondly, whenever in the future we wish to demonstrate some property of the computable functions, we have a choice of using machines or programs. We shall use this power frequently in our study of the nature of computation.

Our examination of the equivalence of programs and machines brought us more than just three characterizations of the computable functions. In fact, a very interesting byproduct is that the proof of theorem 3 can be used to show a feature of the *NICE* language that programming language researchers find important, if not essential.

Careful examination of our constructions, especially the conversion of Turing machines to *NICE* programs, reveals the following very interesting fact about *NICE* programs.

Corollary. *Every NICE program may be rewritten so that it contains no goto statements and only one while statement.*

Proof Sketch. In order to remove *goto's* and extra loops from *NICE* programs, do the following. Take any *NICE* program and:

a) Convert it to a *SMALL* program.
b) Change this into a Turing machine.
c) Translate the machine into a *NICE* program.

The previous results in this section provide the transformations necessary to do this. And, if we examine the final construction from machines to *NICE* programs, we note that the final program does indeed meet the required criteria.

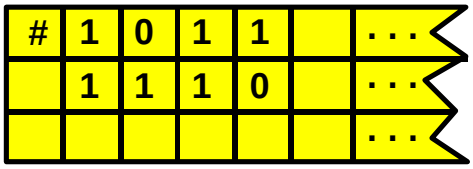
This result has often been referred to as *the fundamental theorem of structured programming* and as such has enjoyed a certain vogue among researchers in this area. Our proof does indeed support the thesis that all programs can be written in structured form. However the methods we employed in order to demonstrate this certainly do not testify to the advantages of structured programming - namely ease of algorithm design and readability of code.

Machine Enhancement

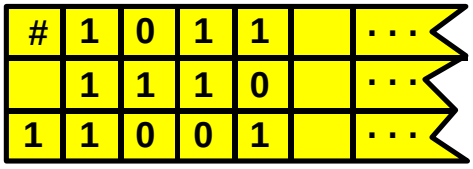
We know now that Turing machines and programming languages are equivalent. Also, we have agreed that they can be used to compute all of the things that we are able to compute. Nevertheless, maybe there is more that we can discover about the limits of computation.

With our knowledge of computer science, we all would probably agree that not too much can be added to programming languages that cannot be emulated in either the *NICE* or the *SMALL* language. But possibly a more powerful machine could be invented. Perhaps we could add some features to the rather restricted Turing machine model and gain computing power. This very endeavor claimed the attention of logicians for a period of time. We shall now review some of the more popular historical attempts to build a better Turing machine.

Let us begin by correcting one of the most unappetizing features of Turing machines, namely the linear nature of their tapes. Suppose we had a work surface resembling a pad of graph paper to work upon? If we wished to add two numbers it would be nice to be able to begin by writing them one above the other like the following.



Then we can add them together in exactly the manner that we utilize when working with pencil and paper. Sweeping the tape from right to left, adding as we go, we produce a tape like that below with the answer at the bottom.



What we have been looking at is the tape of a *3-track Turing machine.* Instructions for a device of this nature are not hard to imagine. Instead of

reading and writing single symbols, the machine reads and writes *columns* of symbols.

The 3-track addition process shown above could be carried out as follows. First, place the tape head at the right end of the input. Now execute the pair of instructions pictured in figure 1 on a sweep to left adding the two numbers together.

Add without carry

0 0 *b*	0 0 0	left	same
0 1 *b*	0 1 1	left	same
0 *b* *b*	0 *b* 0	left	same
1 0 *b*	1 0 1	left	same
1 1 *b*	1 1 0	left	next
1 *b* *b*	1 *b* 1	left	same
# *b* *b*	# *b* *b*	halt	

Add with carry

0 0 *b*	0 0 1	left	previous
0 1 b	0 1 0	left	same
0 *b* b	0 *b* 1	left	previous
1 0 *b*	1 0 0	left	same
1 1 *b*	1 1 1	left	same
1 *b* *b*	1 *b* 0	left	same
# *b* *b*	# b 1	halt	

Figure 1 - Addition using Three Tracks

Comparing this 3-track machine with a 1-track machine designed to add two numbers, it is obvious that the 3-track machine is:

- easier to design,
- more intuitive, and
- far faster

than its equivalent 1-track relative. One might well wonder if we added more power to Turing machines by adding tracks. But alas, this is not true as the proof of the following result demonstrates.

Theorem 1. *The class of functions computed by n-track Turing machines is the class of computable functions.*

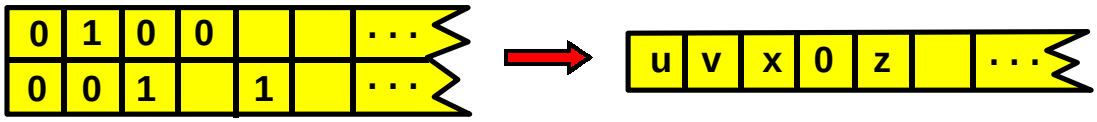
Proof Sketch. Since n-track machines can do everything that 1-track machines can do, we know immediately that every computable function can be computed by an n-track Turing machine. We must now show that for every n-track machine there is an equivalent 1-track machine.

We shall demonstrate the result for a specific number of tracks since concrete examples are always easier to formulate and understand than the general case. So, rather than show this result for all values of n at once, we shall validate it for 2-track machines and claim that extending the result to any number of tracks is an fairly simple expansion of our construction.

Our strategy is to encode columns of symbols as individual symbols. For two-track machines using zero, one and blank we might use the following encoding of pairs into single symbols.

column	0 0	1 0	*b* 0	0 1	1 1	*b* 1	0 *b*	1 *b*	*b* *b*
code	u	v	w	x	y	z	0	1	*b*

The encoding transforms 2-track tapes into 1-track tapes. Here is a simple example.



Now all that is needed is to translate the instructions for a 2-track machine into equivalent instructions for the 1-track machine that uses the encoding. For example, the 2-track machine instruction below may be replaced by that for a 1-track machine to the right.

0 0	1 0	right	next
1 0	1 1	left	same
b *b*	0 *b*	halt	

➔

u v *b*	v y 0	right left halt	next same

To finish the proof, we argue that if the original 2-track machine were run side-by-side with the new 1-track machine their computations would look identical after taking the tape encoding into account.

That takes care of the actual computation. However, we neglected to discuss input and output conventions. Input is simple, we merely assume that the 1-track machine begins computation with the 2-track input encoding upon the tape. Output must be tailored to individual machines. If the output is the two tracks, then all is well. If the output is on the top track, note that our encoding produces the proper answer. We could also cover bottom track output with a new encoding. Thus, the machines compute exactly the same function.

The proof sketch in the last theorem was a machine construction. The result could actually have been proven more quickly with a programming language argument. Recall the NICE language simulation of Turing machines in the model equivalence section. The program had a 1-dimensional array named tape[symbol] that held the symbols written on the Turing machine tape. If we replaced it with a 2-dimensional array tape[track, symbol] or a family of arrays:

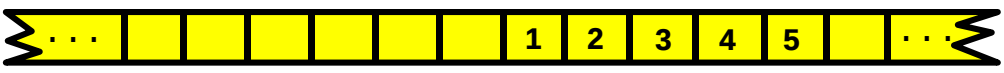
track1[symbol], ... , trackn[symbol]

we could easily simulate an n-track Turing machine. We chose however to examine a more historical machine construction similar to those used in proofs long before structured programming languages were invented.

We shall ponder one more aspect of k-track machines before moving along. Is complexity added to computation by using 1-track machines? Are k-track machines faster, and if so, just how much faster are they? Or, how much slower are 1-track Turing machines? An estimate can be obtained from the above construction.

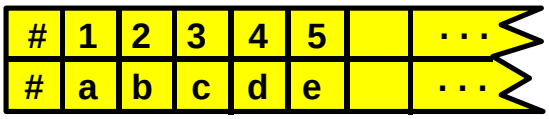
One of the ways in which a Turing machine could terminate its computation was to fall off the left end of the work tape. In the past, Turing machines were often defined with tapes that stretch arbitrarily far in both directions. This meant that the machine's tape head never left the tape. As one might expect, this adds no power to a Turing machine.

Let us explore a method for changing tapes with no ends into tapes with a left end. Suppose a Turing machine were to begin computation on a tape that looked like the following with blank fill on both ends.



and then moved left writing symbols until the tape resembled:

If we folded the tape in half we would retain all of the information as before, but written on a one-ended tape. After folding and inscribing endmarkers, the above tape would look like the 2-track tape pictured below.



We have folded a 1-track tape, arbitrarily long on both ends, onto a one-ended, 2-track tape and now need to transform the instructions of the old machine with the unending tape into those befitting a machine with two tracks on a one-ended tape. This is straightforward. We shall use two companion machines, one named M_t that operates only on the top track, and one named M_b that operates only on the bottom track. Consider the following 1-track instruction.

0 1	1 0	right left	next same

This instruction translates into the following 2-track instructions, one for each of the companion machines. Note that M_b's moves are reversed.

M_t *(uses top track)*

0 0	1 0	right	next
0 1	1 1	right	next
0 *b*	1 *b*	right	next
1 0	0 0	left	same
1 1	0 1	left	same
1 *b*	0 *b*	left	same

M_b *(uses bottom track)*

0 0	0 1	left	next
1 0	1 1	left	next
b 0	*b* 1	left	next
0 1	0 0	right	same
1 1	1 0	right	same
b 1	*b* 0	right	same

When either machine arrives at the leftmost squares (those containing the endmarkers) a transfer is made to the other machine. Here are the details. Suppose the original 1-track machine had n instructions. This means that each of our companion machines, M_t and M_b, has n instructions. We now combine the

machines to form a machine with 2n instructions, the first n for M_t and the second n instructions (n+1, ... , 2n) for M_b. Thus instruction *Ik* of M_t is the companion of instruction *I(n+k)* in M_b.

All that remains is to coordinate the 2-track machine's moves so that it can switch between tracks. This happens when the machine hits the leftmost square, the one containing the endmarkers. We can make the machine bounce off the endmarkers and go to the corresponding instruction of the companion machine by adding the following lines to all of the instructions of M_t and M_b.

add to Ik of M_t

# #	# #	**right**	**I(n+k)**

add to I(n+k) of M_b

# #	# #	**right**	**Ik**

Since the 2-track machine of the previous construction carries out its computation in exactly the *same* manner (except for reading the endmarkers and switching tracks) as the original 1-track machine with an unending tape, we claim that they are equivalent. And, since we know how to change a 2-track machine into a 1-track machine, we state the following theorem without proof.

> **Theorem 2.** *Turing machines with arbitrarily long tapes in both directions compute exactly the class of computable functions.*

Thus far we have discovered that adding extra tracks or unending tapes to Turing machines need not increase their power. These additions were at some additional cost however. In fact, we were forced to introduce extra symbols. This cost us some speed also, but we will worry about that later.

Our next question is to inquire about exactly *how many symbols a Turing machine needs.* Obviously no computation can be done on a machine which uses only blanks since it cannot write. Thus we need at least one additional symbol. In fact, one more will do. But, the proof is easier (and much more fun to read) if we show that any of the computable functions can be computed using only blanks, zeros, and ones. The demonstration that they can be computed using only blanks and one additional symbol will be reserved for readers to carry out in the privacy of their own homes.

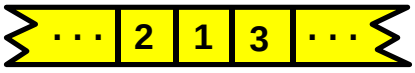
> **Theorem 3.** *Turing machines that use n symbols are no more powerful than those that use two symbols (and blanks).*

> **Proof sketch.** We shall simulate the operation of an n symbol machine with a machine that uses a binary alphabet. The special case of quaternary (n = 4) alphabets is explored below, but it is easily extended to any number of symbols.

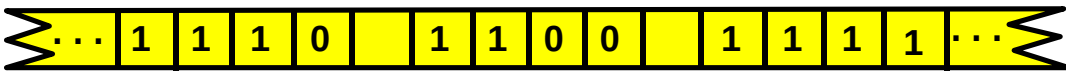
We must mimic a four-symbol computation on a binary tape. So, first we encode the four symbols (0, 1, 2, 3, and *b*) into the binary alphabet according to the following chart.

b	0	1	2	3
0000	1000	1100	1110	1111

We use a tape that is laid out in blocks of squares, one block for each symbol of the 4-symbol alphabet. Consider a tape for the 4-symbol machine that contains the following sequence.



Translating each symbol into a four-bit code with the above chart results in a tape like that below with a four bit block for each symbol.



We must now discuss input and output. If the 4-symbol machine uses binary numbers for both, the new binary machine must:

a) Change the binary input to encoded blocks,
b) Simulate the 4-symbol machine's computation, and
c) Translate the final tape to a binary number.

If the original machine does input and output in the 4-symbol alphabet then the new machine uses the encoding for this.

Steps (a) and (c) are left as exercises. Step (b) is done in the following four steps for each instruction of the original 4-symbol machine.

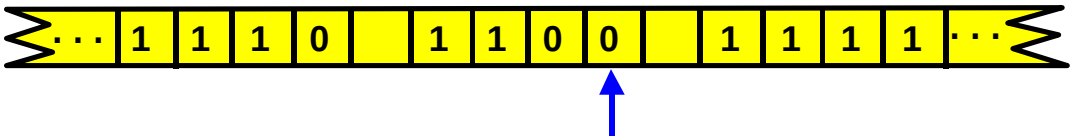
a) Determine what symbol occupies the current block.
b) Write the appropriate symbol in this block.
c) Move the tape head to the correct neighboring block.
d) Transfer control to the appropriate next instruction.

Note that we are merely doing for blocks exactly what was done before for symbols. It is really rather simple.

Now for the sordid details. For each instruction *Ik* of the 4-symbol machine we shall have a group of instructions. The first subgroup of each instruction group will read the encoded symbol. This subgroup begins with the instruction labeled *Ik-read*. Other subgroups will contain

instructions to do the writing, moving, and transfer for each particular symbol read. These subgroups begin with instructions labeled *Ik-saw-b*, *Ik-saw-1*, etc. depending upon which symbol (*b*, 0, 1, 2, 3) was detected by the *Ik-read* group.

It is time for an example. We pick up our new machine's computation with its head at the right end of a block like this:



and execute the reading part (labeled *Ik-read*) of the group of instructions for the original machine's instruction *Ik*. This set of instructions just counts the 1's in the block and reports the symbol encoded by the block.

0	0	left	same
1	1	left	next
b	*b*	right	Ik-saw-b

1	0	left	next
b	*b*	right	Ik-saw-0

1	0	left	next
b	*b*	right	Ik-saw-1

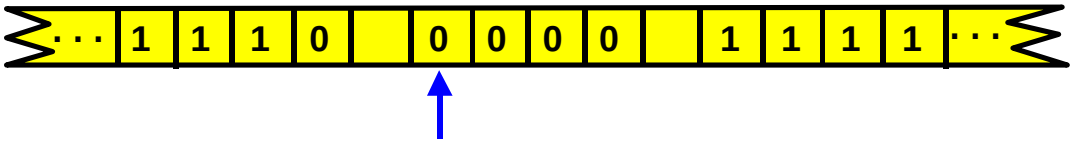
1	0	left	next
b	*b*	right	Ik-saw-2

b	*b*	right	Ik-saw-3

At this time our machine has

- read and decoded the symbol,
- erased the symbol in the block (written a blank),
- placed it's head at the left end of the block,
- and transferred to an instruction to do writing, moving, and transfer to a new instruction.

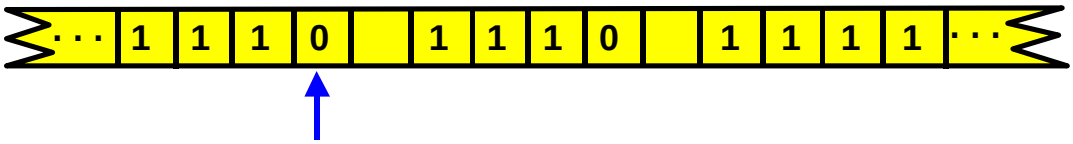
Now the tape looks like this:



We are ready to execute the instruction *Ik-saw-1*. *If Ik* of the original machine said to write a 2, move to the left, and transfer to *Im* upon reading a 1, then the instruction subgroup *Ik-saw-1* consists of:

0	1	right	next
0	1	right	next
0	1	left	next
1	1	left	same
b	*b*	left	Im-read

and after executing them, our new machine ends up in the configuration:



A move to the right is similar in that it merely skips over all of the zeros in the block just written and the next block as well in order to take up position at the right end of the block to the right. If no block exists to the right then a new one must be written containing a blank, but this is a rather simple task.

To finish up the proof, we must show that for each configuration in the computation of the 4-symbol machine, there is an equivalent encoded configuration in the computation of our binary machine. Thus for each and every step of the 4-symbol machine there is an equivalent sequence of steps which the new binary machine executes. From this we can conclude that both machines compute the same function.

(**N.B.** *An interesting problem concerning the transformation of one type of Turing machine to another is the amount of added complexity or steps needed by the new machine. Try to develop some general formulas for this.)*

Due to this sequence of theorems, it will be possible to limit our attention to Turing machines that use a binary alphabet whenever we wish to prove something about the computable functions. For this reason, there is no need to show things about machines with unending tapes, several tracks, or lots of symbols. In other words, whenever something is true for the functions computed by 1-track binary machines it is true for all of the Turing machine computable functions and for the class of computable functions.

So, when we wish to design a particular machine, we shall have no fear about including extra symbols or tracks since we know that if we wish, we can convert it to an equivalent binary, 1-track machine.

Other variations or options have been proposed and used since Turing's day. Features such as additional heads, extra tapes, and even machines which compute in several dimensions have been defined and proven to be equivalent to what we shall now call our *standard* (1-track, one tape, one head, binary alphabet) Turing machine.

N.B. *We note that the argument above and many of the arguments used in proof sketches are not complete proofs. This is not because full proofs are not needed in theoretical computer science - they are very important and will be provided for many theorems. But, proofs for simulation results are very long and tedious. For this reason we have often used informal proofs or proof sketches thus far and indicated how the sketch could be fleshed out to form a complete, detailed proof. It is expected that anyone with a bit of mathematical maturity could easily fill in the needed details and make the proof sketches complete.*

The Theses of Church and Turing

Many logicians and mathematicians have attempted to characterize the computable functions by defining systems in which computation could be carried out, or at least, described. Some of the historical highlights in the formulation of computation were due to Aristotle, Euclid, Frege, Hilbert, and Russell and Whitehead. Some of these mathematicians developed systems for fragments of arithmetic or geometry, but there were always problems and thus none could produce a system in which all of human computation could be formulated properly.

The mid-nineteen-thirties brought Alonzo Church and Alan Turing to the attention of mathematicians throughout the world. Both had developed very different systems in which computation could be carried out and each of them felt that any computation which was humanly or mechanically possible could be carried out within the systems of computation which they formulated. They also felt that under certain common sense rules, no systems of computation could be more powerful than theirs. In other words, they believed (as many others had before) that they had found an answer to a question that had eluded scholars for over two thousand years!

These beliefs have become known as *Church's Thesis* and *Turing's Thesis.* We must note that *neither belief can be proven*, and this is not too difficult to see. After all, how can one prove that all things which are computable can be done by Turing machines unless one somehow lists all of these things and then shows how to compute them. It is the same with systems. How do we know that someone will not propose a system for computation next week that is more powerful than Turing machines?

But, almost everyone *does* believe Church and Turing. This is because lots of evidence for these two beliefs has been presented to date. We shall delve into this after we have precisely formulated the theses of Church and Turing.

First, we need to explore just what is meant by human computation. Let us look closely at programs and machines. They share two major features. First, all computations are ***finitely specified***. This means that the set of instructions followed to carry out a computation must be finite. Exactly like a recipe or a program.

The second shared principle is a little more complicated. We note that all computations carried out in these systems are ***effective*** and ***constructive***. By effective we mean actual or real. No imaginary computations are allowed. By

constructive we mean that it must be possible to show exactly how the computation is performed. We need to be able to reveal the actual sequence of computational steps.

Let us resort to a few examples. Functions we often compute such as:

$$x + y \quad \text{or} \quad x \leq y + 3$$

are constructive because we know exactly how to compute them using simple arithmetic or logical algorithms. We know all of the steps needed to *exactly construct* the answer. We could even design and build an actual computer chip to do the job. There is absolutely no question at all about it. Something a little more difficult like:

$$f(n) = \text{the } n^{th} \text{ digit of } \pi$$

can be computed by an effective, constructive algorithm. This has been a well documented mathematical avocation for years. There is also a constructive algorithm for converting this text to postscript, pdf, or html.

Even some of the partial functions we have seen before are constructive and effective. Consider the prime number recognizer:

$$p(x) = \begin{cases} x \text{ if } x \text{ is a prime integer} \\ \\ \text{diverge otherwise} \end{cases}$$

We know exactly how to design a program that performs this task. An inefficient method might involve checking to see if any integer less than $\sqrt{x}$ divides x evenly. If so, then the routine halts and presents x as the output, otherwise it enters an infinite loop.

But consider the following Boolean function.

$$f(x, y) = \text{if lightning strikes at latitude } x \text{ and longitude } y$$

Is this computable? We feel that it is not since one must wait forever on the spot and observe in order to determine the answer. We cannot think of any effective and constructive way to compute this.

Other popular functions that many wish were constructive are:

$w(n) = \text{the } n^{th} \text{ number from now that will come up on a roulette wheel}$
$h(n) = \text{the horse that will win tomorrow's } n^{th} \text{ race}$

We can statistically try to predict the first, but have no effective way of computing it. The latter is totally out of the question. If these functions were effective and constructive then computer programmers would be millionaires!

With our definition of human computation in hand we shall state the first of the two beliefs.

> **Church's Thesis:** *Every finitely specified, constructive computing procedure can be carried out by a Turing machine.*

By the way, Church did not state the thesis in terms of Turing machines. He stated it in terms of the lambda calculus.

Note that anyone who subscribes to Church's thesis no longer needs to design Turing machines! This is wonderful news. It means that we need not write down lengthy sequences of machine instructions in order to show that something is computable if we can state the algorithm in a more intuitive (but constructive) manner. Part of the justification for this is that *each and every* finitely specified, constructive algorithm anyone has ever thought up has turned out to be Turing machine computable. We shall appeal to Church's thesis often in the sequel so that we may omit coding actual machine descriptions.

Our second belief is credited to Turing and deals with systems of computation rather than individual functions.

> **Turing's Thesis:** *Any formal system in which computation is defined as a sequence of constructive steps is no more powerful than the Turing machine model of computation.*

In the section dealing with machine enhancement we saw some material which could be thought of as evidence for Turing's thesis. And history has not contradicted this thesis. Every formal system of the type specified above which anyone has ever invented has been shown to be no more than equivalent to Turing machines. Some of these include Church's *lambda calculus,* Post's *Post machines,* Markov's *processes,* and Herbrand-Gödel-Kleene *general recursive functions.*

So, it seems that we have achieved our goal and defined computation. Now it is time to examine it.

NOTES

Turing machines were introduced by A. M. Turing in his classic paper:

A. M. TURING, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings, London Mathematical Society* 2:42 (1936-1937), 230-265. Errata appear in 2:43 (1937), 544-546.

Another machine model for computation was discovered independently by:

E. L. POST, "Finite combinatory processes. Formulation I," *Journal of Symbolic Logic* 1 (1936), 103-105.

Still additional computational models can be found in:

N. CHOMSKY, "Three models for the description of language," *IRE Transactions on Information Theory* 2:3 (1956) 113-124.

A. CHURCH, "The Calculi of Lambda-Conversion," *Annals of Mathematics Studies* 6 (1941) Princeton University Press, Princeton, New Jersey.

S. C. KLEENE, "General recursive functions of natural numbers," *Mathematische Annalen* 112:5 (1936) 727-742.

A. A. MARKOV, "Theory of Algorithms," *Trudy Mathematicheskogo Instituta imeni V. A. Steklova* 42 (1954).

E. L. POST, "Formal reductions of the general combinatorial decision problem," *American Journal of Mathematics* 65 (1943) 197-215.

More information on enhanced Turing machines appears in many papers found in the literature. Several titles are:

P. C. FISCHER, A. R. MEYER, and A. L. ROSENBERG, "Real-time simulation of multihead tape units," *Journal of The Association for Computing Machinery* 19:4 (1972) 590-607.

J. HARTMANIS and R. E. STEARNS, "On the computational complexity of algorithms," *Transactions of the American Mathematical Society* 117 (1965) 285-306.

H. WANG, "A variant to Turing's theory of computing machines," *Journal of the Association for Computing Machinery* 4:1 (1957) 63-92.

Church's Thesis was presented in:

A. CHURCH, "An unsolvable problem of elementary number theory," *American Journal of Mathematics* 58 (1936) 345-363.

Other textbooks which contain material on Turing machines include:

M. DAVIS, *Computability and Unsolvability*. McGraw-Hill, New York, 1958.

J. E. HOPCROFT and J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

H. R. LEWIS and C. H. PAPADIMITRIOU, Elements of the Theory of Computation. Prentice-Hall, Englewood Cliffs, N. J., 1981.

M. L. MINSKY, *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, N. J., 1967.

The papers by Church, Kleene, Post, and Turing cited above have been reprinted in the collection:

M. DAVIS, ed., *The Undecidable.* Raven Press, Hewlett, N.Y. 1965.

PROBLEMS

The NICE Programming Language

1. Define a model of computation that does not depend on computers or programming.

2. We used floating point numbers instead of real numbers in our programming language. Why?

3. Add the data types character and string to the *NICE* language. Describe the operations that are needed to manipulate them.

4. Examine the following program:

```
program superexpo(x, y)
var m, n, w, x, y, z: integer;
begin
  w = 1;
  for m = 1 to y do
      begin
        z = 1;
        for n = 1 to w do z = z*x;
        w = z
      end;
  halt(z)
end
```

   What are the values of the function it computes when y equals 1, 2, and 3? Describe this function in general.

5. How many multiplications are performed by the programs named expo and superexpo? (Express your answer in terms of x and y.)

6. We have seen that exponentiation can be programmed with the use of one for-loop and that superexponenentiation can be done with two for-loops. What can be computed with three nested for-loops? How about four?

7. Suppose you are given a program named big(x) and you modify it by replacing all halt(z) statements with the following statement pair.

```
z = z + 1
halt(z)
```

   What does the new program compute? Would you believe anyone who told you that they have written a program that computes numbers larger than those computed by any other program?

8. Write a program which computes the function

$$\text{bar}(x) = \begin{cases} x \text{ if } x \text{ is odd and positive} \\ \\ \text{undefined otherwise} \end{cases}$$

   Combine this with the program for the function fu(x) from the *NICE* language section to get an identity function program.

9. Suppose you have a program which computes the characteristic function for the predicate (or relation) P(x, y). This program provides a 1 as output when P(x, y) is true and a 0 whenever P(x, y) is false for x and y. Can you modify it so that your new program finds and returns for any y, the least x such that P(x, y) is true if there is one? This function is usually called $\mu x P(x, y)$ and defined:

$$\text{least}(y) = \begin{cases} \text{the least } x \text{ for which } P(x,y) \text{ is true} \\ \\ \text{undefined if there is no such } x \end{cases}$$

10. Why is the "undefined" clause needed in the above definition of $\mu x P(x, y)$?

Turing Machines

1. What does the Turing machine of figure 2 that adds 1 to its input do when given #000 as input? What about the inputs: #bbb, #b011, and #11b10?

2. Examine the following Turing machine:

I1	**0**	**1**	**right**	**same**
	1	**0**	**right**	**same**
	b	**b**	**left**	**next**
	#	**#**	**right**	**same**

I2	**0**	**1**	**halt**	
	1	**0**	**left**	**same**
	#	**#**	**halt**	

   What does it do when presented with the inputs #1011, #1111, and #0000? In general, what does this machine accomplish?

3. Design a Turing machine that subtracts 1 from its input.

4. Design a Turing machine that recognizes inputs that read the same forwards and backwards. (The machine should halt with its output equal to 1 for inputs such as #101101 or #11011, and provide the output 0 for #1101 or #001110.)

5. How many instructions does the machine of the previous exercise execute on inputs which are n symbols in length?

6. Design a Turing machine which receives #xby (x and y are binary integers) as input and computes x + y. (You may use the machines that add and subtract 1 as subroutines.)

7. Write down the instructions for a Turing machine which determines whether its input is zero. What happens when this machine is given a blank tape as input?

8. How many instructions are executed by the Turing machines of the last two problems on inputs of length n?

9. Design a Turing machine that computes the fu(x) function of the *NICE* language section.

11. A ***0-1 valued Turing machine*** is a machine that always provides outputs of 0 or 1. Since it halts for all inputs, it computes what is known as a ***total function.*** Assume that $M_i(x, y)$ is a 0-1 valued Turing machine and design a machine which receives y as input and halts if and only if there is an x for which the machine $M_i(x, y) = 1$.

A Smaller Programming Language

1. Describe the ways in which division must change after floating point numbers have been replaced by triples of integers which denote their signs, absolute values and decimal points.

2. Assume that you have programs for the following functions:

```
prime(i) = the i-th prime number
expo(x, y) = x raised to the y-th power.
```

   A pair such as (x, y) can be uniquely encoded as:

```
expo(prime(1),x)*expo(prime(2),y)
```

   and decoded by a suitable division routine. In a similar way, any single dimensional array might be encoded. Describe the way in which an array can be encoded as a single integer. Then write a select(a, k) function which provides the k-th element of the array encoded as the integer a, and a replace(a, k, x) program which sets the k-th element of a to the value of x.

3. How might a two dimensional array be encoded as a single integer? What about an n-dimensional array?

4. Arrays in many programming languages have declared bounds on each dimension. Is this restriction needed here? How might the routines of the last two problems be affected if they were to be written for arbitrarily large arrays?

5. Define integer division. Show that division can be replaced by subtraction in much the same way that multiplication was replaced by addition.

6. If we allow the predecessor operation (x = x - 1) to be included in a programming language, then subtraction is not needed. Show this.

7. Suppose procedure calls had been part of our *NICE* programming language. How might they have been eliminated? What about recursive calls?

8. Many modern programming languages include pointers as a data type. Do they allow us to compute any functions that cannot be computed in our simple language? Why?

9. Dynamic storage allocation is provided by some languages that let programs call for new variables and structures during runtime. Is this necessary for increased computational power?

10. The size of a program could be defined as the number of symbols in the program. (In other words: the length of the program.) Consider two programs (one in the extended language and one in the simplified language) that compute the same function.

    a) How might their sizes differ?
    b) Compare their running times.

11. Let's consider programs in our *SMALL* language which have no input (and thus need no titles). The only one line program that computes a defined function is:

```
1: halt(x)
```

    and this program outputs a zero if we assume that all variables are initialized to zero. The two-line program that computes a larger value than any other two-line program is obviously:

```
1: x = x + 1;
2: halt(x)
```

    and this outputs a 1. We could go to three lines and find that an even larger number (in fact: 2) can be computed and output. We shall now add a little computational power by allowing statements of the form:

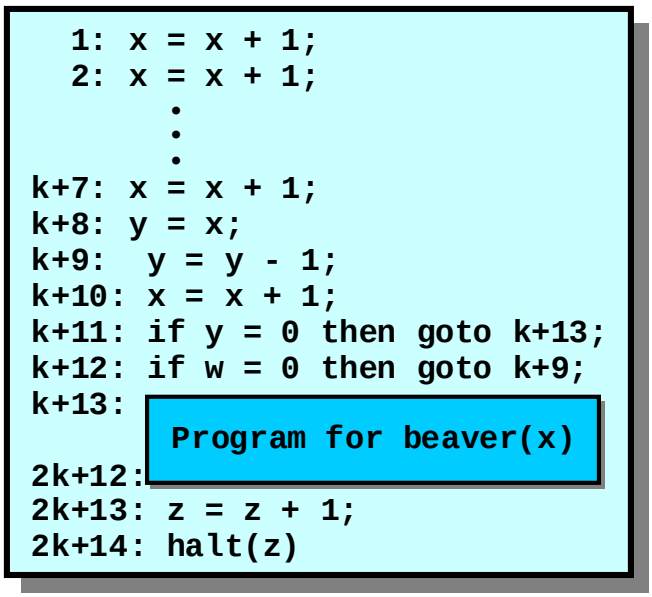
```
k: x = y
```

    and ask some questions. What are the maximum values computed by 4 and 5 line programs of this kind. How about 6? 7? etc.? Do you see a pattern emerging? How about a general formula?

    (Years ago Rado at Bell Laboratories thought up this famous problem. He called it the **Busy Beaver Problem** and stated it as:

    "How many 1's can an n-instruction Turing machine print before halting if it begins with a blank tape as input?")

12. Suppose that someone gives you a *SMALL* language program called beaver(x) that computes the Busy Beaver function of the last exercise. You find that it has exactly k+1 lines of code and ends with a halt(z) statement. After removing the title and the *halt*, it can be embedded as lines k+13 through 2k+12 of the following code to make the program:



Now let's ask some questions about this new program.

a) What value does x posses just before line k+13 is executed?
b) What value is output by this program?
c) What is the value (in words) of beaver(x)?
d) What is the value of z (in words) at line 2k+12?
e) How many lines does this program have?

Comment on this!

Equivalence of the Models

1. Design a "blank-squeezing" Turing machine. That is, a machine which converts #xbby to #xby.

2. Translate the program instruction $x_i = x_k$ into Turing machine instructions.

3. If a *SMALL* program that has n variables executes k instructions, how large can the values of these variables be? How much tape must a Turing machine have to simulate the program?

4. Compare the running times of Turing machines and *SMALL* programs. Assume that one instruction of either can be executed in one unit of time.

5. Translate the Turing machine instructions of problem 2 ($x_i = x_k$) into *NICE* program instructions. Comment on mechanical translation procedures.

6. In the translation from Turing machines to programs an array was used to hold the Turing machine tape. How might scalar variables be employed instead? How would reading, writing and head movement take place?

7. Discuss size trade-offs between Turing machines and programs that compute the same function.

Machine Enhancement

1. Turing machines have often been defined so that they can remain on a tape square if desired. Add the command *stay* to the Turing machine moves and show that this new device is equivalent to the ordinary Turing machine model.

2. Post machines are very much like Turing machines. The major difference is that a Post machine may write or move, but not both on the same instruction. For example, the instruction:

0	**left**	**next**
1	**0**	**same**
b	**halt**	

tells the machine to move if it reads a 0 and to write if it reads a 1. Show that Post machines are equivalent to Turing machines.

3. Endmarkers on our Turing machine tapes were quite useful when we wished not to fall off the left end of the tape during a computation. Show that they are a bit of a luxury and that one can do without them.

4. How much more tape does a two symbol (0, 1, and blank) machine use when it is simulating an n symbol machine? Can this extra space be reduced? How?

5. Design a Turing machine that receives a binary number as input and transforms it into encoded decimal form. (Use the encoding of the machine enhancement section.)

6. Describe the process of changing an encoded decimal into the equivalent binary number.

7. Show that Turing machines which use one symbol and a blank are equivalent to ordinary Turing machines.

8. Describe instructions for multi-tape Turing machines. Specify input and output conventions. Prove that these machines are equivalent to one tape Turing machines.

9. Consider Turing machines that operate on two-dimensional surfaces that look something like infinite chessboards. They now require two additional moves (*up* and *down*) in order to take advantage of their new data structure. Prove that these machines are equivalent to standard Turing machines.

10. Turing machines need not have only one head per tape. Define multiheaded Turing machines. Demonstrate their equivalence to Turing machines that have one head.

11. Consider the problem of recognizing strings which consist of n ones followed be n zeros. How fast can this set be recognized with Turing machines that have:

    a) Two tapes with one head per tape.
    b) One tape and one tape head.
    c) One tape and two tape heads.

    Describe your algorithms and comment on the time trade-offs that seem to occur.

12. A wide-band Turing machine is able to scan several symbols at one time. Define this class of machines and show that they are equivalent to standard Turing machines.

13. Can wide-band Turing machines compute faster than standard Turing machines? Discuss this.

U N SOL V A B I L I T Y

One of the rationales behind our study of computability was to find out exactly what we meant by the term. To do this we looked carefully at several systems of computation and briefly examined the things they could compute. From this study we were able to define the classes of computable functions and computable sets. Then we compared computation via Turing machines and program execution. We found that they were equivalent. Then we examined extensions to Turing machines and found that these added no computational power. After a brief discussion of whether or not Turing machines can perform every computational task we can describe, we came close to assuming that Turing machines (and programs) can indeed compute everything.

Hardly anything is further from the truth! It is not too silly though, for until the 1930's most people (including some very clever mathematicians) felt that everything was computable. In fact, they believed that all of the open problems of mathematics would eventually be solved if someone ingenious enough came along and developed a system in which the problems could be expressed and either verified or refuted mechanically. But there are things which are *not computable* and now we shall attempt to discover and examine a few of them.

Thus our next step in uncovering the nature of computation shall consist of finding out what we cannot compute!

The sections are entitled:

Arithmetization

Finding out what cannot be computed seems like a difficult task. In the very least, we probably shall have to make statements such as:

No Turing machine can compute this!

To verify a claim such as that one, we might have to look at some machines to see if any of them can do the computation in question. And, if we are to start examining machines, we need to know exactly which ones we are talking about. Earlier we discussed naming individual machines. In fact, we gave them delightful names like M_1, M_2, and M_3 but neglected to indicate what any of them really did during their computation. Now is the time to remedy this. After this discussion, any time someone mentions a machine such as M_{942} we shall know *exactly* which machine is being mentioned.

Our first task is to make an *official roster* of Turing machines. It will be sort of a *Who's Who*, except that not only the famous ones, but all of them will be listed. This is called an *enumeration* and the process of forming this list has historically been known as *arithmetization*. It consists of:

a) Encoding all of the Turing machines, and
b) Ordering them according to this encoding.

Now we shall reap a benefit from some of the hard work we undertook while studying computability. We know that we need only consider the *standard* Turing machines - that is, those which use a binary alphabet (0, 1, and *b*) on a one-track tape. Since we know that these compute exactly the class of computable functions, every result about this class of functions applies to these machines. In addition, all results and observations concerning these machines will be true for any other characterization of the computable functions. So, by exploring the properties of the one track, one tape, binary alphabet Turing machines, we shall be also looking at things that are true about programs also.

Let us begin our task. If we take an instruction line such as:

0	1	left	same

lose the nice formatting, and just run the parts together, we get the string:

01leftsame

This is still understandable since we were careful to use only certain words in our Turing machine instructions. We shall modify this a little by translating the words according to the following chart.

left	right	halt	same	next
←	→	↑	s	n

This translation converts our previous instruction line to:

01←s

which is a little more succinct, but still understandable. In the same manner, we can transform an entire instruction such as:

0	1	left	same
1	1	right	I35
b	0	halt	

into three strings which we shall separate by dots and concatenate. The above instruction becomes the following string of characters.

01←s•11→I10001•*b*0↓

(Note that we are using binary rather than decimal numbers for instruction labels. Instead of writing I35 we jotted down I10001.) This is not pretty, but it is still understandable, and *the instruction has been encoded!*

Next, we encode entire machines. This merely involves concatenating the instructions and separating them with double dots. The general format for a machine with n instructions looks something like this:

••*I1*••*I2*••*I3*•• … ••*In*••

As an example, let us take the machine:

0	0	*right*	*next*
1	1	*right*	*same*
b	*b*	*right*	*same*

0	0	*right*	*same*
1	1	*right*	*I1*
b	*b*	*halt*	

(which, by the way, accepts all inputs which contain an even binary number) and encode it as:

••00→n•11→s•*bb*→s••00→s•11→l1•*bb*↓••

It is not too troublesome to interpret these encodings since we *know* what an instruction line looks like and have carefully placed dots between all of the lines and instructions. The next step is turn these encodings into numbers. Then we can deal with them in a arithmetical manner. (This is where the term *arithmetization* comes in.) Assigning numbers to symbols is done according to the following chart.

symbol	**0**	**1**	***b***	**→**	**←**	**↓**	**s**	**n**	**l**	**•**
number	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Now each Turing machine can be encoded as a decimal number. Our last machine has become the rather large number:

9,900,379,113,692,236,990,036,911,381,922,599

and the smallest Turing machine, namely:

0	**0 halt**

(which accepts all strings beginning with zero) is number 9,900,599 in our nifty new, numerical encoding.

These encodings for Turing machines will be referred to as *machine descriptions* since that is what they really are. A nice attribute of these descriptions is that we know what they look like. For example, they always begin with two nines followed immediately by a couple of characters from {0, 1, 2}, and so forth. Thus we can easily tell which integers are Turing machine descriptions and which are not. We know immediately that 10,011,458,544 is not a machine and 991,236,902,369,223,699 has to be the rather reactionary machine which always moves to the right.

Now we shall use Church's thesis for the first time. We can easily design an algorithm for deciding whether or not an integer is a Turing machine description. Thus we may claim that there is a Turing machine which decides this. This makes the set of descriptions computable.

Two important facts emerged from our simple exercise in encoding.

- *Every Turing machine has a unique description.*
- *The set of machine descriptions is a computable set.*

(**N.B.** Here are two points to ponder. First, suppose two machines have the same instructions, but with some of the instruction lines in different order. Are they the same machine even though they have different descriptions? Our statement above says that they are not the same. Is this OK? Next, imagine what an arithmetization of *NICE* programs would have looked like!)

We still do not have our *official roster* of Turing machines, but we are almost there. The list we need shall consist of all Turing machine descriptions in numerical order. Composing this is easy, just go through all of the decimal integers (0, 1, 2, ...) and discard every integer that is not a Turing machine description. This straightforward (tedious, but straightforward) process provides us with a list of machines. Now, we merely number them according to where they appear on the list of machine descriptions. In other words:

M_1 = the first machine on the list
M_2 = the second machine on the list

and so forth. This list of machines is called the ***standard enumeration*** of Turing machines. A machine's place on the list (i.e., the subscript) is called its *index*. So, now we know exactly which machine we're talking about then we mention M_{239} or M_{753}. The *sets accepted* by these machines are also numbered in the same manner. We call them W_1, W_2, and so on. More formally:

$$W_i = \{ x \mid M_i(x) \text{ halts} \}$$

We have now defined a standard enumeration or listing of all the Turing machines: M_1, M_2, M_3, ... as well as a standard enumeration of all the computable sets: $W_1, W_2, W_3, \ldots$

Let us now close by mentioning once again two very important facts about the constructiveness of our standard enumeration.

- ***Given the instructions for a Turing machine, we can find this machine in our standard enumeration.***

- ***Given the index (in the enumeration) of a Turing machine, we can produce its instructions.***

This property of being able to switch between indices and machines combined with Church's thesis allows us to convincingly claim that we can locate the indices of the Turing machines which correspond to any algorithms or computing procedures we use in theorems or examples.

Properties of the Enumeration

The ease with which we formed the official rosters of all Turing machines and the sets that they accept belies its significance. Though it may not seem to be very important, it will be a crucial tool as we begin to formulate and explore the properties of the class of computable sets.

A basic question concerns the *cardinality* of the class of computable sets. Cardinality means size. Thus a set containing exactly three objects has cardinality three. Quite simple really. The cardinalities of the finite sets are:

0, 1, 2, 3, and so on.

Things are not so easy for infinite sets since they do not have cardinalities corresponding to the integers. For this reason, mathematicians employ the special symbol $\aleph_0$ (the Hebrew letter aleph with subscript zero - pronounced *aleph naught)* to represent the cardinality of the set of integers. Let us state this as a definition.

> **Definition.** *The set of nonnegative integers has cardinality* $\aleph_0$.

Many sets have this cardinality. One is the set of even integers. In fact, if it is possible to match up the members of two sets in a one-to-one manner with no elements left over then we say that they have the same cardinality.

> **Definition.** *Two sets have the* ***same cardinality*** *if and only if there is a one-to-one correspondence between them.*

A one-to-one correspondence is merely a matching between the members of two sets where each element is matched with exactly one element of the other set. Here is an example of a one-to-one correspondence between the nonnegative integers and the even nonnegative integers:

0	1	2	3	…	k	…
↓	↓	↓	↓		↓	
0	2	4	6	…	2k	…

It is just a mapping from x to 2x. Since the correspondence contains all of the nonnegative integers and all of the even nonnegative integers we state that these sets have exactly the same size or cardinality, namely $\aleph_0$.

(At this point we need some notation. The size or cardinality of the set A will be written |A|. Thus: $|\{a , b\}| = 2$ and |set of integers| = $\aleph_0$.)

Other examples of sets that have cardinality $\aleph_0$ are all of the (positive and negative) integers, the rational numbers, and the prime numbers. We traditionally speak of these sets as being *countable* since they can be put in one-to-one correspondence with the numbers we use in counting, namely the nonnegative integers. But, most important of all to us at this moment is the fact that our standard enumeration gives us exactly $\aleph_0$ Turing machines. This leads to a result involving the cardinality of the class of computable sets.

Theorem 1. *There are exactly $\aleph_0$ computable sets.*

Proof. We must show two things in order to prove this theorem. First, that there are no more than $\aleph_0$ computable sets, and then that there are at least $\aleph_0$ computable sets.

The first part is easy. From our definition of computable, we know that every computable set is accepted by some Turing machine. Thus

$$|\text{Computable Sets}| \leq |\text{Turing machines}|.$$

Since we can place the Turing machines in one-to-one correspondence with the integers, (this is due to our standard enumeration: M_1, M_2, ...) we know that there are exactly $\aleph_0$ of them. This means that the cardinality of the class of computable sets is no greater than $\aleph_0$. That is:

$$|\text{Computable Sets}| \leq |\text{Turing machines}| = \aleph_0$$

Now we must show that there are, in fact, at least $\aleph_0$ computable sets. (After all - suppose that even though there are an infinite number of Turing machines, many act the same and so the entire collection of machines only accepts a finite number of different sets!) Consider the sequence:

$$\{0\}, \{1\}, \{10\}, \{11\}, \{100\}, \ldots$$

of singleton sets of binary integers. There are exactly $\aleph_0$ of these and they are all computable since each one can be accepted by some Turing machine. Thus:

$$\aleph_0 = |\text{Singleton Sets}| \leq |\text{Computable Sets}| \leq |\text{Turing machines}| = \aleph_0$$

and our theorem is proven.

We now know exactly how many computable sets exist. The next obvious question is to inquire as to whether they exhaust the class of sets of integers.

Theorem 2. *There are more than $\aleph_0$ sets of integers.*

Proof. There are at least $\aleph_0$ sets of integers since there are exactly $\aleph_0$ computable sets. We shall assume that there are exactly that many sets and derive a contradiction. That will prove that our assumption is incorrect and thus there must be more sets.

Our strategy uses a technique named *diagonalization* that was developed by Cantor in the mid nineteenth century. We shall list all of the sets of integers and then define a set that cannot be on the list.

If there are exactly $\aleph_0$ sets of integers then they can be placed in one-to-one correspondence with the integers. Thus there is an enumeration of all the sets of integers. And, if we refer to this infinite list of sets as:

$$S_1, S_2, S_3, \ldots$$

we may be assured that every set of integers appears as some S_i on our list. (Note that we did not explain how we derived the above roster of sets - we just claimed that it exists in some mathematical wonderland!)

Since we assumed that the list of all sets exists, we are allowed to use it in any logically responsible manner. We shall now define a set in terms of this list of sets. The set we shall define will depend directly on the members of the list. We shall call this set D and define membership in it by stating that for each integer i:

$$i \in D \text{ if and only if } i \notin S_i$$

(Again, note that we did not even consider how one goes about computing membership for D, just which integers are members. This is one of the differences between theory and practice.)

Now, what do we know about the set D? First of all, the set D is indeed a set of integers. So, it must appear somewhere on our list ($S_1, S_2, S_3, \ldots$) of sets of integers since the list contains all sets of integers. If the set D appears as the d^{th} set on the list, then $D = S_d$. Now we ask the simple question: *'Does the set D contain the integer d?'*

Watch this argument very closely. If d is a member of D then d must not be a member of S_d since that was how we defined membership in D

above. But we know that D and S_d are precisely the same set! This means that d cannot be a member of D since it is not a member of S_d. Thus d must not be a member of D.

But wait a moment! If d is not a member of D then d has to be a member of S_d, because that is how we defined the set D. Thus d is not a member of D if and only if d is a member of D ! We seem to have a small problem here.

Let's have another look at what just happened. Here is a little chart, which illustrates the above argument.

Due to:	*We know:*
Definition of D	(1) $d \in D$ if and only if $d \notin S_d$
$S_d = D$	(2) $d \notin S_d$ if and only if $d \notin D$
Statements (1) and (2)	(3) $d \notin D$ if and only if $d \notin D$

We shall often use the symbol $\Leftrightarrow$ to mean *if and only if*, and now use it to state the above derivation as:

$$d \in D \Leftrightarrow d \notin S_d \Leftrightarrow d \notin D .$$

As we mentioned earlier, something must be very wrong! And it has to be one of our assumptions since everything else was logically responsible and thus correct. Going back through the proof we find that the only assumption we made was our claim that

there are exactly $\aleph_0$ *sets of integers.*

Therefore there must be more than $\aleph_0$ sets of integers.

This result brings up an interesting topic in mathematics that was very controversial during the mid nineteenth century. There seem to be several kinds of infinity. We have just shown that there is an infinite class (sets of integers) which has cardinality greater than another infinite class (the integers). This larger cardinality is denoted $2^{\aleph_0}$ and is also the cardinality of the class of real numbers. But wait a moment, don't we use real numbers in scientific computation? Well, actually, no. We use *floating-point numbers*, not reals. And now we know one of the reasons we use floating point numbers and not real numbers in programs - there are just too many reals!

Let us now return to the computable sets. Since we have shown that are more sets of integers than there are Turing machines, we may immediately state several corollaries to our last theorem.

Theorem 3. *There are sets that are not computable.*

Corollary. *There are things that cannot be computed by Turing machines or computer programs.*

Here are two more results which follow from the fact that we can enumerate the Turing machines in a manner such that each machine has a unique place in our standard enumeration.

Theorem 4. *For every Turing machine there is an equivalent machine that appears later in the enumeration.*

Theorem 5. *Every computable set appears $\aleph_0$ times in the standard enumeration.*

The proofs of these two theorems have been left as exercises, but they are not difficult to prove. One merely observes that adding additional instructions (which are never executed) to a machine changes it's description but does not alter its behavior.

Universal Machines and Simulation

We know now that there are sets which are not computable. But we do not know of any particular ones except the rather peculiar diagonal set defined in the theorem that declared that there are some sets which Turing machines cannot accept. We must remedy this if we are to make anyone believe that noncomputability is a practical issue.

To find some uncomputable sets, we would like to actually build them, not just know they exist. To accomplish this we need formal construction tools and methods. That is what we shall immediately begin working upon. At the same time, we shall expand our arsenal of theoretical tools and discover more about the nature of computation.

The first thing we must learn how to do formally is very simple and called substitution. Suppose we have the Turing machine M_k which computes:

$$M_k(x, a, b) = ax + b$$

and we wish to change it into a machine which always computes 6x + 57. What we need to do is substitute the integer 6 for the variable a and 57 for b. Or, suppose we changed our mind and wanted to compute 26.5x - 3 instead. That is straightforward too. But it would be nice if there was a general procedure for modifying our ax + b machine to produce another where we have substituted for a and b with particular values we select. And, after we choose values for a and b, we would like to have the new machine built automatically. Furthermore, we want to know how it is done. In other words, the process must be *effective.*

Another way to formulate this is to ask:

> *'Given a machine $M_k(x, a, b)$ which computes ax + b and specific values u and v to be substituted for a and b, is there a computable substitution function s(k, u, v) which provides an index for a new machine in the standard enumeration that always computes ux + v?'*

The two examples of the use of this substitution function s(k, a, b) from our previous ax + b example are:

$$M_{s(k,6,57)}(x) = M_k(x, 6, 57) = 6x + 57$$
$$M_{s(k,26.5,-3)}(x) = M_k(x, 26.5, -3) = 26.5x - e$$

and of course, there are many more.

What the function s(k, a, b) does is to provide a new machine which performs M_k's computation for the specified values of a and b. Also, note that a and b are now fixed and do not appear as input any more in the new machine $M_{s(k,a,b)}$.

Actually doing this with a program is very easy. We just remove the variables a and b from the header and insert the pair of assignments: a=m and b=n at the beginning of the program code.

Theorem 1 precisely states the general case for substitution. (This is also an example of how a very simple concept is sometimes quite difficult to express precisely and formally!)

> **Theorem 1** *(Substitution). There is a computable function s such that for all i, $x_1, \ldots, x_n, y_1, \ldots, y_m$:*
>
> $$M_{s(i,y_1,\ldots,y_m)}(x_1,\ldots,x_n) = M_i(x_1,\ldots,x_n,y_1,\ldots,y_m)$$
>
> **Proof.** We must design an Turing computable algorithm for the substitution function $s(i, y_1, \ldots, y_m)$ which, when given specific values for i and $y_1, \ldots, y_m$ generates the index (or description) of a machine which computes:
>
> $$M_i(x_1, \ldots, x_n, y_1, \ldots, y_m)$$
>
> The machine with index $s(i, y_1, \ldots, y_m)$ operates exactly as described in the following algorithm.
>
> **Move to the right end of the input**
> **(to the right of x_n)**
>
> **Write values of $y_1, \ldots, y_m$ on the input tape.**
>
> **Return to the beginning of the input tape**
> **(to the left of x_1)**
>
> **Commence processing as M_i**
>
> If we are provided with the instructions for M_i, we know how to write down the Turing machine instructions which accomplish all of the above steps. Thus we know *exactly* what $M_{s(i,y_1,\ldots,y_m)}$ looks like. We shall now appeal to Church's Thesis and claim that there is a Turing machine which will can add the instructions for the above preprocessor to M_i.

In fact, the machine can not only generate the instructions for a machine which computes the above algorithm, but also find it in our standard enumeration. (This is clear if we recall our arithmetization of Turing machines and how we can effectively go between machine descriptions and indices in the standard enumeration.)

The machine which locates the index of $M_{s(i,y_1,\ldots,y_m)}$ is exactly the machine which computes $s(i, y_1, \ldots, y_m)$.

This result is well-known in recursive function theory as the *s-m-n theorem* because the function s usually appears as s_n^m. It is an extremely useful result and we shall use it almost every time we build Turing machines from others. It is also very important since there are results in recursion theory which state that all nice enumerations of the computable functions must posses an s-m-n or substitution theorem.

So far we have spoken of Turing machines as special purpose devices. Thus they may have seemed more like silicon chips than general-purpose computers. Our belief that Turing machines can compute everything that is computable indicates that they can do what general-purpose computers do. That is, if programmed correctly, they can be operating systems, compilers, servers, or whatever we wish. As a step in this direction we shall examine their ability to simulate each other - just like real computers! So, we shall design what is always called a *universal Turing machine.* This will also come in handy in the future to prove several strong results about computation.

Let's begin by calling the universal machine M_u. It receives the integers i and x as inputs and carries out the computation of M_i on the input x. It is actually what we in computer science have always called an interpreter. The formal specification for this machine is:

Definition. *M_u is a **universal Turing machine** if and only if for all integers i and x: $M_u(i, x) = M_i(x)$.*

In our design, we shall give this machine two tapes, one above the machine and one below as in the figure 1 below. The top tape shall be used for performing or interpreting the computation of $M_i(x)$. The bottom tape is used to store the machine description (or the program) for M_i and thus provides all of the information we need to know exactly how the simulation should take place.

The universal Turing machine, $M_u(i, x)$ receives i and x as input on its upper tape and begins its computation reading both tape endmarkers in a configuration which looks like that of figure 1.

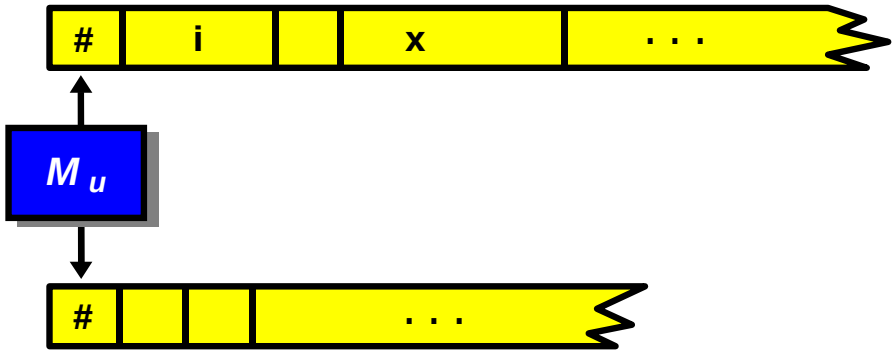
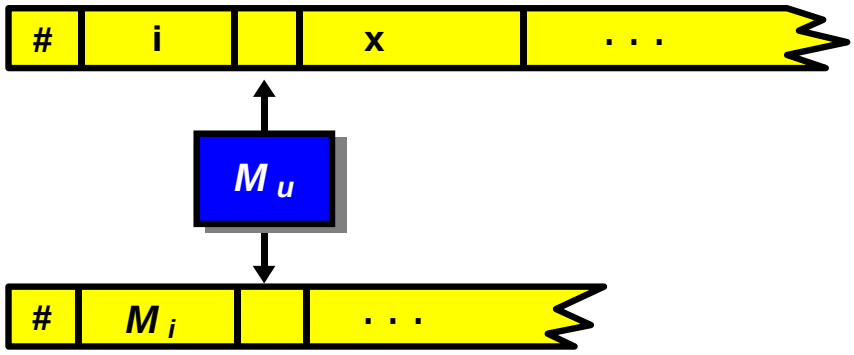


Figure 1 - Universal Turing Machine Initial Configuration

Immediately, the universal machine proceeds to execute the following steps.

a) Write down the description of M_i on the lower tape.
b) Copy the input x at the left of the input (upper) tape.
c) Compute $M_i(x)$ on the upper tape.

After step (a) it has written a description of M_i on its lower tape. This results in a configuration such as the following.



Before continuing, let us examine the description of M_i's instructions that must be written on the lower tape. As an example, suppose instructions I73 and I74 of the machine M_i are:

I73				
	0	1	right	same
	1	b	left	next
	b	0	right	I26

I74				
	0	1	left	next
	1	b	halt	
	b	b	right	I46

The universal machine's lower tape could hold a straightforward transcription of the instruction tables. A fragment of a three-track description tape that holds the above instructions might look like the following.

	*	I	7	3	*	0	1	→	s			*	I	7	4	*	0	1	
…	*					1		←	n			*					1		…
	*						0	→	I	2	6	*							

Exactly how a Turing machine could have done this is left as an exercise. It is not too difficult to design a procedure for doing this though. Recall the arithmetization process where machines were mapped into decimal numbers? First, M_u could jot down the decimal numbers in order, count how many of these are actually Turing machine encodings, and save the decimal number that is the encoding of the i^{th} one.

Translation from the decimal number description to a tape like that shown above is almost automatic. The portion of the decimal number that contains the above two instructions is just:

. . . 9901369124792038110109900479125922381011 1099 . . .

and this quickly translates into:

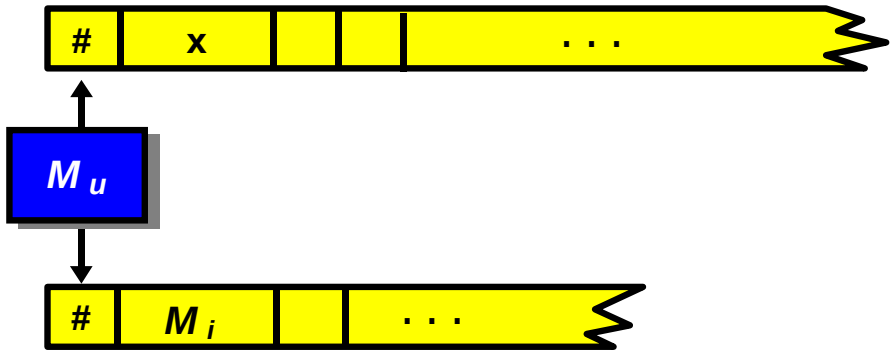
. . . **01→s*1*b*←n*b0→I26**00←n*1*b*↓* . . .

which in turn is easily written on the three-track tape.

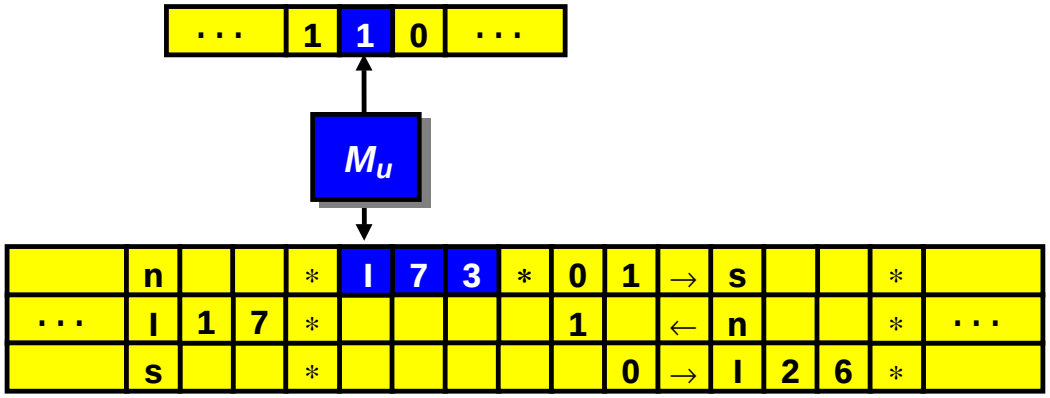
(**N.B.** We need to pause here and note an important assumption that was made when we copied the instructions of M_i on the three-track description tape. We assumed that M_i was a one-tape, one-atrack machine that used only 0, 1, and blank as its symbols. This is allowable because we showed earlier that this class of machines computes everything that is computable and thus is equivalent to any other class of Turing machines, or even programs.

So, if there is a universal machine which simulates these machines, we are simulating the entire class of computable functions.)

At this point the universal machine overprints the integer i and copys x at the left end of the top (or simulation) tape. If we recall that the selection machine which we designed earlier to carry out the operation M(i, x) = x, performing this step is not difficult to imagine. After resetting the tape heads to the left, we arrive at a configuration like that which follows.



At this point the universal machine is ready to begin its simulation of $M_i(x)$. In the sequel we shall look at a simulation example of one step during a computation. Consider the following configuration of our universal machine.



The squares that the universal machine is examining are shaded in blue. This indicates that in the simulation, M_i is reading a 1 and about to execute instruction I73.

In the simulation, the universal machine merely applies the instructions written on the lower tape to the data on the top tape. More precisely, a simulation step consists of:

a) *Reading the symbol on the top tape.*
b) *Writing the appropriate symbol on the top tape.*
c) *Moving the input head (on the top tape).*
d) *Finding the next instruction on the description (bottom) tape.*

Instead of supplying all of the minute details of how this simulation progresses, we shall explain the process with an example of one step and resort once again to our firm belief in Church's thesis to assert that there is indeed a universal Turing machine M_u. We should be able easily to write down the instructions of M_u, or at least write a program which emulates it.

In figure 2, our universal machine locates the beginning or read portion of the instruction to be executed and finds the line of the instruction (on the description tape) that corresponds to reading a 1 on the top tape. Moving one square over on the description tape, it discovers that it should write a blank, and so it does.

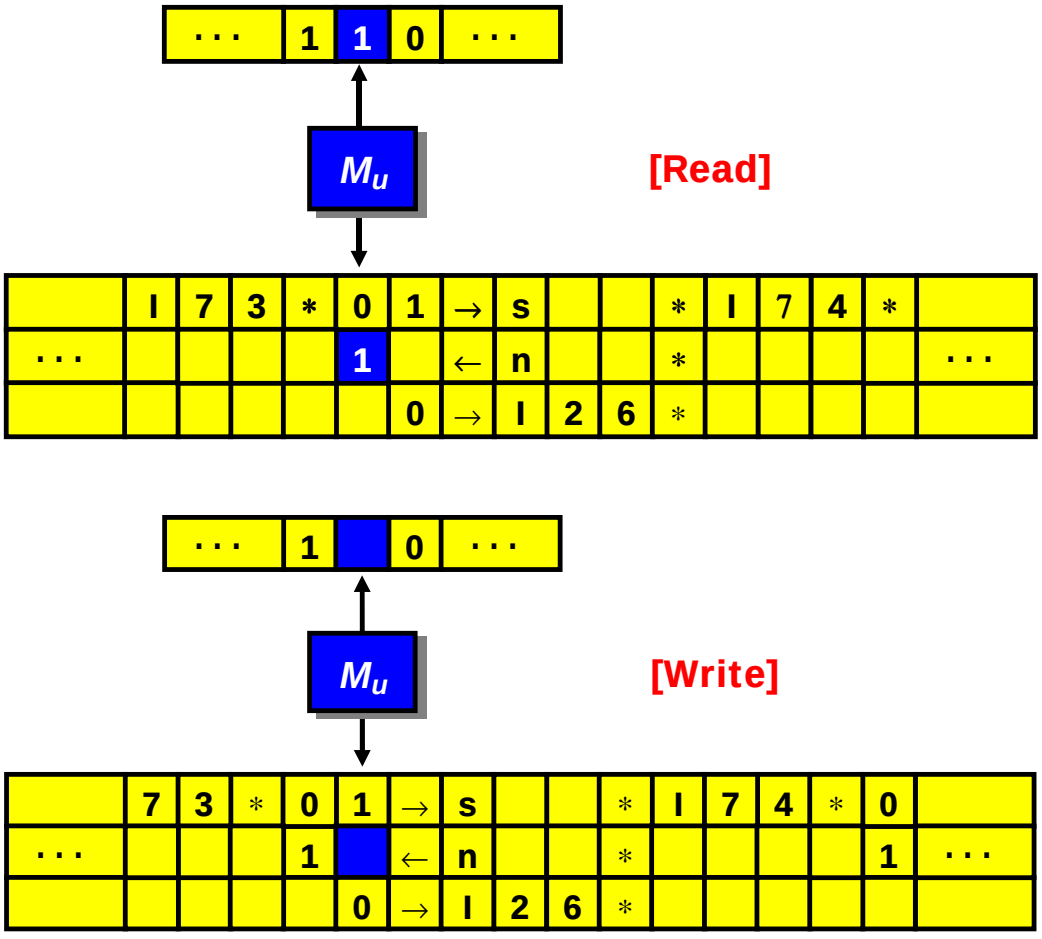


Figure 2 - Universal Turing Machine: Reading and Writing

Next M_i should move and find the next instruction to execute. In figure 3, this is done. The universal machine now moves its simulation tape head one square to the right, finds that it should move the tape head of M_i to the left and does so. Another square to the right is the command to execute the next instruction (I74), so the universal machine moves over to that instruction and prepares to execute it.

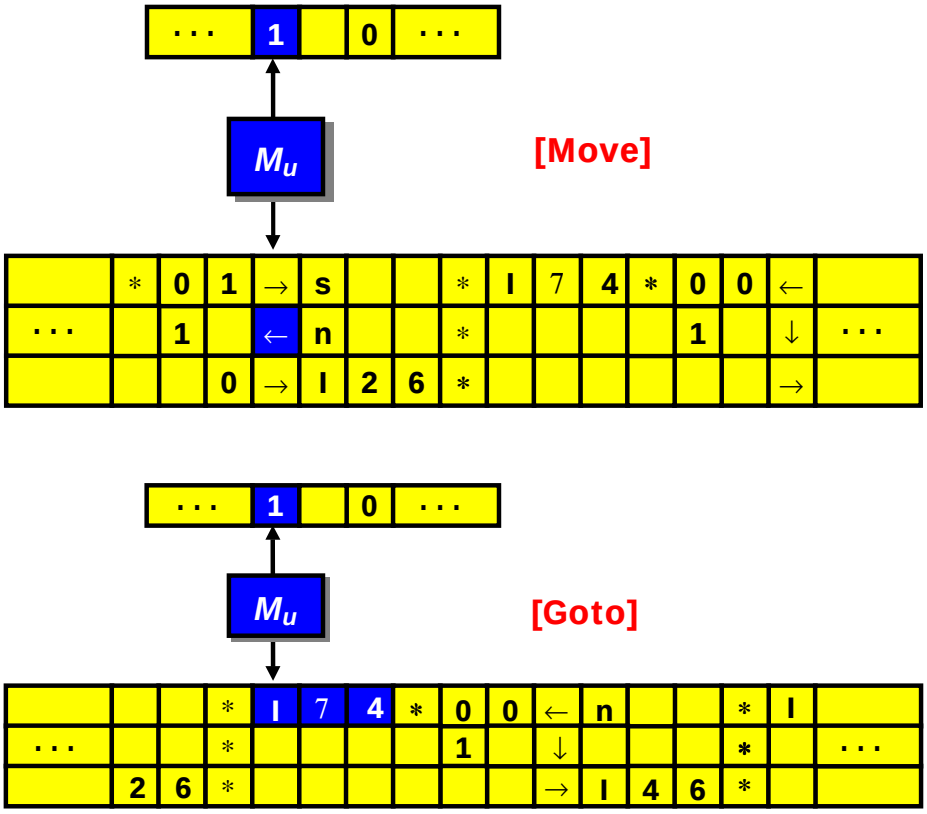


Figure 3 - Universal Turing Machine: Moving and Goto

Several of the details concerning the actual operation of the universal machine have been omitted. Some of these will emerge later as exercises. So, we shall assume that all of the necessary work has been accomplished and state the famous *Universal Turing Machine Theorem* without formal proof.

> **Theorem 2.** *There is a universal Turing machine.*

We do however need another note on the above theorem. The universal machine we described above is a two-tape machine with a three-track tape and as such is not included in our standard enumeration. But if we recall the results about multi-tape and multi-track machines being equivalent to ordinary one-tape machines, we know that an equivalent machine exists in our standard enumeration. Thus, with heroic effort we could have built a binary alphabet, one-tape, one-track universal Turing machine. This provides an important corollary to the universal Turing machine theorem.

> **Corollary.** *There is a universal Turing machine in the standard enumeration.*

At various times previously we mentioned that the universal Turing machine and s-m-n theorems were very important and useful. Here at last is an example of how we can use them to prove a very basic closure property of the computable sets.

Theorem 3. *The class of computable sets is closed under intersection.*

Proof. Given two arbitrary Turing machines M_a and M_b, we must show that there is another machine M_k that accepts exactly what *both* of the previous machines accept. That is for all x:

$$M_k(x) \text{ halts if and only if both } M_a(x) \text{ and } M_b(x) \text{ halt.}$$

or if we recall that the set which M_a accepts is named W_a, another way to state this is that:

$$W_k = W_a \cap W_b.$$

The algorithm for this is quite simple. Just check to see if $M_a(x)$ and $M_b(x)$ both halt. This can be done with the universal Turing machine. An algorithm for this is:

Intersect(x, a, b)

run M_u(a, x), and diverge if M_u diverges
***if* M_u(a, x) halts *then* run M_u(b, x)**
***if* M_u(b, x) halts *then* halt (accept)**

Appealing once more to Church's thesis, we claim that there is a Turing machine which carries out the above algorithm. Thus this machine exists and has a place in our standard enumeration. We shall call this machine M_{int}. And, in fact, for all x:

$$M_{int}(x, a, b) \text{ halts if and only if both } M_a(x) \text{ and } M_b(x) \text{ halt.}$$

At this point we have a machine with three inputs (M_{int}) which halts on the proper x's. But we need a machine with only one input which accepts the correct set. If we recall that the s-m-n theorem states that there is a function s(int, a, b) such that for any integers a and b:

$$M_{s(int, a, b)}(x) = M_{int}(x, a, b).$$

we merely need to look at the output of s(int, a ,b) and then set the index: k = s(int, a, b).

Thus we have designed a Turing machine M_k which satisfies our requirements and accepts the intersection of the two computable sets W_a and W_b.

Solvability and the Halting Problem.

Our development period is over. Now it is time for some action. We have the tools and materials and we need to get to work and discover some things that are not computable. We know they are there and now it is time to find and examine a few.

Our task in this section is to find some noncomputable problems. However we must first discuss what exactly *problems* are. Many of our computational tasks involve questions or decisions. We shall call these problems. For example, some problems involving numbers are:

- *Is this integer a prime?*
- *Does this equation have a root between 0 and 1?*
- *Is this integer a perfect square?*
- *Does this series converge?*
- *Is this sequence of numbers sorted?*

As computer scientists, we are very aware that not all problems involve numbers. Many of the problems that we wish to solve deal with the programs we write. Often we would like to know the answers to questions concerning our methods, or our programs. Some of these problems or questions are:

- *Is this program correct?*
- *How long will this program run?*
- *Does this program contain an infinite loop?*
- *Is this program more efficient than that one?*

A brief side trip to set forth more definitions and concepts is in order. We must describe some other things closely related to problems or questions. In fact, often when we describe problems we state them in terms of *relations* or *predicates.* For example, the predicate *Prime(x)* that indicates prime numbers could be defined:

Prime(x) is true if and only if x is a prime number.

and this predicate could be used to define the set of primes:

$$PRIMES = \{ x \mid Prime(x) \}.$$

Another way to link the set of primes with the predicate for being a prime is to state:

$$x \in PRIMES \text{ if and only if } Prime(x)$$

(**N.B.** Two comments on notation are necessary. We shall use *iff* to mean *if and only if* and will often just mention a predicate as we did above rather than stating that it is true.)

We now have several different terms for problems or questions. And we know that they are closely related. Sets, predicates, and problems can be used to ask the same question. Here are three equivalent questions:

- *Is $x \in PRIMES$?*
- *Is Prime(x) true?*
- *Is x a prime number?*

When we can completely determine the answer to a problem, the value of a predicate, or membership in a set *for all instances* of the problem, predicate, or things that may be in the set; we say that the problem, predicate, or set is *decidable or solvable.* In computational terms this means that there is a Turing machine which can *in every case* determine the answer to the appropriate question. The formal definition of solvability for problems follows.

> **Definition.** *A problem P is **solvable** if and only if there is a Turing machine M_i such that for all x:*
>
> $$M_i(x) = \begin{cases} 1 \text{ if } P(x) \text{ is true} \\ 0 \text{ if } P(x) \text{ is false} \end{cases}$$

If we can always solve a problem by carrying out a computation it is a solvable problem. Many examples of solvable problems are quite familiar to us. In fact, most of the problems we attempt to solve by executing computer programs are solvable. Of course, this is good because it guarantees that if our programs are correct, then they will provide us with solutions! We can determine whether numbers are prime, find shortest paths in graphs, and many other things because these are solvable problems. There are lots and lots of them. But there must be some problems that are not solvable because we proved that there are things which Turing machines (or programs) cannot do. Let us begin by formulating and examining a historically famous one.

Suppose we took the Turing machine M_1 and ran it with its own index as input. That is, we examined the computation of $M_1(1)$. What happens? Well, in this

case we know the answer because we remember that M_1 was merely the machine:

and we know that it only halts when it receives an input that begins with a zero. This is fine. But, how about $M_2(2)$? We could look at that also. This is easy; in fact, there is almost nothing to it. Then we could go on to $M_3(3)$. And so forth. In general, let us take some arbitrary integer i and ask about the behavior of $M_i(i)$. And, let's not ask for much, we could put forth a very simple question: *does it halt?*

Let us ponder this a while. Could we write a program or design a Turing machine that receives i as input and determines whether or not $M_i(i)$ halts? We might design a machine like the universal Turing machine that first produced the description of M_i and then simulated its operation on the input i. This however, does not accomplish the task we set forth above. The reason is because though we would always know if it halted, if it went into an infinite loop we might just sit there and wait forever without knowing what was happening in the computation.

Here is a theorem about this that is very reminiscent of the result where we showed that there are more sets than computable sets.

> **Theorem 1.** *Whether or not a Turing machine halts when given its own index as input is unsolvable.*
>
> **Proof.** We begin by assuming that we *can* decide whether or not a Turing machine halts when given its own index as input. We assume that the problem is solvable. This means that there is a Turing machine that can solve this problem. Let's call this machine M_k and note that for all inputs i:
>
> $$M_k(x) = \begin{cases} 1 \text{ if } M_x(x) \text{ halts} \\ \\ 0 \text{ if } M_x(x) \text{ diverges} \end{cases}$$
>
> (This assertion came straight from the definition of solvability.)
>
> Since the machine M_k exists, we can use it in the definition of another computing procedure. Consider the following machine.

$$M(x) = \begin{cases} \text{halt if } M_k(x) = 0 \\ \\ \text{diverge if } M_k(x) = 1 \end{cases}$$

This is not too difficult to construct from M_k and our universal Turing machine M_u. We just run $M_k(x)$ until it provides an output and then either halt or enter an infinite loop.

We shall apply Church's thesis once more. Since we have developed an algorithm for the above machine M, we may state that is indeed a Turing machine and as such has an index in our standard enumeration. Let the integer d be its index. Now we inquire about the computation of $M_d(d)$. This inquiry provides the following sequence of conclusions. (Recall that iff stands for if and only if.)

$M_d(d)$ halts	iff $M(d)$ halts	(since $M_d = M$)
	iff $M_k(d) = 0$	(see definition of M)
	iff $M_d(d)$ diverges	(see definition of M_k)

Each step in the above deduction follows from definitions stated previously. Thus they all must be true. But there is a slight problem since a contradiction was proven! Thus something must be wrong and the only thing that could be incorrect must be some assumption we made. We only made one, namely our original assumption that the problem was solvable. This means that whether a Turing machine halts on its own index is unsolvable and we have proven the theorem.

Now we have seen an unsolvable problem. Maybe it is not too exciting, but it is unsolvable nevertheless. If we turn it into a set we shall then have a set in which membership is undecidable. This set is named K and is well-known and greatly cherished by recursion theorists. It is:

$$K = \{ i \mid M_i(i) \text{ halts} \}$$

K was one of the first sets to be proven undecidable and thus of great historical interest. It will also prove quite useful in later proofs. Another way to state our last theorem is:

Corollary. *Membership in K is unsolvable.*

Let us quickly follow up on this unsolvability result and prove a more general one. This is possibly the most famous unsolvable problem that exists. It is called the *halting problem* or *membership problem*.

Theorem 2 *(Halting Problem). For arbitrary integers i and x, whether or not $M_i(x)$ halts is unsolvable.*

Proof. This follows directly from the previous theorem. Suppose we could solve halting for $M_i(x)$ on any values of i and x. All we have to do is plug in the value i for x and we are now looking at whether $M_i(i)$ halts. We know from the last theorem that this is not solvable. So the general halting problem (does $M_i(x)$ halt?) must be unsolvable also, since if it were solvable we could solve the restricted version of the halting problem, namely membership in the set K.

This is interesting from the point of view of a computer scientist. It means that no program can ever predict the halting of *all other programs.* Thus we shall never be able to design routines which unfailingly check for infinite loops and warn the programmer, nor can we add routines to operating systems or compilers which always detect looping. This is why one never sees worthwhile infinite loop checkers in the software market.

Let's try another problem. It seems that we cannot tell if a machine will halt on arbitrary inputs. Maybe the strange inputs (such as the machine's own index) are causing the problem. This might be especially true if we are looking at weird machines that halt when others do not and so forth! It might be easier to ask if a machine always halts. After all, this is a quality we desire in our computer programs. Unfortunately that is unsolvable too.

Theorem 3. *Whether or not an arbitrary Turing machine halts for all inputs is an unsolvable problem.*

Proof. Our strategy for this proof will be to tie this problem to a problem that we know is unsolvable. Thus it is much like the last proof. We shall show that halting on one's index is solvable if and only if halting for all inputs is solvable. Then since whether a machine halts on its own index is unsolvable, the problem of whether a machine halts for all inputs must be unsolvable also.

In order to explore this, let's take an arbitrary machine M_i and construct another Turing machine M_{all} such that:

$$M_{all} \text{ halts for all inputs iff } M_i(i) \text{ halts}$$

At this point let us not worry about how we build M_{all}, this will come later.

We now claim that if we can decide whether or not a machine halts for all inputs, we can solve the problem of whether a machine halts on its own index. Here is how we do it. To decide if $M_i(i)$ halts, just ask whether M_{all}

halts on all inputs. But, since we have shown that we cannot decide if a machine halts upon its own index this means that if we are able to construct M_{all}, then we have solved membership in K and proven a contradiction. Thus the problem of detecting halting on all inputs must be unsolvable also.

Let us get to work. A machine like the above M_{all} must be built from M_i. We shall use all of our tools in this construction. As a start, consider:

$$M(x, i) = M_u(i, i) = M_i(i)$$

Note that M does not pay attention to its input x. It just turns the universal machine M_u loose on the input pair (i, i), which is the same as running M_i on its own index. So, no matter what x equals, M just computes $M_i(i)$. Yet another appeal to Church's thesis assures us that M is indeed a Turing machine and exists in the standard enumeration. Let us say that M is machine M_m. Thus for all i and x:

$$M_m(x, i) = M(x, i) = M_u(i, i) = M_i(i).$$

Now we shall call upon the s-m-n theorem. It says that there is a function s(m, i) such that for all i, a, and x:

$$M_{s(m, i)}(x) = M_m(x, i) = M(x, i)$$

If we let all = s(m, i) then we know that for fixed i and all x:

$$M_{all}(x) = M(x, i) = M_u(i, i) = M_i(i)$$

Another way to depict the operation of M_{all} is:

$$M_{all}(x) = \begin{cases} \text{halt if } M_i(i) \text{ halts} \\ \\ \text{diverge if } M_i(i) \text{ diverges} \end{cases}$$

To sum up, from an arbitrary machine M_i we have constructed a machine M_{all} which will halt on all inputs if and only if $M_i(i)$ halts. The following derivation shows this.

$M_i(i)$ halts iff $M_u(i, i)$ halts
iff for all x, M(x, i) halts
iff for all x, $M_m(x, i)$ halts
iff for all x, $M_{s(m,i)}(x)$ halts
iff for all x, $M_{all}(x)$ halts

Each line in the above sequence follows from definitions made above or theorems (s-m-n and universal Turing machine theorems) we have proven before.

Now we have exactly what we were looking for, a machine M_{all} which halts for all inputs if and only if $M_i(i)$ halts. Recalling the discussion at the beginning of the proof, we realize that our theorem has been proven.

Let us reflect on what we have done in this section. Our major accomplishment was to present an unsolvable problem. And, in addition, we presented two more which were related to it. They all concerned halting and as such are relevant to programming and computer science. From this we know that we can never get general answers to questions such as:

- *will this program halt on this data set?*
- *will this program halt on any data set?*

This is indeed a very fine state of affairs! We have shown that there is no way to ever do automatic, general checks on loops or even correctness for the programs we develop. It is unfortunate to close on such a sad note, but the actual situation is even worse! We shall presently find out that hardly anything interesting is solvable.

Reducibility and Unsolvability

A new technique surfaced while proving that the problem of whether or not a Turing machine halts for all inputs is unsolvable. This technique is called *reducibility*. It has traditionally been a very powerful and widely used tool in theoretical computer science as well as mathematics. We shall benefit greatly by investigating it further.

What happened in that proof was to transform an old problem into a new problem by taking instances of the first problem and translating them into instances of the new problem.

We did this also in the proof of the halting problem. Let us closely examine exactly what happened from the point of view of set membership.

First, recall the definition of the diagonal set $K = \{ i \mid M_i(i) \text{ halts} \}$ and define the set of pairs of integers for the general halting problem as $H = \{ \langle i, x \rangle \mid M_i(x) \text{ halts} \}$. We noted that:

$$i \in K \text{ meant exactly the same as } \langle i, i \rangle \in H.$$

We then claimed that we could construct a decision procedure for membership in K based upon deciding membership in H. In other words, to solve membership in K, just take a candidate for membership in K and change it into a candidate for membership in H. If this new element is in H, then the original one had to be in K.

So, we transformed the membership problem for K into that for H by mapping or translating one integer into a pair as follows.

$$i \rightarrow \langle i, i \rangle.$$

(Then, of course, we noted that since membership in K is unsolvable there is no way that membership in H could be solvable since that would imply the solvability of membership in K.)

Now let's return to the proof that deciding whether or not a Turing machine accepts all inputs is unsolvable. Again, we translated a known unsolvable problem into the problem whose unsolvability we were trying to prove. In particular, we took an arbitrary machine M_i and constructed another Turing machine M_{all} such that if M_i halted on its own index (the integer i), then M_{all} halted for all inputs. Then we claimed (and proved) that if the new problem

(halting for all inputs) were solvable then the old one (a Turing machine halting on its own index) had to be solvable too. Thus the new problem must have been unsolvable as well.

This tells us something about the concept of unsolvability as well as providing a new useful technique. Mathematicians often believe that properties that are preserved by mappings or transformations are the most important and in some sense the strongest properties that exist. Thus unsolvability must be a rather serious concept if we cannot get rid of it by changing one problem into another!

OK, back to mappings. Not only did we transform one problem into another, but *we did it in a computable or effective manner*. We took an arbitrary machine M_i and noted that we would like to ask later if $M_i(i)$ halts. From M_i we built another machine $M(x, i)$ which was merely $M_u(i, i)$. We then trotted out Church's thesis and claimed that there was an integer a such that we designed was the machine $M_a(x, i)$. At this point we invoked the s-m-n theorem and asserted that there was a computable function $s(a, i)$ such that

$$M_{s(a, i)}(x) = M_a(x, i) = M(x, i) = M_u(i, i) = M_i(i).$$

And in this manner we were able to show that:

$M_i(i)$ halting means exactly the same as $M_{s(a, i)}(x)$ halting for all x

Thus we used the function s(a,i) to map between problems. Now we turn our attention back to sets since we often formulate things in terms of sets. In fact, sets can be built from problems. Recall that the machine M_i accepts the set W_i (or $W_i = \{ x \mid M_i(x) \text{ halts} \}$) and consider the set definitions in the chart below.

K	**$\{ i \mid M_i(i)$ halts $\}$**	**$\{ i \mid i \in W_i \}$**
E	**$\{ i \mid M_i(x)$ halts for all x $\}$**	**$\{ i \mid \forall x[x \in W_i] \}$**

Note that by showing *'$M_i(i)$ halts if and only if $M_{s(a, i)}(x)$ halts for all x'* we were also demonstrating that:

$$i \in K \text{ if and only if } s(a, i) \in E$$

thus transforming the membership problem for one set (K) into the membership problem for another set (E).

One additional housekeeping item needs discussing. The Turing machine index a is actually a constant rather than a variable since it came from the definition of the particular machine $M(x, i) = M_u(i, i)$. Thus it does not need to be a parameter since it never varies.

With this in mind, we define the function g(i) to be s(a, i). Thus $M_{g(i)}(x)$ is exactly the same machine as $M_{s(a, i)}(x)$ and:

$$i \in K \text{ if and only if } g(i) \in E.$$

Summing up, what we have done is translate (or map) the set K into the set E using the computable function g(i). The Venn diagrams in figure 1 illustrate this. Note that the function g() maps all of K into a portion of E and all of the elements not in K into a portion of E's complement.

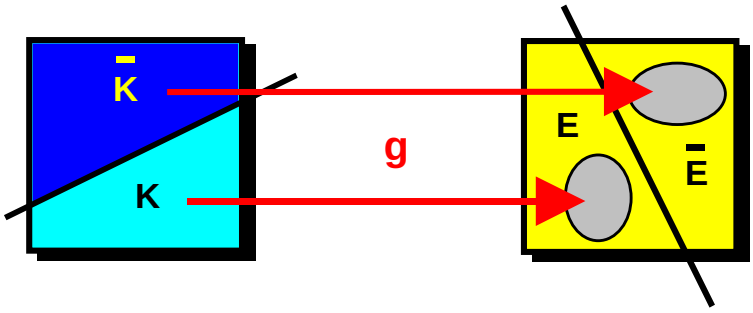


Figure 1 - Mapping between K and E

Note that the mapping does not map K into *all* of E. When given Turing machines whose indices are members of K, it transforms them into new Turing machines that accept all inputs and thus have indices that are members of E.

And, the mapping produces members of E that operate in a very special manner. They make up a family of machines that simulate the original machine's execution with its own index as input $[M_i(i)]$. Since there are many machines that do not perform computations like this, yet halt on all inputs, it is clear that only a portion of E is mapped into.

The formal definition of reducibility between sets appears below. In general it is exactly the same as the mappings we have described in the discussion above and illustrated in figure 1.

> **Definition.** *The set A is **reducible** to the set B (written $A \leq B$) if and only if there is a totally computable function g such that for all x:*
>
> $$x \in A \text{ if and only if } g(x) \in B.$$

That is what we used to do the construction in the last theorem of the last section. It was just a transformation from the set K to the set E. Now we shall tie this into solvability and unsolvability with our next theorem.

Theorem 1. *If A is reducible to B and membership in B is solvable, then membership in A is solvable also.*

Proof. This is very straightforward. First, let $A \leq B$ via the function g. (In other words, $\forall x[x \in A$ iff $g(x) \in B]$.) Now, suppose that the Turing machine M_b solves membership in the set B. Recall that this means that M_b outputs a one when given members of B as input and outputs zeros otherwise.

Consider the machine built from M_b by preprocessing the input with the function g. This gives us the machine $M_b(g(x))$ which we shall claim solves membership in A since g maps members of A into members of B. That is, for all x:

$x \in A$ iff $g(x) \in B$ [since $A \leq B$]
iff $M_b(g(x)) = 1$ [since M_b solves B]

If $M_b(g(x))$ is an actual Turing machine, we are done. We claim it is because M_b is a Turing machine and the function g is computable (thus there is some M_g that computes it). Church's thesis and the s-m-n theorem allow us to find the index of this machine that solves membership in the set A in our standard enumeration.

This leads to an immediate result concerning unsolvability.

Corollary. *If A is reducible to B and membership in A is unsolvable then so is membership in B.*

Let's pause and discuss what we have just discovered. From the theorem and corollary that we just looked at, we can see that we are able perform only the mappings illustrated in figure 2.

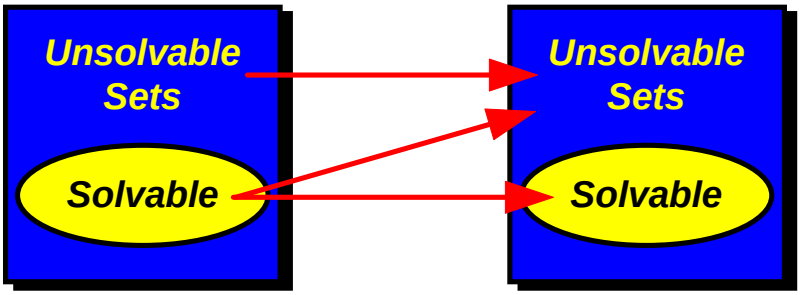


Figure 2 - Allowable reducibility mappings

Note that if a set has a solvable membership problem then we can map from it to any other set except the empty set (since there is nothing there to map into), or the set of all integers (which has an empty complement). But, if a set has an

unsolvable membership problem, we are only able to map from it to unsolvable sets. On the other hand, we are only able to map into solvable sets from other solvable sets. Here is a summation of this in the chart below.

Assume that A is reducible to B

Then if:	***We know:***
A is unsolvable	**B is unsolvable**
A is solvable	**Nothing about B**
B is solvable	**A is solvable**
B is unsolvable	**Nothing about A**

There is some insight about computation to be gained from this. If a set has an unsolvable membership problem then it must be rather difficult to decide whether an integer is a member of the set or not. (That of course is a droll and obvious understatement!) It is certainly more difficult to decide membership in an unsolvable set than one with a solvable membership problem since we are able to do the later but not the first. Thus reducibility takes us from easier problems to more difficult ones. Reducibility (and the intuition of mapping from easy to hard problems) shall surface again in the study of the complexity of computation where we are concerned with the time and space necessary to solve problems and perform computation.

Reducibility provides an additional method for determining the solvability of membership in a set. Here is a summation:

To show that a set has a solvable membership problem:

- ***construct a Turing machine that decides its membership, or***
- ***reduce it to a set with a solvable membership problem.***

To show that a set has an unsolvable membership problem:

- ***prove that no Turing machine decides it's membership, or***
- ***reduce a set with an unsolvable membership problem to it.***

To continue with our discussion about Turing machines and computing membership in certain sets we bring up *properties* of sets at this time. Here is a table containing a few properties and sets of Turing machine indices that correspond to the computable sets possessing that property.

Property	*Set defined from the Property*
Emptiness	$\{ i \mid W_i = \varnothing \}$
Finiteness	$\{ i \mid W_i$ is finite $\}$
Cofiniteness	$\{ i \mid \overline{W_i}$ is finite$\}$
Cardinality	$\{ i \mid W_i$ has exactly k members $\}$
Solvability	$\{ i \mid W_i$ is solvable $\}$
Equality	$\{ \langle a, b \rangle \mid W_a = W_b \}$

That was a list of only a few set properties. Note that the set E defined earlier (W_i contains all integers) was also built from a set property. The set K (machines which halt on their own indices) was not since membership in K depends on the *machine* and not the *set* it accepts.

Speaking about sets, we need to note that there are two *trivial set properties* for the computable sets. One is the property of being computable (all W_i are in this group since they are defined as exactly those sets accepted by Turing machines) and the other is the property not being computable (no W_j are in this group of sets).

Here is a major theorem due to Rice that indicates that nothing is solvable when one asks questions about set properties.

> **Theorem 2** (Rice). *Only trivial set properties are solvable for the computable sets.*
>
> **Proof.** The trivial properties (being or not being a computable set) are indeed easily solvable for the class of computable sets because:
>
> $$\{ i \mid W_i \text{ is computable } \} = \{ \text{all positive integers} \}$$
> $$\{ i \mid W_i \text{ is not computable } \} = \varnothing$$
>
> and these are easily (trivially) solvable.
>
> We need to show that all of other set properties are unsolvable.
>
> If a property is nontrivial then some set must have the property and some set must not. Let us take an arbitrary property and assume that the set W_a has this property. Also, the empty set either has the property or does not. Let us assume that it does not. Thus W_a has the property and $\varnothing$ does not.
>
> (Note that if $\varnothing$ had the property the proof would continue along similar lines. This appears later as an exercise in reducing the complement of K to sets.)
>
> Let us give the name P to the set of all indices (from our standard enumeration) of computable sets that possess this property and show that the unsolvable set K is reducible to P. That is, we must find a computable function g such that for all i:
>
> $$i \in K \text{ if and only if } g(i) \in P.$$

Here is how we define g. For any M_i we construct the machine:

$$M(x,i) = \begin{cases} M_a(x) \text{ if } M_i(i) \text{ halts} \\ \\ \text{diverge otherwise} \end{cases}$$

which operates like M_a if $i \in K$ and diverges on all inputs if $i \notin K$.

Since all M does is to run $M_i(i)$ and then if it halts, turn control over to M_a, M is indeed a Turing machine. Since it is, Church's thesis provides us with an index for M ,and as before, the s-m-n theorem provides a function g such that for all i and x:

$$M_{g(i)} = M(x,i)$$

So, $M_{g(i)}$ is a Turing machine in our standard enumeration of Turing machines. Now let us have a look at exactly what $M_{g(i)}$ accepts. This set is:

$$W_{g(i)} = \begin{cases} W_a \text{ if } i \in K \\ \\ \varnothing \text{ otherwise} \end{cases}$$

because $M_{g(i)}$ acts exactly like M_a whenever $M_i(i)$ halts.

This means that for all i:

$x \in K$	iff $M_i(i)$ halts	[definition of K]
	iff $\forall x[M_{g(i)}(x) = M_a(x)]$	[definition of $M_{g(i)}$]
	iff $W_{g(i)} = W_a$	[$M_{g(i)}$ accepts $W_{g(i)}$]
	iff $g(i) \in P$	[W_a has property P]

Thus we have reduced K to P. Applying the corollary of our last theorem tells us that P is unsolvable and thus whether a computable set has any nontrivial property is likewise an unsolvable problem.

This is a very surprising and disturbing result. It means that almost everything interesting about the computable sets is unsolvable. Therefor we must be very careful about what we attempt to compute.

To conclude the section we present a brief discussion about unsolvability and computer science. Since Turing machines are equivalent to programs, we know

now that there are many questions that we cannot produce programs to answer. This, of course, is because unsolvable problems are exactly those problems that computer programs cannot solve.

Many of the above unsolvable problems can be stated in terms of programs and computers. For example, the halting problem is:

Does this program contain an infinite loop?

and we have shown this to be unsolvable. This means that no compiler may contain a routine that unfailingly predicts looping during program execution.

Several other unsolvable problems concerning programs (some of which are set properties and some of which are not) appear on the following list:

a) Are these two programs equivalent?
b) Will this program halt for all inputs?
c) Does this program accept a certain set? (Correctness)
d) Will this program halt in n^2 steps for an input of length n?
e) Is there a shorter program that is equivalent to this one?
f) Is this program more efficient than that one?

In closing we shall be so bold as to state the sad fact that almost all of the interesting questions about programs are unsolvable!

Enumerable and Recursive Sets

Unfortunately, it seems that very few of the general problems concerning the nature of computation are solvable. Now is the time to take a closer look at some of these problems and in classify them with regard to a finer metric. To do this we need more precision and formality. So, we shall bring forth a little basic mathematical logic.

Examining several of the sets with unsolvable membershp problems, we find that while we cannot decide their membership problems, we are often able to determine when an element *is* a member of some set. In other words, we know if a Turing machine *accepts* a particulay set and halts for some input, then that input is a member of a set. Thus the Turing machine halts for members of the set and provides no information about inputs that are not members. An example is K, the set of Turing machines that halt when given their own indices as input. Recalling that

$$K = \{ i \mid M_i(i) \text{ halts} \} = \{ i \mid i \in W_i \},$$

consider the machine M that can be constructed from the universal Turing machine (M_u) as follows.

$$M(i) = M_u(i, i)$$

Another way to describe M (possibly more intuitively) is:

$$M(i) = \begin{cases} \text{halt if } M_i(i) \text{ halts} \\ \\ \text{diverge otherwise} \end{cases}$$

This *is* a Turing machine. And, since it was just $M_u(i, i)$ we know exactly how to build it and even find its index in our standard enumeration. Furthermore, if we examine it carefully, we discover that it accepts the set K. That is, M will halt for all inputs which are members of K but diverge for nonmembers. There is an important point about this that needs to be stressed.

> *If some integer x is a member of K then M(x) will always tell us so. Otherwise, M(x) provides us with absolutely no information..*

This is because we can detect halting but cannot always detect divergence. After all, if we knew when a machine did not halt, we would be able to solve the

halting problem. In fact, there are three cases of final or terminal behavior in the operation of Turing machines:

a) halting,
b) non-halting which we might detect, and
c) non-detectable divergence.

The latter is the troublesome kind that provides us with unsolvability.

Some of the computable sets have solvable membership problems (for example, the sets of even integers or prime numbers) but many such as K do not. In traditional mathematical logic or recursion theory we name our collection of computable sets the class of *recursively enumerable* sets. There is a reason for this exotic sounding name that will be completely revealed below. The formal definition for members of the class follows.

Definition. *A set is* ***recursively enumerable*** *(abbreviated r.e.) if and only if it can be accepted by a Turing machine.*

We call this family of sets the *class of r.e. sets* and earlier we discovered an enumeration of all of them which we denoted W_1, W_2, ... to correspond to our standard enumeration of Turing machines. Noting that any set with a solvable membership problem is also an r.e. set (as we shall state in a theorem soon) we now present a definition of an important subset of the r.e. sets and an immediate theorem.

Definition. *A set is* ***recursive*** *if and only if it has a solvable membership problem.*

Theorem 1. *The class of recursively enumerable (r.e.) sets properly contains the class of recursive sets.*

Proof. Two things need to be accomplished. First, we state that every recursive set is r.e. because if we can decide if an input is a member of a set, we can certainly accept the set. Next we present a set that does not have a solvable membership problem, but is r.e. That of course, is our old friend, the diagonal set K.

That is fine. But, what else do we know about the relationship between the r.e. sets and the recursive sets? If we note that since we have total information about recursive sets and only partial information about membership in r.e. sets, the following characterization of the recursive sets follows very quickly.

Theorem 2. *A set is recursive if and only if both the set and its complement are recursively enumerable..*

Proof. Let A be a recursive set. Then its complement $\overline{A}$ must be recursive as well. After all, if we can tell whether or not some integer x is a member of A, then we can also decide if x is not a member of A. Thus both are r.e. via the last theorem.

Suppose that A and $\overline{A}$ are r.e. sets. Then, due to the definition of r.e. sets, there are Turing machines that accept them. Let M_a accept the set A and $M_{\overline{a}}$ accept its complement $\overline{A}$. Now, let us consider the following construction.

$$M(x) = \begin{cases} 1 \text{ if } M_a(x) \text{ halts} \\ \\ 0 \text{ if } M_{\overline{a}}(x) \text{ halts} \end{cases}$$

If we can build M as a Turing machine, we have the answer because M *does* solve membership for the set A. *But, it is not clear that M is indeed a Turing machine.* We must explain exactly how M operates. What M must do is to run $M_a(x)$ and $M_{\overline{a}}(x)$ at the same time. This is not hard to do if M has four tapes. It uses two of them for the computation of $M_u(a, x)$ and two for the computation of $M_u(\overline{a}, x)$ and runs them in time-sharing mode (a step of M_a, then a step of $M_{\overline{a}}$). We now note that one of $M_a(x)$ and $M_{\overline{a}}(x)$ *must halt.* Thus M is indeed a Turing machine that decides membership for the set A.

From this theorem come several interesting facts about computation. First, we gain a new characterization of the recursive sets and solvable decision problems, namely, *both the problem and its complement are computable.* We are also soon be able to present our first uncomputable or non-r.e. set. In addition, another closure property for the r.e. sets falls out of this examination. Here are these results.

Theorem 3. *The complement of K is not a recursively enumerable set.*

Proof. The last theorem states that if $\overline{K}$ were r.e. then both it and its complement must be recursive. Since K is not a recursive set, $\overline{K}$ cannot be an r.e. set.

Corollary. *The class of r.e. sets is not closed under complement.*

Remember the halting problem? Another one of the ways to state it is as a membership problem for the set of pairs:

$$H = \{ \langle i, x \rangle \mid M_i(x) \text{ halts} \} = \{ \langle i, x \rangle \mid x \in W_i \}.$$

We have shown that it does not have a solvable membership problem. A little bit of contemplation should be enough to convince anyone that is an r.e. set just like K. But, what about its complement? The last two theorems provide the machinery to show that it also is not r.e.

Theorem 4. *The complement of the halting problem is not r.e.*

Proof. We now know two ways to show that the complement of the halting problem, namely the set $\{\langle i, x\rangle \mid M_i(x) \text{ diverges}\}$ is not an r.e. set. The first is to use theorem 2 that states that a set is recursive if and only if both it and its complement are r.e. If the complement of the halting problem were r.e. then the halting problem would be recursive (or solvable). This is not so and thus the complement of the halting problem must not be r.e.

Another method is to note that if $\{\langle i, x\rangle \mid M_i(x) \text{ diverges}\}$ were r.e. then $\overline{K}$ would have to be r.e. also. This is true because we could use the machine that accepts the complement of the halting problem in order to accept $\overline{K}$. Since $\overline{K}$ is not r.e. then the complement of the halting problem is not either.

The second method of the last proof brings up another fact about reducibilities. It is actually the r.e. version of a corollary to the theorem stating that if a nonrecursive set is reducible to another then it cannot be recursive either.

Theorem 5. *If A is reducible to B and A is not r.e. then neither is B.*

Proof. Let A be reducible to B via the function f. That is, for all x:

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

Let us assume that B is an r.e. set. That means that there is a Turing machine M_b that accepts B. Now construct M in the following manner:

$$M(x) = M_b(f(x))$$

and examine the following sequence of events.

$x \in A$	iff $f(x) \in B$	[since $A \leq B$]
	iff $M_b(f(x))$ halts	[since M_b accepts B]
	iff $M(x)$ halts	[due to definition of M]

This means that if M is a Turing machine (and we know it is because we know exactly how to build it), then M accepts A. Thus A must also be an r.e. set since the r.e. sets are those accepted by Turing machines.

Well, *there* is a contradiction! *A* is *not* r.e. So, some assumption made above must be wrong. By examination we find that the only one that could be wrong was when we assumed that B was r.e.

Now the time has come to turn our discussion to functions instead of sets. Actually, we shall really discuss functions and some of the things that they can do to and with sets. We know something about this since we have seen reducibilities and they are functions that perform operations upon sets.

Returning to our original definitions, we recall that Turing machines compute the computable functions and some compute *total* functions (those that always halt and present an answer) while others compute *partial* functions which are defined on some inputs and not on others. We shall now provide names for this behavior.

Definition. *A function is (total)* ***recursive*** *if and only if it is computed by a Turing machine that halts for every input.*

Definition. *A function is* ***partial recursive*** *(denoted prf) if and only if it can be computed by a Turing machine.*

This is very official sounding and also quite precise. But we need to specify exactly what we are talking about. Recursive functions are the counterpart of recursive sets. We can compute them totally, that is, for *all* inputs. Some intuitive examples are:

$$f(x) = 3x^2 + 5x + 2$$

$$f(x,y) = \begin{cases} x \text{ if } y \text{ is prime} \\ \\ 0 \text{ otherwise} \end{cases}$$

Partial recursive functions are those which do not give us answers for every input. These are exactly the functions we try not to write programs for! This brings up one small thing we have not mentioned explicitly about reducibilities. We need to have answers for every input whenever a function is used to reduce one set to another. This means that the reducibility functions need to be *recursive.* The proper traditional definition of reducibility follows.

Definition. *The set A is* ***reducible*** *to the set B (written* $A \leq B$*) if and only if there is a recursive function f such that for all x:*

$$x \in A \text{ if and only if } f(x) \in B.$$

Another thing that functions are useful for doing is set enumeration (or listing). Some examples of set enumeration functions are:

e(i) = 2∗i = the i^{th} even number
p(i) = the i^{th} prime number
m(i) = the i^{th} Turing machine encoding

These are recursive functions and we have mentioned them before. But, we have not mentioned any general properties about functions and the enumeration of the recursive and r.e. sets. Let us first define what exactly it means for a function to enumerate a set.

Definition. *The function f* ***enumerates*** *the set A (or A =* ***range*** *of f), if and only if for all y,*

a) If $y \in A$*, then there is an x such that* $f(x) = y$ *and*
b) If $f(x) = y$ *then* $y \in A$*.*

Note that partial recursive functions as well as (total) recursive functions can enumerate sets. For example, the function:

$$k(i) = \begin{cases} i \text{ if } M_i \text{ halts} \\ \\ \text{diverge otherwise} \end{cases}$$

is a partial recursive function that enumerates the set K. Here is a general theorem about the enumeration of r.e. sets which explains the reason for their exotic name.

Theorem 6. *A set is r.e. if and only if it is empty or the range of a recursive function.*

Proof. We shall do away with one part of the theorem immediately. If a set is empty then of course it is r.e. since it is recursive. Now what we need to show is that non-empty r.e. sets can be enumerated by recursive functions and that any set enumerated by a recursive function is r.e.

a) If a set is not empty and is r.e. we must find a recursive function that enumerates it.

Let A be a non-empty, r.e. set. We know that there is a Turing machine (which we shall call M_a) which accepts it. Since A is not empty, we may assume that there is some input (let us specify the integer k) which is a member of A. Now consider:

$$M(x,n) = \begin{cases} x \text{ if } M_a(x) \text{ halts in exactly n steps} \\ \\ k \text{ otherwise} \end{cases}$$

We claim that M is indeed a Turing machine and we must demonstrate two things about it. First, the range of M is part of the set A. This is true because M either outputs k (which is a member of A) or some x for which M_a halted in n steps. Since M_a halts only for members of A, we know that the

$$\text{range of } M \subseteq A.$$

Next we must show that our enumerating machin M outputs all of the members of A. For any $x \in A$, $M_a(x)$ must halt in some finite number (let us say m) of steps. Thus $M(x, m) = x$. So, M eventually outputs all of the members of A. In other words:

$$A \subseteq \text{range of } M$$

and we can assert that M exactly enumerates the set A.

(N.B. This is not quite fair since enumerating functions are supposed to have one parameter and M has two. If we define M(z) to operate the same as the above machine with:

x = number of zeros in z
n = number of ones in z

then everything defined above works fine after a little extra computation to count zeros and ones. This is because sooner or later every pair of integers shows up. Thus we have a one parameter machine $M(z) = M(\langle x, n \rangle)$ which enumerates A.)

We also need to show that M does compute a recursive function. This is so because M always halts. Recall that M(x, n) simulates $M_a(x)$ for exactly n steps and then makes a decision of whether to output x or k.

b) The last part of the proof involves showing that if A is enumerated by some recursive function (let us call it f), then A can be accepted by a Turing machine. So, we shall start with A as the range of the recursive function f and examine the following computing procedure.

```
AcceptA(x)

n = 0;
while f(n) ≠ x do n = n + 1;
halt
```

This procedure is computable and halts for all x which are enumerated by f (and thus members of A). It diverges whenever x is not enumerated by f. Since this computable procedure accepts A, we know that A is an r.e. set.

This last theorem provides the reason for the name *recursively enumerable* set. In recursion theory, stronger results have been proven about the enumeration of both the recursive and r.e. sets. The following theorems (provided without proof) demonstrate this.

Theorem 7. *A set is recursive if and only if it is finite or can be enumerated in strictly increasing order.*

Theorem 8. *A set is r.e. if and only if it is finite or can be enumerated in non-repeating fashion.*

NOTES

The landmark paper on unsolvability is:

K. GODEL, "Uber formal unentsheidbare Satze der Principia Mathematica und verwandter Systeme, I," ("On formally undecidable propositions of the Principia Mathematica and related systems, I"), *Monatshefte fur Mathematik und Physik* 38 (1931), 173 -198.

This was followed shortly by the work of Turing which was cited in the notes on computability, and:

A. CHURCH, "A note on the Entscheidungsproblem," *Journal of Symbolic Logic* 1:1 (1936), 40-41, and 1:3 (1936), 101-102.

The study of the class of recursively enumerable sets was initiated in the classic paper:

E. L. POST, "Recursively enumerable sets of positive integers and their decision problems," *Bulletin of the American Mathematical Society* 50 (194), 284-316.

(The above papers have been reprinted in *The Undecidable*, which was cited in the notes on computability. It also includes a translation of Godel's paper.)

All of the books mentioned in the notes on computability contain material on unsolvability and recursiveness. Books which primarily deal with recursive function theory are:

S. C. KLEENE, *Introduction to Metamathematics.* D. Van Nostrand, Princeton, New Jersey, 1952.

M. MACHTEY and P. R. YOUNG, *An Introduction to the General Theory of Algorithms.* North Holland, New York, 1978.

H. ROGERS JR., *The Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

A. YASUHARA, *Recursive Function Theory and Logic.* Academic Press, New York, 1971.

PROBLEMS

Arithmetization

1. Design an algorithm for a Turing machine which recognizes encodings of Turing machine instructions. (Allow it to use the alphabet: {0, 1, *b*, s, n, ...}.) Indicate how this machine could be the basis for one which recognizes Turing machine descriptions.

2. Describe an algorithm for a Turing machine which receives the integer n as input and proceeds to write the description of the n-th Turing machine from the standard enumeration on its tape.

3. Church's thesis tells us that the machine of the previous problem is included in our standard enumeration of Turing machines. Let us call its index k. What happens when it receives the integer k as input?

4. Can you write a program (in any language of your choice) which reproduces itself? This means that when you run the program, its output is the program you just ran. Try it!

5. As we know from our study of computability, programs can be translated into Turing machines. If we check off these Turing machines (those which are transformed programs), we find that there are still some left over in our standard enumeration. Does this mean that there are more Turing machines than programs? Comment.

Properties of the Enumeration

1. Show that there are exactly as many rational numbers as there are nonnegative integers.

2. Show that there are exactly as many programs as there are Turing machines. Discuss problem 5 from the last section in this light.

3. Every Turing machine is equivalent to some other machine in our enumeration. Why? How many other machines is each machine equivalent to? How many times is each Turing-computable function represented in our enumeration? Be sure to justify your answers.

4. Prove that there exist more real numbers between zero and one than there are integers.

5. Any Turing machine may have an infinite number of different outputs. (One for each input.) Exactly how many machines do? Suppose we had a list of triples <k, x, z> where z is the output of Turing machine k on input x. How many items would be on our list?

6. Some infinite sequences of zeros and ones (such as 1010101... or 1101001000...) are not too difficult to compute. In fact we could design Turing machines which compute them. Are there any binary sequences which cannot be computed by Turing machines? Why?

7. Consider the machine which receives x as input and simulates Turing machine number x in our standard enumeration until it wishes to halt. This machine then adds one to the result and halts. In other words, we are examining the machine:

$$M(x) = M_x(x) + 1.$$

What happens when this machine receives its own index as input?

Universal Machines and Simulation

1. State and prove an s-m-n theorem for programs.

2. Describe how the universal Turing machine locates a particular instruction on its description tape.

3. Show that the class of sets accepted by Turing machines is closed under union. (HINT: do not copy the intersection proof.)

4. We know that we can transform Turing machines into programs. In particular, we can find a program equivalent to the universal Turing machine. Design (in the *NICE* language) a program named P_u such that:

$$P_u(i, x) = M_i(x)$$

for every Turing machine M_i and input x.

5. Show that with the program M_u from the last problem and the s-m-n program, we can design a function trans(n) which transforms Turing machines into programs. That is, if trans(i) = k then for all x, $M_i(x) = P_k(x)$ where M_i is a Turing machine and P_k is a program.

Solvability and the Halting Problem

1. Is the following a Turing machine? Please explain either how exactly to build it or precisely why it is not a Turing machine.

$$M(n) = \begin{cases} \text{halt if } M_n(n) \text{ diverges} \\ \\ \text{diverge if } M_n(n) \text{ halts} \end{cases}$$

2. Might the following be a Turing machine? Explain your reasoning.

$$M(i) = \begin{cases} \text{halt if } M_i(n) \text{ halts for some input} \\ \\ \text{diverge otherwise} \end{cases}$$

3. Whether or not an arbitrary Turing machine halts when given the integer 3 as input is obviously a subcase of the general halting problem (just like a machine halting on its own index). Does this fact alone indicate that the problem is unsolvable? Provide precise reasoning which indicates that this is true or a counterexample which shows that it is wrong.

4. We showed that since the problem concerning a machine halting on its own index is unsolvable, the general halting problem for Turing machines is unsolvable. Does this imply that any superset of an unsolvable problem is unsolvable? Provide a proof or a counterexample.

5. Given the Turing machine M_i, consider the machine:

$$M(x) = \begin{cases} M_i(i) \text{ if } x \text{ is a blank} \\ \\ \text{diverge otherwise} \end{cases}$$

If x is blank, when does the machine accept? If x is not a blank, what does the machine accept?

6. Describe the reduction which took place in the last problem. Comment on the unsolvability of the blank tape halting problem.

7. Prove that the membership problem for any finite set is solvable.

8. Define the following problems as predicates or membership problems for sets and prove whether or not they are solvable.

    a) If an arbitrary Turing machine halts for all even numbers.
    b) Whether an arbitrary Turing machine has over 176 instructions.
    c) If you will get an A in the next computer course you take.

9. Show that whether or not an arbitrary Turing machine ever executes a particular one of its instructions is unsolvable. (This is the same as the problem of detecting unreachable code in a program.)

10. Explain precisely what is meant when we say that a Turing machine has entered an infinite loop. Prove that detecting this occurrence for arbitrary Turing machines is unsolvable.

Reducibility and Unsolvability

1. Transform the following problems into set membership problems. Is set membership in these sets solvable? Why?

    a) If an arbitrary Turing machine ever writes a 1 on its tape.
    b) If two arbitrary Turing machines ever accept the same input.
    c) If an arbitrary Turing machine ever runs longer than 193 steps.
    d) If an arbitrary Turing machine accepts at least ten inputs.

2. Suppose there was an enumeration of Turing machines where all of the machines with even indices halted on every input and all of the odd numbered machines did not. Is this possible? Comment on this.

3. Let M_i be a Turing machine which halts for all inputs. Let us further assume that every output of M_i is the index of a Turing machine which halts for every input. That is if for some x, $M_i(x) = k$ then M_k halts for all inputs. Thus M_i outputs a list of Turing machines which always halt. Prove that there is a Turing machine which halts for all inputs that is not on this list.

4. Any general computational problem concerning Turing machines can be stated for programs. Show that general problems which are unsolvable for Turing machines are also unsolvable for programs.

5. Suppose that someone brings you a program which does some task which you find important. If they claim that nobody can write a shorter program to accomplish this task, should you believe them? Is there some general method you can use to check on this?

6. The ***index set*** for a property is the set comprised of *all indices of all Turing machines which accept a set possessing the property*. Another way to state this is to say that for some set property *P*, the index set for *P* is:

$$\{ i \mid W_i \text{ has property P} \}.$$

(Note that these are properties of sets and not properties of individual machines. Thus, if two machines accept the same set, they are either both in an index set or neither is a member.) Two examples of index sets are: the set of all Turing machines which never halt, and the set of all Turing machines which always halt. Two general facts about index sets are:

a) An index set either contains all of the indices of the Turing machines which never halt or it contains none of them.

b) If an index set is *nontrivial* (this means that it has some members, but not all of the integers), then it is infinite and so is its complement.

Show that the above two statements are indeed facts and intuitively explain why $K = \{ i \mid i \in W_i \}$ is not an index set.

6. Only half of the proof for Rice's theorem (that only trivial set properties are solvable) was provided in this section. Prove the remaining half by showing that $\overline{K}$ is reducible to any index set which does not contain the empty set.

Enumerable and Recursive Sets

1. Are the following sets recursive? Are they recursively enumerable? Justify your conjectures.

   a) $\{ x \mid x \text{ is an even integer} \}$
   b) $\{ i \mid M_i \text{ halts for all inputs} \}$
   c) $\{ i \mid M_i \text{ halts only for prime integers} \}$
   d) $\{ i \mid M_i \text{ is not a Turing machine} \}$

2. Prove that if the set A is not recursively enumerable and can be reduced to the set B, then B cannot be recursively enumerable.

3. Show that the following sets are not recursively enumerable.

   a) $\{ i \mid W_i = \emptyset \}$
   b) $\{ i \mid W_i = \text{all integers} \}$

4. Show that if P(x, y) is a recursive predicate then the following is r.e.

   $$\{ x \mid P(x, y) \text{ is true for some } y \}$$

5. A ***complete*** set is a recursively enumerable set to which every recursively enumerable set can be reduced. Show that K is a complete set.

6. Prove that every index set which is recursively enumerable is a complete set.

7. Let f(x) be a recursive function. Is its range recursive? r.e.?

8. Is the image of a partial recursive function recursive? r.e.?

9. Prove that a set is recursive if and only if it is finite or can be enumerated in strictly increasing order.

COMPLEXITY

Thus far we have examined the nature of computation by specifying exactly what we mean by computable and then going a step further and becoming aquainted with several things that are not computable. This was interesting and somewhat useful since we now have a better idea about what is possible and what tasks we should avoid. But we need to delve into issues closer to actual, real-world computation. This brings up the issue of computational cost.

In order to examine this we shall develop a framework in which to classify tasks by their difficulty and possibly identify things that require certain amounts of various resources. In addition we shall discover properties of computational problems which place them beyond our reach in a practical sense. To do this we will examine decision properties for classes of recursive sets and functions with an emphasis opon the difficulty of computation.

The sections include:

Measures and Resource Bounds

In order to answer questions concerning the complexity or difficulty of computation, we must first arrive at some agreement on what exactly is meant by the term *complexity*. One is tempted to confuse this with the complexity of understanding just how some task is computed. That is, if it involves an intricate or tedious computing procedure, then it is complex. But that is a trap we shall leave for the more mathematical when they claim that a proof is difficult. We shall base our notion of difficulty upon the very practical notion of computational cost and in turn define this to be related to the amount of resources used during computation. (Simply put: if something takes a long time then it *must* be hard to do!)

Let us consider the resources used in computation. And, most important are those which seem to limit computation. In particular, we will examine *time* and *space* constraints during computation. This is very much in line with computer science practice since many problems are costly to us or placed beyond our reach due to lack of time or space - even on modern computing equipment.

We shall return to Turing machines in order to examine computational difficulty. This may seem rather arbitrary and artificial, but this choice is reasonable since most natural models of computation are not too far apart in the amounts of time and space used in computation for the same functions. (For example, consider the space used by Turing machines and programs that compute the same functions or decide membership in the same sets. They are very similar indeed!) In addition, the simplicity of the Turing machine model makes our study far less cumbersome.

All we need do is associate a time cost function and a space cost function with each machine in order to indicate exactly how much of each resource is used during computation. We shall begin with time.

Our machine model for time complexity will be the multitape Turing machine. Part of the reason for this is tradition, and part can be explained by examining the computations done by one-tape machines. For example, a two tape machine can decide whether a string is made up of n zeros followed by n ones (in shorthand we write this as 0^n1^n) in exactly 2n steps while a one tape machine might require about nlogn steps for the same task. Arguments like this make multitape machines an attractive, efficient model for computation. We shall

assume that we have a *standard enumeration* of these machines that we shall denote:

$$M_1,\ M_2,\ \ldots$$

and define a time cost function for each machine.

> **Definition** *The **time function** $T_i(n)$ is the maximum number of steps taken by multitape Turing machine M_i on any input of length n.*

These multitape machines are able to solve certain computational problems faster than their one-tape cousins. But intuitively, one might maintain that using these machines is very much in line with computation via programs since tapes are nearly the same as arrays and programs may have lots of arrays. Thus we shall claim that this is a sensible model of computation to use for our examination of complexity.

The tradeoffs gained from using many tapes instead of one or two are presented without proof in the next two theorems. First though, we need some additional mathematical notation.

> ***Definition*** *Given two recursive functions f and g, f = O(g) (pronounced: f is the **order** of g or f is **big OH** of g) if and only if there is a constant k such that $f(n) \leq k*g(n)$ for all but a finite number of n.*

This means that smaller functions are the order of larger ones. For example, x^2 is the order of 2^n or $O(2^n)$. And, if two functions are the same up to a constant, then they are of the same order. Intuitively this means that their graphs have roughly the same shape when plotted. Examine the three functions in figure 1.

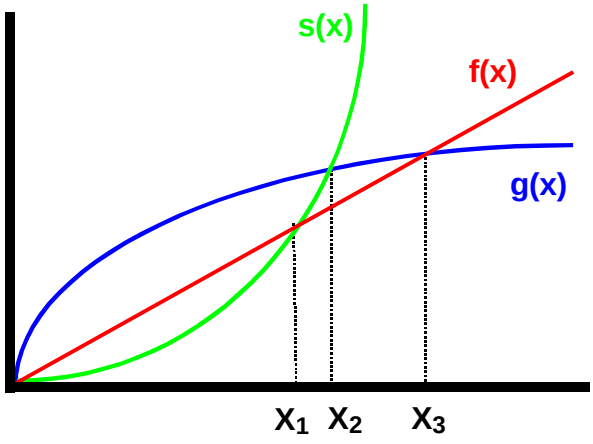


Figure 1 - Three functions

After point x_1, function s(x) is always greater than f(x) and for values of x larger than x_2, it exceeds g(x). Since s(x) grows much faster than f(x) and g(x) we know that for any constant k, after some point $s(x) > k*f(x)$ and $s(x) > k*g(x)$. Due to this, we say that f(x) = O(s(x)) and g(x) = O(s(x)). Similarly, at x_3, f(x) becomes larger than g(x) and since it remains so, we note that g(x) = O(f(x)). (The folk rule to remember here is that small, or slowly growing functions are the order of larger and faster ones.)

Let us think about this concept. Consider a linear function (such as 6x) and a quadratic (like x^2). They do not look the same (one is a line and the other is a curve) so they are not of the same order. But $5x^3 - 2x^2 - 15$ is $O(x^3)$ because it is obviously less than $6x^3$. And $\log_2(n^2)$ is $O(\log_2 n)$. While we're on the subject of logarithms, note that logs to different bases are of the same order. For example:

$$\log_e x = (\log_e 2) \log_2 x.$$

The constant here is $\log_e 2$. This will prove useful soon. Here are the theorems that were promised that indicate the relationship between one-tape Turing machines and multi-tape machines.

> **Theorem 1.** *Any computation which can be accomplished in t(n) time on a multitape Turing machine can be done in $O(t(n)^2)$ time using a one tape Turing machine.*

> **Theorem 2.** *Any computation that can be accomplished in t(n) time on a multitape Turing machine can be done in $O(t(n)\log_2(t(n))$ time using a two tape Turing machine.*

Now we need to turn our attention to the other resource that often concerns us, namely space. Our model will be a little different.

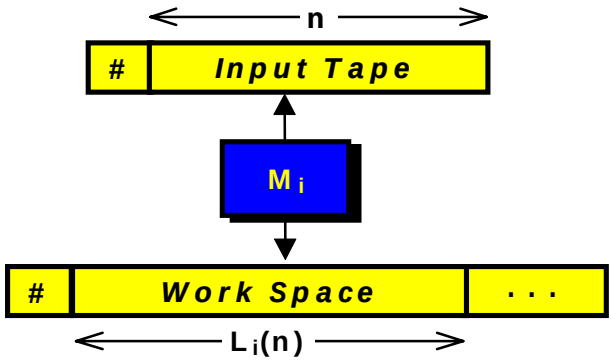


Figure 2 - Space Complexity Model

We shall not allow the Turing machine to use its input tape for computation, which means using a read-only input tape. For computation we shall give the machine one (possibly multi-track) work tape. That looks like the picture in figure 3.

The formal definition of space complexity now follows.

> **Definition.** *The **space function** $L_i(n)$ is the maximum number of work tape squares written upon by Turing machine M_i for any input of length n.*

Several additional comments about our choice of machine model are in order here. Allowing no work to be done on the input tape means that we may have space complexities that are less than the input length. This is essential since sets such as strings of the form 0^n1^n can be recognized in $\log_2 n$ space by counting the 0's and 1's in binary and then comparing the totals. Note also that since we are not concerned with the speed of computation, having several work tapes is not essential.

The machine models used in our examination of complexity have no restrictions as to number of symbols or number of tracks used per tape. This is intentional. Recalling the discussion of tracks and symbols when we studied computability we note that they are interchangeable in that a k symbol horizontal block of a tape can be written as a vertical block on k tracks. And k tracks can be represented on one track by expanding the machine's alphabet.

Thus we may read lots of symbols at once by placing them vertically in tracks instead of horizontally on a tape. We may also do lots of writes and moves on these columns of symbols. Since a similar idea is used in the proof of the next theorem, we shall present it with just the intuition used in a formal proof.

> **Theorem 3** (Linear Space Compression). *Anything that can be computed in s(n) space can also be computed in s(n)/k space for any constant k.*

> **Proof sketch**. Suppose that we had a tape with a workspace that was twelve squares long (we are not counting the endmarker) like that provided below.

#	a	b	c	d	e	f	g	h	i	j	k	l

> Suppose further that we wished to perform the same computation that we are about to do on that tape, but use a tape with a smaller workspace, for example, one that was four squares long or a third of the size of the original tape. Consider the tape shown below.

#	a	b	c	d	*
	e	f	g	h	
	i	j	k	l	

On this one, when we want to go from the square containing the 'd' to that containing the 'e' we see the right marker and then return to the left marker and switch to the middle track.

In this manner computations can always be performed upon tapes that are a fraction of the size of the original. The general algorithm for doing this is as follows for a machine M(x) that uses s(n) space for its computation.

n = the length of the input x
Lay off exactly s(n)/k squares on the work tape
Set up k rows on the work tape
Perform the computation M(x) in this space

There are two constraints that were omitted from the theorem so that it would be more readable. On is that s(n) must be at least $\log_2 n$ since we are to count up to n on the work tape before computing s(n). The other is that we must be able to lay out s(n)/k squares within that amount of tape.

We must mention another slightly picky point. In the last theorem there had to have been some sort of lower bound on space. Our machines do need at least one tape square for writing answers! So, when we talk of using s(n) space we really mean max(1, s(n)) space.

In the same vein, when we speak of t(n) time, we mean max(n+1, t(n)) time since in any nontrivial computation the machine must read its input and verify that it has indeed been read.

This brings up a topic that we shall mention and then leave to the interested reader. It is *real time computation*. By this, we mean that an answer must be presented *immediately* upon finishing the input stream. (For example, strings of the form 0^n1^n are real time recognizable on a 2-tape Turing machine.) In the sequel whenever we speak of O(n) time we mean O(kn) time or *linear time*, not real time. This is absolutely essential in our next theorem on linear time speedup. But first, some more notation.

Definition *The expression $\inf_{n\to\infty} f(n)$ denotes the limit of the greatest lower bound of f(n), f(n+1), f(n+2), ... as n goes to infinity.*

The primary use of this limit notation will be to compare time or space functions. Whenever we can say that

$$\inf_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

we know that the function f *grows faster* than the function g by more than a constant. Another way to say this is that for *every* constant k,

$$f(n) > k*g(n)$$

for all but a finite number of n. In other words, the limit (as n goes to infinity) of f(n) divided by g(n) cannot be bounded by a constant. Thus f *is not* O(g), but larger. For example, f(x) and g(x) could be x^3 and x^2, or even n and $\log_2 n$.

With that out of the way, we may present another version of the last theorem, this time with regard to time. Note that we must use our new *inf* notation to limit ourselves to machines that read their own input. This infers that if a machine does not run for at least on the order of n steps, then it is not reading its input and thus not computing anything of interest to us.

> **Theorem 4** (Linear Time Speedup). *Anything which can be computed in t(n) time can also be computed in t(n)/k time for any constant k if* $inf_{n\to\infty} t(n)/n = \infty$.

The proof of this theorem is left as an exercise since it is merely a more careful version of the linear space compression theorem. All we do is read several squares at once and then do several steps as one step. We note in passing though that if proven carefully the theorem holds for O(n) time also. (Recall that we mean linear time, not real time.)

So far, so good. A naive peek at the last two results might lead us to believe that we can compute things faster and faster if we use little tricks like doing several things at once! Of course we know that this is too good to be true. In fact, practice bears this out to some extent. We can often speed up our algorithms by a constant if we are clever. (And we did not need to do much mathematics to learn that!) Also, we know that there are *best algorithms* for much of the stuff we compute.

This brings up an interesting question. Does everything have a best or most efficient algorithm? Or at least a best algorithm up to a constant? A rather surprising result from abstract complexity theory tells us that there are some problems that have none. And, in fact, there are problems in which computation time can be cut by any amount one might wish.

> **Theorem 5** (Speedup). *For any recursive function g(n), there is a set A such that if the Turing machine M_i decides its membership then there is an equivalent machine M_k such that for all but a finite number of n: $T_i(n) \leq g(T_k(n))$.*

In other words, machine M_k runs as much faster than machine M_i as anyone might wish. Yes, we said that correctly! We can have M_k and M_i computing the same functions with $T_i(n)$ more than $[T_k(n)]^2$. Or even

$$T_i(n) \geq 2^{T_k(n)}$$

This is quite something! It is even rather strange if you think about it.

It means also that there are some problems that can never be computed in a most efficient manner. A strict interpretation of this result leads to an interesting yet rather bizarre corollary.

> **Corollary.** *There are computational problems such that given any program solving them on the world's fastest supercomputer, there is an equivalent program for a cheap programmable pocket calculator that runs faster!*

A close look at the speedup theorem tells us that the corollary is indeed true but must be read *very carefully*. Thoughts like this also lead to questions about these problems that have no best solution. We shall leave this topic with the reassuring comment that even though there are problems like that, they are so strange that most likely none will ever arise in a practical application.

Complexity Classes

All of our computing devices now possess two additional attributes: time and space bounds. We shall take advantage of this and classify all of the recursive sets based upon their computational complexity. This allows us to examine these collections with respect to resource limitations. This, in turn, might lead to the discovery of special properties common to some groups of problems that influence their complexity. In this manner we may learn more about the intrinsic nature of computation. We begin, as usual, with the formal definitions.

> **Definition.** *The class of all sets computable in time t(n) for some recursive function t(n),* ***DTIME(t(n))*** *contains every set whose membership problem can be decided by a Turing machine which halts within O(t(n)) steps on any input of length n.*

> **Definition.** *The class of all sets computable in space s(n) for some recursive function s(n),* ***DSPACE(s(n))*** *contains every set whose membership problem can be decided by a Turing machine which uses at most O(s(n)) tape squares on any input of length n.*

We must note that we defined the classes with bounds of *order* t(n) and *order* s(n) for a reason. This is because of the linear space compression and time speedup theorems presented in the section on measures. Being able to use order notation brings benefits along with it. We no longer have to mention constants. We may just say n^2 time, rather than $3n^2 + 2n - 17$ time. And the bases of our logarithms need appear no longer. We may now speak of nlogn time or logn space. This is quite convenient!

Some of the automata theoretic classes examined in the study of automata fit nicely into this scheme of complexity classes. The smallest space class, DSPACE(1), is the class of regular sets and DSPACE(n) is the class of sets accepted by deterministic linear bounded automata. And, remember that the smallest time class, DTIME(n), is the class of sets decidable in *linear* time, not real time.

Our first theorem, which follows almost immediately from the definitions of complexity classes, assures us that we shall be able to find all of the recursive sets within our new framework. It also provides the first characterization of the class of recursive sets we have seen.

> **Theorem 1.** *The union of all the DTIME(t(n)) classes or all of the DSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof. This is quite stratighforward. From the original definitions, we know that membership in any recursive set can be decided by some Turing machine that halts for every input. The time and space functions for these machines name the complexity classes that contain these sets. In other words, if M_a decides membership in the recursive set A, then A is obviously a member of DTIME($T_a(n)$) and DSPACE($L_a(n)$).

On the other hand, if a set is in some complexity class then there must be some Turing machine that decides its membership within some recursive time or space bound. Thus a machine which always halts decides membership in the set. This makes all of the sets within a complexity class recursive.

We now introduce another concept in computation: *nondeterminism*. It might seem a bit strange at first, but examining it in the context of complexity is going to provide us with some very important intuition concerning the complexity of computation. We shall provide two definitions of this phenomenon.

The first is the historical definition. Early in the study of theoretical computing machines, the following question was posed.

Suppose a Turing machine is allowed several choices of action for an input-symbol pair. Does this increase its computational power?

Here is a simple example. Consider the following Turing machine instruction.

0	**1**	**right**	**next**
1	**1**	**left**	**same**
1	**0**	**halt**	
b	**1**	**left**	**I75**

When the machine reads a one, it may either print a one and move left or print a zero and halt. This is a choice. And the machine may choose either of the actions in the instruction. If it is possible to reach a halting configuration, then the machine accepts.

We need to examine nondeterministic computation in more detail. Suppose that the above instruction is I35 and the machine containing it is in configuration: #0110(I35)110 reading a one. At this point the instruction allows it to make either choice and thus enter one of two different configurations. This is pictured below as figure 1.

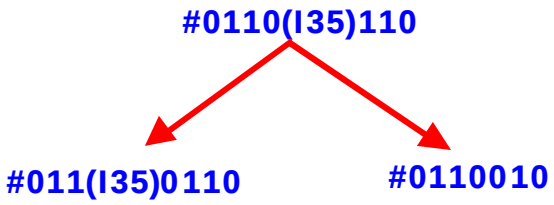


Figure 1 - A Computational Choice

We could now think of a computation for one of these new machines, not as a mystical, magic sequence of configurations, but as a tree of configurations that the machine could pass through during its computation. Then we consider paths through the computation tree as possible computations for the machine. Then if there is a path from the initial configuration to a halting configuration, we say that the machine halts. A more intuitive view of set acceptance may be defined as follows.

> **Definition.** *A nondeterministic Turing machine accepts the input x if and only if there is a path in its computation tree that leads from the initial configuration to a halting configuration.*

Here are the definitions of nondeterministic classes.

> **Definition.** *For a recursive function t(n) is* ***NTIME(t(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that halt within O(t(n)) steps for every input of length n.*

> **Definition.** *For a recursive function s(n),* ***NSPACE(s(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that use at most O(s(n)) tape squares for any input of length n.*

(NB. We shall see NSPACE(n) in the context of formal languages. It is the family of context sensitive languages or sets accepted by nondeterministic linear bounded automata.)

Now that we have a new group of computational devices, the first question to ask is whether or not they allow us to compute anything new. Our next theorem assures us that we still have the recursive sets. It is given with a brief proof sketch since the details will be covered in other results below.

> **Theorem 2.** *The union of all the NTIME(t(n)) classes or all of the NSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof Sketch. There are two parts to this. First we maintain that since the recursive sets are the union of the DTIME or DSPACE classes, then they are all contained in the union of the NTIME or NSPACE classes.

Next we need to show that any set accepted by a nondeterministic Turing machine has a decidable membership problem. Suppose that a set is accepted by a t(n)-time nondeterministic machine. Now recall that the machine accepts if and only if there is a path in its computation tree that leads to a halting configuration. Thus all one needs to do is to generate the computation tree to a depth of t(n) and check for halting configurations.

Now let us examine nondeterministic acceptance from another viewpoint. A path through the computation tree could be represented by a sequence of rows in the instructions that the machine executes. Now consider the following algorithm that receives an input and a path through the computation tree of some nondeterministic Turing machine M_m.

```
Verify(m, x, p)
Pre:  p[] = sequence of rows in instructions to be
      executed by Mm as it processes input x
Post: halts if p is a path through the computation tree

i = 1;
config = (I1)#x;
while config is not a halting configuration and i≤ k do
  i = i + 1;
  if row p(i) in config's instruction can be executed by Mm
     then set config to the new configuration
     else loop forever
if config is a halting configuration then halt(true)
```

This algorithm verifies that p is indeed a path through the computation tree of M_m and if it leads to a halting configuration, the algorithm halts and accepts. Otherwise it either loops forever or terminates without halting. In addition, the algorithm is *deterministic*. There are <u>*no*</u> choices to follow during execution.

Now let us examine paths through the computation tree. Those that lead to halting configurations show us that the input is a member of the set accepted by the machine. We shall say that these paths *certify* that the input is a member of the set and call the path a *certificate of authenticity* for the input. This provides a clue to what nondeterministic operation is really about.

Certificates do not always have to be paths through computation trees. Examine the following algorithm for accepting nonprimes (composite numbers) in a nondeterministic manner.

```
NonPrime(x)
nondeterministically determine integers y and z;
if y*z = x then halt(true)
```

Here the certificate of authenticity is the pair <y, z> since it demonstrates that x is not a prime number. We could write a completely deterministic algorithm which when given the triple <x, y, z> as input, compares y*z to x and certifies that x is not prime if x = y∗z.

This leads to our second definition of nondeterministic operation. We say that the following deterministic Turing machine M uses certificates to *verify membership in the set A.*

M(x, c) halts if and only if c provides a proof of $x \in A$

The nondeterministic portion of the computation is finding the certificate and we need not worry about that. Here are our definitions in terms of verification.

> **Definition.** *The class of all sets nondeterministicly acceptable in time t(n) for a recursive function t(n),* ***NTIME(t(n))*** *contains all of the sets whose members can be verified by a Turing machine in at most O(t(n)) steps for any input of length n and certificate of length $\leq t(n)$.*

Note that certificates must be shorter in length than t(n) for the machine to be able to read them and use them to verify that the input is in the set.

We should also recall that nondeterministic Turing machines and machines which verify from certificates do not decide membership in sets, but accept them. This is an important point and we shall come back to it again.

At this point we sadly note that the above wonderfully intuitive definition of nondeterministic acceptance by time-bounded machines does not extend as easily to space since there seems to be no way to generate certificates in the worktape space provided.

We mentioned earlier that there is an important distinction between the two kinds of classes. In fact, important enough to repeat. *Nondeterministic machines accept sets, while deterministic machines decide membership in sets.* This is somewhat reminiscent of the difference between recursive and recursively enumerable sets and there are some parallels. At present the

differences between the two kinds of classes is not well understood. In fact, it is not known whether these methods of computation are equivalent. We do know that

$$\begin{aligned} &DSPACE(1) = NSPACE(1) \\ &DSPACE(s(n)) \subseteq NSPACE(s(n)) \\ &DTIME(t(n)) \subseteq NTIME(t(n)) \end{aligned}$$

for every recursive s(n) and t(n). Whether DSPACE(s(n)) = NSPACE(s(n)) or whether DTIME(t(n)) = NTIME(t(n)) remain famous open problems. The best that anyone has achieved so far is the following result that is presented here without proof.

Theorem 3. *If $s(n) \geq \log_2 n$ is a space function, then $NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$.*

Our next observation about complexity classes follows easily from the linear space compression and speedup theorems. Since time and space use can be made more efficient by a constant factor, we may state that:

$$\begin{aligned} &DTIME(t(n)) = DTIME(k*t(n)) \\ &NTIME(t(n)) = NTIME(k*t(n)) \\ &DSPACE(s(n)) = DSPACE(k*s(n)) \\ &NSPACE(s(n)) = NSPACE(k*s(n)) \end{aligned}$$

for every recursive s(n) and t(n), and constant k. (Remember that t(n) means max(n+1, t(n)) and that s(n) means max(1, s(n)) in each case.)

While we are comparing complexity classes it would be nice to talk about the relationship between space and time. Unfortunately not much is known here either. About all we can say is rather obvious. Since it takes one unit of time to write upon one tape square we know that:

$$TIME(t(n)) \subseteq SPACE(t(n))$$

because a machine cannot use more than t(n) tape squares if it runs for t(n) steps. Going the other way is not so tidy. We can count the maximum number of steps a machine may go through before falling into an infinite loop on s(n) tape and decide that for some constant c:

$$SPACE(s(n)) \subseteq TIME(2^{cs(n)})$$

for both deterministic and nondeterministic complexity classes. And, in fact, this counting of steps is the subject of our very next theorem.)

Theorem 4. *If an s(n) tape bounded Turing machine halts on an input of length n then it will halt within $O(2^{cs(n)})$ steps for some constant c.*

Proof Sketch. Consider a Turing machine that uses O(s(n)) tape. There is an equivalent machine M_i that uses two worktape symbols and also needs no more than O(s(n)) tape. This means that there is a constant k such that M_i never uses more than k∗s(n) tape squares on inputs of length n.

We now recall that a machine configuration consists of:

a) the instruction being executed,
b) the position of the head on the input tape,
c) the position of the head on the work tape, and
d) a work tape configuration.

We also know that if a machine repeats a configuration then it will run forever. So, we almost have our proof.

All we need do is count machine configurations. There are $|M_i|$ instructions, n+2 input tape squares, k∗s(n) work tape squares, and $2^{ks(n)}$ work tape configurations. Multiplying these together provides the theorem's bound.

One result of this step counting is a result relating nondeterministic and deterministic time. Unfortunately it is nowhere near as sharp as theorem 3, the best relationship between deterministic and nondeterministic space. Part of the reason is that our simulation techniques for time are not as good as those for space.

Corollary. *$NTIME(t(n)) \subseteq DTIME(2^{ct(n)^2})$*

Proof. $NTIME(t(n)) \subseteq NSPACE(t(n)) \subseteq DSPACE(t(n)^2) \subseteq DTIME(2^{ct(n)^2})$ because of theorems 3 and 4. (We could have proven this from scratch by simulating a nondeterministic machine in a deterministic manner, but the temptation to use our last two results was just too tempting!)

Our first theorem in this section stated that the union of all the complexity classes results in the collection of all of the recursive sets. An obvious question is whether one class can provide the entire family of recursive sets. The next result denies this.

Theorem 5. *For any recursive function s(n), there is a recursive set that is not a member of DSPACE(s(n)).*

Proof. The technique we shall use is diagonalization over *DSPACE(s(n))*. We shall examine every Turing machine that operates in s(n) space and define a set that cannot be decided by any of them.

First, we must talk about the machines that operate in O(s(n)) space. For each there is an equivalent machine which has one track and uses the alphabet {0,1,*b*}. This binary alphabet, one track Turing machine also operates in O(s(n)) space. (Recall the result on using a binary alphabet to simulate machines with large alphabets that used blocks of standard size to represent symbols.) Let's now take an enumeration M_1, M_2, ... of these one track, binary machines and consider the following algorithm.

```
Examine(i, k, x)
Pre: n = length of x

lay out k*s(n) tape squares on the work tape;
run Mi(x) within the laid off tape area;
if Mi(x) rejects then accept else reject
```

This is merely a simulation of the binary Turing machine M_i on input x using k∗s(n) tape. And, the simulation lasts until we know whether or not the machine will halt. Theorem 4 tells us that we only need wait some constant times $2^{cs(n)}$ steps. This is easy to count to on a track of a tape of length k∗s(n). Thus the procedure above is recursive and acts differently than M_i on input x if $L_i(n) \leq k*s(n)$.

Our strategy is going to be to feed the Examine routine all combinations of k and i in hopes that we shall eventually knock out all s(n) tape bounded Turing machines.

Thus we need a sequence of pairs <i, k> such that each pair occurs in our sequence infinitely often. Such sequences abound. A standard is:

<1,1>, <1,2>, <2,1>, <1,3>, <2,2>, <3,1>, ...

For each input x we take the x^{th} pair in the sequence. The decision procedure for the set we claim is not s(n) space computable is now:

```
Diagonal(x)
select the x-th pair <i, k> from the sequence;
Examine(i, k, x)
```

Two things need to be verified. First, we need to show that the above decision procedure can be carried out by some Turing machine. We note that M_i comes from an enumeration of two work tape symbol machines and then appeal to Church's thesis for the actual machine construction for the decision procedure. Next we need to prove that this procedure cannot be carried out by an s(n) space bounded Turing machine.

Suppose that the Diagonal procedure is indeed s(n) space computable. Then there is some two worktape symbol, s(n) space bounded Turing machine M_i which computes the above Diagonal procedure. And there is a constant k such that for all but a finite number of inputs, M_i uses no more than k∗s(n) tape squares on inputs of length n. In particular, there is an x such that <j, k> is the x^{th} pair in our sequence of pairs and the computation of $M_i(x)$ requires no more than k∗s(n) tape. (In fact there are infinitely many of these x since the pair <j, k> appears infinitely often in the sequence.) In this case

$$M_j(x) \neq \text{Examine}(j, k, x) = \text{Diagonal}(x)$$

which is a contradiction. Thus M_i cannot be an s(n) bounded machine and the set defined by our Diagonal procedure cannot be a member of DSPACE(s(n)).

It should come as no surprise that the same result holds for nondeterministic space as well as time classes. Thus we do have a hierarchy of classes since none of them can hold all of the recursive sets. This seems in line with intuition since we believe that we can compute bigger and better things with more resources at our disposal.

Our next results explore the amount of space or time needed to compute new things for classes with resource bounds that are tape or time functions. (Recall that *tape* or *time* functions are bounds for actual Turing machines.) We consider these resource bounds to be *well behaved* and note in passing that there are strange functions about which are not tape or time functions.

Theorem 6 (Space Hierarchy). *If r(n) and s(n) are both at least O(n), s(n) is a space function, and* $\inf_{n\to\infty} r(n)/s(n) = 0$, *then* $DSPACE(s(n)) - DSPACE(r(n)) \neq \varnothing$.

Proof Sketch. The proof is very similar to that of the last theorem. All that is needed is to change the space laid off in the Examine routine to s(n). Since s(n) grows faster than any constant times r(n), the diagonalization proceeds as scheduled. One note. What makes this simulation and diagonalization possible is that s(n) is a space function.

This allows us to lay off s(n) tape squares in s(n) space. Thus the diagonalization does produce a decision procedure for a set which is s(n) space decidable but not r(n) space decidable.

The major reason the simulation worked was that we were able to lay out s(n) tape squares. This is because we could compute s(n) by taking the machine it was a tape function for and run it on all inputs of length n to find the longest one. This requires O(n) space. If s(n) is even *more well behaved* we can do better.

Definition. *A recursive function s(n) is* ***efficiently space computable*** *if and only if it can be computed within s(n) space.*

If s(n) is efficiently space computable, then the space hierarchy theorem is true for s(n) down to $O(\log_2 n)$ because we can lay out the required space for the simulation and keep track of which input symbol is being read.

Many functions are efficiently space computable, including such all time favorites such as $\log_2 n$ and $(\log_2 n)^k$. An exercise dealing with efficient space computability will be to prove that all space functions that are at least O(n) are efficiently space computable.

Combining the space hierarchy theorem with the linear space compression theorem provides some good news at last. If two functions differ only by a constant, then they bound the same class. But if one is larger than the other by more than a constant then one class is properly contained in the other.

Sadly the result for time is not as sharp. We shall need one of our functions to always be efficiently time computable and do our simulation with two tapes. Here is the theorem.

Theorem 7 (Time Hierarchy). *If r(n) and t(n) are both at least O(n), t(n) is efficiently time computable, and*

$$\inf_{n \to \infty} \frac{r(n)\log_2 r(n)}{t(n)} = 0$$

then DTIME(t(n)) - DTIME(r(n)) ≠ ∅.

Reducibilities and Completeness.

We now have a framework in which to study the recursive functions in relation to one of the major considerations of computer science: computational complexity. We know that there is a hierarchy of classes but we do not know very much about the classes other than this. In this section we shall present a tool for refining our rough framework of complexity classes.

Let us return to an old mathematical technique we saw in chapter two. There we mapped problems into each other and used this to be make statements about unsolvability. We shall do the same at the subrecursive level except that we shall use mappings to help us determine the complexity of problems. And even use the same kind of mappings that were used before. Here is the definition of reducibility one more time.

> **Definition.** *The set A is* ***many-one reducible*** *to the set B (written* $A \leq_m B$*) if and only if there is a recursive function g(x) such that for all x:* $x \in A$ *if and only if* $g(x) \in B$.

With the function g(x) we have mapped all of the members of A into the set B. Integers not in A get mapped into B 's complement. Symbolicly:

$$g(A) \subseteq B \quad \text{and} \quad g(\overline{A}) \subseteq \overline{B}$$

Note that the mapping is *into* and that several members of A may be mapped into the same element of B by g(x).

An important observation is that we have restated the membership problem for A in terms of membership in B. This provides a new way to decide membership in A. Let's have a look. If the Turing machine M_b decides membership in B and M_g computes g(x) then membership in A can be decided by:

$$M_a(x) = M_b(M_g(x)).$$

(In other words, compute g(x) and then check to see if it is in B.)

Now for the complexity of all this. It is rather straightforward. Just the combined complexity of computing g(x) and then membership in B. In fact, the time and space complexities for deciding membership in A are:

$$L_a(n) = \text{maximum}[L_g(n), L_b(\,|g(x)|\,)]$$
$$T_a(n) = T_g(n) + T_b(\,|g(x)|\,)$$

Thinking a little about the length of g(x) and its computation

we easily figure out that the number of digits in g(x) is bounded by the computation space and time used and thus:

$$|g(x)| \leq L_g(n) \leq T_g(n)$$

for x of length n. This makes the complexity of the decision problem for A:

$$L_a(n) \leq \text{maximum}[L_g(n), L_b(L_g(n))]$$
$$T_a(n) \leq T_g(n) + T_b(L_g(n))$$

An essential aside on space below O(n) is needed here. If $L_b(n)$ is less than O(n) and |g(x)| is O(n) or greater we can often compute symbols of g(x) one at a time as needed and then feed them to M_b for its computation. Thus we need not write down all of g(x) - just the portion required at the moment by M_b. We do however need to keep track of where the read head of M_b is on g(x). This will use $\log_2|g(x)|$ space. If this is the case, then

$$L_a(n) \leq \text{maximum}[L_g(n), L_b(|g(x)|), \log_2|g(x)|]$$

Mapping functions which are almost straight character by character translations are exactly what is needed. As are many $\log_2 n$ space translations.

What we would like to do is to be able to say something about the complexity of deciding membership in A in terms of B's complexity and not have to worry about the complexity of computing g(x). It would be perfect if $A \leq_m B$ meant that A is no more complex than B. This means that computing g(x) must be less complex than the decision problem for B. In other words, the mapping g(x) must preserve | not influence) complexity. In formal terms:

> **Definition.** *Let $A \leq_m B$ via g(x). The recursive function g(x) is **complexity preserving** with respect to **space** if and only if there is a Turing machine M_b which decides membership in B and a Turing machine M_g which computes g(x) such that:*
>
> $$maximum[L_g(n), L_b(L_g(n))] = O(L_b(n)).$$

> **Definition.** *Let $A \leq_m B$ via g(x). The recursive function g(x) is **complexity preserving** with respect to **time** if and only if there is a Turing machine M_b which decides membership in B and a Turing machine M_g which computes g(x) such that:*
>
> $$T_g(n) + T_b(L_g(n)) = O(T_b(n)).$$

These complexity preserving mappings now can be used to demonstrate the relative complexities of decision problems for sets. In fact, we can often pinpoint a set's complexity by the use of complexity preserving reducibilities. The next two theorems explain this.

Theorem 1. *If $A \leq_m B$ via a complexity preserving mapping and B is in DSPACE(s(n)) then A is in DSPACE(s(n)) also.*

Proof. Follows immediately from the above discussion. That is, if $M_a(x) = M_b(g(x))$ where g(x) is a complexity preserving mapping from A to B, then $L_a(n) = O(L_b(n))$.

Theorem 2. *If $A \leq_m B$ via a complexity preserving mapping and the best algorithm for deciding membership in A requires O(s(n)) space then the decision problem for B cannot be computed in less than O(s(n)) space.*

Proof. Suppose that the membership problem for B can be computed in less than O(s(n)) space. Then by theorem 1, membership in A can be computed in less. This is a contradiction.

Now, just what have we accomplished? We have provided a new method for finding upper and lower bounds for the complexity of decision problems. If $A \leq_m B$ via a complexity preserving mapping, then the complexity of A's decision problem is the lower bound for B's and the complexity of B's decision problem is the upper bound for A's. Neat. And it is true for time too.

An easy example might be welcome. We can map the set of strings of the form $0^n\#1^n$ into the set of strings of the form w#w (where w is a string of 0's and 1's). The mapping is just a character by character transformations which maps both 0's and 1's into 0's and maps the marker (#) into itself. Thus we have shown that $0^n\#1^n$ is no more difficult to recognize than w#w.

Now that we have provided techniques for establishing upper and lower complexity bounds for sets in terms of other sets, let us try the same thing with classes. In other words, why not bound the complexity for an entire class of sets all at once?

Definition. *The set A is **hard** for a class of sets if and only if every set in the class is many-one reducible to A.*

What we have is a way to put an upper bound on the complexity for an entire class. For example if we could show that deciding w#w is hard for the context free languages then we would know that they are all in $DSPACE(\log_2 n)$ or $DTIME(n)$. Too bad that we cannot! When we combine hardness with complexity preserving reducibilities and theorem 1, out comes an easy corollary.

Corollary. *If $A \in DSPACE(s(n))$ is hard for $DSPACE(r(n))$ via complexity preserving mappings then $DSPACE(r(n)) \subseteq DSPACE(s(n))$.*

And, to continue, what if a set is hard for the class which contains it? This means that the hard set is indeed the most difficult set to recognize in the class. We have a special name for that.

> **Definition.** *A set is **complete** for a class if and only if it is a member of the class and hard for the class.*

Think about this concept for a bit. Under complexity preserving mappings complete sets are the most complex sets in the classes for which they are complete. In a way they represent the class. If there are two classes with complete sets, then comparing the complete sets tells us a lot about the classes. So, why don't we name the classes according to their most complex sets?

> **Definition.** *The class of sets no more complex (with respect to space) than A **DSPACE(A)** contains all sets B such that for each Turing machine which decides membership in A there is some machine which decides membership in B in no more space.*

Of course we can do the same for time and nondeterministic resources also. We must note that we defined the complexity relationship between A and B very carefully. This is because sets with speedup may be used to name or denote classes as well as those with best algorithms. Thus we needed to state that *for each algorithm for A there is one for B which is no worse,* so if the set B has speedup, then it can still be in *DSPACE(A)* even if A has speedup too. Now for a quick look at what happens when we name a class after its complete set.

> **Theorem 3.** *If the set A is DSPACE(s(n))-complete via complexity preserving reducibilities then DSPACE(s(n)) = DSPACE(A).*
>
> **Proof.** Theorem 2 assures us that every set reducible to A is no more difficult to decide membership in than A. Thus *DSPACE(S(N))* 5 *DSPACE(A).* And, since A is a member of *DSPACE(S(N))* then all members of *DSPACE(A)* must be also since they require less time to decide membership in than A.

That was not unexpected. Seems like we set it up that way! We now go a bit further with our next question. Just what kinds of complexity classes have complete sets? Do all of them? Let us start by asking about the functions which are resource bounds for classes with complete sets and go on from there.

> **Theorem 4.** *Every space complexity class which has a complete set via complexity preserving reducibilities is some DSPACE(s(n)) where s(n) is a space function.*

Proof. Let *DSPACE(r(n))* have the complete set A. Since A is a member of *DSPACE(r(n))*, there must be a Turing machine M_a which decides membership in A in O(r(n)) space. Thus $L_a(n) = O(r(n))$. Which makes *DSPACE*($L_a(n)$) 5 *DSPACE(r(n))*. Theorem 1 assures us that *DSPACE(r(n))* 5 *DSPACE*($L_a(n)$ since all of the the members of *DSPACE(r(n))* are reducible to A.

Theorem 5. *If there is a best algorithm for deciding membership in the set A then there is a space function s(n) such that DSPACE(A) = DSPACE(s(n)).*

Proof. Almost trivial. The space function s(n) which we need to name the complexity class is just the space function for the Turing machine which computes the most efficient algorithm for deciding membership in A.

Now we know that space functions can be used to name some of our favorite complexity classes: those with complete sets, and those named by sets with best decision procedures. Let us turn to the classes named by sets which have no best decision procedure. These, as we recall were strange. They have speedup and can be computed by sequences of algorithms which run faster and faster.

Theorem 6. *If the set S has speedup then there is no recursive function s(n) such that DSPACE(s(n)) = DSPACE(S).*

All of the results mentioned in the last series of theorems (3 - 6) are true for time and both kinds of nondeterministic class. Well, except for theorem 6. For time the speedup must be at least $n\log_2 n$. But that is not too bad.

We would like to have a theorem which states that all classes with space or time functions as their resource bounds have complete sets. This *is* true for a few of these classes, namely those with polynomial bounds. For now, let us leave this section with the reassuring that complete sets can define classes and none of them have speedup.

Complexity Classes

All of our computing devices now possess two additional attributes: time and space bounds. We shall take advantage of this and classify all of the recursive sets based upon their computational complexity. This allows us to examine these collections with respect to resource limitations. This, in turn, might lead to the discovery of special properties common to some groups of problems that influence their complexity. In this manner we may learn more about the intrinsic nature of computation. We begin, as usual, with the formal definitions.

> **Definition.** *The class of all sets computable in time t(n) for some recursive function t(n),* ***DTIME(t(n))*** *contains every set whose membership problem can be decided by a Turing machine which halts within O(t(n)) steps on any input of length n.*

> **Definition.** *The class of all sets computable in space s(n) for some recursive function s(n),* ***DSPACE(s(n))*** *contains every set whose membership problem can be decided by a Turing machine which uses at most O(s(n)) tape squares on any input of length n.*

We must note that we defined the classes with bounds of *order* t(n) and *order* s(n) for a reason. This is because of the linear space compression and time speedup theorems presented in the section on measures. Being able to use order notation brings benefits along with it. We no longer have to mention constants. We may just say n^2 time, rather than $3n^2 + 2n - 17$ time. And the bases of our logarithms need appear no longer. We may now speak of nlogn time or logn space. This is quite convenient!

Some of the automata theoretic classes examined in the study of automata fit nicely into this scheme of complexity classes. The smallest space class, DSPACE(1), is the class of regular sets and DSPACE(n) is the class of sets accepted by deterministic linear bounded automata. And, remember that the smallest time class, DTIME(n), is the class of sets decidable in *linear* time, not real time.

Our first theorem, which follows almost immediately from the definitions of complexity classes, assures us that we shall be able to find all of the recursive sets within our new framework. It also provides the first characterization of the class of recursive sets we have seen.

> **Theorem 1.** *The union of all the DTIME(t(n)) classes or all of the DSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof. This is quite stratighforward. From the original definitions, we know that membership in any recursive set can be decided by some Turing machine that halts for every input. The time and space functions for these machines name the complexity classes that contain these sets. In other words, if M_a decides membership in the recursive set A, then A is obviously a member of DTIME($T_a(n)$) and DSPACE($L_a(n)$).

On the other hand, if a set is in some complexity class then there must be some Turing machine that decides its membership within some recursive time or space bound. Thus a machine which always halts decides membership in the set. This makes all of the sets within a complexity class recursive.

We now introduce another concept in computation: *nondeterminism*. It might seem a bit strange at first, but examining it in the context of complexity is going to provide us with some very important intuition concerning the complexity of computation. We shall provide two definitions of this phenomenon.

The first is the historical definition. Early in the study of theoretical computing machines, the following question was posed.

Suppose a Turing machine is allowed several choices of action for an input-symbol pair. Does this increase its computational power?

Here is a simple example. Consider the following Turing machine instruction.

0	**1**	**right**	**next**
1	**1**	**left**	**same**
1	**0**	**halt**	
b	**1**	**left**	**I75**

When the machine reads a one, it may either print a one and move left or print a zero and halt. This is a choice. And the machine may choose either of the actions in the instruction. If it is possible to reach a halting configuration, then the machine accepts.

We need to examine nondeterministic computation in more detail. Suppose that the above instruction is I35 and the machine containing it is in configuration: #0110(I35)110 reading a one. At this point the instruction allows it to make either choice and thus enter one of two different configurations. This is pictured below as figure 1.

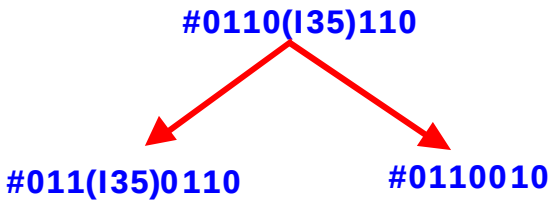


Figure 1 - A Computational Choice

We could now think of a computation for one of these new machines, not as a mystical, magic sequence of configurations, but as a tree of configurations that the machine could pass through during its computation. Then we consider paths through the computation tree as possible computations for the machine. Then if there is a path from the initial configuration to a halting configuration, we say that the machine halts. A more intuitive view of set acceptance may be defined as follows.

> **Definition.** *A nondeterministic Turing machine accepts the input x if and only if there is a path in its computation tree that leads from the initial configuration to a halting configuration.*

Here are the definitions of nondeterministic classes.

> **Definition.** *For a recursive function t(n) is* ***NTIME(t(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that halt within O(t(n)) steps for every input of length n.*

> **Definition.** *For a recursive function s(n),* ***NSPACE(s(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that use at most O(s(n)) tape squares for any input of length n.*

(NB. We shall see NSPACE(n) in the context of formal languages. It is the family of context sensitive languages or sets accepted by nondeterministic linear bounded automata.)

Now that we have a new group of computational devices, the first question to ask is whether or not they allow us to compute anything new. Our next theorem assures us that we still have the recursive sets. It is given with a brief proof sketch since the details will be covered in other results below.

> **Theorem 2.** *The union of all the NTIME(t(n)) classes or all of the NSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof Sketch. There are two parts to this. First we maintain that since the recursive sets are the union of the DTIME or DSPACE classes, then they are all contained in the union of the NTIME or NSPACE classes.

Next we need to show that any set accepted by a nondeterministic Turing machine has a decidable membership problem. Suppose that a set is accepted by a t(n)-time nondeterministic machine. Now recall that the machine accepts if and only if there is a path in its computation tree that leads to a halting configuration. Thus all one needs to do is to generate the computation tree to a depth of t(n) and check for halting configurations.

Now let us examine nondeterministic acceptance from another viewpoint. A path through the computation tree could be represented by a sequence of rows in the instructions that the machine executes. Now consider the following algorithm that receives an input and a path through the computation tree of some nondeterministic Turing machine M_m.

```
Verify(m, x, p)
Pre:  p[] = sequence of rows in instructions to be
      executed by Mm as it processes input x
Post: halts if p is a path through the computation tree

i = 1;
config = (I1)#x;
while config is not a halting configuration and i≤ k do
  i = i + 1;
  if row p(i) in config's instruction can be executed by Mm
     then set config to the new configuration
     else loop forever
if config is a halting configuration then halt(true)
```

This algorithm verifies that p is indeed a path through the computation tree of M_m and if it leads to a halting configuration, the algorithm halts and accepts. Otherwise it either loops forever or terminates without halting. In addition, the algorithm is *deterministic*. There are <u>*no*</u> choices to follow during execution.

Now let us examine paths through the computation tree. Those that lead to halting configurations show us that the input is a member of the set accepted by the machine. We shall say that these paths *certify* that the input is a member of the set and call the path a *certificate of authenticity* for the input. This provides a clue to what nondeterministic operation is really about.

Certificates do not always have to be paths through computation trees. Examine the following algorithm for accepting nonprimes (composite numbers) in a nondeterministic manner.

```
NonPrime(x)
nondeterministically determine integers y and z;
if y*z = x then halt(true)
```

Here the certificate of authenticity is the pair <y, z> since it demonstrates that x is not a prime number. We could write a completely deterministic algorithm which when given the triple <x, y, z> as input, compares y*z to x and certifies that x is not prime if x = y∗z.

This leads to our second definition of nondeterministic operation. We say that the following deterministic Turing machine M uses certificates to *verify membership in the set A.*

M(x, c) halts if and only if c provides a proof of $x \in A$

The nondeterministic portion of the computation is finding the certificate and we need not worry about that. Here are our definitions in terms of verification.

> **Definition.** *The class of all sets nondeterministicly acceptable in time t(n) for a recursive function t(n),* ***NTIME(t(n))*** *contains all of the sets whose members can be verified by a Turing machine in at most O(t(n)) steps for any input of length n and certificate of length $\leq t(n)$.*

Note that certificates must be shorter in length than t(n) for the machine to be able to read them and use them to verify that the input is in the set.

We should also recall that nondeterministic Turing machines and machines which verify from certificates do not decide membership in sets, but accept them. This is an important point and we shall come back to it again.

At this point we sadly note that the above wonderfully intuitive definition of nondeterministic acceptance by time-bounded machines does not extend as easily to space since there seems to be no way to generate certificates in the worktape space provided.

We mentioned earlier that there is an important distinction between the two kinds of classes. In fact, important enough to repeat. *Nondeterministic machines accept sets, while deterministic machines decide membership in sets.* This is somewhat reminiscent of the difference between recursive and recursively enumerable sets and there are some parallels. At present the

differences between the two kinds of classes is not well understood. In fact, it is not known whether these methods of computation are equivalent. We do know that

$$\begin{aligned} &DSPACE(1) = NSPACE(1) \\ &DSPACE(s(n)) \subseteq NSPACE(s(n)) \\ &DTIME(t(n)) \subseteq NTIME(t(n)) \end{aligned}$$

for every recursive s(n) and t(n). Whether DSPACE(s(n)) = NSPACE(s(n)) or whether DTIME(t(n)) = NTIME(t(n)) remain famous open problems. The best that anyone has achieved so far is the following result that is presented here without proof.

Theorem 3. *If $s(n) \geq \log_2 n$ is a space function, then $NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$.*

Our next observation about complexity classes follows easily from the linear space compression and speedup theorems. Since time and space use can be made more efficient by a constant factor, we may state that:

$$\begin{aligned} &DTIME(t(n)) = DTIME(k*t(n)) \\ &NTIME(t(n)) = NTIME(k*t(n)) \\ &DSPACE(s(n)) = DSPACE(k*s(n)) \\ &NSPACE(s(n)) = NSPACE(k*s(n)) \end{aligned}$$

for every recursive s(n) and t(n), and constant k. (Remember that t(n) means max(n+1, t(n)) and that s(n) means max(1, s(n)) in each case.)

While we are comparing complexity classes it would be nice to talk about the relationship between space and time. Unfortunately not much is known here either. About all we can say is rather obvious. Since it takes one unit of time to write upon one tape square we know that:

$$TIME(t(n)) \subseteq SPACE(t(n))$$

because a machine cannot use more than t(n) tape squares if it runs for t(n) steps. Going the other way is not so tidy. We can count the maximum number of steps a machine may go through before falling into an infinite loop on s(n) tape and decide that for some constant c:

$$SPACE(s(n)) \subseteq TIME(2^{cs(n)})$$

for both deterministic and nondeterministic complexity classes. And, in fact, this counting of steps is the subject of our very next theorem.)

Theorem 4. *If an s(n) tape bounded Turing machine halts on an input of length n then it will halt within $O(2^{cs(n)})$ steps for some constant c.*

Proof Sketch. Consider a Turing machine that uses O(s(n)) tape. There is an equivalent machine M_i that uses two worktape symbols and also needs no more than O(s(n)) tape. This means that there is a constant k such that M_i never uses more than k∗s(n) tape squares on inputs of length n.

We now recall that a machine configuration consists of:

a) the instruction being executed,
b) the position of the head on the input tape,
c) the position of the head on the work tape, and
d) a work tape configuration.

We also know that if a machine repeats a configuration then it will run forever. So, we almost have our proof.

All we need do is count machine configurations. There are $|M_i|$ instructions, n+2 input tape squares, k∗s(n) work tape squares, and $2^{ks(n)}$ work tape configurations. Multiplying these together provides the theorem's bound.

One result of this step counting is a result relating nondeterministic and deterministic time. Unfortunately it is nowhere near as sharp as theorem 3, the best relationship between deterministic and nondeterministic space. Part of the reason is that our simulation techniques for time are not as good as those for space.

Corollary. *$NTIME(t(n)) \subseteq DTIME(2^{ct(n)^2})$*

Proof. $NTIME(t(n)) \subseteq NSPACE(t(n)) \subseteq DSPACE(t(n)^2) \subseteq DTIME(2^{ct(n)^2})$ because of theorems 3 and 4. (We could have proven this from scratch by simulating a nondeterministic machine in a deterministic manner, but the temptation to use our last two results was just too tempting!)

Our first theorem in this section stated that the union of all the complexity classes results in the collection of all of the recursive sets. An obvious question is whether one class can provide the entire family of recursive sets. The next result denies this.

Theorem 5. *For any recursive function s(n), there is a recursive set that is not a member of DSPACE(s(n)).*

Proof. The technique we shall use is diagonalization over *DSPACE(s(n))*. We shall examine every Turing machine that operates in s(n) space and define a set that cannot be decided by any of them.

First, we must talk about the machines that operate in O(s(n)) space. For each there is an equivalent machine which has one track and uses the alphabet {0,1,*b*}. This binary alphabet, one track Turing machine also operates in O(s(n)) space. (Recall the result on using a binary alphabet to simulate machines with large alphabets that used blocks of standard size to represent symbols.) Let's now take an enumeration M_1, M_2, ... of these one track, binary machines and consider the following algorithm.

```
Examine(i, k, x)
Pre: n = length of x

lay out k*s(n) tape squares on the work tape;
run Mi(x) within the laid off tape area;
if Mi(x) rejects then accept else reject
```

This is merely a simulation of the binary Turing machine M_i on input x using k∗s(n) tape. And, the simulation lasts until we know whether or not the machine will halt. Theorem 4 tells us that we only need wait some constant times $2^{cs(n)}$ steps. This is easy to count to on a track of a tape of length k∗s(n). Thus the procedure above is recursive and acts differently than M_i on input x if $L_i(n) \leq k*s(n)$.

Our strategy is going to be to feed the Examine routine all combinations of k and i in hopes that we shall eventually knock out all s(n) tape bounded Turing machines.

Thus we need a sequence of pairs <i, k> such that each pair occurs in our sequence infinitely often. Such sequences abound. A standard is:

<1,1>, <1,2>, <2,1>, <1,3>, <2,2>, <3,1>, ...

For each input x we take the x^{th} pair in the sequence. The decision procedure for the set we claim is not s(n) space computable is now:

```
Diagonal(x)
select the x-th pair <i, k> from the sequence;
Examine(i, k, x)
```

Two things need to be verified. First, we need to show that the above decision procedure can be carried out by some Turing machine. We note that M_i comes from an enumeration of two work tape symbol machines and then appeal to Church's thesis for the actual machine construction for the decision procedure. Next we need to prove that this procedure cannot be carried out by an s(n) space bounded Turing machine.

Suppose that the Diagonal procedure is indeed s(n) space computable. Then there is some two worktape symbol, s(n) space bounded Turing machine M_i which computes the above Diagonal procedure. And there is a constant k such that for all but a finite number of inputs, M_i uses no more than k∗s(n) tape squares on inputs of length n. In particular, there is an x such that <j, k> is the x^{th} pair in our sequence of pairs and the computation of $M_i(x)$ requires no more than k∗s(n) tape. (In fact there are infinitely many of these x since the pair <j, k> appears infinitely often in the sequence.) In this case

$$M_j(x) \neq \text{Examine}(j, k, x) = \text{Diagonal}(x)$$

which is a contradiction. Thus M_i cannot be an s(n) bounded machine and the set defined by our Diagonal procedure cannot be a member of DSPACE(s(n)).

It should come as no surprise that the same result holds for nondeterministic space as well as time classes. Thus we do have a hierarchy of classes since none of them can hold all of the recursive sets. This seems in line with intuition since we believe that we can compute bigger and better things with more resources at our disposal.

Our next results explore the amount of space or time needed to compute new things for classes with resource bounds that are tape or time functions. (Recall that *tape* or *time* functions are bounds for actual Turing machines.) We consider these resource bounds to be *well behaved* and note in passing that there are strange functions about which are not tape or time functions.

Theorem 6 (Space Hierarchy). *If r(n) and s(n) are both at least O(n), s(n) is a space function, and* $\inf_{n\to\infty} r(n)/s(n) = 0$, *then* $DSPACE(s(n)) - DSPACE(r(n)) \neq \varnothing$.

Proof Sketch. The proof is very similar to that of the last theorem. All that is needed is to change the space laid off in the Examine routine to s(n). Since s(n) grows faster than any constant times r(n), the diagonalization proceeds as scheduled. One note. What makes this simulation and diagonalization possible is that s(n) is a space function.

This allows us to lay off s(n) tape squares in s(n) space. Thus the diagonalization does produce a decision procedure for a set which is s(n) space decidable but not r(n) space decidable.

The major reason the simulation worked was that we were able to lay out s(n) tape squares. This is because we could compute s(n) by taking the machine it was a tape function for and run it on all inputs of length n to find the longest one. This requires O(n) space. If s(n) is even *more well behaved* we can do better.

Definition. *A recursive function s(n) is* ***efficiently space computable*** *if and only if it can be computed within s(n) space.*

If s(n) is efficiently space computable, then the space hierarchy theorem is true for s(n) down to $O(\log_2 n)$ because we can lay out the required space for the simulation and keep track of which input symbol is being read.

Many functions are efficiently space computable, including such all time favorites such as $\log_2 n$ and $(\log_2 n)^k$. An exercise dealing with efficient space computability will be to prove that all space functions that are at least O(n) are efficiently space computable.

Combining the space hierarchy theorem with the linear space compression theorem provides some good news at last. If two functions differ only by a constant, then they bound the same class. But if one is larger than the other by more than a constant then one class is properly contained in the other.

Sadly the result for time is not as sharp. We shall need one of our functions to always be efficiently time computable and do our simulation with two tapes. Here is the theorem.

Theorem 7 (Time Hierarchy). *If r(n) and t(n) are both at least O(n), t(n) is efficiently time computable, and*

$$\inf_{n \to \infty} \frac{r(n)\log_2 r(n)}{t(n)} = 0$$

then DTIME(t(n)) - DTIME(r(n)) ≠ ∅.

NOTES

Hartmanis and Stearns begin the study of computational complexity on Turing machines. The early papers on time and space as well as multitape simulation and real-time computation are:

J. HARTMANIS and R. E. STEARNS. "On the computational complexity of algorithms," *Trans. AMS 117 (1965)*, 285-305.

J. HARTMANIS, P. M. LEWIS II, and R. E. STEARNS. "Hierarchies of memory limited computations," *Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logical Design (1965)*, 179-190.

P. M. LEWIS II, R. E. STEARNS, and J. HARTMANIS. "Memory bounds for the recognition of context-free and context-sensitive languages," *Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logical Design (1965)*, 191-202.

F.C. HENNIE and R.E. STEARNS. "Two-tape simulation of multitape Turing machines," *J. ACM 13:4 (1966)*, 533-546.

M. O. RABIN. "Real-time computation," *Israel J. Math. 1 (1963)*, 203-211.

The speedup theorem as well as a axiomatic theory of complexity came from:

M. BLUM. "A machine-independent theory of the complexity of recursive functions," *J. ACM 14:2 (1967)*, 322-336.

The theorem on the relationship between deterministic and nondeterministic space classes is from:

W. J. SAVITCH. "Relationships between nondeterministic and deterministic tape complexities," *J. Comput. and System Sci 4:2 (1970)*, 177-192.

Cobham was the first to mention the class P and the initial NP-complete set was discovered by Cook. Karp quickly produced more and the last reference is an encyclopedia of such sets.

A. COBHAM. "The instrinsic computational difficulty of functions," *Proc. 1964 Congress for Logic, Mathematics, and the Philosophy of Science*. North Holland, 1964, 24-30.

S. A. COOK. "The complexity of theorem proving procedures," *Proc. 3rd Annual ACM Symp. on the Theory of Computation (1971)*, 151-158.

R. M. KARP. "Reducibility among combinatorial problems," *Complexity of Computer Computations*, Plenum Press, NY, 1972, 85-104.

M. R. GAREY and D. S. JOHNSON. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* H. Freeman, San Francisco, 1978.

More material on complexity may be found in any of the general theory of computation texts mentioned in the notes for the section on computability.

PROBLEMS

The sets described below shall be used in many of the problems for this chapter. As usual, w represents a string of 0's and 1's and the superscript R stands for *string reversal*.

$A = 0^n1^n$	$B = 0^n1^n0^n$	$C = 0^n1^m0^n1^m$
$D = w\#w^R$	$E = w\#w$	$F = ww$

Measures and Resource Bounds

1. Present the least time consuming algorithms you are able for deciding membership in sets A, D, and F above on multitape Turing machines. Analyze the space required for these algorithms.

2. Develop the most space efficient algorithms you can for membership in the sets A, D, and F above. How much time is needed for algorithm?

3. How many times a Turing machine reverses the direction of its tape heads can be a measure of complexity too. Try to compute membership in the sets A, D, and F above with the fewest number of tape head reversals. Describe your algorithms.

4. Assume that one unit of ink is used to print a symbol on a Turing machine tape square. How much ink is needed to decide membership in sets A, D, and F above?

5. Try time as a complexity measure on one tape (rather than multitape) Turing machines. Now how much time do algorithms for deciding A, D, and F take?

6. Let T(n), S(n), R(n), and I(n) denote the time, space, reversals, and ink needed for Turing machine M to process inputs of length n. Why must a machine always use at least as much time as space? Thus $S(n) \leq T(n)$. What are the relationships between the above four measures of complexity.

7. Define a measure of complexity which you feel properly reflects the actual cost of computation. Determine the complexity of deciding membership in the sets A, D, and F above.

8. Using multitape Turing machines as a computational model, precisely prove that linear speedup in time is possible.

9. Let us use Turing machines with a read-only input tape and a single work tape as our computational model. In addition, let us only use 0, 1, and blank as tape symbols. Is there a difference in space complexity when multitrack work tapes are used as opposed to one track work tapes? (That is, if a task requires m tape squares on a k track machine, how much space is needed on a one track model?)

10. Solve exercise 9 for time instead of space.

11 Show that the simulation of a k tape Turing machine's computation by a one tape Turing machine can be done in the square of the original time.

12. If $\inf_{n\to\infty} f(n)/g(n) = 0$, then what is the relationship between the functions f and g? Does f = O(g)? Does g = O(f)?

13. If $\inf_{n\to\infty} f(n)/g(n) = k$, for some constant k, then what is the relationship between the functions f and g? Again, does f = O(g)? Does g = O(f)?

14. What are the time and space requirements for LL parsing (from the presentation on languages)?

15. Using the speedup theorem, prove the bizarre corollary concerning supercomputers and pocket calculators.

Complexity Classes

1. Show that n^2 and 2^n are time functions.

2. Demonstrate that $(\log_2 n)^2$ is a space function.

3. Prove that for every recursive function f(n) there exists a time function t(n) and a space function s(n) which exceed f(n) for all values of n.

4. Show that there are recursive functions which cannot be time or space functions.

5. Verify that a set is a member of DSPACE(n) if its members can be recognized within O(n) space for all but a finite number of n.

6. Prove that if a set can be *accepted* by some deterministic Turing machine, then its membership problem can be *decided* in the same amount of space or time.

7. Show that the family of DSPACE classes is closed under union, intersection, and complement.

8. Prove that the family of NSPACE classes is closed under concatenation and Kleene star.

9. Design a nondeterministic Turing machine which accepts members of the above sets A and C. (That is, $A \cup C$.) Is this quicker than doing it in a deterministic manner?

10. Present a nondeterministic Turing machine which finds roots of polynomials. (That is, given a sequence of integers a_0 , ... , a_n it figures out a value of x such that: $a_0 + a_1x + a_2x^2 + ... + a_nx^n = \delta$ for some specified error bound δ.)

11. Describe how in a nondeterministic manner one could accept pairs of finite automata which do not accept the same set.

12. Show that if a space function is at least O(n) then it is efficiently space computable.

13. How much space is needed to compute $\log_2 n$?

14. Prove the time hierarchy theorem using $r(n)^2$ rather than $r(n)\log_2 r(n)$.

Reducibilities and Completeness

1. Show that the set A above is reducible to the sets D and F by mappings which can be computed by finite automata.

2. Provide a mapping from set D above to set E. Analyze the space requirements of the mapping.

3. Show that set A above is reducible to set C. Take a Turing machine M_c which decides membership in C, combine it with the machine which maps A into C, and produce a machine which decides membership in A. Further, do this so that the space requirements are only $O(\log_2 n)$.

4. Precisely state and prove theorems 1 and 2 for time complexity.

5. Show that if the set A is a member of the class DTIME(B) then DTIME(A) is a subclass of DTIME(B). Be sure to consider the cases where the sets A and B have speedup.

6. Prove theorem 4 for nondeterministic time classes.

7. Prove that complexity classes named by sets with speedup cannot be named by recursive functions. (If S has speedup then there is no recursive f such that DSPACE(A) = DSPACE(f).)

The Classes *P* and *NP*

1. Estimate the time (in seconds, years, etc.) that it would take to solve a problem which takes $O(2^n)$ steps on a typical computer. Do this for various values of n.

2. Prove that any set in P can be reduced to any other set in P via some polynomial time mapping.

3. Verify theorem 2.

4. Show that the existence of a set in NP whose complement was not also in NP would lead to a proof that $P \neq NP$.

5. Demonstrate that the entire satisfiability problem is indeed in NP.

6. Present an algorithm for converting a Turing machine instruction into the set of clauses needed in the proof that SAT is NP-complete.

Intractable Problems

1. Convert the set of clauses $\{(v_1), (v_2,\overline{v_3}), (v_1,\overline{v_2},v_3)\}$ into proper 3-SAT format.

2. Convert the set of clauses $\{(v_1,\overline{v_2}), (v_1, v_2 , v_3 , v_4)\}$ into proper 3-SAT format.

3. How much time does it take a nondeterministic Turing machine to recognize a member of 3-SAT? Assume that there are n clauses.

4. Show precisely that 0-1INT is in NP.

5. What time bound is involved in recognizing cliques in graphs? Before solving this, define the data structure used as input. Might different data structures (adjacency matrix or edge list) make a difference in complexity?

6. Verify that the transformation from 3-SAT to CLIQUE can be done in polynomial time.

7. Prove that the vertex cover problem is NP-complete.

8. Show how to construct in polynomial time the graph used in the proof that COLOR is NP-complete

9. Suppose you wanted to entertain n people. Unfortunately some of them do not get along with each other. What is the complexity of deciding how many parties you must have to entertain all of the people on your list?

AUTOMATA

Thus far we have been concerned with two major topics: the discovery of an appropriate model for computation and an examination of the intrinsic properties of computation in general. We found that since Turing machines are equivalent to prograns, they form an appropriate model for computation. But we also discovered that in some sense they possessed far too much computational power. Because of this we ran into great difficulty when we tried to ask questions about them, and about computation in general. Whenever we wished to know something nontrivial, unsolvability sprang forth. And even in the solvable or recursive realm, we found intractability.

Now we shall attempt to overcome this lack of information about computation by restricting the power or our computational model. We hope that this will force some of the decision problems in which we are interested into the zone of solvability.

The sections include:

Finite Automata

Let us begin by removing almost all of the Turing machine's power! Maybe then we shall have solvable decision problems and still be able to accomplish *some* computational tasks. Also, we might be able to gain insight into the nature of computation by examining what computational losses we incur with this loss of power.

If we do not allow writing or two-way operation of the tape head, we have what has been traditionally called a *finite automaton.* This machine is only allowed to read its input tape and then, on the basis of what it has read and processed, accept or reject the input. This restricted machine operates by:

a) Reading a symbol,
b) Transferring to a new instruction, and
c) Advancing the tape head one square to the right.

When it arrives at the end of its input it then accepts or rejects depending upon what instruction is being executed.

This sounds very simple. It is merely a one-way, semi-literate Turing machine that just decides membership problems for a living! Let us examine one. In order to depict one, all we need to do is jot down Turing machine instructions in one large table, leave out the write part (that was not a pun!), and add a note which indicates whether the machine should accept. Here is an example:

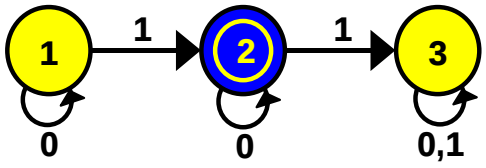
Instruction	*Read*	*Goto*	*Accept?*
1	0 1	same next	no
2	0 1	same next	yes
3	0 1	same same	no

Look closely at this machine. It stays on instruction one until it reads a one. Then it goes to instruction two and accepts any input unless another one arrives (the symbol 1 - not another input). If two or more ones appear in the input, then it ends up executing instruction three and does not accept when the input is over. And if no ones appear in the input the machine remains on instruction one and does not accept. So, this machine accepts only inputs that contain *exactly one 1.*

(**N.B.** Endmarkers are not needed since the machine just moves to the right. Accepting happens when the machine finishes the input while executing an instruction that calls for acceptance. Try this machine out on the inputs 000, 0100, 1000100, etc.)

Traditionally these machines have not had instructions but *states.* (Recall A. M. Turing's *states of mind*.) Another way to represent this same machine is to put the *next instruction* or *next state* or *goto* portion under the input in a table like that in figure 1.

There is another traditional method to describe finite automata which is extremely intuitive. It is a picture called a *state graph.* The states of the finite automaton appear as vertices of the graph while the transitions from state to state under inputs are the graph edges. The state graph for the same machine also appears in figure 1.



State	Input		Accept?
	0	1	
1	1	2	no
2	2	3	yes
3	3	3	no

Figure 1 - Finite Automaton Representations

(Note that the two circles that surround state two mean acceptance.)

Before continuing let's examine the computation of a finite automaton. Our first example begins in state one and reads the input symbols in turn changing states as necessary. Thus a computation can be characterized by a sequence of states. (Recall that Turing machine configurations needed the state plus the tape content. Since a finite automaton never writes, we always know what is on the tape and need only look at a state as a configuration.) Here is the sequence for the input 0001001.

Input Read:	**0**		**0**		**0**		**1**		**0**		**0**		**1**		
States:	**1**	→	**1**	→	**1**	→	**1**	→	**2**	→	**2**	→	**2**	→	**3**

Our next example is an elevator controller. Let's imagine an elevator that serves two floors. Inputs are calls to a floor either from inside the elevator or from the floor itself. This makes three distinct inputs possible, namely:

0 - no calls
1 - call to floor one
2 - call to floor two

The elevator itself can be going up, going down, or halted at a floor. If it is on a floor it could be waiting for a call or about to go to the other floor. This provides us with the six states shown in figure 2 along with the state graph for the elevator controller.

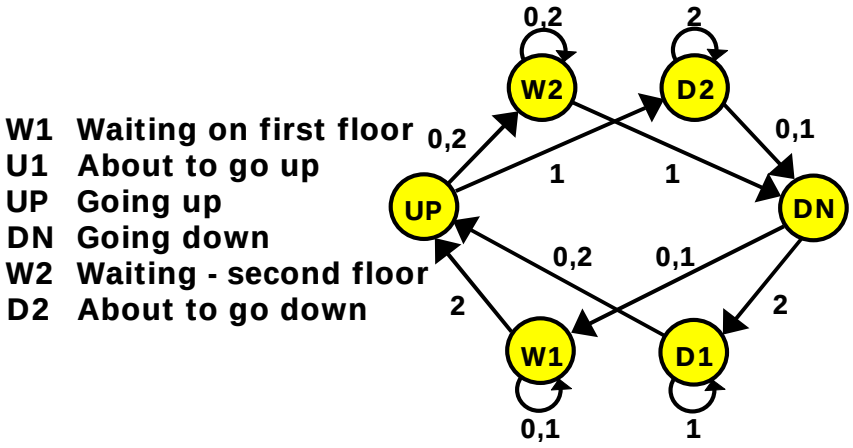


Figure 2 - Elevator Control

A state table for the elevator is provided below as table 1.

State	*Input*		
	none	call to 1	call to 2
W1 (wait on 1)	W1	W1	UP
U1 (start up)	UP	U1	UP
UP	W2	D2	W2
DN	W1	W1	U1
W2 (wait on 2)	W2	DN	W2
D2 (start down)	DN	DN	D2

Table 1 - Elevator Control

Accepting and rejecting states are not included in the elevator design because acceptance is not an issue. If we were to design a more sophisticated elevator, it might have states that indicated:

a) power failure,
b) overloading, or
c) breakdown

In this case acceptance and rejection might make sense.

Let us make a few small notes about the design. If the elevator is about to move (i.e. in state U1 or D2) and it is called to the floor it is presently on it will stay. (This may be good - try it next time you are in an elevator.) And if it is moving (up or down) and gets called back the other way, it remembers the call by going to the U1 or D2 state upon arrival on the next floor. Of course the elevator does not do things like open and close doors (these could be states too) since that would have added complexity to the design. Speaking of complexity, imagine having 100 floors.

That is our levity for this section. Now that we know what a finite automaton is, we must (as usual) define it precisely.

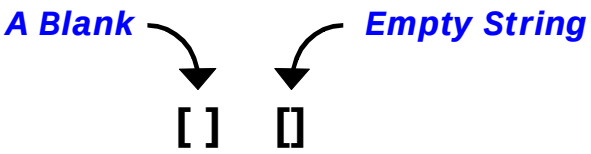
Definition. *A **finite automaton** M is a quintuple $M = (S,I,\delta,s_0,F)$ where:*
S is a finite set (of states)
I is a finite alphabet (of input symbols)
δ: $S \times I \rightarrow S$ (next state function)
$s_0 \in S$ (the starting state)
$F \subseteq S$ (the accepting states).

We also need some additional notation. The next state function is called the *transition function* and the accepting states are often called *final states.* The entire machine is usually defined by presenting a state table or a state graph. In this way, the states, alphabet, transition function, and final states are constructively defined. The starting state is usually the lowest numbered state. Our first example of a finite automaton is:

$$M = (\{s_1 , s_2 , s_3\}, \{0,1\}, \delta, s_1, \{s_2\})$$

where the transition function δ, is defined explicitly by either a state table or a state graph.

At this point we must make a slight detour and examine a very important yet seemingly insignificant input string called the *empty* string. It is a string without any symbols in it and is denoted as ε. It is *not* a string of blanks. An example might make this clear. Look between the brackets in the picture below.



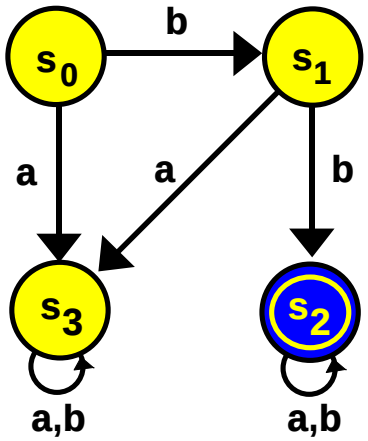
Let's look again at a computation by our first finite automaton. For the input 010, our machine begins in s_1, reads a 0 and goes to $\delta(s_1,0) = s_1$, then reads a 1 and goes to $\delta(s_1,1) = s_2$, and ends up in $\delta(s_2,0) = s_2$ after reading the final 0. All of that can be put together as:

$$\delta(\delta(\delta(s_1,0),1),0) = s_2$$

We call this transition on strings δ^* and define it as follows.

> **Definition.** *Let $M = (S,I,\delta,s_0,F)$. For any input string x, input symbol a, and state s_i, the* ***transition function on strings*** *δ^* takes the values:*
>
> $\delta^*(s_i,*e) = s_i$
>
> $\delta^*(s_i,a) = \delta(s_i,a)$
>
> $\delta^*(s_i,xa) = \delta(\delta^*(s_i,x),a).$

That certainly was terse. But, δ^* is really just what one expects it to be. It merely applies the transition function to the symbols in the string. Let's look at this for the example in figure 3.



This machine has a set of states = $\{s_0, s_1, s_2, s_3\}$ and operates over the input alphabet {a, b}. Its starting state is s_0 and its set of final or accepting states, $F = \{s_2\}$. The transition function is fully described twice in figure 3; once in figure 3a as a state table and once in figure 3b as a state graph.

State	Input		Accept?
	a	b	
0	3	1	no
1	3	2	no
2	2	2	yes
3	3	3	no

Figure 3 - Finite Automaton

If the machine receives the input bbaa it goes through the sequence of states:

$$s_0, s_1, s_2, s_2, s_2$$

while when it gets an input such as abab it goes through the state transition:

$$s_0, s_3, s_3, s_3, s_3$$

Now we shall become a bit more abstract. When a finite automaton receives an input string such as:

$$x = x_1 x_2 \ldots x_n$$

where the x_i are symbols from its input alphabet, it progresses through the sequence:

$$s_{k_1}, s_{k_2}, \ldots, s_{k_{n+1}}$$

where the states in the sequence are defined as:

$$\begin{aligned} s_{k_1} &= s_0 \\ s_{k_2} &= \delta(s_{k_1}, x_1) = \delta(s_0, x_1) \\ s_{k_3} &= \delta(s_{k_2}, x_2) = \delta^*(s_0, x_1 x_2) \\ &\vdots \\ s_{k_{n+1}} &= \delta(s_{k_n}, x_n) = \delta^*(s_0, x_1 x_2 \ldots x_n) \end{aligned}$$

Getting back to a more intuitive reality, the following table provides an assignment of values to the symbols used above for an input of bbaba to the finite automaton of figure 3.

i	**1**	**2**	**3**	**4**	**5**	**6**
x_i	**b**	**b**	**a**	**b**	**a**	
s_{k_i}	**s_0**	**s_1**	**s_2**	**s_2**	**s_2**	**s_2**

We have mentioned acceptance and rejection but have not talked too much about it. This can be made precise also.

> **Definition.** *The **set** (of strings) **accepted** by the finite automaton* $M = (S,I,\delta,s_0,F)$ *is:* $T(M) = \{ x \mid \delta^*(s_0,x) \in F \}$

This set of accepted strings (named T(M) to mean T*apes of M*) is merely all of the strings for which M ended up in a final or accepting state after processing the string. For our first example (figure 1) this was all strings of 0's and 1's that contain *exactly* one 1. Our last example (figure 3.1.3) accepted the set of strings over the alphabet {a, b} which began with *exactly* two b's.

Closure Properties

Removing power from Turing machines provided us with a new machine. We also found a new class of sets. Now it is time to examine this class. Our first questions concern operations on sets within the class.

Set *complement* is our first operation to examine. Since we are dealing with strings (rather than numbers), we must redefine this operation. Thus the definition of complement will slightly different. This is to be expected because even though 0100 and 100 are the same number, they are different strings. Here is the definition. *If a set contains strings over an alphabet, then its complement contains all of the strings (over the alphabet) not in the set.*

> **Theorem 1.** *If a set is accepted by a finite automaton then its complement can be accepted by a finite automaton.*

> **Proof**. Let the finite automaton $M = (S, I, \delta, s_0, F)$ accept the set T(M). We must now show that there is a finite automaton which accepts the set of strings over the alphabet I which M does not accept. Our strategy will be to look at the operation of M, accepting when it rejects and rejecting when M accepts. Since we know that strings which take M to a state of F are accepted by M and those which take M into S-F are rejected, then our course is fairly clear. Consider the machine:
>
> $$M' = (S, I, \delta, s_0, S\text{-}F).$$
>
> It is exactly the same as M except for its final or accepting states. Thus it should accept when M rejects. When we precisely examine this, we find that for all strings x over the alphabet I:
>
> $$\begin{aligned} x \in T(M) &\text{ if and only if } \delta^*(s_0,x) \in F \\ &\text{if and only if } \delta^*(s_0,x) \notin S\text{-}F \\ &\text{if and only if } x \notin T(M') \end{aligned}$$
>
> and so $T(M') = \overline{T(M)}$.

The last proof contained an example of our first method of dealing with finite automata: *rearranging an existing machine.* Now we shall employ another strategy: *combining two machines.* This is actually going to be parallel

processing. Suppose we take the two machines whose state graphs are in figure 1.

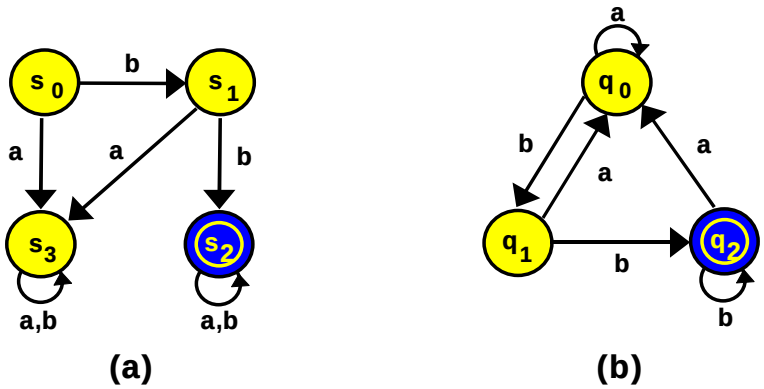
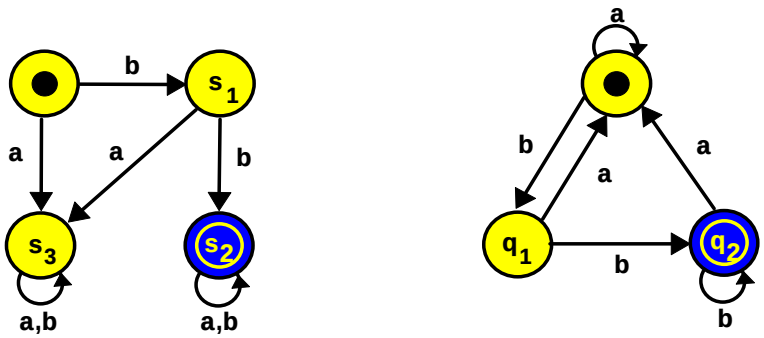


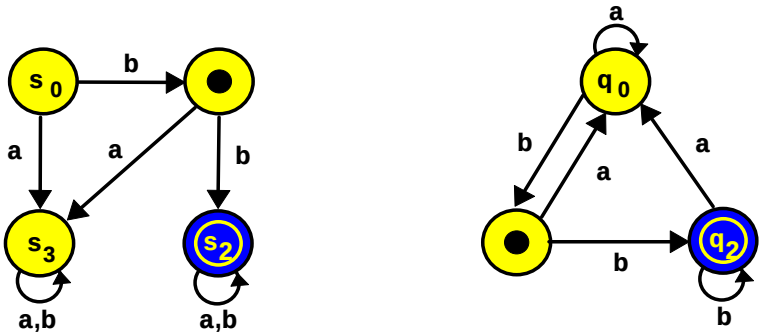
Figure 1 - Finite Automaton Examples

We have seen the machine (M_1) of figure 1a before, it accepts all strings (over {a, b}) which begin with two b's. The other machine (M_2) accepts strings which end with two b's. Let's try to combine them into one machine which accepts strings which either begin or end with two b's.

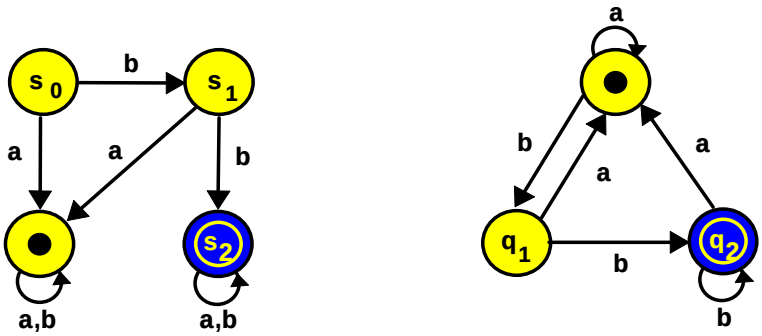
Why not run both machines at the same time on an input? We could keep track of what state each machine is in by placing pebbles upon the appropriate states and then advancing them according to the transition functions of each machine. Both machines begin in their starting states, as pictured in the state graphs below



with pebbles on s_0 and q_0. If both machines now read the symbol b on their input tapes, they move the pebbles to new states and assume configurations like these



with pebbles on s_1 and q_1. The pebbles have advanced according to the transition functions of the machines. Now let's have them both read an a. At this point, they both advance their pebbles to the next state and enter the configurations



with pebbles on s_3 and q_0.

With this picture in mind, let's trace the computations of both machines as they process several input strings. Pay particular attention to the *pairs* of states the machines go through. Our first string is bbabb, which will be accepted by both machines.

Input:	**b**	**b**	**a**	**b**	**b**	
M_1's states	**s_0**	**s_1**	**s_2**	**s_2**	**s_2**	**s_2**
M_2's states	**q_0**	**q_1**	**q_2**	**q_0**	**q_1**	**q_2**

Now let us look at an input string neither will accept: babab.

Input:	**b**	**a**	**b**	**a**	**b**	
M_1's states	**s_0**	**s_1**	**s_3**	**s_3**	**s_3**	**s_3**
M_2's states	**q_0**	**q_1**	**q_0**	**q_1**	**q_0**	**q_1**

And finally, the string baabb which will be accepted by M_2 but not M_1.

Input:	b	a	a	b	b	
M_1's states	s_0	s_1	s_3	s_3	s_3	s_3
M_2's states	q_0	q_1	q_0	q_0	q_1	q_2

If we imagine a *multiprocessing* finite automaton with two processors (one for M_1 and one for M_2), it would probably look just like the pictures above. Its *state* could be a state pair (one from each machine) corresponding to the pebble positions. Then, if a pebble ended up on an accepting state for either machine (that is, either s_2 or q_2), our multiprocessing finite automaton would accept.

This is not difficult at all! All we need to do is to define a new class of machines and we can accept several things at once. (Note that the new machines accept unions of sets accepted by finite automata.) Or, if we think for a bit, we find that we do not need to define a new class for this. The next result shows that we already have this facility with finite automata. The proof of the theorem demonstrates this by careful manipulation of the symbolic definition of finite automata.

Theorem 2. *The class of sets accepted by finite automata is closed under union.*

Proof Sketch. Let $M_1 = (S, I, \delta, s_0, F)$ and $M_2 = (Q, I, \gamma, q_0, G)$ be two arbitrary finite automata. To prove the theorem we must show that there is another machine (M_3) which accepts every string accepted by M_1 or M_2.

We shall try the multiprocessing pebble machine concept and work it into the definition of finite automata. Thus the states of M_3 are pairs of states (one from M_1 and one from M_2). This works out nicely since the set of pairs of states from S and Q is known as the *cross product* (written $S \times Q$) between S and Q. The starting state is obviously $<s_0 , q_0>$. Thus:

$$M_3 = (S \times Q, I, \xi, <s_0, q_0>, H)$$

where ξ and H will be described presently.

The transition function ξ is just a combination of δ and γ, since it simulates the advance of pebbles on the state graphs. It uses δ to change the states in S and γ, to change the states in Q. In other words, if a is a symbol of I:

$$\xi(<s_i , q_i >, a) = <\delta(s_i, a), \gamma(q_i, a)>.$$

Our final states are all of the pairs which contain a state from F or a state from G. In cross product notation this is:

$$H = (F \times Q) \cup (S \times G).$$

We have now defined M_3 and know that it is indeed a finite automaton because it satisfies the definition of finite automata. We claim it does accept $T(M_1) \cup T(M_2)$ since it mimics the operation of our intuitive multiprocessing pebble machine. The remainder of the formal proof (which we shall leave as an exercise) is merely an induction on the length of input strings to show that for all strings x over the alphabet I:

$$x \in T(M_1) \cup T(M_2) \text{ iff } \delta^*(s_0,x) \in F \text{ or } \gamma^*(q_0,x) \in G$$
$$\text{iff } \xi^*(<s_0 , q_0 >,x) \in H.$$

Thus by construction we have shown that the class of sets accepted by finite automata is closed under union.

By manipulating the notation we have shown that two finite automata can be combined. Let's take a look at the result of applying this construction to the machines of figure 1. (This is pictured in figure 2.)

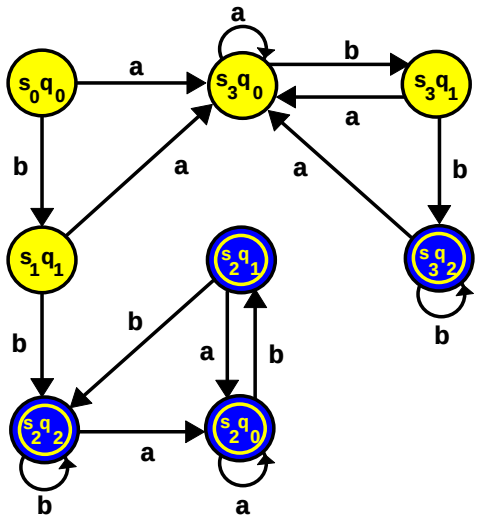


Figure 2 - Union of M_1 and M_2

Note that not all pairs of states are included in the state graph. (For example, $<s_0, q_1>$ and $<s_1, q_2>$ are missing.) This is because it is impossible to get to these states from $<s_0 , q_0>$.

This is indeed a complicated machine! But, if we are a bit clever, we might notice that if the machine enters a state pair containing s_2, then it remains in pairs containing s_2. Thus we can combine all of these pairs and get the smaller but equivalent machine of figure 3.

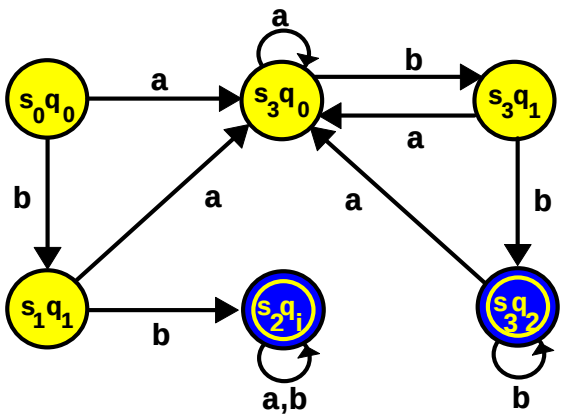


Figure 3 - Reduced Union Machine

This is very likely what anyone would design if they had to build a machine from scratch rather than use the construction detailed in the theorem. Lest anyone think that finding smaller, equivalent machines is always this simple, we must admit that this was an elementary example and we did some prior planning to provide such clean results.

The final Boolean operation is set intersection. We state this here and leave the proof as an exercise in Boolean algebra identities (De Morgan's Laws) or machine construction - as you wish!

> **Theorem 3.** *The class of sets accepted by finite automata is closed under intersection.*

Nondeterministic Operation

So far, every step taken by a finite automaton has been *exactly* determined by the state of the machine and the symbol read. No choices have existed. This mode of operation is called *determinism* and machines of this ilk are known as *deterministic finite automata*. Finite automata need not be so unambiguous. We could have defined them to have some choices of which state to advance to on inputs. Examine figure 1, it provides the state graph of such a machine.

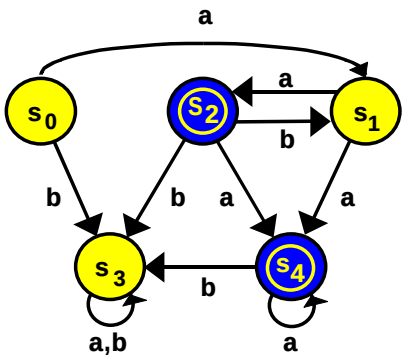


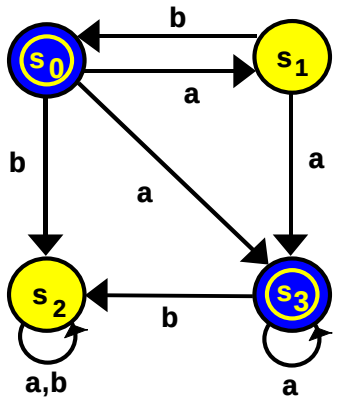
Figure 1 - Nondeterministic Finite Automaton

If the input to this machine begins with the string ab, it will progress from s_0 to s_1 to s_2. Now, if the next input symbol is an a, it has a choice of whether to go back to s_1 or to move to s_4. So, now what does it do? Well, it transfers to the state which will eventually take it into a final state if possible. Simple as that! But, how does it know when it has not seen the remainder of the input yet? Do not worry about that - it always chooses the right state!

The above explanation of the operation of the machine of figure 1 *is* a slight bit mystical. But we can (if we wish) define machines with choices in their transition functions. And we can (if we wish) define acceptance to take place whenever a correct sequence of states under the input will end up in a final state. In other words, *if the machine can get to a final state* in some proper manner, it will accept. Now for a formal definition.

> **Definition.** *A **nondeterministic** finite automaton is the quintuple* $M = (S, I, \delta, S_0, F)$ *where S, I, and F are as before but:*
>
> $S_0 \in S$ *(a set of starting states), and*
> $\delta(s,a) \subseteq S$ *for each* $s \in S$ *and* $a \in I$ *.*

Now instead of having a starting *state* and a *transition function*, we have a starting *state set* and a *set of transition states*. More on this later. For now, note that the only differences in the finite automaton definitions was that the machine now has a choice of states to start in and a choice of transitions under a state-symbol pair. A reduced version of the last nondeterministic machine is presented in figure 2.



State	Input a	Input b	Accept?
0	1,3	2	yes
1	3	0	no
2	2	2	no
3	3	2	yes

Figure 2 - Reduced Nondeterministic Machine

(**N.B.** We must note that the transition indicator δ is not a function any more. To be precise about this we must state that it has become a *relation*. In other words, since δ(s, a) is a *set*, it indicates which states are members of δ(s, a). If that is confusing then forget it and consider δ(s, a) to be a *set*.) Our next step is to define acceptance for nondeterministic finite automata. We could extend the transition indicator so that it will handle strings. Since it provides a set of next states, we will just note the set of states the machine is in after processing a string. Let's look at a picture. Think about the last machine (the one in figure 2). Now imagine what states it might go through if it processed all possible strings of length three. Now, look at figure 3.

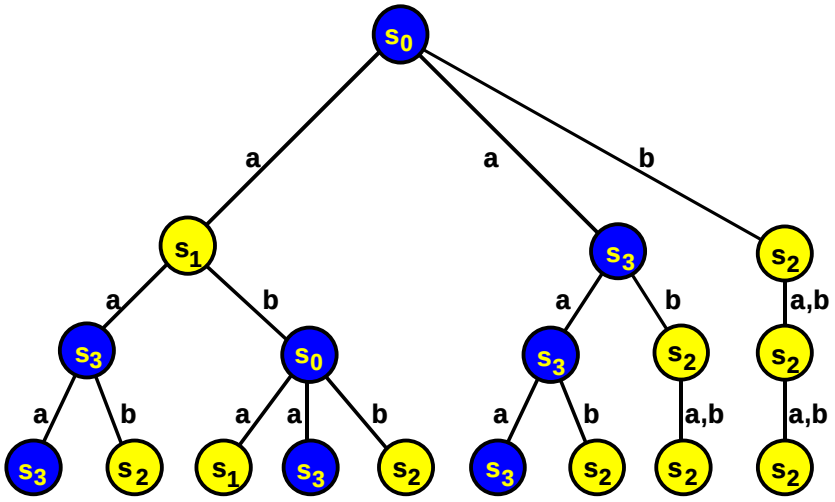


Figure 3- Computation Tree

In processing the string abb, the machine ends up in s_2, but it can get there in two different ways. These are:

$$s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_2$$
$$s_0 \rightarrow s_3 \rightarrow s_2 \rightarrow s_2$$

Likewise, the machine can end up in 3 after processing the string aaa. But since the automaton is nondeterministic, it can have the option of ending up in several states after processing an input. For example, after aba, the set of states $\{s_1 , s_2 , s_3\}$ is reached.

In fact, a *set* of states is reached by the machine after processing an input. This set depends upon the choices the automaton was offered along the way. And, if a final state was in the set reached by the machine, we accept. In the above example only the strings aba and aaa can be accepted because they were the only strings which took the automaton into s_3.

This gives a definition of δ^* as the *set of states reached by the automaton* and the tapes accepted as:

$$T(M) = \{ x \mid \delta^*(s_0,x) \cap F \neq \varnothing \}$$

On the other hand, instead of extending δ to strings as with the deterministic case, we shall discuss sequences of state transitions under the state transition indicator δ. The following formal definition of acceptance merely states that a string is accepted by a nondeterministic finite automaton if there is a sequence of states (or path through the computation tree) which leads from a starting

state to a final state under the input string. (Note that we do not discuss how a machine might find this sequence - that does not matter! We just say that the input is accepted if there exists such a sequence of states.)

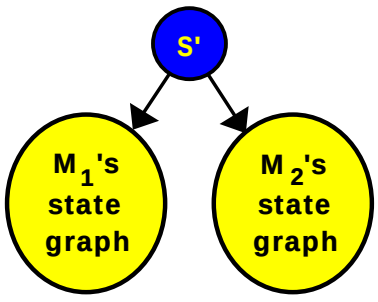
> **Definition.** *The input* $x = x_1 \ldots x_n$ *is* ***accepted*** *by the nondeterministic finite automaton* $M = (S, I, \delta, S_0, F)$ *if and only if there is a sequence of states:* $s_{k_1}, s_{k_2}, \ldots, s_{k_{n+1}}$ *where:*
>
> *a)* $s_{k_1} \in S_0$
>
> *b) for each* $i \leq n$: $s_{k_{i+1}} \in \delta(s_{k_i}, x_i)$
>
> *c)* $s_{k_{n+1}} \in F$.

With this definition of acceptance in mind, we define the set of *tapes accepted* by a nondeterministic finite automaton (denoted T(M) as before) as exactly those inputs for which there is a sequence of states leading from a starting state to an accepting state.

Nondeterminism is useful because it allows us to define very simple machines which perform certain operations. As an example, let's revisit the union closure problem. As usual, suppose we have the two finite automata $M_1 = (S, I, \delta, s_0, F)$ and $M_2 = (Q, I, \gamma, q_0, G)$ and wish to build a machine which accepts the union of the sets they accept. Our new union machine contains the states of both M_1 and M_2 plus a new starting state. This new starting state leads into the states of either M_1 or M_2 in a nondeterministic manner. That is, under the first input symbol, we would advance to an appropriate state in M_1 *or* an appropriate state in M_2. Formally, if $I = \{0,1\}$, the transition indicator of the union machine is ξ, and its starting state s' then:

$$\xi(s', 0) = \{\delta(s_0, 0), \gamma(q_0, 0)\}$$
$$\xi(s', 1) = \{\delta(s_0, 1), \gamma(q_0, 1)\}$$

The rest of the transition relation ξ is just a combination of δ and γ, and the state graph for the new union machine might resemble:



Thus the union machine is $M = (S \cup Q \cup \{s'\}, I, \xi, s', F \cup G)$. Acceptance takes place whenever there is a sequence to an accepting state through either the state graph of M_1 or that of M_2.

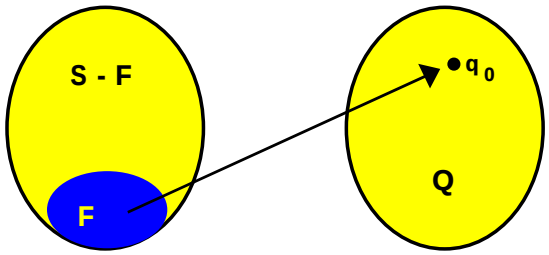
Let's try another closure property and build a nondeterministic machine which realizes it. This will be a string property called *concatenation* or juxtaposition. For strings this is easy, the concatenation of x and y is xy. If we have the sets A and B then the concatenation of them is defined as:

$$AB = \{ xy \mid x \in A \text{ and } y \in B \}.$$

If A and B can be accepted by the deterministic finite automata $M_1 = (S, I, \delta, s_0, F)$ and $M_2 = (Q, I, \gamma, q_0, G)$ we need to try and construct a machine M_3 which will accept AB. Let us look at:

$$M_3 = (S \cup Q, I, \xi, \{s_0\}, G)$$

and define ξ so that M_3 accepts AB. Our strategy will be to start out processing as M_1 might and then when M_1 wishes to accept a part of the input, switch to M_2 and continue on. With any luck we will end up in G, the final state set of M_2. Nondeterminism will be used to make the change from M_1 to M_2. The transition relation ξ will operate just like δ on the state set S and like γ on the state set Q except for the final states $F \subseteq S$. There it will include a transition to the starting state q_0 of M_2. One might picture this as:



and define the transition relation precisely as:

$$\xi(s_i, a) = \{\delta(s_i, a)\} \text{ for } s_i \in S\text{-}F$$
$$\xi(s_i, a) = \{\delta(s_i, a), q_0\} \text{ for } s_i \in F$$
$$\xi(q_i, a) = \{\gamma(q_i, a)\} \text{ for } q_i \in Q$$

By the definition of acceptance for nondeterministic machines, M_3 will accept a string if and only if there is a sequence of states (under the direction of ξ) which leads from s_0 to a state in G. Suppose z is a member of AB. Then:

$$z \in AB \text{ iff } z = xy \text{ where } x \in A \text{ and } y \in B$$
$$\text{iff } x \in T(M_1) \text{ and } y \in T(M_2)$$
$$\text{iff } \delta^*(s_0,x) \in F \text{ and } \gamma^*(q_0,y) \in G$$

This means that there is a sequence of states

$$s_{k_1}, s_{k_2}, \ldots, s_{k_n}$$

in S from $s_0 = s_{k_1}$ to $s_{k_n} \in F$ under δ and x. Also, there is a sequence of states in Q

$$q_{k_1}, q_{k_2}, \ldots, q_{k_m}$$

from $q_0 = q_{k_1}$ to $q_{k_m} \in G$ under γ and y. Since ξ is defined to be just like δ on S and like γ on Q, these sequences of states in S and Q exist under the influence of ξ and x and ξ and y. We now note that instead of going to the last state s_{k_n} in the sequence in S, ξ could have directed transferred control to q_0. Thus there is a sequence:

$$s_0, s_{k_2}, \ldots, s_{k_{n-1}}, q_0, q_{k_2}, \ldots, q_{k_m}$$

under ξ and z = xy which proceeds from s_0 to G. We can now claim that

$$T(M_3) = AB = T(M_1)T(M_2)$$

noting that the only way to get from s_0 to a state in G via ξ is via a string in AB.

A final note on the machine M_3 which was just constructed to accept AB. If A contains the empty word then of course s_0 is a final state and thus q_0 also must be a member of the starting state set. Also, note that if B contains the empty word, then the final states of M_3 must be $F \cup G$ rather than merely G.

Now we shall move along and look at multiple concatenation. Suppose we concatenated the same set together several times. Putting this more formally and in superscript notation, let:

$$A^0 = \{\varepsilon\}$$
$$A^1 = A$$
$$A^2 = AA$$
$$A^3 = AA^2 = AAA$$

and so forth. The general scheme for this is:

$$A^0 = \{\varepsilon\}$$
$$A^{i+1} = A^iA$$

To sum everything up, we consider the union of this infinite sequence of concatenations and call it the *Kleene closure* of the set A. Here is the definition.

> **Definition.** *The **Kleene Closure** (written A^*) of the set A is the union of A^k over all integer values of k.*

This operator is known as the *Kleene Star Operator* and so A^* is pronounced *A star*. One special case of this operator's use needs to be mentioned. If A is the set {a,b} (in other words: an alphabet), then A^* is the set of all strings over the alphabet. This will be handy.

To accept the Kleene closure of a set accepted by a deterministic finite automaton a construction similar to that used above for concatenation works nicely. The strategy we shall use is to allow a reset to the starting state each time a final state is entered. (That is, start over whenever a string from the original set could have ended.) We shall present the construction below and leave the proof of correctness as an exercise.

For a deterministic finite automaton $M = (S, I, \delta, s_0, F)$ we must build a (nondeterministic) machine which accepts $[T(M)]^*$. As we mentioned, our strategy will be to allow the transition relation to reset to s_0 whenever a final state is reached. First we shall introduce a new starting state s' and the following transition relation γ for all a U I:

$$\gamma(s',a) = \{\delta(s_0,a)\}$$
$$\gamma(s_i,a) = \{\delta(s_i,a)\} \text{ for } s_i \in S\text{-}F$$
$$\gamma(s_i,a) = \{ s_0, \delta(s_i,a)\} \text{ for } s_i \in F$$

The machine which accepts $[T(M)]^*$ is $M' = (S\cup\{s'\},I,\gamma,\{s'\},F\cup\{s'\})$. Now that we have seen how useful nondeterministic operation can be in the design of finite automata, it is time to ask whether they have more power than their deterministic relatives.

Theorem 3. *The class of sets accepted by finite automata is exactly the same class as that accepted by nondeterministic finite automata.*

Proof Sketch. Two things need to be shown. First, since deterministic machines are also nondeterministic machines (which do not ever make choices) then the sets they accept are a subclass of those accepted by nondeterministic automata.

To show the other necessary relation we must demonstrate that every set accepted by a nondeterministic machine can be accepted in a deterministic manner. We shall bring forth again our pebble automaton model.

Let $M_n = (S, I, \delta, S_0, F)$ be an arbitrary nondeterministic finite automaton. Consider its state graph. Now, place a pebble on each state in S_0. Then, process an input string under the transition relation δ. As δ calls for transitions from state to state, move the pebbles accordingly. Under an input a, with a pebble on s_i, we pick up the pebble from s_i and place pebbles on every state in $\delta(S_i, a)$. This is done for all states which have pebbles upon them. Indeed, this is parallel processing with a vengeance! (Recall the computation tree of figure 3, we are just crawling over it with pebbles - or processors.) After the input has been processed, we accept if any of the final states have pebbles upon them.

Since we moved pebbles on all of the paths M_n could have taken from a starting state to a final state (and we did not miss any!), we should accept whenever M_n does. And, also, if we accept then there was indeed a path from a starting state to a final state. Intuitively it all seems to work.

But, can we define a deterministic machine which does the job? Let's try. First let us define some states which correspond to the pebbled states of M_n. Consider:

q_1 means a pebble upon s_0
q_2 means a pebble upon s_1
q_3 means pebbles upon s_0 and s_1
q_4 means a pebble upon s_2
q_5 means pebbles upon s_0 and s_2
⋮
$q_{2^{n+1}-1}$ means pebbles upon $s_0, s_1, \ldots, s_n$

This is our state set Q. The starting state is the q_k which means pebbles on all of S_0. Our set of final states (G) includes all q_i which have a pebble on a state of F. All that is left is to define the transition function γ and we have defined a deterministic finite automaton $M_d = (Q, I, \gamma, q_k, G)$ which should accept the same set as that accepted by M_n.

This is not difficult. Suppose q_i means pebbles upon all of the states in $s' \subseteq S$. Now we take the union of $\delta(s_k,a)$ over all $s_k \in s'$ and find the q_j which means pebbles on all of these states. Then we define $\gamma(q_i, a) = q_j$.

The remainder of the proof involves an argument that M_n and M_d accept the same set.

With this theorem and our previous constructions firmly in mind, we now state the following.

Corollary. *The class of sets accepted by finite automata is closed under concatenation and Kleene closure.*

A cautionary remark must be made at this time. It is rather wonderful to be able to use nondeterminism in our machine constructions. But, when these machines are converted to deterministic finite automata via the above construction, there is an *exponential* state explosion! A four state machine would have fifteen states when converted to a deterministic machine. Imagine what a 100 state machine would look like when converted! Fortunately most of these machines can be reduced to a more manageable size.

Regular Sets and Expressions

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (And, since the advent of VLSI systems sometimes finite automata *are* circuits!) Computer scientists adore them because they adapt very nicely to algorithm design, for example the lexical analysis portion of compiling and translation. Mathematicians are intrigued by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept. This is partially what this section is about.

We shall build expressions from the symbols 0, 1, +, and & using the operations of union, concatenation, and Kleene closure. Several intuitive examples of our notation are:

a) 01 means a zero followed by a one (concatenation)
b) 0+1 means either a zero or a one (union)
c) 0^* means ^ + 0 + 00 + 000 + ... (Kleene closure)

With parentheses we can build larger expressions. And we can associate meanings with our expressions. Here's how:

Expression	Set Represented
$(0+1)^*$	all strings over {0,1}.
$0^*10^*10^*$	strings containing exactly two ones.
$(0+1)^*11$	strings which end with two ones.

That is the intuitive approach to these new expressions or formulas.
Now for a precise, formal view. Several definitions should do the job.

Definition. *0, 1, ε, and $\varnothing$ are* ***regular expressions.***

Definition. *If α and β are regular expressions, then so are $(\alpha\beta)$, $(\alpha + \beta)$, and $(\alpha)^*$.*

OK, fine. Regular expressions are strings put together with zeros, ones, epsilons, stars, plusses, and matched parentheses in certain ways. But why did we do it? And what do they mean? We shall answer this with a list of what various general regular expressions represent. First, let us define what some specific regular expressions represent.

a) 0 represents the set {0}
b) 1 represents the set {1}
c) ε represents the set {ε} (the empty string)
d) $\varnothing$ represents the empty set

Now for some general cases. If α and β are regular expressions representing the sets A and B, then:

a) $(\alpha\beta)$ represents the set AB
b) $(\alpha + \beta)$ represents the set $A \cup B$
c) $(\alpha)^*$ represents the set A^*

The sets which can be represented by regular expressions are called *regular sets.* When writing down regular expressions to represent regular sets we shall often drop parentheses around concatenations. Some examples are $11(0 + 1)^*$ (the set of strings beginning with two ones), 0^*1^* (all strings which contain a possibly empty sequence of zeros followed by a possibly null string of ones), and the examples mentioned earlier. We also should note that {0,1} is not the only alphabet for regular sets. Any finite alphabet may be used.

From our precise definitions of the regular expressions and the sets they represent we can derive the following nice characterization of the regular sets. Then, very quickly we shall relate them to finite automata.

Theorem 1. *The class of regular sets is the smallest class containing the sets {0}, {1}, {ε}, and $\varnothing$ which is closed under union, concatenation, and Kleene closure.*

See why the above characterization theorem is true? And why we left out the proof? Anyway, that is all rather neat but, what exactly does it have to do with finite automata?

Theorem 2. *Every regular set can be accepted by a finite automaton.*

Proof. The singleton sets {0}, {1}, {ε}, and $\varnothing$ can all be accepted by finite automata. The fact that the class of sets accepted by finite automata is closed under union, concatenation, and Kleene closure completes the proof.

Just from closure properties we know that we can build finite automata to accept all of the regular sets. And this is indeed done using the constructions

from the theorems. For example, to build a machine accepting $(a + b)a^*b$, we design:

M_a which accepts {a},
M_b which accepts {b},
M_{a+b} which accepts {a, b} (from M_a and M_b),
M_{a^*} which accepts a^*,
and so forth

until the desired machine has been built. This is easily done automatically, and is not too bad after the final machine is reduced. But it would be nice though to have some algorithm for converting regular expressions directly to automata. The following algorithm for this will be presented in intuitive terms in language reminiscent of language parsing and translation.

Initially, we shall take a regular expression and break it into subexpressions. For example, the regular expression $(aa + b)^*ab(bb)^*$ can be broken into the three subexpressions: $(aa + b)^*$, ab, and $(bb)^*$. (These can be broken down later on in the same manner if necessary.) Then we number the symbols in the expression so that we can distinguish between them later. Our three subexpressions now are: $(a_1a_2 + b_1)^*$, a_3b_2, and $(b_3b_4)^*$.

Symbols which lead an expression are important as are those which end the expression. We group these in sets named *FIRST* and *LAST*. These sets for our subexpressions are:

Expression	*FIRST*	*LAST*
$(a_1a_2 + b_1)^*$	a_1 , b_1	a_2 , b_1
a_3b_2	a_3	b_2
$(b_3b_4)^*$	b_3	b_4

Note that since the *FIRST* subexpression contained a union there were two symbols in its *FIRST* set. The *FIRST* set for the entire expression is: $\{a_1 , a_3 , b_1\}$. The reason that a_3 was in this set is that since the first subexpression was starred, it could be skipped and thus the first symbol of the next subexpression could be the first symbol for the entire expression. For similar reasons, the *LAST* set for the whole expression is $\{b_2 , b_4\}$.

Formal, precise rules do govern the construction of the *FIRST* and *LAST* sets. We know that *FIRST*(a) = {a} and that we always build *FIRST* and *LAST* sets from the bottom up. Here are the remaining rules for *FIRST* sets.

Definition. *If* α *and* β *are regular expressions then:*

a) $FIRST(\alpha + \beta) = FIRST(\alpha) \cup FIRST(\beta)$

b) $FIRST(\alpha^*) = FIRST(\alpha) \cup \{\varepsilon\}$

c) $FIRST(\alpha\beta) = \begin{cases} FIRST(\alpha) \text{ if } \varepsilon \notin FIRST(\alpha) \\ FIRST(\alpha) \cup FIRST(\beta) \text{ otherwise} \end{cases}$

Examining these rules with care reveals that the above chart was not quite what the rules call for since empty strings were omitted. The correct, complete chart is:

Expression	*FIRST*	*LAST*
$(a_1a_2 + b_1)^*$	a_1, b_1, ε	a_2, b_1, ε
a_3b_2	a_3	b_2
$(b_3b_4)^*$	b_3, ε	b_4, ε

Rules for the *LAST* sets are much the same in spirit and their formulation will be left as an exercise.

One more notion is needed, the set of symbols which might follow each symbol in any strings generated from the expression. We shall first provide an example and explain in a moment.

Symbol	a_1	a_2	a_3	b_1	b_2	b_3	b_4
FOLLOW	a_2	a_1, a_3, b	b_2	a_1, a_3, b_1	b_3	b_4	b_3

Now, how did we do this? It is almost obvious if given a little thought. The *FOLLOW* set for a symbol is all of the symbols which could come next. The algorithm goes as follows. To find *FOLLOW*(a), we keep breaking the expression into subexpressions until the symbol a is in the *LAST* set of a subexpression. Then *FOLLOW*(a) is the *FIRST* set of the next subexpression. Here is an example. Suppose that we have $\alpha\beta$ as our expression and know that $a \in LAST(\alpha)$. Then *FOLLOW*(a) = *FIRST*(β). In most cases, this is the way it we compute *FOLLOW* sets.

But, there are three exceptions that must be noted.

1) If an expression of the form $a\gamma^*$ is in α then we must also include the *FIRST* set of this starred subexpression γ.

2) If α is of the form β^* then *FOLLOW*(a) also contains α's *FIRST* set.

3) If the subexpression to the right of α has an ε in its *FIRST* set, then we keep on to the right unioning *FIRST* sets until we no longer find an ε in one.

Another example. Let's find the *FOLLOW* set for b_1 in the regular expression $(a_1 + b_1a_2^*)^*b_2^*(a_3 + b_3)$. First we break it down into subexpressions until b_1 is in a *LAST* set. These are:

$$(a_1 + b_1\, a_2{}^*)^* \qquad b_2^* \qquad (a_3 + b_3)$$

Their *FIRST* and *LAST* sets are:

Expression	FIRST	LAST
$(a_1 + b_1a_2^*)^*$	a_1, b_1, ε	$a_1, b_1, a_2, \varepsilon$
b_2^*	b_2, ε	b_2, ε
$(a_3 + b_3)$	a_3, b_3	a_3, b_3

Since b_1 is in the *LAST* set of asubexpression which is starred then we place that subexpression's *FIRST* set $\{a_1 , b_1\}$ into *FOLLOW*(b_1). Since a_2^* came after b_1 and was starred we must include a_2 also. We also place the *FIRST* set of the next subexpression (b_2^*) in the *FOLLOW* set. Since that set contained an ε , we must put the next *FIRST* set in also. Thus in this example, all of the *FIRST* sets are combined and we have:

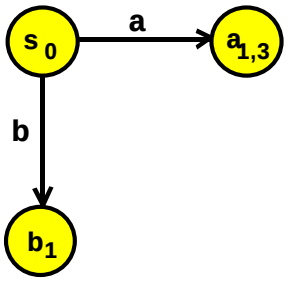
$$FOLLOW(b_1) = \{a_1 , b_1 , a_2 , b_2 , a_3 , b_3\}$$

Several other *FOLLOW* sets are:

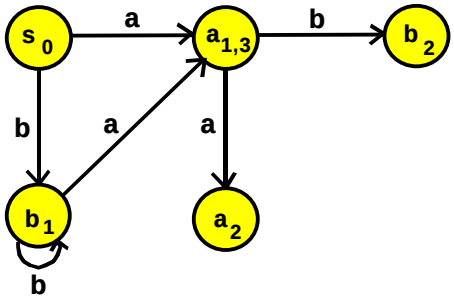
$$FOLLOW(a_1) = \{a_1 , b_1 , b_2 , a_3 , b_3\}$$
$$FOLLOW(b_2) = \{b_2 , a_3 , b_3\}$$

After computing all of these sets it is not hard to set up a finite automaton for any regular expression. Begin with a state named s_0. Connect it to states

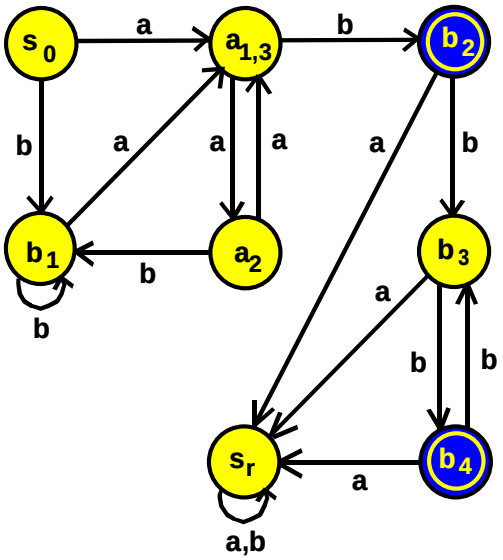
denoting the *FIRST* sets of the expression. (By *sets* we mean: split the *FIRST* set into two parts, one for each type of symbol.) Our first example $(a_1a_2 + b_1)^*a_3b_2(b_3b_4)^*$ provides:



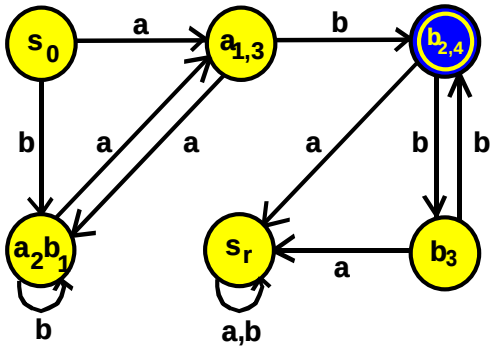
Next, connect the states just generated to states denoting the *FOLLOW* sets of all their symbols. Again, we have:



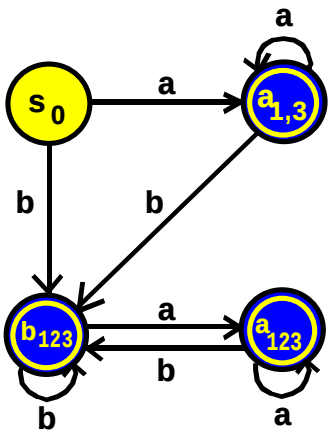
Continue on until everything is connected. Any edges missing at this point should be connected to a rejecting state named s_r. The states containing symbols in the expression's *LAST* set are the accepting states. The complete construction for our example $(aa + b)^*ab(bb)^*$ is:



This construction did indeed produce an equivalent finite automaton, and in not too inefficient a manner. Though if we note that b_2 and b_4 are basically the same, and that b_1 and a_2 are similar, we can easily streamline the automaton to:



Our construction method provides:



for our final example. There is a very simple equivalent machine.
Try to find it!

We now close this section with the equivalence theorem concerning finite automata and regular sets. Half of it was proven earlier in the section, but the translation of finite automata into regular expressions remains. This is not included for two reasons. First, that it is *very* tedious, and secondly that nobody ever actually does that translation for any practical reason! (It is an interesting demonstration of a correctness proof which involves several levels of iteration and should be looked up by the interested reader.)

> **Theorem 3.** *The regular sets are exactly those sets accepted by finite automata.*

Decision Problems for Finite Automata

Now we wish to examine decision problems for the sets accepted by finite automata (or the regular sets). When we tried to decide things concerning the r.e. sets we were disappointed because everything nontrivial seemed to be unsolvable. Our hope in defining finite automata as a much weaker version of Turing machines was to gain solvability at the expense of computational power. Let us see if we have succeeded. Our first result indicates that we have.

Theorem 1. *Membership is solvable for the regular sets.*

Proof. This is very easy indeed. With a universal Turing machine we can simulate any finite automaton. To decide whether it accepts an input we need just watch it for a number of steps equal to the length of that input.

So far, so good. In order to decide things a bit more intricate than membership though, we need a very useful technical lemma which seems very strange indeed until we begin to use it. It is one of the most important results in finite automata theory and it provides us with a handle on the finiteness of finite automata. It tells us that only the information stored in the states of a finite automaton is available during computation. (This seems obvious, but the proof of the following lemma points this out in a very powerful manner.)

Lemma (Pumping). *Let $M = (S, I, \delta, s_0, F)$ be a finite automaton. Then for any string x accepted by M whose length is no less than the size of S, there are strings u, v, and w (over the alphabet I) such that:*

a) $x = uvw$,
b) v is not the empty string, and
c) for all $k \geq 0$, $uv^k w \in T(M)$.

Proof. Let $x = x_1 \ldots x_n$. Suppose that x is accepted by the finite automaton M and has length n where n is not smaller than the size of S, the state set of M. Then as M processes x, it goes through the sequence of states:

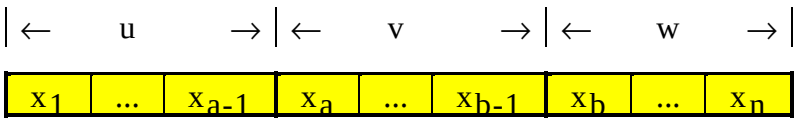
$$s_{j_1}, \ldots, s_{j_{n+1}}$$

where M is in state s_{j_i} as it reads x_i, and:

a) $s_{j_1} = s_0$,
b) for each $i \leq n$, $\delta(s_{j_i}, x_i) = s_{j_{i+1}}$, and
c) $s_{j_{n+1}} \in F$.

Since M has no more than n states (our initial assumption about the length of x not being less than the number of M's states), at least one of the states in the sequence must be repeated because there are n+1 states in the sequence. We shall assume that this repeat occurs at s_{j_a} and s_{j_b} where $a < b$.

Now let's consider the string $x_a \ldots x_{b-1}$, the portion of the input which is processed by M as it goes through the sequence of states $s_{j_a}, \ldots, s_{j_b}$. We shall say that v is this substring of x and note that since $a < b$, $v \neq \varepsilon$. Now we shall assign the remaining characters of x to u and w. The following picture illustrates this.



It is obvious that uvw = x. And, when M processes x:

a) $\delta^*(s_{j_1}, u) = \delta^*(s_0, u) = s_{j_a}$ [since $s_{j_1} = s_0$]
b) $\delta^*(s_{j_a}, v) = s_{j_b} = s_{j_a}$ [since $s_{j_b} = s_{j_a}$]
c) $\delta^*(s_0, uv) = s_{j_b} = s_{j_a}$ [same reason]
d) $\delta^*(s_{j_b}, w) = \delta^*(s_{j_a}, w) = s_{j_{n+1}} \in F$ [same again]

In other words, M enters and leaves the substring v in *the same state*. Now we shall examine exactly what this means. If we were to omit v and process uw, M would leave u in $s_{j_a} = s_{j_b}$ and finish w in $s_{j_{n+1}}$ just as before. Thus uw is in T(M). If we were to make M process uvvw then M would leave uv in $s_{j_b} = s_{j_a}$, leave uvv in s_{j_b} and finish w in the same state as before. Thus $uvvw \in T(M)$. In fact, no matter how many times we add another v between u and w, M always leaves and enters each v in s_{j_a} and therfore finishes the entire input in the same final state. Thus for any $k \geq 0$, $uv^k w \in T(M)$.

If we go back and examine our proof of the pumping lemma, we find that we can prove something a little more powerful. In fact, something that will come in handy in the future. Something which will make our lives much more pleasing. Here it is.

> **Corollary.** *The substring v of x can be forced to reside in any portion of x which is at least as long as the number of states of the automaton.*
>
> **Proof.** Just note that in any substring of x which is no shorter than the number of states, we can find a repeated state while processing. This provides a v and the proof proceeds as before.

This technical lemma (referred to as the *pumping lemma* from now on) is one of the most useful results in theoretical work involving finite automata and the regular sets. It is the major tool used to

a) detect non-regular sets, and
b) prove decision problems solvable for finite automata.

The usefulness of the pumping lemma comes from the fact that it dramatically points out one of the major characteristics of finite automata, namely that they have only a finite amount of memory. Another way to state this is to say that if a finite automaton has n states, then it *can only remember n different things*! In fact, if $\delta^*(s_0, x) = \delta^*(s_0, y)$ then the machine with the transition function δ cannot tell the difference between x and y. They look the same to the machine since they induce the same last state in computations involving them. And, if a finite automaton accepts a very long string, then chances are that this string contained repetitive patterns.

Our first use of the pumping lemma will be to present a non-regular set. This is the favorite example for computer science theorists and illustrates the method almost always used to prove sets non-regular.

> **Theorem 2**. *The set of strings of the form $\{0^n1^n\}$ for any $n \geq 0$ is not a regular set.*
>
> **Proof.** Assume that the set of strings of the form 0^n1^n is a regular set and that the finite automaton $M = (S, I, \delta, s_0, F)$ accepts it. Thus every string of the form 0^k1^k for k larger than the size of the state set S will be accepted by M.
>
> If we take one of these strings for some $k > |S|$ then the pumping lemma assures us that there are strings u, v, and w such that:

a) $uvw = 0^k1^k$,
b) $v \neq \varepsilon$, and
c) for all $n \geq 0$, $uv^nw \in T(M)$.

Invoking the corollary to the pumping lemma assures us that the substring v can be in the midst of the 0's if we wish. Thus $v = 0^m$ for some (nonzero) $m \leq k$. This makes $uw = 0^{k-m}1^k$ and the pumping lemma states that $uw \in T(M)$. Since uw is not of the form 0^n1^n we have a contradiction. Thus our assumption that strings of the form 0^n1^n can be accepted by finite automata and be regular set is incorrect.

As we mentioned earlier, almost *all* of the proofs of non-regularity involve the same technique used in the proof of the last theorem. One merely needs to examine the position of v in a long string contained in the set. Then either remove it or repeat it several times. This will always produce an improper string!

Next, we shall use the deflation aspect of the pumping lemma in order to show that emptiness is solvable for regular sets.

Theorem 3. *If a finite automaton accepts any strings, it will accept one of length less than the size of its state set.*

Proof. Let M be a finite automaton which accepts the string x and that the length of x is no less than the size of M's state set. Assume further that M accepts no strings shorter than x. (This is the opposite of our theorem.)

Immediately the pumping lemma asserts that there are strings u, v, and w such that $uvw = x$, $v \neq \varepsilon$, and $uw \in T(M)$. Since $v \neq \varepsilon$, uw is shorter than $uvw = x$. Thus M accepts shorter strings than x and the theorem follows.

Here is a sequence of corollaries which follow from the last theorem. In each case the proof merely involves checking membership for all strings of length less than the size of the state set for some finite automaton or the machine which accepts the complement of the set it accepts. (Recall that the class of regular sets, namely those accepted by finite automata is closed under complement.)

Corollary (Emptiness Problem)**.** *Whether or not a finite automaton accepts anything is solvable.*

Corollary (Emptiness of Complement)**.** *Whether or not a finite automaton rejects anything is solvable.*

Corollary. *Whether or not a finite automaton accepts everything is solvable.*

Another cautionary note about these decision problems is in order. It is quite refreshing that many things are solvable for the regular sets and it is wonderful that several solvable decision problems came immediately from one lemma and a theorem. But, it is very *expensive* to try and solve these problems. If we need to look at all of the input strings of length less than the size of the state set (let's say that it is of size n) then we are looking at almost 2^n strings! Imagine how long this takes when n is equal to 100 or so!

Flushed with success we shall attempt (and succeed) at another decision problem which is unsolvable for the class of recursively enumerable sets.

Theorem 4. *Whether a regular set is finite is solvable.*

Proof Sketch. We know that if a finite automaton accepts any strings at all then some will be of length less than the size of the state set. (This does not help directly, but it gives a hint as to what we need for this theorem.) The deletion aspect [uw Y T(M)] of the pumping lemma was used to prove this. Let's use the inflation aspect [uv^nw Y T(M)] of the lemma to look for an infinite set.

For starters, if we were to find a string accepted by a finite automaton M which was longer than or equal to the size of it's state set, we could use the aforementioned inflation aspect of the pumping lemma to show that machine M must accept an infinite number of strings. This means that:

a finite automaton accepts only strings of length less than the size of its set of states, if and only if it accepts a finite set.

Thus, to solve the finiteness problem for M = (S, I, δ, s_0, F), we need to determine whether or not:

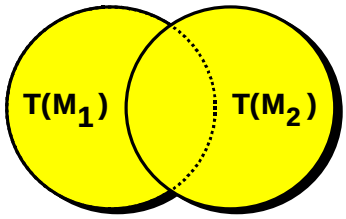
$$T(M) - \{\text{strings of length} < |S|\} = \varnothing.$$

A question now arises as to how many inputs we must examine in order to tell if M will accept an input longer than or equal to the size of its state set. The answer is that we only need to consider input strings up to twice the size of the state set. (The proof of this is left as an exercise, but it is much the same as the proof of the emptiness problem.) This ends our proof sketch.

The next decision problem is included because it demonstrates another technique; using old problems to solve a new one. We see this in the areas of unsolvability and complexity when we use reducibility to show problems unsolvable or intractable. Here we show that the problem of set equivalence is reducible to the emptiness problem and thus solvable.

Theorem 5. *Whether two regular sets are identical is solvable.*

Proof. Let's take two finite automata (M_1 and M_2) and examine a picture of the sets they accept.



If the intersection is the same as both sets then indeed they are identical. Or, on the other hand, if the areas outside the intersection are empty then both sets are identical. Let's examine these outside areas.

$T(M_1) \cap \overline{T(M_2)}$ $\qquad$ $T(M_2) \cap \overline{T(M_1)}$

The picture on the left represents the set accepted by M_1 and rejected by M_2 while that on the right is the set which M_2 accepts and M_1 rejects.

If these two areas (or sets) are empty then the sets accepted by M_1 and M_2 are exactly the same. This means that the equivalence problem for $T(M_1)$ and $T(M_2)$ is exactly the same as the emptiness problem for:

$$[\, T(M_1) \cap \overline{T(M_2)} \,] \cup [\, T(M_2) \cap \overline{T(M_1)} \,]$$

So, if we can solve the emptiness problem for the above set, then we can solve the equivalence problem for $T(M_1)$ and $T(M_2)$. Since the regular sets are closed under union, complement, and intersection; the above set is a regular set. And, we know that emptiness is solvable for the class of regular sets.

Here is yet another cautionary note. The last proof was quite slick and elegant, but one should not do the construction in an attempt to prove that two finite

automata accept the same set. We know that the complexity of any algorithm which attempts to do this is very large since it would take a while to do emptiness for a machine formed from the unions and intersections of four different machines.

We shall close this section on a cheerful note with an intuitive, imprecise statement about finite automata and the class of regular sets.

> **Folk Theorem.** *Just about everything concerning finite automata or the regular sets is solvable!*

Pushdown Automata

In the last section we found that restricting the computational power of computing devices produced solvable decision problems for the class of sets accepted by finite automata. But along with this ability to solve problems came a rather sharp decrease in computational power. We discovered that finite automata were far too weak to even tell if an input string was of the form a^nb^n. In this section we shall extend the power of finite automata a little so that we can decide membership in sets which cannot be accepted by finite automata.

Let's begin. In fact, let's provide a finite automaton with a data structure which will allow it to recognize strings of the form a^nb^n. To tell if a string is of the form a^nb^n we need to match the a's with the b's. We could use a counter for this, but thinking ahead a bit, there is a computer science way to do this. We shall allow the machine to build a pile of discs as it processes the a's in its input. Then it will unpile these disks as it passes over the b's. Consider the following algorithm for a machine of this kind.

```
place the input head on the leftmost
      input symbol

while symbol read = a
   advance head
   place disc on pile

while symbol read = b and pile contains discs
   advance head
   remove disc from pile

if input has been scanned
     and pile = empty then accept
```

Figure 1 - a^nb^n Recognition Algorithm

It is clear exactly what happens when the algorithm of figure 1 is used on the input aaabbb. The machine reads the a's and builds a pile of three discs. Then it reads the b's and removes the discs from the pile one by one as each b is read. At this point it has finished the input and its pile is empty so it accepts. If it was given aabbb, it would place two discs on the pile and then remove them

as it read the first two b's. Then it would leave the second while loop with one b left to read (since the pile was empty) and thus not accept. For aaabb it would end with one disk on the pile and not accept that input either. When given the input string aabbab, the machine would finish the second loop with ab yet to be read. Now, try the strings aaa and bbb as exercises. What happens?

We now have a new data structure (a pile) attached to our old friend, the finite automaton. During the last algorithm several conventions implicitly arose. They were:

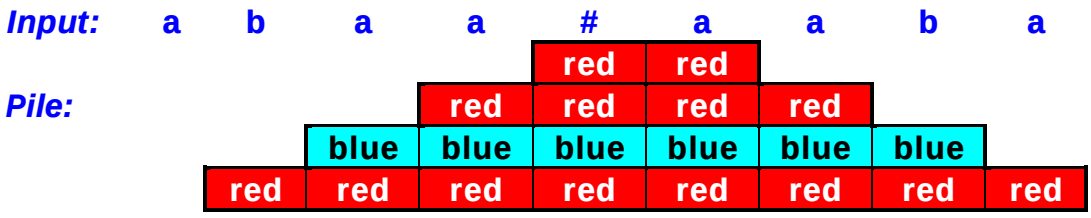
- The tape head advanced on each step,
- Discs were placed on *top* of the pile, and
- An empty pile means acceptance.

Let us now attempt something a bit more difficult. Here's where we shall use a structure more powerful than a counter. Why not try to recognize strings of the form $w\#w^R$ where w is a string over the alphabet {a, b} and w^R is the *reversal* of the string w? (Reversal is just turning the string around end for end. For example, $abaa^R = aaba$.) Now we need to do some comparing, not just counting. Examine the algorithm of figure 2.

```
place input head upon leftmost input symbol

while symbol being scanned ≠ #
   if symbol scanned = a, put red disk on pile
   if symbol scanned = b, put blue disk on pile
   advance input head to next symbol

advance input head past #

repeat
  if (symbol scanned = a and red disk on pile)
     or (symbol scanned = b and blue disk on pile)
        then remove top disk; advance input head
until (pile is empty) or (no input remains)
      or (no disk removed)

if input has been read and pile is empty then accept
```

Figure 2 - Accepting Strings of the Form $w\#w^R$

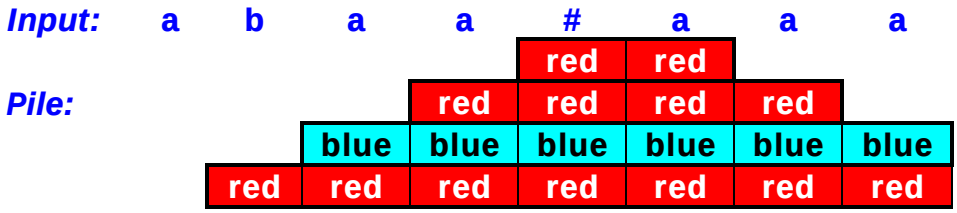
We will now look at what happens to the disc pile when this machine processes the input string abaa#aaba. Here is a picture:



At the right end of the picture, the machine reads the a and removes the red disk from the stack. Since the stack is empty, it accepts.

Our first machine (figure 1) used its discs to count the a's and match them against the b's. The second machine (figure 2) used the pile of discs to full advantage in that it compared actual symbols, not just the number of them. Note how this machine recorded the symbols before the marker (#) with discs and then matched them against the symbols following the marker. Since the input was completely read and the pile was empty, the machine accepted.

Now, here is what happens when the string abaa#aaab is processed:



In this case, the machine stopped with ab yet to read and discs on the pile since it could not match an a with the blue disc. So, it rejected the input string. Try some more examples as exercises.

The machines we designed algorithms for above in figures 1 and 2 are usually called *pushdown automata*. All they are is finite automata with auxiliary storage devices called *stacks*. (A *stack* is merely a pile. And symbols are normally placed on stacks rather than various colored discs.) The rules involving stacks and their contents are:

a) Symbols must always be placed upon the top of the stack.
b) Only the top symbol of a stack can be read.
c) No symbol other than the top one can be removed.

We call placing a symbol upon the stack a *push* operation and removing one from the top of the stack a *pop* operation.

Figure 3 provides a picture of one of these machines.

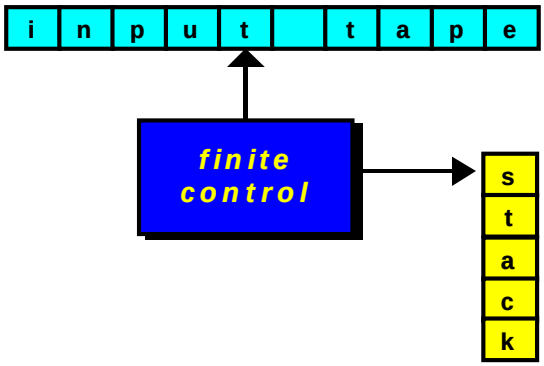


Figure 3 - A Pushdown Automaton

Pushdown automata can be presented as state tables in very much the same way as finite automata. All we need to add is the ability to place (or *push*) symbols on top of the stack and to remove (or *pop*) symbols from the top of the stack. Here is a state table for our machine of figure 1 which accepts strings of the form $a^n b^n$.

state	read	pop	push	goto
1	a		A	1
	b	A		2
2	b	A		2

Note that this machine operates exactly the same as that of the algorithm in figure 1. During operation, it:

a) reads a's and pushes A's on the stack in state 1,
b) reads b's and pops A's from the stack in state 2, and
c) accepts if the stack is empty at the end of the input string.

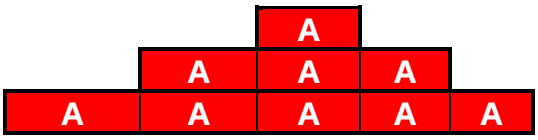
Thus, the states of the pushdown machine perform exactly the same as the while loops in the algorithm presented in figure 1.

If a pushdown machine encounters a configuration which is not defined (such as the above machine being in state 2 reading an a, or any machine trying to pop a symbol from an empty stack) then computation is terminated and the machine rejects. This is very similar to Turing machine conventions.

A trace of the computation for the above pushdown automaton on the input aaabbb is provided in the following picture:

Input Remaining:	aaabbb	aabbb	abbb	bbb	bb	b	
State:	s_1	s_1	s_1	s_1	s_2	s_2	s_2

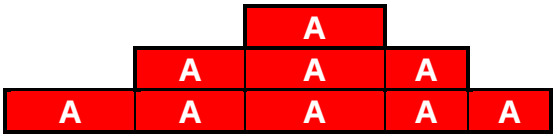
Stack:



and a trace for the input aaabb is:

Input Remaining:	aaabb	aabb	abb	bb	b	
State:	s_1	s_1	s_1	s_1	s_2	s_2

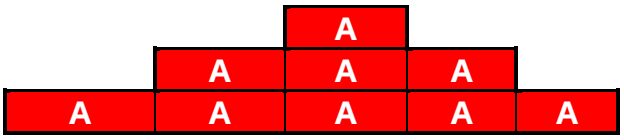
Stack:



In the first computation (for input aaabbb), the machine ended up in state s_2 with an empty stack and accepted. The second example ended with an A on the stack and thus the input aaabb was rejected. If the input was aaabba then the following would take place:

Input Remaining:	aaabba	aabba	abba	bba	ba	a
State:	s_1	s_1	s_1	s_1	s_2	s_2

Stack:



In this case, the machine terminates computation since it does not know what to do in s_2 with an a to be read on the input tape. Thus aaabba also is rejected.

Our second machine example (figure 2) has the following state table.

state	*read*	*pop*	*push*	*goto*
1	a		A	1
	b		B	1
	#		2	
2	a	A		2
	b	B		2

Note that it merely records a's and b's on the stack until it reaches the marker (#) and then checks them off against the remainder of the input.

Now we are prepared to precisely define our new class of machines.

Definition. *A **pushdown automaton** (pda) is a quintuple* $M = (S, \Sigma, \Gamma, \delta, s_0)$, *where:*

S *is a finite set (of states),*
Σ *is a finite (input) alphabet,*
Γ *is a finite (stack) alphabet,*
δ: $S \times \Sigma \times \Gamma \cup \{\varepsilon\} \rightarrow \Gamma^* \times S$ *(transition function),*
and $s_0 \in S$ *(the initial state).*

In order to define computation we shall revert to the conventions used with Turing machines. A *configuration* is a triple <s, x, α> where s is a state, x a string over the input alphabet, and α a string over the stack alphabet. The string x is interpreted as the input yet to be read and α is of course the content of the stack. One configuration *yields* another (written $C_i \rightarrow C_k$) when applying the transition function to it results in the other. Some examples from our first machine example are:

$$\begin{aligned} \langle s_1, aaabbb, \varepsilon\rangle &\rightarrow \langle s_1, aabbb, A\rangle \\ \langle s_1, aabbb, A\rangle &\rightarrow \langle s_1, abbb, AA\rangle \\ \langle s_1, abbb, AA\rangle &\rightarrow \langle s_1, bbb, AAA\rangle \\ \langle s_1, bbb, AAA\rangle &\rightarrow \langle s_2, bb, AA\rangle \\ \langle s_2, bb, AA\rangle &\rightarrow \langle s_2, b, A\rangle \\ \langle s_2, b, A\rangle &\rightarrow \langle s_2, \varepsilon, \varepsilon\rangle \end{aligned}$$

Note that the input string decreases in length by one each time a configuration yields another. This is because the pushdown machine reads an input symbol every time it goes through a step.

We can now define *acceptance* to take place when there is a sequence of configurations beginning with one of the form $\langle s_0, x, \varepsilon\rangle$ for the input string x and ending with a configuration $\langle s_i, \varepsilon, \varepsilon\rangle$. Thus a pushdown automaton accepts when it finishes its input string with an empty stack.

There are other conventions for defining pushdown automata which are equivalent to that proposed above. Often machines are provided with an initial stack symbol Z_0 and are said to terminate their computation whenever the stack is empty. The machine of figure 2 might have been defined as:

state	read	pop	push	goto
0	a	Z_0	A	1
	b	Z_0	B	1
	#	Z_0		2
1	a		A	1
	b		B	1
	#			2
2	a	A		2
	b	B		2

if the symbol Z_0 appeared upon the stack at the beginning of computation. Including it in our original definition makes a pushdown automaton a *sextuple* such as:

$$M = (S, \Sigma, \Gamma, \delta, s_0, Z_0).$$

Some definitions of pushdown automata require the popping of a stack symbol on every move of the machine. Our example might now become:

state	read	pop	push	goto
0	a	Z_0	A	1
	b	Z_0	B	1
	#	Z_0		2
1	a	A	AA	1
	a	B	AB	1
	b	A	BA	1
	b	B	BB	1
	#	A	A	2
	#	B	B	2
2	a	A		2
	b	B		2

where in state s_1 the symbols which were popped are placed back upon the stack.

Another very well known convention is to have pushdown machines *accept by final state*. This means that the automaton must pass the end of the input in an accepting or final state. Just like finite automata. Now we have a final state subset and our machine becomes:

$$M = (S, \Sigma, \Gamma, \delta, s_0, Z_0, F)$$

and our tuples get larger and larger.

Converting this example to this format merely involves detecting when the stack has only one symbol upon it and changing to an accepting state if things are satisfactory at this point. We do this by placing special sentinels (X and Y) on the bottom of the stack at the beginning of the computation. Here is our example with s_3 as an accepting state. (Note that the machine accepts by empty stack also!).

state	read	pop	push	goto
0	a		X	1
	b		Y	1
	#			3
1	a		A	1
	b		B	1
	#			2
2	a	A		2
	a	X		3
	b	B		2
	b	Y		3

All of these conventions are equivalent (the proofs of this are left as exercises) and we shall use any convention which seems to suit the current application.

Now to get on with our examination of the exciting new class of machines we have defined. Our first results compare them to other classes of automata we have studied.

Theorem 1. *The class of sets accepted by pushdown automata properly includes the regular sets.*

Proof. This is very easy indeed. Since finite automata are just pushdown machines which do not ever use their stacks, all of the regular sets can be accepted by pushdown machines which accept by final state. Since strings of the form $a^n b^n$ can be accepted by a pushdown machine (but not by any finite automaton), the inclusion is proper.

Theorem 2. *All of the sets accepted by pushdown automata are recursive.*

Proof. For the same reason that the regular sets are recursive. Pushdown machines are required to process an input symbol at each step of their computation. Thus we can simulate them and see if they accept.

Corollary. *The class of recursively enumerable sets properly contains the class of sets accepted by pushdown automata.*

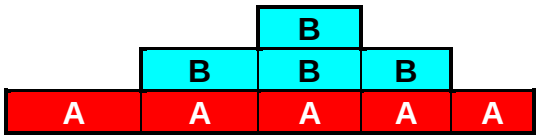
As you may have noticed (and probably were quite pleased about), the pushdown machines designed thus far have been deterministic. These are usually called *dpda*'s and the *nondeterministic* variety are known as *npda*'s. As with nondeterministic finite automata, npda's may have several possible moves for a configuration. The following npda accepts strings of the form ww^R by nondeterministicly deciding where the center of the input string lies.

state	read	pop	push	goto
0	a		A	0
	a	A		1
	b		B	0
	b	B		1
1	a	A		1
	b	B		1

Note that in s_0 there are choices of either to remain in s_0 and process the w part of ww^R or to go to s_1 and process the w^R portion of the input. Here is an example computation of the machine on the string abbbba.

Input Remaining:	**abbbba**	**bbbba**	**bbba**	**bba**	**ba**	**a**	
State:	**s_0**	**s_0**	**s_0**	**s_0**	**s_1**	**s_1**	**s_1**

Stack:



One should note that the machine does in fact change from s_0 to s_1 when bba remains to be read and a B is on top of the stack. We shall not prove it here, but *nondeterminism does indeed add power to pushdown machines*. In fact, the set of strings of the form ww^R cannot be accepted by a deterministic pushdown automaton because it would have no way to detect where the center of the input occurs.

Our last example of a set which can be accepted by a pushdown automaton is a set of simple arithmetic expressions. Let's take v as a variable, + as an operator, and put them together (in the proper manner) with parentheses. We get expressions such as:

v+v+v, or v+(v+v), or (v+v)+(v+v)

but not expressions like:

v+v+v+, or (v+v)(v+v), or (v+(v+(v+v)).

The strategy for the machine design is based on the method in which these simple expressions are generated. Since an expression can be:

a) a variable,
b) a variable, followed by a plus, followed by an expression, or
c) an expression enclosed in parentheses.

Here is a simple but elegant, one state, nondeterministic machine which decides which of the above three cases is being used and then verifies it. The machine begins computation with the symbol E upon its stack.

read	*pop*	*push*
v	E	OE
v	E	
(	E	EP
+	O	
)	P	

With a little effort this nondeterministic machine can be turned into a deterministic machine. Then, more arithmetic operators (such as subtraction, multiplication, etc.) can be added. At this point we have a major part of a parser for assignment statements in programming languages. And, with some output we could generate code exactly as compilers do. This is discussed in the treatment on formal languages.

Unsolvable Problems for Pushdown Automata

Soon after introducing pushdown automata we proved that their membership problem was solvable. Well, this was not quite true since we did it only for deterministic machines. But, it should not be too difficult to extend this result to the nondeterministic variety. Soon we shall when we examine formal languages. Also at that time we shall show that the emptiness and finiteness problems for pushdown automata are solvable too. Unfortunately most of the other decision problems for these machines are unsolvable. So, it would appear that stacks produce some unsolvability.

As usual we shall use reducibility. We shall map some undecidable Turing machine (or actually r.e. set) problems into pushdown automata problems. To do these reductions we need to demonstrate that pushdown automata can analyze Turing machine computations. Recall from before that a *Turing machine configuration* is a string that describes exactly what the machine is doing at the moment. It includes:

a) what is written on the tape,
b) which symbol the machine is reading, and
c) the instruction the Turing machine is about to execute.

If we dispense with endmarkers and restrict ourselves to the two symbol alphabet {0, 1, *b*} then a configuration is a string of the form x(Ik)y or x where x and y are strings over the alphabet {0, 1, *b*}. (We interpret the configuration x(Ik)y to mean that the string xy is on the tape, the machine is about to execute instruction *Ik*, and is reading the first symbol of the string y. A configuration of the form x with no instruction in it means that the machine has halted with x on its tape.) Let us prove some things as we develop the ability of pushdown machines to examine Turing machine computations.

Lemma. *The set of strings that are valid Turing machine configurations is a regular set.*

Proof. This is simple because a Turing machine configuration is of the form $[0+1+b]^*$ or of the form $[0+1+b]^*(I1[0+1]^*)[0+1+b]^*$. (Note that we used square brackets in our expressions because the parentheses were present in the Turing machine configurations.)

Since these are regular expressions and represent a regular set.

A *computation* is merely a sequence of configurations (C_i) separated by markers. It can be represented by a string of the form:

$$C_1\#C_2\# \ldots \#C_n$$

Since the regular sets are closed under concatenation and Kleene star these kinds of strings form a regular set also.

Now let us examine just what makes a sequence of valid Turing machine configurations *not a valid halting computation* for a particular machine. Either:

1) C_1 is not an initial configuration,
2) C_n is not a halting configuration,
3) one of the C_i contains an improper instruction, or
4) for some $i < n$, some C_i does not yield C_{i+1}.

Parts (1) and (2) are easy to recognize. An initial configuration is merely a string of the form (I1)[0+1+*b*]* and a halting configuration is a string of the form [0+1+*b*]*. These are both regular sets. Part (3) is regular also because Turing machine instructions are required to be members of a sequence beginning at I1 and there can be only a finite number of them for a particular Turing machine. Thus this is regular also because a finite automata can recognize finite sets.

Let us now put this in the form of a lemma that we shall use later when we wish to build a pushdown machine which check Turing machine computations.

Lemma. *For each Turing machine there are finite automata that are able to detect:*

a) initial configurations,
b) halting configurations, and
c) configurations with improper instructions.

The final situation that leads to a string not being a valid Turing machine computation is part (4) of our previous list. This when a configuration does not yield the next in the sequence. We shall show now that a nondeterministic pushdown machine can detect this.

Lemma. *For each Turing machine, there is a pushdown automaton that accepts pairs of configurations (for that Turing machine) such that the first does not yield the second.*

Proof. Let C_1 and C_2 be valid configurations for a particular Turing machine. We shall build a pushdown automaton which accepts the input $C_1\#C_2$ whenever it is not the case that $C_1 \rightarrow C_2$. The automaton will

operate in a nondeterministic manner by first selecting the reason that C_1 does not yield C_2, and then verifying that this happened. Let us analyze the reasons why C_1 might not yield C_2.

a) *No configuration can follow C_1.* There are three possible reasons for this, namely:

1) C_1 is a halting configuration.
2) The instruction in C_1 is not defined for the symbol read.
3) C_1 = (Ik)x and the Turing machine wishes to move left.

Since these three conditions are finite, a finite automaton could detect them, and so can our pushdown machine.

b) *The instruction in C_2 is incorrect.* In other words it does not follow from the situation in C_1. This can also be detected by a finite automaton.

c) *The tape symbols in C_2 are wrong.* Suppose that

C_1 =

x	a	(	I	k	)	c	y

and instruction Ik calls for the Turing machine to write the symbol d and transfer to instruction Im if it reads the symbol c. Then it should be the case that

C_2 =

x	(	I	m	)	a	d	y

if the Turing machine moved left, and

C_2 =

x	a	d	(	I	m	)	y

if the Turing machine moved a square to the right. Suppose that instead of things being as they should, we end up with the following configuration:

C_2 =

u	(	I	m	)	v

The errors that could take place have been tabulated in the following table.

left move error:	**$u \neq x$**	**or**	**$v \neq ady$**
right move error:	**$u \neq xad$**	**or**	**$v \neq y$**

The pushdown machine first nondeterministicly selects which of the four errors has taken place and then verifies it. Here is how this works. Our pushdown automaton first scans xacy (the tape in C_1) and uses its stack as a counter to select the position of where the error will be in C_2. It then remembers (in its finite control) exactly what symbol should be in that location on C_2. Now it counts off squares of C_2 until it reaches the site of the error and verifies that the wrong symbol is in that location.

To recap, our pushdown automaton first selects the error that makes C_1 not yield C_2 and then verifies that it did indeed take place.

Now we possess all of the tools necessary to show that pushdown machines can analyze strings and detect whether or not they are Turing machine computations. This is done by enumerating what could be making the string not a proper computation, and then verifying that something of this sort took place.

Theorem 1. *For every Turing machine, there is a pushdown automaton that accepts all strings that are not valid halting computations for the Turing machine.*

Proof Sketch. The pushdown machine in question merely selects (nondeterministicly) the reason that its input cannot be a valid halting computation for the Turing machine. Then it verifies that this error took place. The preceding lemmas provide the list of errors and the methods needed to detect them.

This theorem is very interesting in its on right. as well as being a nice example of nondeterministic computation. In fact, the *guess and verify* strategy cannot be carried out on a deterministic device unless the choices remain constant no matter how long the input gets.

It also indicates that pushdown automata can in some sense analyze the computations of Turing machines. (By the way, npda *cannot* detect *valid* halting computations for Turing machines. This should become clear after observing linear bounded automata that is a more powerful device that is able to recognize valid computations.) The major use of this theorem, however, is to

prove that several decision problems involving pushdown machines are unsolvable. This will be done by reducing unsolvable problems for the r.e. sets to problems for pushdown automata.

Theorem 2. *Whether or not a pushdown automaton accepts every string over its alphabet is unsolvable.*

Proof. We shall reduce the emptiness problem for Turing machines (is $W_i = \varnothing$?) to this problem (known as the Σ^* problem since Σ is the input alphabet) for pushdown machines.

Let M_i be a Turing machine and let $P_{g(i)}$ be the pushdown automaton that accepts the set of strings that are not valid halting computations for Turing machine M_i. Then note that:

$$\begin{aligned} W_i = \varnothing \quad & \text{iff } \forall x[M_i(x) \text{ never halts}] \\ & \text{iff } \forall ax[M_i(x) \text{ has no halting computations}] \\ & \text{iff } P_{g(i)} \text{ accepts every input} \end{aligned}$$

That was the reduction from the emptiness problem for Turing machines to the Σ^* problem for pushdown machines. Recalling our results on reducibilities indicates that both must be unsolvable. (Also they are not r.e.)

Acceptance of strings that are not valid halting computations of Turing machines leads to several other unsolvability results concerning pushdown automata. Two which appear in the exercises are the *equivalence problem* (whether two machines accept the same set) and the *cofiniteness problem* (whether the complement of the set accepted by a pushdown machine is finite). They are proven in the same manner.

Linear Bounded Automata

The last machine model of computation which we shall examine is the *linear bounded automaton* or *lba*. These were originally developed as models for actual computers rather than models for the computational process. They have become important in the theory of computation even though they have not emerged in applications to the extent which pushdown automata enjoy.

Here is the motivation for the design of this class of machines. Computers are finite devices. They do not have unending amounts of storage like Turing machines. Thus any actual computation done on a computer is not as extensive as that which could be completed on a Turing machine. So, to mimic (or maybe model) computers, we must restrict the storage capacity of Turing machines. This should not be as severely as we did for finite automata though. Here is the definition.

> **Definition.** *A **linear bounded automaton** (lba) is a multi-track Turing machine which has only one tape, and this tape is exactly the same length as the input.*

That seems quite reasonable. We allow the computing device to use just the storage it was given at the beginning of its computation. As a safety feature, we shall employ endmarkers (* on the left and # on the right) on our lba tapes and never allow the machine to go past them. This will ensure that the storage bounds are maintained and help keep our machines from leaving their tapes.

At this point, the question of accepting sets arises. Let's have linear bounded automata accept just like Turing machines. Thus for lba halting means accepting.

For these new machines computation is restricted to an area bounded by a constant (the number of tracks) times the length of the input. This is very much like a programming environment where the sizes of values for variables is bounded.

Now that we know what these devices are, let's look at one. A set which cannot be accepted by pushdown machines (this is shown in the material on formal languages) is the set of strings whose length is a perfect square. In symbols this is:

$$\{a^n \mid n \text{ is a perfect square } \}.$$

Here is the strategy. We shall use a four track machine with the input written on the first track. The second and third tracks are used for scratch work while the fourth track is holds strings of square length which will be matched against the input string.

To do this we need to generate some strings of square length. The second and third tracks are used for this. On the second track we will build strings of length k = 1, 2, 3, and so forth. After each string is built, we construct (on the fourth track) a string whose length is the square of the length of the string on the second track by copying the second track to the fourth exactly that many times. The third track is used to count down from k. Here is a little chart which explains the use of the tracks.

track	*content*
1	a^n (input)
2	a^k
3	a^{k-m}
4	a^{mk}

Then we check to see if this is the same length as the input. The third track is used for bookkeeping. The algorithm is provided as figure 1.

```
repeat
  clear the 3rd and 4th tracks
  add another a to the 2nd track
  copy the 2nd track to the 3rd track
  while there are a's written on the 3rd track
     delete an a from the 3rd track
     add the 2nd track's a's to those on 4th track
until overflow takes place or 4th track = input
if there was no overflow then accept
```

Figure 1 - Recognition of Perfect Square Length Inputs

Now we've seen something of what can be done by linear bounded automata. We need to investigate some of the decision problems concerning them. The first problem is the halting problem.

Theorem 1. *The halting problem is solvable for linear bounded automata.*

Proof. Our argument here will be based upon the number of possible configurations for an lba. Let's assume that we have an lba with one track (this is allowed because can use additional tape symbols to simulate tracks as we did with Turing machines), k instructions, an alphabet of s

tape symbols, and an input tape which is n characters in length. An lba configuration is the same as a Turing machine configuration and consists of:

a) an instruction,
b) the tape head's position, and
c) the content of the tape.

That is all. We now ask: how many different configurations can there be? It is not too difficult to figure out. With s symbols and a tape which is n squares long, we can have only s^n different tapes. The tape head can be on any of the n squares and we can be executing any of the k instructions. Thus there are only

$$k*n*s^n$$

possible different configurations for the lba.

Let us return to a technique we used to prove the pumping lemma for finite automata. We observe that if the lba enters the same configuration twice then it will do this again and again and again. It is stuck in a loop.

The theorem follows from this. We only need to simulate and observe the lba for $k*n*s^n$ steps. If it has not halted by then it must be in a loop and will never halt.

Corollary. *The membership problems for sets accepted by linear bounded automata are solvable.*

Corollary. *The sets accepted by linear bounded automata are all recursive.*

Let's pursue this notion about step counting a bit more. We know that an lba will run for no more than $k\&n\&s^n$ steps because that is the upper bound on the number of configurations possible for a machine with an input of length n. But, let us ask: *exactly* how many configurations are *actually* reached by the machine? If we knew (and we shall soon) we would have a sharper bound on when looping takes place. Thus to detect looping, we could count steps with an lba by using an extra track as a step counter.

Now let's make the problem a little more difficult. Suppose we had a nondeterministic linear bounded automaton (nlba). We know what this is; merely a machine which has more than one possible move at each step. And, if it can achieve a halting configuration, it accepts. So we now ask: how many configurations can an nlba reach for some input? We still have only $k*n*s^n$

possible configurations, so if we could detect them we could count them using n tape squares. The big problem is how to detect them. Here is a rather nifty result which demonstrates nondeterminism in all of its glory. We start with a series of lemmata.

Lemma. *For any nondeterministic linear bounded automaton there is another which can locate and examine m configurations reachable (by the first lba) from some input if there are at least m reachable configurations.*

Proof. We have an nlba and an integer m. In addition we know that there are at least m configurations reachable from a certain input. Our task is to find them.

If the nlba has k instructions, one track, s symbols, and the input is length n, then we know that there are at most k&n&s^n possible configurations (C_i) reachable from the starting configuration (which we shall call C_0). We can enumerate them and check whether the nlba can get to them. Consider:

```
x = 0
for i = 1 to k*n*s^n
    generate C_i
    guess a path from C_0 to C_i
    verify that it is a proper path
    if C_i is reachable then x = x + 1
verify that x ≥ m (otherwise reject)
```

This is a perfect example of the *guess and verify* technique used in nondeterministic operation. All we did was exploit our definition of nondeterminism. We looked at all possible configurations and counted those which were reachable from the starting configuration.

Note also that every step above can be carried out using n tape squares and several tracks. Our major problem here is to count to k*n*s^n. We need to first note that for all except a few values of n, this is smaller than $(s+1)^n$ and we can count to this in base s+1using exactly n tape squares.

Since we verify that we have indeed found at least m configurations our algorithm does indeed examine the appropriate number of reachable configurations if they exist.

Lemma. *For any nondeterministic linear bounded automaton there is another which can compute the number of configurations reachable from an input.*

Proof. As before we begin with an arbitrary machine which has k instructions, one track, s symbols, and an input of length n. We shall iteratively count the number of configurations (n_i) reachable from the initial configuration. Consider:

```
n0 = 1
i = 0
repeat
  i = i + 1
  ni = 0
  m := 0
  for j = 1 to k*n*s^n
    generate Cj
    guess whether Cj can be reached in i steps or less
    if path from C0 to Cj is verifiable then
       ni = ni + 1
       if reached in less than i steps then m = m + 1
  verify that m = ni-1 (otherwise reject)
until ni = ni-1
```

The guessing step is just the algorithm of our last lemma. We do it by finding all of the configurations reachable in less than i steps and seeing if any of them is C_j or if one more step will produce C_j. Since we know n_{i-1}, we can verify that we have looked at all of them.

The remainder is just counting. We do not of course have to save all of the n_i, just the current one and the last one. All of this can be done on n squares of tape (and several tracks). Noting that we are done when no more reachable configurations can be found finishes the proof.

Theorem 2. *The class of sets accepted by nondeterministic linear bounded automata is closed under complement.*

Proof. Most of our work has been done. To build a machine which accepts the complement of the set accepted by some nlba involves putting the previous two together. First find out exactly how many configurations are reachable. Then examine all of them and if any halting configurations are encountered, reject. Otherwise accept.

Our final topic is decision problems. Unfortunately we have seen the only important solvable decision problem concerning linear bounded automata. (At least there was one!) The remaining decision problems we have examined for other classes of machines are unsolvable. Most of the proofs of this depend upon the next lemma.

> **Lemma.** *For every Turing machine there is a linear bounded automaton which accepts the set of strings which are valid halting computations for the Turing machine.*

The proof of this important lemma will remain an exercise. It should not be too hard to see just how an lba could check a string to see if it is a computation though. After all, we did a rather careful analysis of how pushdown machines recognize invalid computations.

> **Theorem 3.** *The emptiness problem is unsolvable for linear bounded automata.*
>
> **Proof.** Note that if a Turing machine accepts no inputs then it does not have any valid halting computations. Thus the linear bounded automaton which accepts the Turing machine's valid halting computations accepts nothing. This means that if we could solve the emptiness problem for linear bounded automata then we could solve it for Turing machines.

In the treatment of formal languages we shall prove that the class of sets accepted by linear bounded automata properly contains the class of sets accepted by pushdown machines. This places this class in the hierarchy

$$\textbf{fa} \subset \textbf{pda} \subset \textbf{lba} \subset \textbf{TM}$$

of classes of sets computable by the various machine models we have been examining.

(By the way, we could intuitively indicate why lba's are more powerful than pushdown machines. Two observations are necessary. First, a tape which can be read and written upon is as powerful a tool as a stack. Then, note that a pushdown machine can only place a bounded number of symbols on its stack during each step of its computation. Thus its stack cannot grow longer than a constant times the length of its input.)

The other relationship we need is not available. Nobody knows if nondeterministic linear bounded automata are more powerful than ordinary ones.

NOTES

Finite automata literature began with three classic papers:

D. A. HUFFMAN, "The synthesis of sequential switching circuits," *Journal of the Franklin Institute 257:3-4 (1954)*, 161-190 and 275-303.

G. H. MEALY, "A method for synthesizing sequential circuits," *Bell System Technical Journal 34:5 (1955)*, 1045-1079.

E. F. MOORE, "Gedanken experiments on sequential machines," in *Automata Studies*, 129-153, Princeton University Press, Princeton, New Jersey, 1956.

Nondeterministic machines were first examined by Rabin and Scott. This and other papers which present closure properties concerning finite automata are:

Y. BAR-HILLEL, M. PERLES, and E. SHAMIR, "On formal properties of simple phrase structure grammars," *Zeitschrift fur Phonetik, Sprachwissenschaft, und Kommunikationsforshung 14 (1961)*, 143-172.

S. GINSBURG and E. H. SPANIER, "Quotients of context free languages," *Journal of the Association for Computing Machinery 10:4 (1963)*, 487-492.

M. O. RABIN and D. SCOTT, "Finite automata and their decision problems," *IBM Journal of Research and Development 3 (1959)*, 114-125.

Regular sets and their relation to finite automata appear in:

J. A. BRZOZOWSKI, "A survey of regular expressions and their applications," *IEEE Transactions on Electronic Computers 11:3 (1962)*, 324-335.

S. C. KLEENE, "Representation of events in nerve nets and finite automata." in *Automata Studies*, 3-42, Princeton University Press, Princeton, New Jersey 1956.

R. McNAUGHTON and H. YAMADA, "Regular expressions and state graphs for automata." *IEEE Transactions on Electronic Computers 9:1 (1960)*, 39-47.

Bar-Hillel, Perles and Shamir presented the pumping lemma and many of its uses. Other decision problems and their solutions were first examined by Moore.

Pushdown automata emerged in:

A. G. OETTINGER, "Automatic syntactic analysis and the pushdown store," *Proceedings of Symposia on Applied Mathematics 12*, American Mathematical Society, Providence, Rhode Island, 1961.

Many papers and books have been written about topics which include pushdown machines. The general texts referenced in chapters one and four mention lots of them. We shall only cite the paper containing the unsolvable decision problem results for pda.

J. HARTMANIS, "Context free languages and Turing machine computations," *Proceedings of Symposia on Applied Mathematics 19*, American Mathematical Society, Providence, Rhode Island, 1967.

Linear bounded automata were developed by Myhill and examined in:

N. IMMERMAN, "Nondeterministic space is closed under complementation." *SIAM Journal of Computing 17:5 (1988)*, 935-938.

P. S. LANDWEBER, "Decision problems of phrase structure grammars." *IEEE Transactions on Electronic Computers 13 (1964)*, 354-362.

J. MYHILL, "Linear bounded automata," *WADD TR-57-624*, 112-137, Wright Patterson Air Force Base, Ohio, 1957.

Several general texts on automata theory and formal languages are listed at the end of the languages chapter.

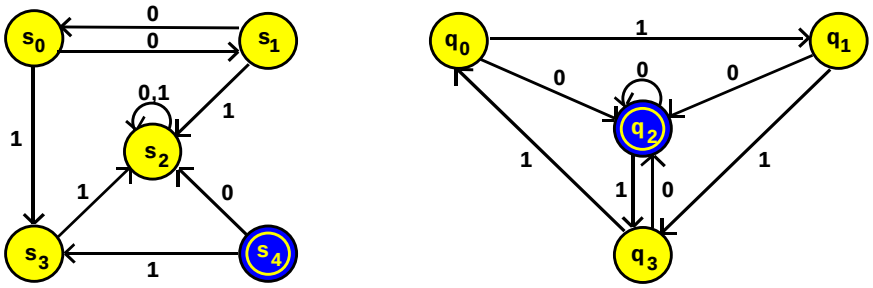
PROBLEMS

Finite Automata

1. Draw the state graphs for the finite automata which accept sets of strings composed of zeros and ones which:

   a) Are a multiple of three in length.
   b) End with the string 00.
   c) Possess runs of even numbers of 0's and odd numbers of 1's.

2. Describe the sets accepted by the finite automata pictured below.



3. Design a finite automaton that will control an elevator that serves three floors. Describe the states of the machine, intuitively present its operating strategy, and provide a state graph for your automaton.

4. Define formally and provide state tables for the finite automata that accept strings of zeros and ones which:

   a) Never contain three adjacent ones.
   b) Have a one as the next to last symbol.
   c) Contain an even number of zeros or an odd number of ones - not both!

5. String searching and pattern matching can be done easily by finite automata. Design a machine that accepts only strings containing 101 as a substring. Now do the same (design a machine) for the substring 00101.

6. Develop an algorithm to design finite automata for substring searching. The input should be the substring and the output is a finite automaton state table. Be sure to specify your data structures and intuitively describe the algorithm.

Closure Properties

1. Suppose that the finite automata M_i and M_k accept strings over the alphabet {0,1}. Design an automaton which accepts strings of the form x#y where x is accepted by M_i and y is accepted by M_k.

2. Prove that the class of sets accepted by finite automata is closed under intersection. In other words, given M_i and M_k construct the finite automaton M_m such that:

$$T(M_m) = T(M_i) \cap T(M_k)$$

3. Let x^R denote the *reversal* of the string x. (For example, if x = 010011 then x^R = 110010 and so forth.) Prove that the sets accepted by finite automata are closed under *string reversal* by constructing for any finite automaton, a new machine that accepts the reversals of the strings accepted by the original automaton.

4. Show that for each finite automaton, there is another machine that accepts only strings that are the front two thirds of the strings the first automaton accepted.

5. The *minus* operator on sets is usually defined as:

$$A - B = \{ x \mid x \in A \text{ and } x \notin B\}.$$

Prove that the class of sets accepted by finite automata is closed under minus.

6. Let x be a particular string. For arbitrary strings y and z such that z = yx, the *quotient* operator (/) can be defined:

$$z/x = yx/x = y.$$

(For example: 11010/10 = 110 and 11101/01 = 111.) This operator can be applied to sets as follows:

$$A/x = \{ y \mid yx \cup A\}.$$

(That is: $0^*110^*/10 = 0^*1$ and $0^*11(01)^*/11 = 0^*$.) Show that the class of sets accepted by finite automata is closed under quotient by constructing for any x and M_i, a machine M_k for which:

$$T(M_k) = T(M_i)/x.$$

7. *Set quotient* may be defined as:

$$A/B = \{ x \mid xy \in A \text{ for some } y \in B\}.$$

Show that the class of sets accepted by finite automata is closed under set quotient. That is, for M_i and M_k, design an M_m in such a way that:

$$T(M_m) = T(M_i)/T(M_k).$$

8. An *epsilon move* takes place when a finite automaton reads and changes state but does not move its tape head. (This is like a *stay* move for Turing machines.) Does this new operation add power to finite automata? Justify your answer.

Regular Sets and Expressions

1. What are the regular expressions for sets of strings composed of zeros and ones which:

   a) Are a multiple of three in length.
   b) End with the string 00.
   c) Possess runs (substrings) containing only even numbers of zeros and odd numbers of ones.

2. Derive the regular expressions for the sets accepted by the finite automata whose state graphs are pictured in the second problem of the first section.

3. Design finite automata that will accept the sets represented by the following regular expressions.

   a) $11(10 + 01)^*1^*01$
   b) $(0 + 11 + 01)^*0^*(01)^*$
   c) $(0 + 1)^*0^*101$

4. Show that the set of all binary integers that are the sum of exactly four (no more, no less!) positive squares is a regular set. (HINT: They are all found by substituting for m and n in the formula $4^n(8m + 7)$.)

5. Review the encodings of Turing machines from chapter one. Are these encodings a regular set? Discuss this in terms of nondeterministic Turing machines.

6. Derive and present the rules for determining *LAST* sets for regular expressions. Argue that they are correct.

7. Develop an algorithm for determining *FOLLOW* sets for any symbol in a regular expression. (You may assume that procedures for computing *FIRST* and *LAST* sets are available.)

Decision Problems for Finite Automata

1. Can finite automata accept sets of strings of the form:

   a) $0^n1^*[(0 + 11)^*(1 + 00)^*]^*0^*1^n$
   b) ww where w is a string of zeros and ones
   c) ww where w is a string of zeros

2. Can the following sets of strings be accepted by finite automata? Justify your answers!

   a) $\{ 1^n \mid n \text{ is a prime number} \}$
   b) $\{ 0^{2n}1^{2m} \mid n \text{ and } m \text{ are integers} \}$
   c) $\{ x \mid x \text{ is a binary power of two} \}$
   d) $\{ x \mid \text{the center symbol of } x \text{ is a } 1 \}$

3. Show that the regular sets are not closed under infinite union by producing an infinite family of regular sets whose union is not regular.

4. Consider a programming language in which only the following instructions occur.

```
x = 0
x = y
x = y + 1
repeat x
end
```

The symbols x and y stand for strings from a specified alphabet. A *correct program* is one which contains only the above instructions and in which an *end* eventually follows each *repeat*. *Nesting* is said to occur whenever two or more *repeat* instructions are encountered before reaching an *end*. The *depth of nesting* for a program is the number of consecutive *repeat* instructions.

Can the following sets of correct programs be accepted by finite automata?

a) Programs with depth of nesting no greater than two.
b) All correct programs.

5. Prove that every infinite regular set has an infinite regular subset.

6. Are all subsets of a regular set regular? Why?

7. Two states of a finite automaton are said not to be *equivalent* if there is a string which takes one into an accepting state and the other into a rejecting state. How many strings must be checked in order to determine whether two states are equivalent? Develop an algorithm for this.

8. Design an algorithm to determine whether a finite automaton accepts an infinite set. Prove that your algorithm is correct.

9. Exhibit an algorithm that detects whether one finite automaton accepts a subset of the set accepted by another machine. Show that this procedure works.

10. Examine the emptiness problem algorithm for finite automata. How much time does it require to analyze an automaton that has n states and uses m symbols?

Pushdown Automata

1. Design a pushdown automaton which accept strings of the form $1^*0^n1^n$ and one which accepts strings which contain twice as many zeros as ones.

2. Can pushdown automata accept sets of strings of the form:

   a) $0^n1^*[\,(0 + 11)^*(1 + 00)^*\,]^*0^*1^n$
   b) ww where w is a string of zeros and ones
   c) ww where w is a string of zeros

3. Prove that acceptance by empty stack is equivalent to accepting by final state for pushdown automata.

4. Provide pushdown machines that accept sets of strings composed of zeros and ones which are:

   a) of the form $1^n\,0^n$ or $1^n\,0^{2n}$.
   b) not of the form ww.

5. Consider pushdown automata that write output on separate one directional tapes (that is, they never go back to change any of what they have written). This basically means that they may write a string as part of each instruction. Design a machine that changes infix arithmetic expressions to postfix expressions.

6. Design a pushdown machine that generates output which will change postfix expressions into assembly language code.

7. Define pushdown automata with two stacks. Prove that they can simulate Turing machines.

8. When a pushdown machine executes an instruction and does not move its reading head, we say that it has made an *epsilon move*. Does this new capability add power to these automata? Why?

Unsolvable Problems for Pushdown Automata

1. Prove that the equivalence problem (whether two arbitrary machines accept the same set) for pushdown automata is unsolvable. (HINT: relate it to a pushdown machine problem you know is unsolvable.) Do the same for the set inclusion problem.

2. Show that whether or not a pushdown machine accepts everything but a finite set is unsolvable.

3. Design a pushdown automaton that accepts strings of the form x#y where x is a Turing machine configuration and y is the reversal of the configuration yielded by x. From this, develop two machines that accept sets whose intersection is a set of valid Turing machine computations.

4. Show that the problem of whether two pushdown automata accept sets with no elements in common is unsolvable.

Linear Bounded Automata

1. Demonstrate that multiheaded linear bounded automata are equivalent to those we defined.

2. Explain why multitrack linear bounded automata are equivalent to ordinary one track machines.

3. Design a linear bounded automaton which accepts strings of the form $0^n1^n0^n$.

4. Analyze the space and time complexity for the linear bounded automaton that accepted the set of squares.

5. Prove that linear bounded automata can accept sets of valid Turing machine computations.

6. Show that emptiness and finiteness are unsolvable for linear bounded automata.

7. Prove that equivalence is unsolvable for linear bounded automata.

LANGUAGES

Machines have been emphasized so far. This has been very useful in the sorts of problems we have been examining. We have primarily been concerned with problems of detection and numerical computation. Automata proved to be appropriate devices to aid in this study.

Now our focus will change to from recognition to generation of patterns or strings. This will be done with formal systems called *grammars*. Then the languages generated with these grammars will be tied to the machine models developed in the first and third chapters. And (of course) we shall study the properties possessed by the languages generated by the grammars.

The sections include:

Grammars

Thus far we have been doing a lot of computation. We have been doing arithmetic, finding squares, and even comparing numbers of symbols in strings. For example, in our study of automata, we found that pushdown machines could recognize strings of the form a^nb^n. This was done by counting the a's and then checking them off against the b's. Let's ask about something that must happen *before* we can recognize a string. How was that string generated in the first place? What kind of algorithm might be used to produce strings of the form a^nb^n? Consider the following computing procedure.

```
write down the symbol #
while the string is not long enough
    keep replacing the # with the string a#b
replace the # with the string ab
```

This is a very simple procedure. We should be convinced that it does indeed produce strings of the form a^nb^n. Watch what happens when we execute the algorithm. We start with a #, then replace it with a#b, then it grows to aa#bb, and so forth until we've had enough. Then we just replace the # with a final ab and we have a string of a's followed by exactly the same number of b's.

If we analyze what took place, we discover that we were applying three rules in our algorithm. Those rules were:

a) start with a #,
b) replace the # with a#b, and
c) replace the # with ab.

We should note two things here. We begin with a # and end up by replacing it with the string ab. Then, when the # is gone no more rules apply. We could use a little shorthand notation (we'll use an *arrow* instead of the phrase *gets replaced with*) and write our rules as follows.

a) Start $\rightarrow$ #
b) # $\rightarrow$ a#b
c) # $\rightarrow$ ab

So far this looks pretty good. Now we get rid of the # and replace it by the capital letter A. Also, we represent the word Start by its first letter, the symbol S. Now we have this set of rules:

$S \rightarrow A$
$A \rightarrow aAb$
$A \rightarrow ab$

and these rules tell how to replace upper case letters with strings. If we are to use these rules to generate strings, the main algorithm becomes:

```
string = S
while capital letters exist in the string
    apply an appropriate replacement rule
```

Note that this algorithm works for any group of rules like those provided above. We just replace a capital letter with the right hand side (the symbols following the arrow) of *any* of the rules in which it appears as the left hand side. This is fairly simple! Generating strings seems to be very easy with the proper set of rules.

Let's try generating a string. For example, aaabbb. Here are the steps:

S
A
aAb
aaAbb
aaaAbb

That's not too bad. We just apply the right rules in the proper order and everything works out just fine.

Let's try something not quite so easy. Let's take a close look at something important to the computer scientist - an arithmetic assignment statement. It is a string such as:

x = y + (z*x).

Let's consider how it was generated. It contains variables and some special symbols we recognize as arithmetic operators. And, in fact, it has the general form:

<variable> = <expression>

So, the starting rule for these statements could be:

$$S \rightarrow V{=}E$$

if we substitute V and E for <variable> and <expression>. This is fine. But we are not done yet. There are still things we do not have rules on how to generate. For example, what's an expression? It could be something with an operator (the symbols + or ∗) in the middle. Or it could be just a variable. Or even something surrounded by parentheses. Putting all of this into rules yields:

$$\begin{aligned} S &\rightarrow V{=}E \\ E &\rightarrow E{+}E \\ E &\rightarrow E{*}E \\ E &\rightarrow (E) \\ E &\rightarrow V \end{aligned}$$

So far, so good. We're almost there. We need to define variables. They are just letters like x, y, or z. We write this as

$$V \rightarrow x \mid y \mid z$$

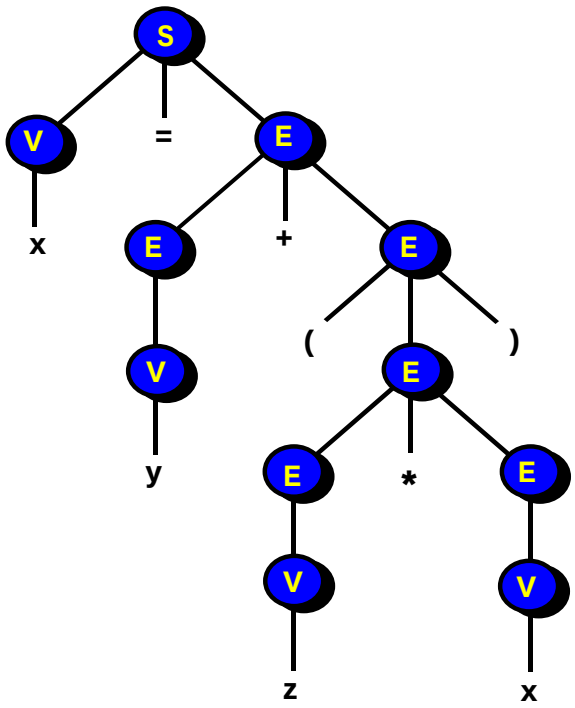
where the vertical line means *or* (so that V can be x, or y, or z). This is a little more shorthand notation.

Note that there are several ways to generate our original arithmetic statement from the set of rules we have derived. Two sequences of steps that generate the string are:

S	S
V=E	V=E
x=E	V=E+E
x=E+E	V=E+(E)
x=V+E	V=E+(E∗E)
x=y+E	V=E+(E∗V)
x=y+(E)	V=E+(E∗x)
x=y+(E∗E)	V=E+(V∗x)
x=y+(V∗E)	V=E+(z∗x)
x=y+(z∗E)	V=V+(z∗x)
x=y+(z∗V)	V=y+(z∗x)
x=y+(z∗x)	x=y+(z∗x)

These sequences are called *derivations* (of the final string from the starting symbol S). Both of the above are special derivations. That on the left is a *leftmost derivation* because the capital letter on the left is changed according to some rule in each step. And, of course the derivation on the right is called a

rightmost derivation. Another way to present a derivation is with a diagram called a *derivation tree*. An example is:



We need to note here that derivations are done in a *top-down manner*. By this we mean that we begin with an S and apply rules until only small letters (symbols of our target alphabet) remain. There is also a *bottom-up* process named *parsing* associated with string generation that we shall investigate later. It is used when we wish to determine exactly how a string was generated.

As we have seen, sets of strings can be generated with the aid of sets of rules. (And, we can even reverse the process in order to determine just how the string was generated.) *Grammars* are based upon these sets of rules. Here is how we shall define them.

Definition. *A **grammar** is a quadruple $G = (N, T, P, S)$ where:*

N is a finite set (of nonterminal symbols),
T is a finite alphabet (of terminal symbols),
P is a finite set (of productions) of the form $\alpha \rightarrow \beta$
where α and β belong to $(N \cup T)^*$, and
$S \in N$ (is the starting symbol).

At this point we should be rather comfortable with the definition of what a grammar is. Our last example involving assignment statements had {S, E, V} as the nonterminal set and {x, y, z, +, *, (,), =} as its alphabet of terminals. The

terminals and nonterminals must of course be disjoint if we wish things to not get awfully confusing.

Generating strings with the use of grammars should also be a comfortable topic. (Recall that we begin with the starting symbol and keep replacing nonterminals until none remain.) Here is a precise definition of one step in this generating process.

> **Definition.** *Let $G = (N, T, P, S)$ be a grammar. Let α, β, γ, and ξ be strings over $N \cup T$. Then $\gamma\alpha\xi$ **yields** $\gamma\beta\xi$ (written $\gamma\alpha\xi \Rightarrow \gamma\beta\xi$) under the grammar G if and only if P contains the production $\alpha \rightarrow \beta$.*

A *derivation* is just a sequence of strings where each string yields the next in the sequence. A sample derivation for our first example (strings of the form $a^n b^n$) is:

$$S \Rightarrow A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaabbb.$$

Our next definition concerns the sets generated by grammars. These are called the *languages* generated by the grammars.

> **Definition.** *The **language** generated by $G = (N, T, P, S)$ (denoted $L(G)$) is the set of all strings in T^* which can be derived from S.*

That is all there is to grammars and formal languages. Well, that is not quite so. We shall spend the remainder of the chapter examining their properties and relating them to the machines we have examined thus far. To begin this process, we will now set forth a hierarchy of languages developed by Noam Chomsky and thus called the *Chomsky hierarchy*.

First, the unrestricted grammars that are named *type 0 grammars* (with their companion languages named, the type 0 languages). Anything goes here. Productions are of the form mentioned in the definition. Right and left hand sides of productions can be any kind of strange combination of terminals and nonterminals. These are also known as *phrase structure grammars*.

Next, the *type 1 grammars*. These have much better behaved productions. Their right hand sides must not be shorter than their left hand sides. In other words, productions are of the form $\alpha \rightarrow \beta$ where the length of β is at least as long as the length of α. These are known as *length preserving* productions. The type 1 grammars and languages are called *context sensitive* because productions such as:

$$aAB \rightarrow abA$$

which depend upon context (note that AB cannot change to a bA unless an a is next to it) are allowed and encouraged.

Type 2 grammars are simpler still. The productions must be length preserving and have single nonterminal symbols on the left hand side. That is, rules of the form $A \rightarrow \alpha$. The examples at the beginning of the section are of this type. These are also known as the *context free* grammars and languages

Last (and least) are the *type 3 grammars*. Not much is allowed to happen in their productions. Just $A \rightarrow dB$ or $A \rightarrow d$ where d is a terminal symbol and both A and B are nonterminals. These are very straightforward indeed. They are called *regular* or *right linear* grammars.

One last definitional note. As a mechanism to allow the empty string as a member of a language we shall allow a production of the form

$$S \rightarrow \varepsilon$$

to appear in the rules for types 1 through 3 if the starting symbol S *never* appears on the right hand side of any production. Thus ε can be generated by a grammar but not used to destroy the length preserving nature of these grammars. A complete type 2 grammar for generating strings of the form $a^n b^n$ for $n \geq 0$ is:

$$S \rightarrow A$$
$$S \rightarrow \varepsilon$$
$$A \rightarrow aAb$$
$$A \rightarrow ab$$

As a recap of our definitions, here is a chart that summarizes the restrictions placed upon productions of all of the types of grammars. The strings α and β are of the form $(N \cup T)^*$, A and B are nonterminals, and d is a terminal symbol.

Type	Production	Restriction	Name
0	$\alpha \rightarrow \beta$		Phrase Structure
1	$\alpha \rightarrow \beta$	$\lvert\alpha\rvert \rightarrow \lvert\beta\rvert$	Context Sensitive
2	$A \rightarrow \beta$	$1 \geq \lvert\beta\rvert$	Context Free
3	$A \rightarrow dB$ $A \rightarrow d$		Regular or Right Linear

We shall close this section on grammars by presenting the grammar for a language which will later be shown not to be context free (type 2) like the previous examples. We need something a bit more complicated in order to show off our new string generating system. So, we shall design a context sensitive (type 1) grammar.

Let's try a grammar for strings of the form $0^n1^n0^n$. Here is the strategy. There are three main steps. They are:

a) Generate a string of the form $0^n(AB)^n$
b) Change this to one of the form $0^nA^nB^n$
c) Convert the A's to 1's and the B's to 0's

Thus our terminal set is {0,1} and we have A and B as nonterminals so far. We shall introduce another nonterminal C to help coordinate the above steps and make sure that they take place in the proper order. Here are the details for the three steps above.

a) Generate an equal number of 0's, A's, and B's with the productions

$$S \rightarrow 0SAB$$
$$S \rightarrow 0CB$$

These productions generate strings such as:

000CBABAB.

(By the way, the symbol C will eventually be like an A. For the moment, it will help change letters into zeros and ones.)

b) With the initial zeros are in place, we must group all of the A's and all of the B's together. This is done by the very context sensitive production:

$$BA \rightarrow AB$$

A derivation sequence of this might be:

000CBABAB
000CABBAB
000CABABB
000CAABBB

c) Now it is time to change letters into numbers. The symbol C is used as a transformer. It moves to the right changing A's to ones until it reaches the first B. Thus, we need:

$$CA \rightarrow 1C$$

At this point we have generated:

00011CBBB

and can change all of the B's to zeros with:

$$CB \rightarrow 10$$
$$0B \rightarrow 00$$

until we get the final string of terminal symbols we desire. A derivation sequence of this last part is:

00011CBBB
0001110BB
00011100B
000111000

Now let's collect the productions of the grammar we have designed.

$S \rightarrow 0SAB$	$CA \rightarrow 1C$
$S \rightarrow 0CB$	$CB \rightarrow 10$
$BA \rightarrow AB$	$0B \rightarrow 00$

And here are three more example derivations of the same string

S	S	S
0SAB	0SAB	0SAB
00SABAB	00SABAB	00SABAB
000CBABAB	000CBABAB	00SAABB
000CABBAB	000CBAABB	000CBAABB
0001CBBAB	000CABABB	000CABABB
0001CBABB	000CAABBB	000CAABBB
0001CABBB	0001CABBB	0001CABBB
00011CBBB	00011CBBB	00011CBBB
0001110BB	0001110BB	0001110BB
00011100B	00011100B	00011100B
000111000	000111000	000111000

The grammar presented above seems to generate strings of the appropriate form, but we really should back up and look at it again with the following questions in mind.

a) Are the same number of 0's, A's and B's generated?
b) What happens if some B's change to 0's too early?
c) Can all of the correct strings be generated?

It should be clear that our grammar does indeed generate strings of the form $0^n1^n0^n$. We should also note that the context sensitive nature of type 1 productions was very important in the design of our grammar. This was what allowed us to first arrange the A's and B's and then turn them into 1's and 0's.

Language Properties

We have defined a new way to look at things that are computed. Rather than examine inputs and attempt to recognize them as members of sets, we now try to generate them. This seems to work well for sets of strings with particular patterns (such as $0^n1^n0^n$). But will we be able to generate sets with arithmetic properties? For example, what about the set of primes? Or, sets with computational properties such as indices for Turing machines which halt on their own index.

It turns out that we *can* generate members of the set of primes and the set K. One of our first tasks will be to show this by relating certain classes of languages to classes of automata. This means that we will *not* show how to generate primes or sets of machine indices with weird properties. We'll just show that it *can* be done.

Reconsider the context sensitive language example (strings of the form $0^n1^n0^n$) given at the end of the section on grammars. A construction from that example will be used in the next theorem to demonstrate the relationship between the Type 0 languages and Turing machines.

Theorem 1. *The Type 0 languages and the recursively enumerable sets are identical.*

Proof Sketch. Half of this can be accomplished by showing that any Type 0 language can be accepted by some Turing machine. This is not very difficult for a nondeterministic machine. All it does is to attempt to generate its input using the production rules of the grammar. The method is straight forward. It just writes down the starting symbol and then (nondeterministicly) applies productions until the input string is generated. If so, then it accepts. And if not - who cares? Since we know that nondeterminism does not add power to Turing machines, we may now claim that:

$$\text{Type 0 Languages} \subseteq \text{R.E. Sets.}$$

Showing that each recursively enumerable set is a Type 0 language is a bit more involved. We shall use the same technique we used to show that Turing machines could be duplicated by programs - Turing machine simulation. Our strategy is to design a grammar which generates an initial Turing machine configuration and then keeps changing it the same way the Turing machine would until the machine halts. If this happens then the original input is reconstructed and provided as the grammar's output.

So, given a Turing machine, in fact a one tape, one track machine with the alphabet {0,1,*b*}, let's build a grammar which generates the strings accepted by that machine. First, we shall generate a string of symbols with the productions:

$$S \rightarrow \{X\}$$
$$X \rightarrow XA0$$
$$X \rightarrow XB1$$

The productions can generate strings such as:

{XB1A0B1B1A0B1}

Now let's do some rearranging of the kind that we did when we generated strings of the form $0^n1^n0^n$. We shall use the productions:

$$XB \rightarrow BX$$
$$XA \rightarrow AX$$
$$0B \rightarrow B0$$
$$0A \rightarrow A0$$
$$1B \rightarrow B1$$
$$1A \rightarrow A1$$

These productions now produce a string which looks like:

{BABBABX101101}

Now, if we quickly change the X with:

$$X \rightarrow \}\{(I1)\#$$

we have generated a string which looks like this.

{BABBAB}{(I1)#101101}

Note that now we have an initial Turing machine configuration at the right end of the string. And, there are brackets to mark where the ends of the configuration. On the left is a copy of the input in A's and B's with endmarkers too. If we leave this copy alone, we shall have it later on if it is needed. Summing up, we have generated a string of the form:

{input copy}{initial configuration}

What comes next is reminiscent of universal Turing machines and simulation. We shall now show how our grammar changes of one configuration to the next thereby simulating the Turing machine's computation. We now need to simulate the machine during its computation. Let us provide productions which do this for the Turing machine about to execute instruction I43 while reading a 1. Suppose this instruction called for the machine to print a 0, move left, and go to I26. Then we would include the productions:

$$0(\text{I43})1 \rightarrow (\text{I26})00$$
$$1(\text{I43})1 \rightarrow (\text{I26})10$$
$$b(\text{I43})1 \rightarrow (\text{I26})b0$$

Blanks require more work since there are two kinds. There are those which we have seen before (which look like a *b*) and those on the left end of the tape (which is marked by the } sign). So, if the machine wished to print a 1, move right, and go to *I43* after reading a blank in instruction I35, we would include the productions:

$$(\text{I35})b \rightarrow 1(\text{I43})$$
$$(\text{I35})\} \rightarrow 1(\text{I43})\}$$

With productions such as those above for each instruction, our grammar can mimic the Turing machine computation until it wishes to halt. Let's assume that the machine prints 0 and halts when it reads a 1 on instruction I26. We shall make the instruction vanish and introduce a special erasing nonterminal E. This is done with the production:

$$(\text{I26})1 \rightarrow \text{E}0$$

Let us recap slightly with a small example. Suppose we began with the tape {BABBAB}{(I1)#101101} and processed it until we reached the configuration below. If we then followed the last few instructions defined above, our last few productions might produce the sequence:

{BABBAB}{#10101(I35)*b*101}
{BABBAB}{#101011(I43)101}
{BABBAB}{#10101(I26)1001}
{BABBAB}{#101011E0001}

At this point we use the erasing nonterminal E to destroy everything between the configuration endmarkers as it moves up and down the string. We use the following productions to do this. (Note that they're type 0 and not type 1 since they reduce the string length.)

$$E0 \rightarrow E$$
$$E1 \rightarrow E$$
$$0E \rightarrow E$$
$$1E \rightarrow E$$
$$\#E \rightarrow E$$

A sample derivation sequence for this stage of the string generation might be:

{BABBAB}{#101011E001}
{BABBAB}{#101011E01}
{BABBAB}{#10101E01}
{BABBAB}{#1010E01}
{BABBAB}{#101E01}
{BABBAB}{#101E1}
{BABBAB}{#10E1}
{BABBAB}{#1E1}
{BABBAB}{#E1}
{BABBAB}{#E}
{BABBAB}{E}

At last we have only our input and some markers remaining. So, we change the eraser E into a translator T and allow it to change the A's and B's into 0's ands 1's. Then the T goes away and leaves the string of terminal symbols which was the original input to the Turing machine. Here are the productions:

$$\}\{E\} \rightarrow T$$
$$AT \rightarrow T0$$
$$BT \rightarrow T1$$
$$\{T \rightarrow \varepsilon$$

For our example, this provides the following sequence of strings which end the derivation.

{BABBAB}{E}
{BABBABT
{BABBAT1
{BABBT01
{BABT101
{BAT1101
{BT01101
{T101101
101101

In short, derivations mimic computations. We claim that for every Turing machine we can design a grammar that:

a) is able to generate any starting configuration,
b) proceed through the machine's computation steps, and
c) leave behind the original input if the machine halts.

In our grammar the terminal set was {0,1} and *all other symbols were nonterminals.* Thus if we can generate a string of 0's and 1's from the grammar, then *the machine had to halt* when given that string as input.

Thus for each Turing machine there is a type 0 language which is exactly the set of strings accepted by the machine.

Corollary. *Every language is a recursively enumerable set.*

It is time to show that context sensitive languages are the sets accepted by linear bounded automata. We state the result without proof because the ideas of the last proof sketch should provide a mechanism for anyone wishing to prove this theorem. The only trick is to add two extra tracks to the machine being simulated. One holds the input copy and the other holds the instruction being executed. This means that no erasing - just translation - need be done at the end.

Theorem 2. *The sets accepted by nondeterministic linear bounded automata are the context sensitive languages.*

We can now use some of what we know about linear bounded automata to point the way to another language property. Since halting is solvable for deterministic linear bounded automata, might the context sensitive languages be recursive too? Yes, they are. Here's why.

Theorem 3. *Every context sensitive (or type 1) language is recursive.*

Proof. There is a rather elegant proof of this which comes from severval previous theorems. First, recall the theorem which stated that the class of sets accepted by nondeterministic linear bounded automata is closed under complement. This means that the context sensitive languages are closed under complement. Now recall the most recent corollary we have seen. It said that all languages, including the context sensitive languages, are recursively enumerable. So, we can conclude that every context sensitive language and its complement are both recursively enumerable. We round out the proof by recalling another theorem that stated that sets are recursive if and only if they and their complements are recursively enumerable.

Corollary. *Every type 2 and type 3 language is a recursive set.*

Proof. Because the type 3 languages form a subclass of the type 2 languages, which in turn are a subclass of the type 1 languages.

One of the advantages of having a general scheme such as our language and grammar hierarchy is that we can prove things for several of the classes at once. Another nice thing about these grammar schemes is that closure properties are often easier to show than for classes of automata. Here are some examples.

Let us begin with the old classics. Consider the following two grammars:

$$G_i = (N_i , T_i , P_i , S_i) \text{ and}$$
$$G_k = (N_k , T_k , P_k , S_k).$$

If we wish to construct a grammar which generates a language that is the union of the languages generated by G_i and G_k, only the following steps are necessary.

a) Rename the nonterminals so that none appear in both grammars.

b) Combine the sets of terminals, nonterminals, and productions to form these sets for the new grammar.

c) Introduce a new starting nonterminal S, and add the two productions: $S \rightarrow S_i$ and $S \rightarrow S_k$ to those of the original grammars.

d) If $S_i \rightarrow \varepsilon$ or $S_k \rightarrow \varepsilon$ then we add $S \rightarrow \varepsilon$ and take all other epsilon rules out of the grammar.

It should be easy to see that the new grammar does generate all of the strings in both languages except for type 3 languages. There we have a little problem with

the starting productions since they are not of the correct form. This requires a bit more work and is left as an exercise.

Since that was so easy, we will now produce a grammar which does the concatenation of $L(G_i)$ and $L(G_k)$. Here we

a) Rename nonterminals and join the grammars as before.

b) Make a new starting symbol S and add $S \to S_i S_k$ to the production set.

c) If $S_i \to \varepsilon$ then delete this and add $S \to S_j$. Likewise if $S_j \to \varepsilon$.

Again we have a bit of trouble with type 3 grammars since the production in step b above is not of the proper form. Here we must modify the final productions of G_i. This means for each production like $A \to a$ we add one like $A \to aS_k$ to our grammar.

Now let us make a grammar which generates the Kleene closure (or the Kleene star operator) of any language of types 0, 1, and 2 merely by introducing two new starting symbols S' and S'', deleting $S \to \varepsilon$ if it exists, and adding the productions:

$$S'' \to \varepsilon$$
$$S'' \to S'S$$
$$S'' \to S$$
$$S' \to S'S$$
$$S' \to S$$

Now let's sum it all up and state these closure properties as an official theorem.

Theorem 4. *The classes of languages of each type are closed under union, concatenation, and Kleene closure.*

Regular Languages

From the general properties of formal languages presented in the last section we already know a lot about the type 3 or *regular languages*. We know that they are recursive and in fact can be easily recognized by linear bounded automata (since they are a subclass of the type 1 languages). We know also that they are closed under the operations of union, concatenation, and Kleene star.

Now recall the definition of regular sets and some of the characterization results for this class. In particular, we know that the class of regular sets is the smallest class of sets containing the finite sets which is closed under union, concatenation, and Kleene closure. Now note that any finite set can be very easily generated by a regular grammar. For example to generate the set {aa, ab} we use the grammar:

$$S \to aA,\ \ S \to aB,\ \ A \to a,\ \ B \to b$$

This allows us to immediately state a theorem and corollary.

> **Theorem** 1. *Every regular set is a regular (type 3) language.*
>
> **Corollary.** *Sets accepted by finite automata can be generated by regular grammars.*

This is a very good result considering that we do not know exactly how to generate any regular sets, just that they can be generated by some grammar. We do know quite a few things about type 3 languages since they've now been placed in our automata hierarchy as a superclass of the regular sets and a subclass of the context sensitive languages or the class of sets accepted by linear bounded automata. But, we have not even seen one yet and must remedy this immediately. Let's examine one which is very familiar to all computer scientists. Namely numerical constants which are used in programs. These are strings which look like this:

$$-1752.40300E+25$$

There seem to be three parts, an integer, a fraction (or decimal), and an exponent. Let's label these I, F, and X respectively and construct a list of the possible combinations. Some are:

I.F, I, I., and .F

for positive constants with no exponents. Imagine how many combinations there are with signs and exponents!

Now we need a grammar which will generate them. We'll use C as a starting symbol. Next, we note that they all begin with a sign, an integer, or a dot. Here are our beginning productions.

$$C \rightarrow -I \qquad C \rightarrow .F$$
$$C \rightarrow dI \qquad C \rightarrow d$$

(We shall use the terminal symbol d instead of actual digits since we do not want to have to do *all* of the productions involving 0 through 9.) Note that all of these productions begin with a terminal. Also we should note that we could not just use $C \rightarrow I$ since it is not of the proper form.

Next comes the integer part of the constant. This is not hard to do.

$$I \rightarrow dI$$
$$I \rightarrow .F$$
$$I \rightarrow d$$

Note that we allowed the choice of ending the constant at this integer stage or allowing it to progress into the fractional portion of the constant.

Fractional parts of the constant are easily done at this point. The decimal point has already been generated and if desired, the exponential part may come next. These productions are:

$$F \rightarrow dF$$
$$F \rightarrow EA$$
$$F \rightarrow d$$

There might seem to be a production of the wrong kind above in the middle. But this is not so since E is actually a terminal. Also, the nonterminal A has appeared. Its job will be to lead in to exponents. Here is how it does this.

$$A \rightarrow +X \qquad X \rightarrow dX$$
$$A \rightarrow -X \qquad X \rightarrow d$$

Here is the entire grammar as figure 1. It has {d,.,+,-,E} as its alphabet of terminals and {C,I,F,A,X} as the nonterminal set.

C → –I	I → dI	F → dF	A → +X	X → dX
C → dI	I → .F	F → EA	A → –X	X → d
C → .F	I → d	F → d		
C → d				

Figure 1 - Grammar for Constants

It is interesting to see what derivations look like for type 3 grammars. A derivation tree for the string dd.dE-dd appears as figure 2.

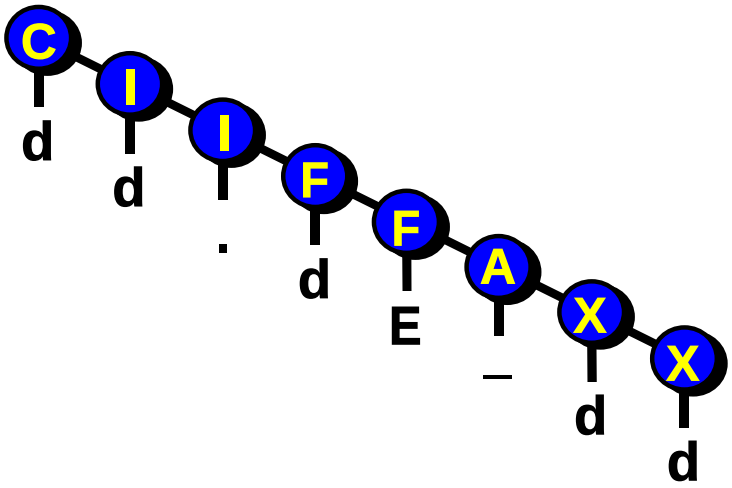


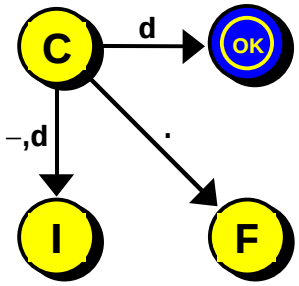
Figure 2 - Regular Grammar Derivation Tree

There is not much in the way of complicated structure here. It is exactly the sort of thing that a finite automaton could handle. In fact, let us now turn the grammar into a finite automaton.

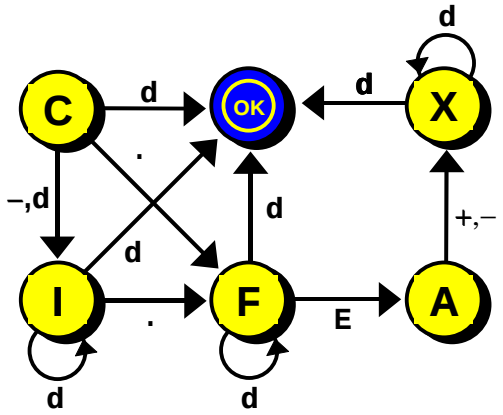
In order to do this conversion we must analyze how a constant is assembled. We know that the derivation begins with a C so we shall use that as a starting state and examine what comes next. This is fairly clear. It must be either a minus sign, a digit, or a dot. Here is a chart.

Production	Action
C → –I	look for a minus and an integer
C → dI	look for a digit and an integer
C → .F	look for a dot and a fraction
C → d	look for a digit and quit

Our strategy will be to go from the state which refers to the part of the constant we're building (i.e. the nonterminal) to the next part of the constant. This is a machine fragment which covers the beginning productions of our grammar for constants.



We should note that we are building a *nondeterministic* finite automaton. The above fragment had a choice when it received a digit (d) as input. To continue, we keep on connecting the states (nonterminals) of the machine according to productions in the grammar just as we did before. When everything is connected according to the productions, we end up with the state diagram of a machine that looks like this:



At this point we claim that a derivation from our grammar corresponds to a computation of the machine. Also we note that there are some missing transitions in the state graph. These were the transitions for symbols which were out of place or errors. For example: strings such as --112, 145.-67, 0.153E-7+42 and the like. Since this *is* a nondeterministic machine, these are not necessary but we shall invent a new state R which rejects all incorrect inputs and put it in the state table below. The states H and R are the halting (we used OK above) and rejecting states.

State	Inputs					Accept?
	d	.	–	+	E	
C	H, I	F	I	R	R	no
I	H, I	F	R	R	R	no
F	H, F	R	R	R	A	no
A	R	R	X	X	R	no
X	H, X	R	R	R	R	no
H	R	R	R	R	R	yes
R	R	R	R	R	R	no

From the above example it seems that we can transform type 3 or regular grammars into finite automata in a rather sensible manner. (In fact, a far easier way than changing regular expressions into finite automata.) It is now time to formalize our method and prove a theorem.

Theorem 2. *Every regular (type 3) language is a regular set.*

Proof Sketch. Let G = (N, T, P, S) be a type 3 grammar. We must construct a nondeterministic finite automaton which will accept the set generated by the language of G. As in our example, let's use the nonterminals as states. Also, we shall add special halting and rejecting states (H and R). Thus our machine is:

$$M = (N \cup \{H, R\}, T, \delta, S, \{H\})$$

which accepts the strings in L(G).

The transition relation δ of M is defined as follows for all states (or nonterminals) A and C as well as all input symbols (or terminals) b.

a) $C \in \delta(A, b)$ iff $A \rightarrow bC$
b) $H \in \delta(A, b)$ iff $A \rightarrow b$
c) $R \in \delta(A, b)$ iff it is not the case that $A \rightarrow bC$ or $A \rightarrow b$
d) $\{R\} = \delta(R, b) = \delta(H, b)$ for all $b \in T$

We need one more note. If there is a production of the form $S \rightarrow \varepsilon$ then S must also be a final state.

This is merely a formalization of our previous example. The machine begins with the starting symbol, and examines the symbols which appear in the input. If they could be generated, then the machine moves on to the next nonterminal (or state).

The remainder of the proof is the demonstration that derivations of the grammar are equivalent to computations of the machine. This is merely claimed here and the result follows.

Another example occurs in figure 3. It contains a grammar which generates 0^*10^* and the finite automaton which can be constructed from it using the methods outlined in the theorem. Note that since there are choices in both states A and S, this is a nondeterministic finite automaton.

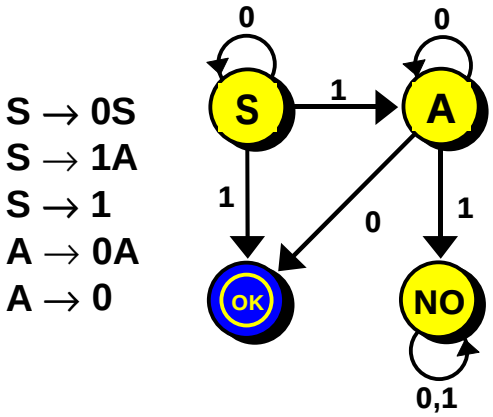


Figure 3 - Automaton and Grammar for 0^*10^*

With this equivalence between the regular sets and the type 3 languages we now know many more things about this class of languages. In particular we know about some sets they cannot generate.

> **Theorem 3.** *The set of strings of the form $a^n b^n$ cannot be generated by a regular (type 3) language.*

> **Corollary.** *The regular (type 3) languages are a proper subclass of the context free (type 2) languages.*

Context Free Languages

Now that we know a lot about the regular languages and have some idea just why they were named that, let us proceed up the Chomsky hierarchy. We come to the type 2 or context free languages. As we recall they have productions of the form $A \rightarrow \alpha$ where α is a nonempty string of terminals and nonterminals and A is a nonterminal.

Why not speculate about the context free languages before examining them with care. We have automata-language pairings for all the other languages. And, there is only one class of machines remaining. Furthermore, the set of strings of the form $a^n b^n$ is context free. So, it might be a good bet that the context free languages have something to do with pushdown machines.

This section will be devoted to demonstrating the equivalence between context free languages and the sets accepted by pushdown automata. Let's begin by looking at a grammar for our favorite context free example: the set of strings of the form $a^n b^n$.

$$S \rightarrow aSB$$
$$S \rightarrow aB$$
$$B \rightarrow b$$

It is somewhat amusing that this is almost a regular grammar. The extra symbol on the right hand side of the first production is what makes it nonregular and that seems to provide the extra power for context free languages.

Now examine the following nondeterministic pushdown machine which reads a's and pushes B's on its stack and then checks the B's against b's.

read	pop	push
a	S	SB
a	S	B
b	B	B

It is obvious that our machine has only one state. It should also be easy to see that if it is presented with a stack containing S at the beginning of the computation then it will indeed accept strings of the form $a^n b^n$ by empty stack (recall that this means that it accepts if the stack is empty when the machine reaches the end of its input string). There is a bit of business with the S on top of the stack while we are reading a's. But this is what tells the machine that

we're on the first half of the input. When the S goes away then we cannot read a's any more.

If there are objections to having a symbol on the stack at the beginning of the computation, we can design the following equivalent two state machine which takes care of this.

state	read	pop	push	goto
1	a		SB	2
	a		B	2
	a	S	SB	2
2	a	S	B	2
	b	B		2

See what we did? Just took all of the instructions which popped an S and duplicated them as the first instruction.

Anyway, back to our primary objective. We've seen a grammar and a machine which are equivalent. And, in fact, the translation was rather simple. The following chart sums it up.

Production	read	pop	push
$A \rightarrow b\alpha$	b	A	α
$A \rightarrow b$	b	A	

So, if we could have grammars with the proper sorts of productions, (namely right hand sides beginning with terminals), we could easily make pushdown machines which would accept their languages. We shall do just this in a sequence of grammar transformations which will allow us to construct grammars with desirable formats. These transformations shall prove useful in demonstrating the equivalence of context free languages and pushdown automata as well as in applications such as parsing.

First we shall consider the simplest kind of production in captivity. This would be one that has exactly one nonterminal on its right hand side. We shall claim that we do not need productions of that kind in out grammars. Then we shall examine some special forms of grammars.

> **Definition.** *A **chain rule** (or singleton production)is one of the form $A \rightarrow B$ where both A and B are nonterminals.*

Theorem 1. (Chain Rule Elimination). *For each context free grammar there is an equivalent one which contains no chain rules.*

The proof of this is left as an exercise in substitution. It is important because it is used in our next theorem concerning an important normal form for grammars. This normal form (due to Chomsky) allows only two simple kinds of productions. This makes the study of context free grammars much, much simpler.

Theorem 2. (Chomsky Normal Form). *Every context free language can be generated by a grammar with productions of the form $A \to BC$ or $A \to b$ (where A, B, and C are nonterminals and b is a terminal).*

Proof. We begin by assuming that we have a grammar with no chain rules. Thus we just have productions of the form $A \to b$ and productions with right hand sides of length two or longer. We need just concentrate on the latter.

The first step is to turn everything into nonterminal symbols. For a production such as $A \to BaSb$, we invent two new nonterminals (X_a and X_b) and then jot down:

$$A \to BX_aSX_b$$
$$X_a \to a$$
$$X_b \to b$$

in our set of new productions (along with those of the form $A \to b$ which we saved earlier).

Now the only offending productions have all nonterminal right hand sides. We shall keep those which have length two right hand sides and modify the others. For a production such as $A \to BCDE$, we introduce some new nonterminals (of the form Z_i) and do a cascade such as:

$$A \to BZ_1$$
$$Z_1 \to CZ_2$$
$$Z_2 \to DE$$

Performing these two translations on the remaining productions which are not in the proper form produces a Chomsky normal form grammar.

Here is the translation of a grammar for strings of the form a^nb^n into Chomsky Normal form. Two translation steps are performed. The first changes terminals to X_i's and the second introduces the Z_i cascades which make all productions the proper length.

Original	⇒	Rename	⇒	Cascade
$S \to aSb$		$S \to X_aSX_b$		$S \to X_aZ_1$ $Z_1 \to SX_b$
$S \to ab$		$S \to X_aX_b$ $X_a \to a$ $X_b \to b$		$S \to X_aX_b$ $X_a \to a$ $X_b \to b$

Chomsky normal form grammars are going to prove useful as starting places in our examination of context free grammars. Since they are very simple in form, we shall be able to easily modify them to get things we wish. In addition, we can use them in proofs of context free properties since there are only two kinds of productions.

Although this is still not what we are looking for, it *is* a start. What we really desire is another normal form for productions named *Greibach Normal Form*. In this, all of the productions are of the form $A \to b\alpha$ where α is a (possibly empty) string of nonterminals.

Starting with a Chomsky normal form grammar, we need only to modify the productions of the form $A \to BC$. This involves finding out what strings which begin with terminals ate generated by B. For example: suppose that B generates the string $b\beta$. Then we could use the production $A \to b\beta C$. A translation technique which helps do this is *substitution*. Here is a formalization of it.

> **Substitution.** Consider a grammar which contains a production of the form $A \to B\alpha$ where A and B are nonterminals and α is a string of terminals and nonterminals. Looking at the remainder of the grammar containing that production, suppose that:
>
> $$B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$
>
> (where the β_i are strings of terminals and nonterminals) is the collection of all of the productions which have B as the left hand side. We may replace the production $A \to B\alpha$ with:
>
> $$A \to \beta_1\alpha \mid \beta_2\alpha \mid \ldots \mid \beta_n\alpha$$
>
> without changing the language generated by the grammar.

This is very promising since all we need do is substitute until terminal symbols pop up at the beginning of the right hand sides in all of our productions. Unfortunately, this is easier said than done! The thing that shall provide

problems is recursion. A production such as $A \rightarrow AB$ keeps producing A's at the front of the string when we expand it. So, we must remove this kind of recursion from our productions.

Definition. *A production of the form $A \rightarrow A\alpha$ is **left recursive**.*

Theorem 3 (Left Recursion Removal). *Any context free language can be generated by a grammar which contains no left recursive productions.*

Proof Sketch. Accomplishing this is based upon the observation that left recursive productions such as $A \rightarrow A\beta$ generate strings of the form $A\beta^*$. Here is what we do. First, divide the productions which have A on the left hand side into two groups:

1) $A \rightarrow A\beta_i$, and
2) $A \rightarrow \alpha_j$ where α_j does not start with an A.

We retain the $A \rightarrow \alpha_j$ type productions since they're not left recursive. Each production of the form $A \rightarrow A\beta_i$ is replaced by a set of productions which generate $\alpha_j\beta_i^*$. A new nonterminal 'A' comes forth to aid in this endeavor. Thus for all α_i and β_j we produce:

$$A \rightarrow \alpha_j\ A'$$
$$A' \rightarrow \beta_i$$
$$A' \rightarrow \beta_i\ A'$$

and add them to our rapidly expanding grammar.

Since neither the α_j nor the β_i can begin with an A', none of the productions in the above group are left recursive. Noticing that A does now generate strings of the form $\alpha_j\beta_i^*$ completes the proof.

Here is an example. Suppose we begin with the Chomsky Normal form grammar fragment:

$$A \rightarrow AB$$
$$A \rightarrow AC$$
$$A \rightarrow DA$$
$$A \rightarrow a$$

and divide it into the two groups indicated by the construction in the left recursion removal theorem. We now have:

$\mathbf{A \to A\beta}$	$\mathbf{A \to \alpha}$
$A \to AB$	$A \to DA$
$A \to AC$	$A \to a$

The β_i mentioned in the proof are B and C. And the α_j are DA and a.
Now we retain the $A \to \alpha$ productions and build the three new groups mentioned in the proof to obtain:

$\mathbf{A \to \alpha}$	$\mathbf{A \to \alpha A'}$	$\mathbf{A' \to \beta}$	$\mathbf{A' \to \beta A'}$
$A \to DA$	$A \to DAA'$	$A' \to B$	$A' \to B'$
$A \to a$	$A \to aA'$	$A' \to C$	$A' \to CA'$

That removes *immediate* left recursion. But we're not out of the woods quite yet. There are more kinds of recursion. Consider the grammar:

$$A \to BD$$
$$B \to CC$$
$$C \to AB$$

In this case A can generate ABCD and recursion has once more caused a difficulty. This *must* go. Cyclic recursion will be removed in the proof of our nest result, the ultimate normal form theorem. And, as a useful byproduct, this next theorem provides us with the means to build pushdown machines from grammars.

Theorem 4. (Greibach Normal Form). *Every context free language can be generated by a grammar with productions of the form $A \to a\alpha$ where a is a terminal and α is a (possibly empty) string of nonterminals.*

Proof Sketch. Suppose we have a context free grammar which is in Chomsky normal form. Let us rename the nonterminals so that they have a nice ordering. In fact, we shall use the set $\{A_1, A_2, \ldots, A_n\}$ for the nonterminals in the grammar we are modifying in this construction.

We first change the productions of our grammar so that the rules with A_i on the left hand side have either a terminal or some A_k where $k > i$ at the beginning of their right hand sides. (For example: $A_3 \to A_6\alpha$, but not $A_2 \to A_1\alpha$.) To do this we start with A_1 and keep on going until we reach A_n rearranging things as we go. Here's how.

Assume that we have done this for all of the productions which have A_1 up through A_{i-1} on their left hand sides. Now we take a production involving A_i. Since we have a Chomsky normal form grammar, it will be

of the form $A_i \rightarrow b$, or $A_i \rightarrow A_k B$. We need only change the second type of production if $k \leq i$.

If $k < i$, then we apply the *substitution* translation outlined above until we have productions of the form:

$$A_i \rightarrow a\beta, \text{ or}$$
$$A_i \rightarrow A_j\beta \text{ where } j \geq i.$$

(Note that no more than i-1 substitution steps need be done.) At this point we can use the left recursion removal technique if $j = i$ and we have achieved our first plateau.

Now let us see what we have. Some new nonterminals (A_i') surfaced during left recursion removal and of course we have all of our old terminals and nonterminals. But all of our productions are now of the form:

$$A_i \rightarrow a\alpha,\ A_i \rightarrow A_k\alpha, \text{ or } A_i' \rightarrow \alpha$$

where $k > i$ and α is a string of old and new nonterminals. (This is true because we began with a Chomsky normal form grammar and had no terminal symbols on the inside. This is intuitive, but quite nontrivial to show!)

An aside. We are in very good shape now because recursion can *never* bother us again! All we need do is convince terminal symbols to appear at the beginning of every production.

The rest is all downhill. We'll take care of the $A_i \rightarrow A_k\alpha$ productions first. Start with $i = n-1$. (The rules with A_n on the left *must* begin with terminals since they do not begin with nonterminals with indices less than n and A_n is the last nonterminal in our ordering.) Then go backwards using *substitution* until the productions with A_1 on the left are reached. Now all of our productions which have an A_i on the left are in the form $A_i \rightarrow a\alpha$.

All that remain are the productions with the A_i' on the left. Since we started with a Chomsky normal form grammar these must have one of the A_i at the beginning of the right hand side. So, substitute. We're done.

That was rather quick and seems like a lot of work! It is. Let us put all of the steps together and look at it again. Examine the algorithm presented in figure 1 as we process an easy example.

GreibackConversion(G)
Pre: *G is a Chomsky Normal Form Grammar*
Post: *G is an equivalent Greibach Normal Form Grammar*

Rename nonterminals as $A_1, A_2, \ldots, A_n$
for **i := 1** ***to*** **n** ***do***
  for **each production with A_i on the left hand side** ***do***
    while **$A_i \rightarrow A_k\alpha$** ***and*** **k < i substitute for A_k**
    if **$A_i \rightarrow A_i\alpha$** ***then*** **remove left recursion**
for **i := n-1** ***downto*** **1** ***do***
  for **each production $A_i \rightarrow A_k\alpha$ substitute for A_k**
for **each $A_i' \rightarrow A_k\alpha$ substitute for A_k**

Figure 1 - Greibach Normal Form Conversion

Consider the Chomsky normal form grammar

$S \rightarrow AB$
$A \rightarrow a \mid SA$
$B \rightarrow b \mid SB$

We order the nonterminals S, A, and B. The first step is to get the right hand sides in descending order. $S \rightarrow AB$ is fine as is $A \rightarrow a$. For $A \rightarrow SA$ we follow the steps in the first for loop and transform this production as indicated below:

Original	⇒	Substitute	⇒	Remove Recursion
$A \rightarrow SA$		$A \rightarrow ABA$		$A \rightarrow aA'$
				$A' \rightarrow BA$
				$A' \rightarrow BAA'$

To continue, $B \rightarrow b$ is what we want, but $B \rightarrow SB$ needs some substitution.

Original	⇒	Substitute	⇒	Substitute
$B \rightarrow SB$		$B \rightarrow ABB$		$B \rightarrow aBB$
				$B \rightarrow aA'BB$

Now we execute the remaining for loops in the algorithm and use substitution to get a terminal symbol to lead the right hand side on all of our productions. The right column shows the final Greibach normal form grammar.

Original	⇒	Substitute
$B \rightarrow aBB$		$B \rightarrow aBB$
$B \rightarrow aA'BB$		$B \rightarrow aA'BB$
$A \rightarrow a$		$A \rightarrow a$
$A \rightarrow aA'$		$A \rightarrow aA'$
$S \rightarrow AB$		$S \rightarrow aB$
		$S \rightarrow aA'B$
$A' \rightarrow BA$		$A' \rightarrow aBBA$
		$A' \rightarrow aA'BBA$
$A' \rightarrow BAA'$		$A' \rightarrow aBBAA'$
		$A' \rightarrow aA'BBAA'$

Granted, Greibach normal form grammars are not always a pretty sight. But, they do come in handy when we wish to build a pushdown machine which will accept the language they generate. If we recall the transformation strategy outlined at the beginning of the section we see that these grammars are just what we need. Let's do another example. The grammar:

$S \rightarrow cAB$
$A \rightarrow a \mid aBS$
$B \rightarrow b \mid bSA$

can be easily transformed into the one state, nondeterministic machine:

Read	Pop	Push
c	S	AB
a	A	
a	A	BS
b	B	
b	B	SA

which starts with S on its stack and accepts an input string if its stack is empty after reading the input.

Production	read	pop	push
$A \rightarrow a$	a	A	
$A \rightarrow a\alpha$	a	A	α

Machine

It should be quite believable that Greibach normal form grammars can beeasily transformed into pushdown machine. The following chart depicts the formalization for the general algorithm used in this transformation.

Let's add a bit of ammunition to our belief that we can change grammars into machines. Examine a leftmost derivation of the string cabcabb by the latest grammar and compare it to the computation of the pushdown machine as it recognizes the string.

Grammar	Machine	
Derivation	Input Read	Stack
S	ε	S
cAB	c	AB
caBSB	ca	BSB
cabSB	cab	SB
cabcABB	cabc	ABB
cabcaBB	cabca	BB
cabcabB	cabcab	B
cabcabb	cabcabb	ε

Note that the leftmost derivation of the string *exactly* corresponds to the input read so far plus the content of the machine's stack. Whenever a pushdown machine is constructed from a Greibach normal form grammar this happens. Quite handy! This recognition technique is known as *top-down parsing* and is the core of the proof of our next result.

> **Theorem 5.** *Every context free language can be accepted by a nondeterministic pushdown automaton.*

From examining the constructions which lead to the proof of our last theorem we could come to the conclusion that one state pushdown machines are equivalent to context free languages. But of course pushdown automata can have many states. Using nondeterminism it is possible to turn a multistate automaton into a one state machine. The trick is to make the machine *guess which state it will be in when it pops each symbol off of its stack*. We shall close this section by stating the last part of the equivalence and leave the proof to a more advanced text on formal languages.

> **Theorem 6.** *Every set accepted by a pushdown automaton can be accepted by a one state pushdown machine.*

> **Theorem 7.** *The class of languages accepted by pushdown machines is the class of context free languages.*

Parsing and Deterministic Languages

We have noted the usefulness of finite automata (for circuit models, pattern recognition, and lexical scanning) and linear bounded automata (as computer models). But we have not discussed one of the most important areas of computing, namely *translation*. Therefore, we shall now turn our attention to the role of context free languages and pushdown automata in translation, compiling, and parsing.

As seen in an early example in this chapter, assignment statements may be generated by context free grammars. In fact, *most of the syntax for programming languages is context free.* Thus, if we can construct grammars for programming languages, we should be able to use pushdown automata to recognize correct programs and aid in their translation into assembly language.

In order to recognize programs with correct syntax all we need to do is write down a grammar that is able to generate all correct programs and then build a pushdown machine that will recognize this language. Actually, we know exactly how to build the machine. We convert the grammar to Greibach normal form and jot down the one state pushdown machine that recognizes strings from the language which are generated by the grammar. Unfortunately, there is one small problem. Machines produced from Greibach normal form grammars are often *nondeterministic.* This removes the utility from the process since we cannot convert these machines into the kind of programs we are used to writing. (After all, we normally do not do nondeterministic programming on purpose.)

Here is a small aside. We should note that we know that the context free languages *are* recursive and thus recognizable by programs which always halt. So if we desired, we could have the program do some backtracking and go through all possible derivations described by the grammar. But unfortunately this makes our computation time somewhere between n^2 and n^3 steps when recognizing strings of length n. And, we really do not want to use the entire class of context free languages, only the deterministic ones.

This is exactly what we shall do. By resorting to the same strategy we used to get away from unsolvability we can eliminate nondeterministic languages by simplifying the grammars we design. Here is the first step.

> **Definition.** *A context free grammar is an **s-grammar** if and only if every production's right hand side begins with a terminal symbol and this terminal is different for any two productions with the same left-hand side.*

This looks good. We can easily build machines for these grammars and can count on them being deterministic. Let us try an example. Our old and trusted friend:

$$S \rightarrow aSb$$
$$S \rightarrow ab$$

It is of course not in the necessary form, but with the operation shown in figure 1, we can fix that.

Given a set of productions of the form:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$$

invent a new nonterminal Z and replace this set by the collection:

$$A \rightarrow \alpha Z$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Figure 1 - Factoring

We shall now try this on the above grammar for $a^n b^n$. The α is of course just the terminal symbol a while the β's are Sb and b. After factoring we come up with the following grammar.

$$S \rightarrow aA$$
$$A \rightarrow Sb$$
$$A \rightarrow b$$

This is not quite what we want, but substituting for S will allow us to write down the following equivalent s-grammar.

$$S \rightarrow aA$$
$$A \rightarrow aAb$$
$$A \rightarrow b$$

Designing machines for these grammars is something we have done before. For a production of the form $A \rightarrow a\alpha$ we just:

read the a, pop the A, and push α onto the stack.

The machine accepts by empty stack at the end of the input. There is one minor adjustment that must be made. We do not wish terminal symbols to be pushed onto the stack. We can prevent this by presenting the s-grammar in Greibach normal form by turning all terminals that do not begin the right-hand

side into nonterminals. Or, we could modify the pushdown machine so that it just pops a terminal whenever it is on top of the stack and on the input string.

Thus in designing the pushdown machine, we always push the tail of a production's right hand side and pop the nonterminal on the left whenever we read the proper symbol. We also pop terminals when they appear on top of the stack and under the reading head at the same time.

Let us continue. At times factoring produces productions which are not compatible with s-grammars. For example, the fragment from a grammar for arithmetic assignment statements

$$E \rightarrow T \mid T+E$$

becomes the following when we factor it.

$$E \rightarrow TZ$$
$$Z \rightarrow \varepsilon \mid +E$$

and we can substitute for the T until we get terminals leading all of the right-hand sides of the productions. Our above example contains something we have not considered, an epsilon rule ($Z \rightarrow \varepsilon$). This was not exactly what we wished to see. In fact it messes up our parser. After all, just how do we read nothing and pop a Z? And when do we know to do it? The following definitions lead to an answer to this question.

> **Definition.** *The **select set** for the production $A \rightarrow a\alpha$ (which is written SELECT($A \rightarrow a\alpha$)) where A is a nonterminal and α is a possibly empty string of terminal and nonterminial symbols is the set {a}.*

> **Definition.** *A context free grammar is a **q-grammar** if and only if every production's right hand side begins with a terminal symbol or is ε, and whenever two productions possess the same left-hand side they have different select sets.*

Thus if we have productions with matching left-hand sides, they must behave like s-grammars, and the select sets guarantee this. Select sets solve that problem, but what exactly is the select set for an epsilon rule? It should be just the terminal symbol we expect to see next. If we knew what should follow the A in the epsilon rule $A \rightarrow \varepsilon$ then we would know when to pop the A without using up an input symbol. The next definitions quickly provide the precise way to start setting this up.

Definition. *The **follow set** for the nonterminal A (written FOLLOW(A)) is the set of all terminals a for which some string $\alpha A a \beta$ can be derived from the starting symbol S. (Where α and β are possibly empty strings of both terminals and nonterminals.)*

Definition. *SELECT($A \rightarrow \varepsilon$) = FOLLOW(A).*

That was not so difficult. We merely apply an epsilon rule whenever a symbol that follows the left-hand side nonterminal comes along. This makes a lot of sense. There is one small catch though. We *must not advance* past this symbol on the input string at this time. Here is our first example in a new, elegant form since it now contains an epsilon rule:

$$S \rightarrow \varepsilon \mid aSb$$

It is easy to see that the terminal symbol b must follow the S since S remains between the a's and b's until it disappears via the epsilon rule. A machine constructed from this grammar appears as figure 2.

read	*pop*	*push*	*advance?*
a	**S**	**Sb**	**yes**
b	**S**		**no**
b	**b**		**yes**

Figure 2 - A Parser for a q-grammar

Note that technically we are no longer designing pushdown machines. Actually we have designed a *top-down parser*. The difference is that parsers can advance along the input string (or not) as they desire. But we should note that they still are really pushdown automata.

Since our last example was a bit different than a pushdown automaton, let us examine the computation it goes through when presented with the string aabb. In this example, the stack bottom is to the left and the blue, italic input symbols have been read.

Stack	*Input*	*Action*
S	aabb	Apply S $\rightarrow$ aSb
bS	*a* abb	Apply S $\rightarrow$ aSb
bbS	*aa* bb	Apply S $\rightarrow \varepsilon$
bb	*aa* bb	Verify b
b	*aab* b	Verify b
	aabb	Accept

Another way to look at this computation is to note that a leftmost derivation of aabb took place. In fact, if we were to concatenate the input read and the stack up to the point where the b's were verified, we would have the following leftmost derivation.

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

So, a top-down parser executes and verifies a leftmost derivation based on the input and stack symbols it sees along the way. We shall explore this phenomenon later.

Let us now go one step farther. Why require a production to have a right hand beginning with a terminal? Why not allow productions of the form:

$$E \rightarrow TZ$$

that we got when we factored the above some productions for arithmetic expressions earlier? This seems more intuitive and natural. Besides, it is no fun transforming grammars into Greibach Normal Form. Things would be far easier if we had a way of predicting when to apply the rule as we did for epsilon rules. Modifying our select set definition a little helps since it allows select sets to indicate when to apply productions.

> **Definition.** *The **select set** for the production $A \rightarrow \alpha$ (which is written as SELECT($A \rightarrow \alpha$)) where A is a nonterminal and α is a string of terminal and nonterminial symbols is the set of all terminal symbols which begin strings derived from α.*

This tells us that if our pushdown machine has an E on its stack and is reading a symbol from SELECT($E \rightarrow TZ$), then that is the production that should be applied by pushing TZ on the stack. Here is a new class of deterministic context free grammars based upon select sets.

> **Definition.** *A context free grammar is an **LL(1) grammar** if and only if two productions with the same left-hand sides have different select sets.*

Two items need to be cleared up. First, LL(1) means that the parser we build from the grammar is processing the input string from *left to right* (the first L) using *leftmost derivations* (the second L). This is exactly what the parsers we have been designing have been doing. They create a leftmost derivation and check it against the input string. In addition we *look ahead* one symbol. By that we mean that the parser can examine the next symbol on its input string without advancing past it. The parser in figure 2 is an example of this.

Next, we have neglected to fully define select sets for arbitrary productions. Intuitively, this will be all of the terminal symbols first produced by the production, which we call its FIRST set. If the production can lead to the empty string, then the FOLLOW set of the left-hand side nonterminal must be included. Here are the formal definitions.

> **Definition.** *FIRST($A \rightarrow \alpha$) is the set of all terminal symbols a such that some string of the form $a\beta$ can be derived from α.*

> **Definition.** *SELECT($A \rightarrow \alpha$) contains FIRST($A \rightarrow \alpha$). If ε can be derived from α then it also contains FOLLOW(A).*

It is time for a theoretical aside. The proof of this next theorem is left as an exercise in careful substitution.

> **Theorem 1.** *The class of languages generated by q-grammars is the same as the class of LL(1) languages.*

Our next step is to work out computing procedures for FIRST, FOLLOW, and SELECT. We have defined them, but have not considered just how to find them. We begin by finding out which nonterminals can generate the empty string ε using the algorithm of figure 3. (In the following algorithms, capital Roman letters are nonterminals, small Roman letters from the beginning of the alphabet are terminals, x can be either, and Greek letters (except for ε) are strings of both.)

```
NullFinder(G, E)
PRE:   G = (N, T, S, P) is a context free grammar
POST:  E = set of nonterminals which generate ε

delete all productions containing terminals from P
E := ∅
repeat
  for each epsilon rule A → ε in P
      add A to E
      delete all A → α productions from P
      delete occurrences of A in all productions
until no more epsilon rules remain in P
```

Figure 3 - Finding nullable nonterminals

Let's pause a moment and insure that the algorithm for finding nonterminals which generate the null string is correct. We claim that it terminates because one nonterminal is deleted every time the loop is executed. Correctness comes

from thinking about just how a nonterminal A could generate the null string ε. There must be a production $A \rightarrow \alpha$ where all of the nonterminals in α in turn generate ε. And one of them must generate ε directly.

Computing FIRST sets starts with a relation named BEGIN that is computed according to the recipe:

for all productions: $A \rightarrow \alpha x \beta$
    if ε can be derived from α then $x \in$ BEGIN(A)

(Note that x is either a terminal or nonterminal and α is a possibly empty string of nonterminals.) Then FIRST(A) is merely the set of terminal symbols in the reflexive, transitive closure of BEGIN(A). Now we have all of the terminals that show up at the beginning of strings generated by each nonterminal. (For completeness we should add that for a terminal symbol a, FIRST(a) = {a}.) And at last we have:

for all productions $A \rightarrow \alpha$
    if $\alpha = \beta x \gamma$ and ε can be derived from β
        then FIRST($A \rightarrow \alpha$) contains FIRST(x)

Some more relations are needed to do FOLLOW. Let us begin. We need to detect symbols that come after others and which end derivations.

for all productions: $A \rightarrow \alpha A \beta x \gamma$
    if ε can be derived from β then $x \in$ AFTER(A)

for all productions: $A \rightarrow \alpha x \beta$
    if ε can be derived from β then $x \in$ END(A)

So, AFTER(A) is all of the symbols that immediately follow A. If we set LAST(A) to be the reflexive, transitive closure of END(A) then it is the set of symbols which end strings generated from A.

Now, watch closely. FOLLOW(A) is the set of all terminal symbols a such that there exist symbols B and x where:

$A \in$ LAST(B), $x \in$ AFTER(B), and $a \in$ FIRST(x)

Try to write that out in prose. Best way to be sure that it is correct and to understand the computing procedure. For now, let us do an example. How about one of the all time favorite grammars for arithmetic expressions (which we factor on the right):

$E \rightarrow T \mid T{+}E$	$E \rightarrow TA$
	$A \rightarrow {+}E \mid \varepsilon$
$T \rightarrow F \mid F{*}T$	$T \rightarrow FB$
	$B \rightarrow {*}T \mid \varepsilon$
$F \rightarrow x \mid (E)$	$F \rightarrow x \mid (E)$

Here are all of the relations that we described above for the nonterminals of the factored grammar.

	E	A	T	B	F
BEGIN	T	+	F	*	x, (
FIRST	x, (	+	x, (	*	x, (
AFTER	)		A		B
END	T, A	E	F, B	T	x,)
LAST	E,A,T,B,F,x,)	E,A,T,B,F,x,)	T,B,F,x,)	T,B,F,x,)	F,x,)
FOLLOW	)	+,)	+,)	+, *,)	)

Putting this all together we arrive at the following select sets and are able to construct a parser for the grammar. The general rule for LL(1) parsers is to pop a production's left-hand side and push the right hand side when reading the select set. Here is our example.

Production	Select Set
$E \rightarrow TA$	{ x, (}
$A \rightarrow {+}E$	{ + }
$A \rightarrow \varepsilon$	{) }
$T \rightarrow FB$	{ x, (}
$B \rightarrow {*}T$	{ * }
$B \rightarrow \varepsilon$	{ +,) }
$F \rightarrow x$	{ x }
$F \rightarrow (E)$	{ (}

Parser

read	pop	push	advance?
x,(	E	TA	no
+	A	E	yes
)	A		no
x,(	T	FB	no
*	B	T	yes
+,)	B		no
x	F		yes
(	F	E)	yes
)	)		yes

That wasn't so bad after all. We did a lot of work, but came up with a deterministic parser for a LL(1) grammar that describes arithmetic expressions for a programming language. One more note on top-down parsers though. They're usually presented with instructions to either pop the top stack symbol (for an epsilon rule) or to replace the top stack symbol with a string (for other productions). No advance of the input is made in either case. It is always implicitly assumed that when the top of the stack matches the input that the

parser pops the symbol and advances. This is a *predict* and *verify* kind of operation based on leftmost derivations.

Often parsers are presented in a slightly different format. All of the *match* or *verify* operations with terminals on the input and stack are understood and a *replace table* is provided as the parser. This table shows what to place on the stack when the top symbol and the next input symbol are given. Here is our last example in this form.

Stack	Input Symbol				
	+	*	x	(	)
E			TA	TA	
A	+E				pop
T			FB	FB	
B	pop	*T			pop
F			x	(E)	

See how the two formats describe the same parser? In the new from we just examine the input and the stack and replace the stack symbol by the string in the appropriate box of the parser table. By the way, the blank boxes depict configurations that should not occur unless there is an error. In a real parser one might have these boxes point to error messages. Quite useful.

At this point we have defined a subclass (in fact a proper one) of the deterministic context free languages. And, we know how to build parsers for their grammars and check to see if the grammars are in the proper form. We even have some tools that can be used to place grammars in the correct form if needed. These are:

- omitting useless nonterminals,
- omitting unreachable nonterminals,
- factoring,
- substitution, and
- recursion removal.

A word about recursion. It should be obvious that left recursion (immediate or cyclic) is not compatible with top-down parsing. The following result should reinforce this.

Theorem 2. *Every LL(1) language has a nonrecursive grammar.*

Proof. Assume that there is an LL(1) language whose grammars all have recursion. This means that in each LL(1) grammar for the language there is a nonterminal A such that $A\alpha$ can be derived from it. Let us further assume that A is not a useless nonterminal.

This means that there is a sequence of nonterminals $X_1, X_2, \ldots, X_k$ such that for some sequence of strings $\beta_1, \beta_2, \ldots, \beta_k$:

$$A \Rightarrow X_1\beta_1 \Rightarrow X_2\beta_2 \Rightarrow \ldots \Rightarrow X_k\beta_k \Rightarrow A\alpha.$$

Take the nonterminals in the above sequence (A and the X_i). Now examine the select sets for all productions where they form the left-hand side. Each of these select sets must have FIRST(A) in it. (And, FIRST(A) is not empty since A is not useless.) We claim that either all of these nonterminals are useless or that at least one of them forms the left side of more than one production. Since A is not useless, then there must be at least two productions with the same left side and overlapping select sets. Thus the grammar is not LL(1).

So, if we wish to translate or parse something here is the recipe:

Develop a grammar for it,
Convert the grammar to LL(1) form, and
Construct the parser.

This always brings success since LL(1) grammars provide parsers which are nonrecursive and deterministic.

Unfortunately this is not always possible. In fact, we have no way of knowing whether a grammar can be placed in LL(1) form. This can be seen from the following sequence of results that terminate with our old nemesis unsolvability. (The second result, theorem 4, is left for an advanced treatment of formal languages.)

Theorem 3. *An arbitrary Turing machine's set of valid computations is a context free language if and only if the machine halts for only a finite set of inputs.*

Proof. Left as an exercise in use of the pumping lemma.

Theorem 4. *The class of deterministic context free languages is closed under complement.*

Theorem 5. *It is unsolvable whether an arbitrary context free language is:*
a) a regular set,
b) an s-language,
c) an LL(1) language, or
d) a deterministic context free language.

Proof. We shall reduce the finiteness problem for Turing machines to the problem of deciding whether or not a context free language is in a subclass that is closed under complement.

Take an arbitrary Turing machine M_i and construct the context free language $L_{g(i)}$ that is the set of invalid computations for the Turing machine. Now examine the complement of $L_{g(i)}$: the Turing machine's valid computations. If the Turing machine halted for only a finite number of inputs then the set of valid computations is a regular set (and thus also a deterministic context free language, or LL(1) language, or s-language). Both $L_{g(i)}$ and its complement are in all of these subclasses of the context free languages. That is:

$T(M_i)$ is finite iff { valid computations } is finite
iff the complement of $L_{g(i)}$ is regular
iff $L_{g(i)}$ is regular

Thus being able to decide whether an arbitrary context free language is in one of these subclasses allows us to decide finiteness from recursively enumerable sets.

Another problem with deterministic parsing occurs when we wish to know how a statement was formed. If we examine the grammar:

$$S \rightarrow V = E$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow V$$
$$V \rightarrow x \mid y \mid z$$

we find that it generates all of the assignment statements generated by the grammar at the beginning of this chapter. But unfortunately it is ambiguous. For example, the statement:

$$x = y + z*x$$

can be generated by the two rightmost derivations in figure 4. Note that in the tree on the left an expression is generated which must be evaluated with the

multiplication first as x = y + (z∗x) while that on the right generates one which would be evaluated as x = (y + z)∗x.

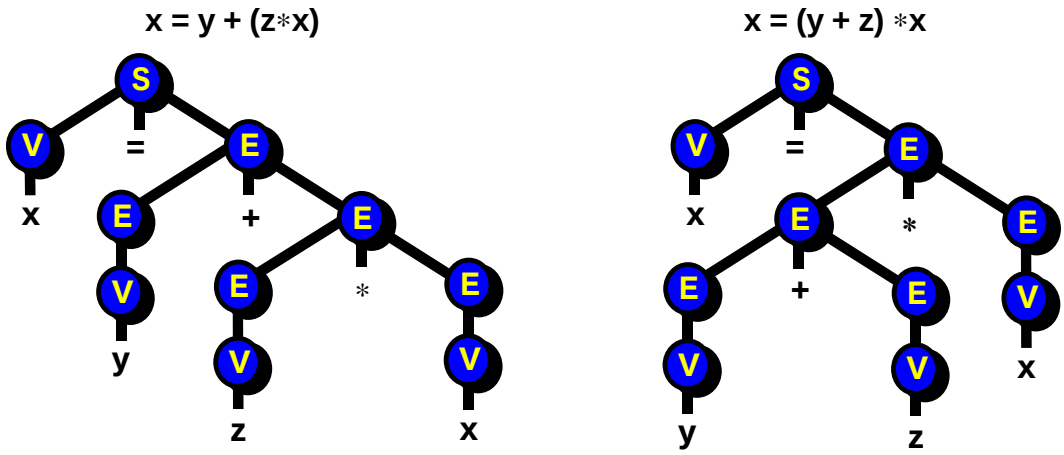


Figure 4 - Ambiguous Rightmost Derivations

Thus, it is not clear whether we should add or multiply first when we execute this statement. In our previous work with LL(1) grammars this was taken care of for us - none of them could be ambiguous! But since some languages *always* have ambiguous grammars (we call these languages *inherently ambiguous*) we need to be careful.

A shortcoming of the class of LL(1) languages we should mention is that they do not include all of the deterministic context free languages. We might think to extend our lookahead set and use LL(k) parsers by looking into a window k symbols wide on the input as we parse. But this leads to very large parser tables. Besides, applications that require more than one symbol of lookahead are scarce. And anyway, the LL(k) languages do not include all of the deterministic context free languages.

In our quest for all of the deterministic context free languages, we shall turn from top-down or predictive parsing to the reverse. Instead of predicting what is to come and verifying it from the input we shall use a bottom-up approach. This means that rather than beginning with the starting symbol and generating an input string, we shall examine the string and attempt to work our way back to the starting symbol. In other words, we shall reconstruct the parse. We will process the input and decide how it was generated using our parser stack as a notebook. Let us begin by examining a string generated by (you guessed it!) the grammar:

$$A \rightarrow aB$$
$$B \rightarrow Ab$$
$$B \rightarrow b$$

The following chart provides an intuitive overview of this new approach to parsing. Here the string aabb is parsed in a bottom-up manner. We shall discuss the steps after presenting the chart.

Step	Stack	Examine	Input	Maybe	Action
0			aabb		
1		a	abb	A → aB	push
2	a	a	bb	A → aB	push
3	aa	b	b	B → b	apply
4	aa	B	b	A → aB	apply
5	a	A	b	B → Ab	push
6	aA	b		B → Ab	apply
7	a	B		A → aB	apply
8		A			accept

In step 1 we moved the first input symbol (a) into the examination area and guessed that we might be working on A → aB. But, since we had not seen a B we put off making any decisions and pushed the a onto the stack to save it for awhile. In step 2 we did the same. In step three we encountered a b. We knew that it could come from applying the production B → b. So we substituted a B for the b. (Remember that we are working backwards.) In step 4 we looked at the stack and sure enough discovered an a. This meant that we could surely apply A → aB, and so we did. (We moved the new A in to be examined after getting rid of the B which was there as well as the a on the stack since we used them to make the new A.) In step 5 we looked at the stack and the examination area, could not decide what was going on (except something such as B → Ab possibly), so we just put the A onto the stack. Then in step 6 we looked at the b in the examination area and the A on the stack and used them both to make a B via the production B → Ab. This B entered the examination area. Looking at this B and the stacked a in step 7 we applied A → aB to them and placed an A in the examination area. Since nothing was left to do in step 8 and we were examining our starting symbol we accepted.

See what happened? We looked at our input string and whenever we could figure out how a symbol was generated, we applied the production that did it. We in essence worked our way *up* the derivation tree. And, we used the stack to save parts of the tree to our left that we needed to tie in later. Since our grammar was unambiguous and deterministic, we were able of do it.

Now let us do it all over with some new terminology and some mixing up of the above columns. When we push an input symbol into the stack we shall call it a *shift*. And when we apply a production we shall call it a *reduce* operation. We

shall *shift* our guesses onto the stack with input symbols. For example, if we see an a and guess that we're seeing the results of applying the production A → aB, we shift the pair (a,aB) onto the stack. After we *reduce*, we shall place a guess pair on the stack with the nonterminal we just produced. Here we go.

Step	Stack	Input	Action
0		aabb	shift(aB)
1	(a,aB)	abb	shift(aB)
2	(a,aB)(a,aB)	bb	shift(b)
3	(a,aB)(a,aB)(b,b)	b	reduce(B → b)
4	(a,aB)(a,aB)(B,aB)	b	reduce(A → aB)
5	(a,aB)(A,Ab)	b	shift(Ab)
6	(a,aB)(A,Ab)(b,Ab)		reduce(B → Ab)
7	(a,aB)(B,aB)		reduce(A → aB)
8	(A,)		accept

Our new parsing technique involves keeping notes on past input on the stack. For instance, in step 5 we have an a (which might be part of an aB) at the bottom of our stack, and an A (which we hope shall be part of an Ab) on top of the stack. We then use these notes to try to work backwards to the starting symbol. This is what happens when we do reduce operations. This is the standard bottom-up approach we have always seen in computer science. Our general method is to do a rightmost derivation except that we do it backwards! Neat. What we did at each step was to examine the stack and see if we could do a reduction by applying a production to the top elements of the stack. If so, then we replaced the right hand side symbols (which were at the top of the stack) with the left-hand side nonterminal.

After doing a reduction we put the new nonterminal on the stack along with a guess of what was being built. We also did this when we shifted a terminal onto the stack. Let us examine these guesses. We tried to make them as accurate as possible by looking at the stack before pushing the (symbol, guess) pair. We should also note that the pair (a,aB) means that we have placed the a on the stack and think that maybe a B will come along. On the other hand, the pair (b,Ab) indicates that the top two symbols on the stack are A and b, and, we have seen the entire right hand side of a production. Thus we always keep track of what is in the stack.

Now for another enhancement. We shall get rid of some duplication. Instead of placing (a, aB) on the stack we shall just put a|B on it. This means that we have seen the part of aB which comes before the vertical line - the symbol a. Putting aB| on the stack means that we have a and B as our top two stack symbols. Here is the same computation with our new stack symbols.

Step	Stack	Input	Action
0		aabb	shift(a|B)
1	a|B	abb	shift(a|B)
2	a|B, a|B	bb	shift(b|)
3	a|B, a|B,b|	b	reduce(B → b)
4	a|B, a|B,aB|	b	reduce(A → aB)
5	a|B, A|b	b	shift(Ab|)
6	a|B, A|b, Ab|		reduce(B → Ab)
7	a|B, aB|		reduce(A → aB)
8	A|		accept

Let's pause a bit and examine these things we are placing on the stack. They are often called *states* and do indicate the state of the input string we have read and partially parsed. States are made up of *items* that are just productions with an indicator that tells us how far on the right hand side we have progressed. The set of items for the previous grammar is:

$A \rightarrow |aB$ $B \rightarrow |Ab$ $B \rightarrow |b$
$A \rightarrow a|B$ $B \rightarrow A|b$ $B \rightarrow b|$
$A \rightarrow aB|$ $B \rightarrow Ab|$

Recall what an item means. $A \rightarrow a|B$ means that we have seen an a and hope to see a B and apply the production. Traditionally we also invent a new starting symbol and add a production where it goes to the old starting symbol. In this case this means adding the items:

$S_0 \rightarrow |A$ $S_0 \rightarrow A|$

to our collection of items.

There are lots and lots of items in a grammar. Some are almost the same. Now it is time to group equivalent items together. We take a closure of an item and get all of the equivalent ones. These *closures* shall form the stack symbols (or states) of our parser. These are computed according to the following procedure.

Closure(I, CLOSURE(I))
PRE: I is an item
POST: CLOSURE(I) contains items equivalent to I

place I in CLOSURE(I)
for **each (A → α\|Bβ) in CLOSURE(I) and (B → γ)**
    place (B → \|γ) in CLOSURE(I)

Figure 5 - Closure Computation for Items

We should compute a few closures for the items in our grammar. The only time we get past the first step above is when the vertical bar is to the left of a nonterminal. Such as in B → |Ab. Let's do that one. We place B → |Ab in CLOSURE(B → |Ab) first. Then we look at all A → α productions and put A → |α in CLOSURE(B → |Ab) also. This gives us:

$$\text{CLOSURE}(B \rightarrow |Ab) = \{B \rightarrow |Ab,\ A \rightarrow |aB\}$$

Some more closures are:

$$\text{CLOSURE}(S \rightarrow |A) = \{S \rightarrow |A,\ A \rightarrow |aB\}$$
$$\text{CLOSURE}(S \rightarrow A|) = \{S \rightarrow A|\}$$
$$\text{CLOSURE}(A \rightarrow a|B) = \{A \rightarrow a|B,\ B \rightarrow |Ab,\ B \rightarrow |b,\ A \rightarrow |aB\}$$

Thus the closure of an item is a collection of all items which represent the same sequence of things placed upon the stack recently. These items in the set are what we have seen on the input string and processed. The productions represented are all those which might be applied soon. The last closure presented above is particularly interesting since it tells us that we have seen an a and should be about to see a B. Thus either Ab, b, or aB could be arriving shortly. States will built presently by combining closures of items.

Let's return to our last table where we did a recognition of aabb. Note that in step 2 a|B was on top to the stack and the next input was b. We then placed b| on the stack. Traditionally sets of items called states are placed upon the stack and so the process of putting the next state on the stack is referred to as a *GOTO*. Thus from step 2 to step 3 in the recognition of aabb we execute:

$$\text{GOTO}(a|B,\ b) = b|.$$

In step 3 we reduced with the production B → b and got a B. We then placed aB| on the stack. In our new terminology this is:

$$GOTO(a|B, B) = aB|.$$

It is time now to precisely define the GOTO operation. For a set of items (or state) Q and symbol x this is:

$$GOTO(Q, x) = \{CLOSURE(A \rightarrow \alpha x|\beta)\} \text{ for all } A \rightarrow \alpha|x\beta \in Q$$

Check out the operations we looked at above and those in the previous acceptance table. Several more examples are:

$$\begin{aligned} GOTO(\{S \rightarrow |A, A \rightarrow |aB\}, A) &= CLOSURE(S \rightarrow A|) \\ &= \{S \rightarrow A|\} \end{aligned}$$

$$\begin{aligned} GOTO(\{S \rightarrow |A, A \rightarrow |aB\}, a) &= CLOSURE(A \rightarrow a|B) \\ &= \{A \rightarrow a|B, B \rightarrow |Ab, B \rightarrow |b, A \rightarrow |aB\} \end{aligned}$$

So, all we need do is add a new starting production ($S_0 \rightarrow S$) to a grammar and execute the following state construction algorithm in order to generate all the states we require in order to do parsing.

```
Q0 = CLOSURE(S0 → |S)
i = 0
k = 1
repeat
    for each |x in Qi
        if GOTO(Qi, x) is a new state then
            Qk = GOTO(Qi, x)
            k = k + 1
    i = i + 1
until no new states are found
```

Figure 6 - State Construction Algorithm

The seven states determined for our example grammar using the above algorithm are:

$Q_0 = \{S \rightarrow |A, A \rightarrow |aB\}$
$Q_1 = \{S \rightarrow A|\}$
$Q_2 = \{A \rightarrow a|B, B \rightarrow |Ab, B \rightarrow |b, A \rightarrow |aB\}$
$Q_3 = \{A \rightarrow aB|\}$
$Q_4 = \{B \rightarrow A|b\}$
$Q_5 = \{B \rightarrow b|\}$
$Q_6 = \{B \rightarrow Ab|\}$

and the relationships formed from the GOTO(Q, x) relationship are:

	A	B	a	b
Q_0	Q_1		Q_2	
Q_2	Q_4	Q_3	Q_2	Q_5
Q_4				Q_6

Note that not all state-symbol pairs are represented. We do know why there are no states to go to from Q_1, Q_3, Q_5, and Q_6 - right? Because they are states that contain items we shall reduce. After that we shall place another item on the stack as part of the reduction process.

All that remains is to build a parser that will carry out the sample computation we presented above. It is easy. Here are the rules for building the parser table. (Recall that the stack symbols or states are along the left side while grammar symbols lie across the top.)

On the row for state Q_i:

a) if GOTO(Q_i,a) = Q_k then *shift*(Q_k) under a
b) if $A \rightarrow \alpha| \in Q_i$ then *reduce*($A \rightarrow \alpha$) under FOLLOW(A)
c) if $S_0 \rightarrow S| \in Q_i$ then *accept* under the endmarker

Figure 7 - Parser Table Construction

That is all there is to it. Quite simple. Another note - we shall attach a GOTO table to the right side of our parser table so that we know what to place on the stack after a reduction. The parser for our sample grammar is provided below. The words *shift* and *reduce* have been omitted because they refer always to states and productions respectively and there should be no problem telling which is which. (Aliases for the states have been provided so that the table is readable.)

State		Input			GOTO	
Name	Alias	a	b	end	A	B
Q_0	|aB	Q_2			Q_1	
Q_1	A|			*accept*		
Q_2	a|B	Q_2	Q_5		Q_4	Q_2
Q_3	aB|		$A \rightarrow aB$			
Q_4	A|b		Q_6			
Q_5	b|		$B \rightarrow b$	$B \rightarrow b$		
Q_6	Ab|		$B \rightarrow Ab$	$B \rightarrow Ab$		

We know intuitively how these parsers work, but need to specify some things precisely. *Shift* operations merely push the indicated state on the stack. A *reduce* operation has two parts. For a reduction of $A \rightarrow \alpha$ where the length of α is k, first pop k states off the stack. (These are the right hand side symbols for the production.) Then if Q_i is on top of the stack, push GOTO(Q_i ,A) onto the stack. So, what we are doing is to examine the stack and push the proper state depending upon what was at the top and what was about to be processed. And last, begin with Q_0 on the stack. Try out our last example and note that exactly the same sequence of moves results.

Now let us label what we have been doing. Since we have been processing the input from left to right and doing rightmost derivations, this is called *LR parsing*. And the following theorem ties the LR languages into our framework.

Theorem 6. *The following classes are equivalent.*

a) Deterministic context free languages.
b) LR(1) languages.
c) LR(0) languages with endmarkers.
d) Languages accepted by deterministic pushdown automata.

Summary

We have encountered five major classes of languages and machines in our examination of computation. Now seems like a good time to sum up some of the things we have discovered for all of these classes. This shall be done in a series of charts.

The first sets forth these classes or families in descending order. Each is a proper subclass of those above it. (Note that the last column provides a set in the class which does not belong to the one below.)

Class	*Machine*	*Language*	*Example*
Recursively Enumerable Sets	Turing Machines	Type 0	K
Recursive Sets			diagonal sets
Context Sensitive Languages	Linear Bounded Automata	Type 1	$0^n1^n0^n$
Context Free Languages	Pushdown Automata	Type 2	invalid TM computations
Deterministic Context Free Languages	Deterministic Pushdown Automata	LR(1)	a^nb^n
Regular Sets	Finite Automata	Type 3	

Next, we shall list the closure properties which were proven for each class or mentioned in either the historical notes or exercises. Complement is indicated by '¬' and concatenation is indicated by a dot.

Class	¬	∪	∩	•	*
r.e.	no	yes	yes	yes	yes
recursive	yes	yes	yes	yes	yes
csl	yes	yes	yes	yes	yes
cfl	no	yes	no	yes	yes
dcfl	yes	no	no	no	no
regular	yes	yes	yes	yes	yes

Our last chart indicates the solvability or unsolvability of the decision problems we have examined thus far. (S stands for solvable, U for unsolvable, and ? for unknown.)

Class	$x \in L$	$L = \varnothing$	L finite	$L_i \subset L_j$	$L_i = L_j$	$L = \Sigma^*$	L cofinite
r.e.	U	U	U	U	U	U	U
recursive	S	U	U	U	U	U	U
csl	S	U	U	U	U	U	U
cfl	S	S	S	U	U	U	U
dcfl	S	S	S	U	?	S	S
regular	S	S	S	S	S	S	S

Notes

It all began with Noam Chomsky. Soon, however *BNF* (Backus Normal form or Backus-Naur Form) was invented to specify the syntax of programming languages. The classics are:

J. W. BACKUS, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proceedings of the International Conference on Information Processing (1959)*, UNESCO, 125-132.

N. CHOMSKY, "Three models for the description of languages," *IRE Transactions on Information Theory 2:3 (1956)*, 113-124.

P. NAUR et. al., "Report of the algorithmic language ALGOL 60," *Communications of the Association for Computing Machinery 3:5 (1960)*, 299-314. Revised in *6:1 (1963)*, 1-17.

Relationships between classes of languages and automata were soon investigated. In order of language type we have:

N. CHOMSKY, "On certain formal properties of grammars," *Information and Control 2:2 (1959)*, 137-167.

S. Y. KURODA, "Classes of languages and linear bounded automata," *Information and Control 7:2 (1964)*, 207-223.

P. S. LANDWEBER, "Three theorems on phrase structure grammars of type 1." *Information and Control 6:2 (1963)*, 131-136.

N. CHOMSKY, "Context-free grammars and pushdown storage," *Quarterly Progress Report 65 (1962)*, 187-194, MIT Research Laboratory in Electronics, Cambridge, Massachusetts.

J. EVEY, "Application of pushdown store machines," *Proceedings of the 1963 Fall Joint Computer Conference*, 215-227, AFIPS Press, Montvale, New Jersey.

N. CHOMSKY and G. A. MILLER, "Finite state languages," *Information and Control 1:2 (1958)*, 91-112.

Normal forms for the context free languages are due to Chomsky (in the 1959 paper above) and:

S. A. GREIBACH, "A new normal form theorem for context-free phrase structure grammars," *Journal of the Association for Computing Machinery 12:1 (1965)*, 42-52.

Most of the closure properties and solvable decision problems for context free languages were discovered by Bar-Hillel, Perles, and Shamir in the paper cited in chapter 3. They also invented the pumping lemma. A stronger form of this useful lemma is due to:

W. G. OGDEN, "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory 2:3 (1969)*, 191-194.

The text by Hopcroft and Ullman is a good place to find material about automata and formal languages, as is the book by Lewis and Papadimitriou. (These were cited in chapter 1.) Several formal languages texts are:

S. GINSBURG, *The Mathematical Theory of Context-free Languages*, McGraw-Hill, New York, 1966.

M. A. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts, 1978.

G. E. REVESZ, *Introduction to Formal Languages*, McGraw-Hill, New York, 1983.

A. SALOMAA, *Formal Languages*, Academic Press, New York, 1973.

Knuth was the first to explore LR(k) languages and their equivalence to deterministic context free languages. The early LR and LL grammar and parsing papers are:

D. E. KNUTH, "On the translation of languages from left to right," *Information and Control 8:6 (1965)*, 607-639.

A. J. KORENJAK, "A practical method for constructing LR(k) processors," *Communications of the Association for Computing Machinery 12:11 (1969)*, 613-623.

F. L. DE REMER, "Generating parsers for BNF grammars," *Proceedings of the 1969 Spring Joint Computer Conference*, 793-799, AFIPS Press, Montvale, New Jersey.

and two books about compiler design are:

A. V. AHO and J. D. ULLMAN, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

P. M. LEWIS II, D. J. ROSENCRANTZ, and R. E. STEARNS, *Compiler Design Theory*, Addison-Wesley, Reading, Massachusetts, 1976.

PROBLEMS

Grammars

1. Construct a grammar that defines variables and arrays for a programming language. Add arithmetic assignment statements. Then include labels. Make sure to explain what the productions accomplish.

2. Provide a grammar for the *while, for,* and *case* statements of our *NICE* programming language. Then define blocks. (Note that at this point you have defined most of the syntax for our *NICE* language.)

3. What type of grammar is necessary in order to express the syntax of the *SMALL* language? Justify your answer.

4. Build a grammar that generates Boolean expressions. Use this as part of a grammar for conditional (or *if-then-else*) statements.

Language Properties

1. Design a grammar that generates strings of the form $0^n1^m0^n1^m$. Provide a convincing argument that it does what you intended. What type is it?

2. Furnish a grammar for binary numbers that are powers of two. What type is it? Now supply one for strings of ones that are a power of two in length. Explain your strategy.

3. Construct a grammar that generates strings of the form ww where w is a string of zeros and ones. Again, please hint at the reasons for your methods.

4. Show precisely that for each Type 0 language there is a Turing machine that accepts the strings of that language.

5. Prove that the Type 1 or context sensitive languages are equivalent to the sets accepted by linear bounded automata.

6. Prove that all types of languages are closed under the operation of string reversal.

Regular Languages

1. Derive a regular expression for programming language constants such as those defined in this chapter.

2. Construct a regular grammar for strings of the form 1^*0^*1.

3. What is the regular expression for the language generated by the regular grammar:

   $S \rightarrow 1A \mid 1$
   $A \rightarrow 0S \mid 0A \mid 0$

4. Design a *deterministic* finite automaton that recognizes programming language constants.

5. What is the equivalent regular grammar for the following finite automaton?

State	Input		Accept?
	0	1	
0	0	2	no
1	1	0	yes
2	2	1	no

6. Prove that every regular set can be generated by some regular grammar.

7. Show that if *epsilon rules* of the form $A \rightarrow \varepsilon$ are allowed in regular grammars, then only the regular sets are generated.

8. A *left linear* grammar is restricted to productions of the form $A \rightarrow Bc$ or of the form $A \rightarrow c$ where as usual A and B are nonterminals and c is a terminal symbol. Prove that these grammars generate the regular sets.

Context Free Languages

1. Construct a context free grammar which generates all strings of the form $a^n b^* c^n$.

2. A nonterminal can be *reached* from another if it appears in some string generated by that nonterminal. Prove that context free grammars need not contain any nonterminals that cannot be reached from the starting symbol.

3. Show that productions of the form A → B (i.e. chain rules) need never appear in context free grammars.

4. Produce a Chomsky Normal Form grammar for assignment statements.

5. Develop a Chomsky Normal form Grammar that generates all Boolean expressions.

6. Express the following grammar in Chomsky Normal form.

   S → =VE
   E → +EE | -EE | V
   V → a | b

7. Convert the grammar of the last problem to Greibach Normal form.

8. Place our favorite grammar (S → 1S0 | 10) in Greibach Normal form.

9. If you think about it, grammars are merely bunches of symbols arranged according to certain rules. So, we should be able to generate grammars with other grammars. Design three context free grammars which generate grammars which are:

   a. context free,

   b. in Chomsky Normal form, and

   c. in Greibach Normal form.

10. Prove that any set which can be accepted by a pushdown automaton is a context free language.

11. Show that the context free languages can be accepted by deterministic linear bounded automata. [Hint: use Greibach Normal form grammars.]

12. An *epsilon move* is one in which the tape head is not advanced. Show that epsilon moves are unnecessary for pushdown automata.

Context Free Language Properties

1. Is the set of strings of the form $0^n1^m0^n1^m$ (for $n \leq 0$) a context free language? Justify your conjecture.

2. Show that the set of strings of prime length is not a context free language.

3. We found that the context free languages are not closed under intersection. But they are closed under a less restrictive property - *intersection with regular sets*. Prove this.

4. Suppose that L is a context free language and that R is a regular set. Show that L - R is context free. What about R - L?

5. Demonstrate that while the set of strings of the form w#w (where w is a string of a's and b's) is not a context free language, its complement is one.

6. Select a feature of your favorite programming language and show that its syntax is not context free.

7. Precisely work out the algorithm for deciding the emptiness problem for context free languages. Why is your algorithm correct?

8. Show that the grammars of problems 3 and 6 generate infinite languages.

Parsing and Deterministic Languages

1. Why is the following grammar not an s-grammar? Turn it into one and explain each step as you do it.

   $S \rightarrow AB$
   $A \rightarrow aA \mid aC$
   $B \rightarrow bC$
   $C \rightarrow Bc \mid c$

2. Develop an s-grammar for strings of the form $a^n b^n c^m d^m$. Show why your grammar is an s-grammar.

3. Rewrite the following grammar as a q-grammar. Explain your changes.

   $E \rightarrow T \mid T+E$
   $T \rightarrow x \mid (E)$

4. Construct a q-grammar for Boolean expressions.

5. Show that every LL(1)-language is indeed a q-language.

6. Discuss the differences between pushdown automata and top-down parsers. Can parsers be made into pushdown automata?

7. Show that the class of languages accepted by deterministic pushdown automata is closed under complement.

8. Construct the tables for an LR parser that recognizes arithmetic assignment statements. (Modify the grammar provided in this chapter to include endmarkers.)

9. Outline a general method for converting LR parsers to deterministic pushdown automata (with epsilon moves).