PROGRAMS GRAMMARS ARGUMENTS

Joachim Lambek

Contents

Preface

These notes began as notes for a course called "Computability and Mathematical Linguistics" taught at McGill University for about 25 years, beginning in 1974. It was quite successful, but after Professor Lambek and I retired, there was no one who was sufficiently interested in teaching the course as designed and it eventually disappeared. There was a proposal to add quite a bit of logic to the notes and publish it jointly with Phil Scott and me, but this project never much went beyond putting the original notes (augmented by the paper on the French verb) into TEX. Now, nearly fifteen years later, Professor Lambek and I decided to put this online in case there is any interest. There are two additional comments by me on pages 82 and 115.

Michael Barr, Montreal, 2007-08-16.

Chapter 1

Calculations on an abacus

1.1 What is a calculation?

Some people say that a calculation is what can be done on an electronic computer, while others would be satisfied with pencil and paper. To give a precise answer to this question, Turing in 1936 invented an abstract device, later called a "Turing machine". His idea proved to be very fruitful, both in answering some theoretical questions on so-called "word problems" and in stimulating the development of the modern electronic computer. However, there is an easier concept that goes back to antiquity and will do the same thing.

The word "calculate" is derived from the Latin word for pebble. At one time these pebbles were placed into grooves on a table, called "abacus" from a Greek word for table. Let us here define an **abacus** as consisting of

(a) a potentially infinite number of **locations** A, B, $C \ldots$, that may be distinguished from one another, and

(b) a potentially infinite number of **pebbles** that need not be distinguishable.

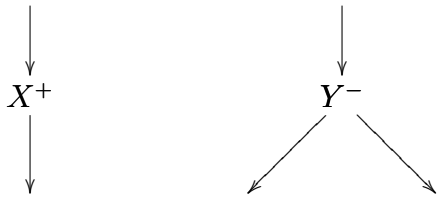
Let it also be understood that

(c) each location can hold as many pebbles as one wishes.

It will turn out that for a given calculation only a finite number of locations and pebbles are needed, but we do not wish to impose a bound on this number beforehand. In the same way, when talking about calculations with paper and pencil, one does not want to limit the amount of paper and the number of pencils to be used.

Essentially there are only two operations one can perform on an abacus. One can put a pebble into a designated location, or one can take a pebble from a designated location. The first operation can always be performed, the second can only be performed if the

location is not empty. Accordingly there are two **elementary instructions**, as indicated by the following diagrams:

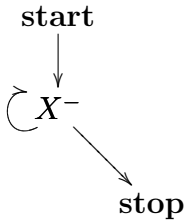


The first instruction reads: go to location X, deposit one pebble, then go to wherever the outgoing arrow points. The second instruction reads: go to location Y, remove one pebble if this can be done; in that case go to wherever the left arrow points; in the contrary case (that is, when Y found to be empty), go to wherever the right arrow points. It may be convenient to consider two further elementary instructions:

The first of these says: at the beginning go to wherever the arrow points. The second one says: **stop**.

By a **program** we shall mean a finite number of such elementary instructions fitted together in a specified manner. We shall not make this precise at the moment, instead we shall illustrate some simple programs by their flow diagrams.

1.1.1 Example Empty location X.



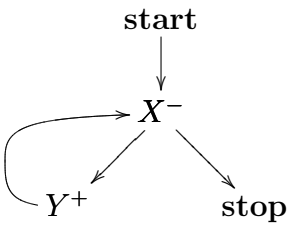
X
x
0

What does this picture mean? Instruction: remove one pebble from location X. If this can be done, return to the same instruction; otherwise, stop. Suppose, for example, there are 3 pebbles at X to start with, then successive contents of location X are indicated as follows:

X
3
2
1
0

In general, if the initial content of X is x pebbles, then its final content is 0 pebbles.

1.1.2 EXAMPLE Transfer content of X to Y.



X	Y
x	y
0	$x+y$

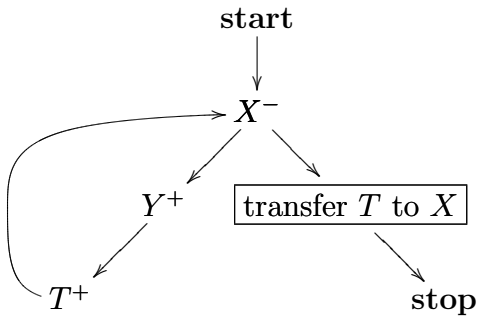
What does this flow diagram mean?
Instruction (1): remove one pebble from location X. If this can be done, go to instruction (2). If it cannot be done, stop.
Instruction (2): add one pebble to location Y. Then go to instruction (1). For example, suppose there are 2 pebbles at location X and 3 pebbles at location Y to start with.

X	Y
2	3
1	4
0	5

hen successive contents of those two locations are as follows:

We shall use the word **configuration** to denote a complete listing of the contents of all locations, but we only mention those locations which are nonempty at some stage of a calculation. In the present example, an initial configuration of x pebbles at X and y pebbles at Y is transformed by the program to a terminal configuration of 0 pebbles at X and $y+x$ pebbles at Y.

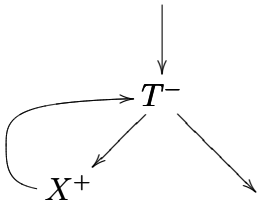
1.1.3 EXAMPLE copy content of X into Y.



X	Y
x	y
x	$y+x$

In discussing the performance of this program, we assume that T is empty at the beginning of the calculation. It follows that T is also empty at the end. We call T a **temporary storage** location. It serves the same purpose as scratch paper when calculating with paper and pencil.

In the above flow diagram, we have used $\boxed{T \to X}$, which means "transfer content of T to X" as a so-called **subroutine**. In accordance with Example 1.1.2, the rectangle is to be replaced by



Note that the **start** and **stop** instructions within the subroutine should not be included, although the arrows going to and from them should be as the entry and exit points of the subroutines. We will always put a subroutine in a box.

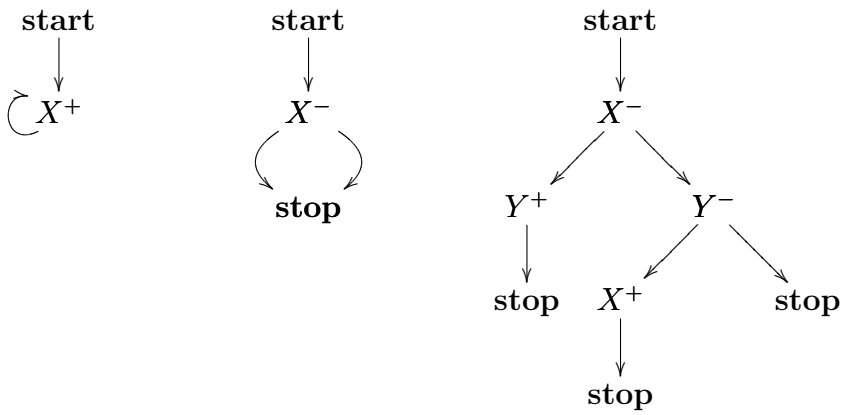
Assuming the same initial configuration as in Example 1.1.2, with T empty, here are successive configurations for Example 1.1.3.

X	Y	T
2	3	0
1	4	1
0	5	2
2	5	0

The astute reader will have noticed that the programs of Examples 1.1.2 and 1.1.3 could have served stone-age man in the task of adding two numbers.

Exercises

1. What do the following programs do:



(In the second and third diagrams the **stop**instruction has been written more than once to simplify the drawing of the diagrams. Nonetheless, there is only one stop. This is

necessary in order that when a program is used as a subroutine, it will have just one exit.)

2. Draw the flow diagram for a program which will result in interchanging the contents of locations X and Y.

1.2 Calculable functions

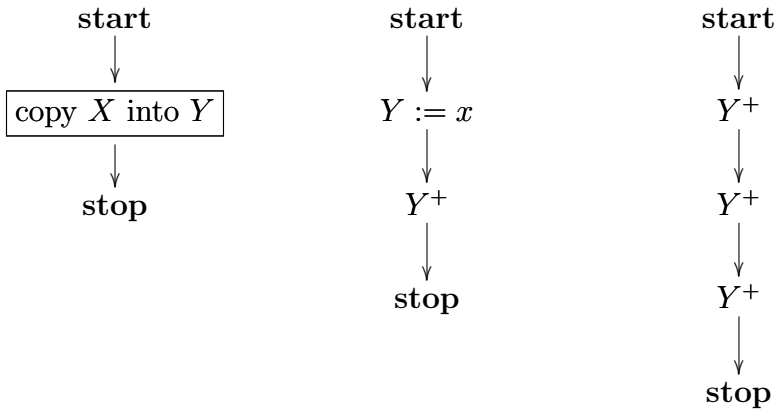
It is convenient to include 0 among the natural numbers. The set of **natural numbers** is thus $\mathbf{N} = \{0, 1, 2, \ldots\}$. By a **numerical function** of one variable we mean a mapping $f : \mathbf{N} \longrightarrow \mathbf{N}$, and we write $y = f(x)$ to indicate that the value of f at x is y. By a numerical function of two variables we mean a mapping $f : \mathbf{N} \times \mathbf{N} \longrightarrow \mathbf{N}$, and we write $z = f(x, y)$ to indicate that f assigns to the pair (x, y) the value z. We are interested in calculating numerical functions of any number of variables on an abacus.

We shall say that a program **calculates** the function $y = f(x)$ with the help of the temporary storage location T if it transforms the configuration $\begin{array}{ccc} X & Y & T \\ \hline x & 0 & 0 \end{array}$ to $\begin{array}{ccc} X & Y & T \\ \hline x & f(x) & 0 \end{array}$. It is assumed that Y and T are empty at the beginning of the calculation. T may be occupied at some stages during the calculation, but must be empty at the end. No other locations are supposed to be involved in the calculation, as we did not specify any. The fact that X has the same content at the end of the calculation as at the beginning is required for technical reasons: we don't want the input to be lost. Otherwise we could have permitted the final configuration $\begin{array}{ccc} X & Y & T \\ \hline 0 & f(x) & 0 \end{array}$.

We will denote by $\boxed{Y := f(x)}$ a machine that calculates assigns to the location Y the value of the function f evaluated at the contents x of the location X and leaves all other locations unchanged, that is the machine that calculates $y = f(x)$. Similarly a machine that calculates $z = f(x, y)$ will be denoted $\boxed{Z := f(y, z)}$.

For example, the **identity** function $y = x$, the **successor** function $y = S(x) = x + 1$,

and the **constant** function $y = 3$ are calculated by the following three programs:



This is shown by the initial and final configurations:

X	Y	T
x	0	0
x	x	0

X	Y	T
x	0	0
x	$S(x)$	0

X	Y	T
x	0	0
x	3	0

Note that the subroutine $\boxed{\text{copy } X \text{ into } Y}$ is exactly the same as $\boxed{Y := x}$.

Hard working students are invited to draw the flow diagram for the constant function $y = 17$.

Aside from such trivial functions, what other functions can be calculated on an abacus? Consider the following numerical functions:

$y =$ the $(x + 1)$st prime number,

$y =$ the number of primes less than x,

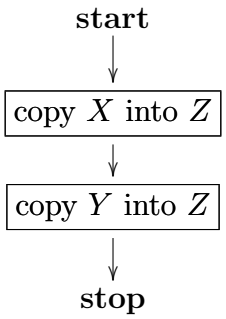
$y =$ the $(x + 1)$st digit of π.

All these can be calculated on the abacus, and for two of them we shall construct programs later. In fact, at the moment, the reader will probably not be able to think of a numerical function that cannot be calculated on an abacus.

More generally, we say that a program **calculates** a numerical function $z = g(x_1, x_2, \ldots, x_n)$ of n variables with the help of temporary storage locations T_1, T_2, $\ldots, T_k$, if its initial and final configurations are as follows:

X_1	X_2	$\cdots$	X_n	Z	T_1	T_2	$\cdots$	T_n
x_1	x_2	$\cdots$	x_n	0	0	0	$\cdots$	0
x_1	x_2	$\cdots$	x_n	$g(x_1, \ldots, x_n)$	0	0	$\cdots$	0

For example, here is a simple minded program for calculating $z = x + y$:



We shall presently give another program for calculating addition which is more sophisticated, but which has the advantage of serving as a model for programs that will calculate multiplication and exponentiation.

What is meant by the sum $x + y$ of two numbers? We do know that $x + 0 = x$. Moreover, if we happen to know the value of $x + 173$, then we also know that $x + 174$ is one more.

We shall write "$S(y)$" for the **successor** of y. (We are not allowed to call this $y + 1$ until after addition has been defined.) We then have quite generally

$$x + S(y) = S(x + y)$$

The sum of two numbers is now completely determined, once we decide to define the natural numbers by

$$1 = S(0), \qquad 2 = S(1), \qquad 3 = S(2), \ldots \qquad \text{etc.}$$

For example, we have

$$3 + 2 = 3 + S(1) = S(3 + 1)$$

where

$$3 + 1 = 3 + S(0) = S(3 + 0) = S(3) = 4$$

hence

$$3 + 2 = S(4) = 5$$

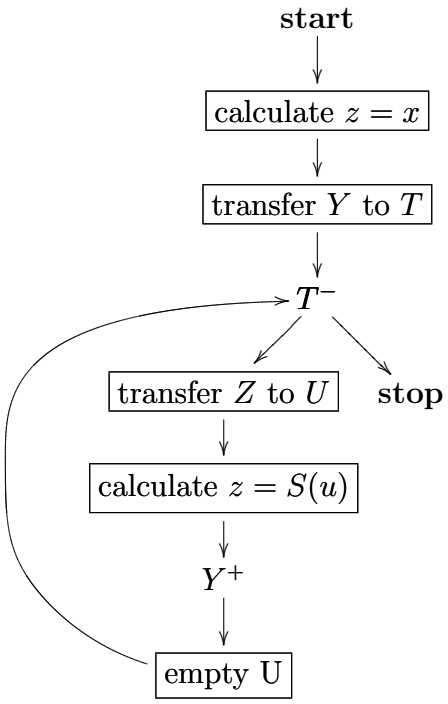
The definition of addition discussed here is called a **recursive** definition.

We shall now present programs for the functions $z = x + y$, $z = x * y$, $z = x^y$. Note that these functions are defined recursively as follows:

$$\begin{cases} x + 0 = x \\ x + S(y) = S(x + y) \end{cases} \qquad \begin{cases} x * 0 = 0 \\ x * S(y) = x * y + x \end{cases} \qquad \begin{cases} x^0 = 1 \\ x^{S(y)} = x^y * x \end{cases}$$

where $1 = S(0)$.

Here is the program to calculate $z = x + y$:



X	Y	Z	T	U
x	y	0	0	0
x	0	x	y-1	0
x	0	0	y-1	x
x	1	x+1	y-2	0
.	.	.	.	.
x	y	x+y	0	0

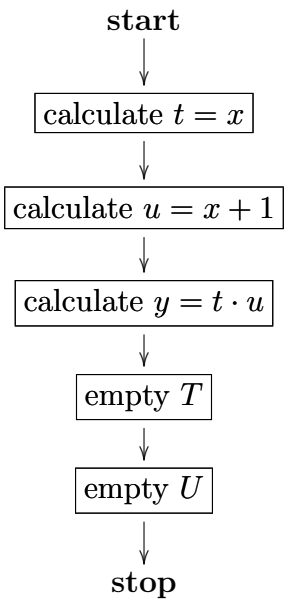
Some of the configurations which appear during the calculation are shown in the above table. In addition to the temporary storage locations T and U, there are also other temporary storage locations which are used in the subroutines.

To calculate $z = x * y$ and $z = x^y$ one need only replace the subroutine Z:=x in the above by Z:= 0 and Z:= 1, respectively, and the subroutine Z:= S(u) by Z:= u+x and Z:= u * x, respectively.

Exercises

1. Draw the flow diagram of a program for the function $z = x^2$. (You may regard this as a special case of $z = x * y$ or of $z = x^y$.)

2. Which function $y = f(x)$ is calculated by the following program:



3. Draw the flow diagram of a program to calculate $y = x^2 + 2x + 2$.
4. The function $z = x \uparrow y$ is defined thus: $x \uparrow 0 = 1, x \uparrow S(y) = x^{(x \uparrow y)}$. For instance, $x \uparrow 3 = x^{(x^x)}$. Write out a program.

1.3 Primitive recursive functions are calculable

The functions $x + y$, $x * y$, and $x \uparrow y$ are all examples of "primitive recursive" functions. Probably all the numerical functions you can think of are primitive recursive. By a numerical function of k variables we understand a function $y = f(x_1, x_2, \ldots, x_k)$, where $x_1, x_2, \ldots, x_k$ and y all range over the set of natural numbers. When $k = 0$, this reduces to $y = c$, where c is a natural number.

We shall now list a number of numerical functions and some schemes for producing new functions from known functions.

R–1. **successor** function: $f(x) = S(x)$

R–2. **constant** functions: $f(x_1, \ldots, x_n) = k$

R–3. **projection** functions: $f(x_1, \ldots, x_n) = x_k$

R–4. **substitution** scheme:

$$f(x_1, \ldots, x_n) = g(h1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$$

R–5. **recursion** scheme:

$$\begin{cases} f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1}) \\ f(x_1, \ldots, x_{n-1}, S(y)) = h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y)) \end{cases}$$

Remarks. R–2 and R–3 are families of functions, one for each pair of natural numbers n and k. R–3 includes the identity function $f(x) = x$ as a special case, when $n = k = 1$. R–4 produces one function f of n variables, given one function g of k variables and k functions h_1, ..., h_k of n variables. R–5 recursively defines a function f of n variables, given a function g of $n-1$ variables and a function h of $n+1$ variables. Special cases of R–5 are the following:
R–5, case $n = 1$,

$$\begin{cases} f(0) = c \\ f(S(y)) = h(y, f(y)) \end{cases}$$

R–5, case $n = 2$,

$$\begin{cases} f(x, 0) = g(x) \\ f(x, S(y)) = h(x, y, f(y)) \end{cases}$$

Informally we may define primitive recursive functions as all functions of type R–1, R–2 and R–3, as well as all functions derivable from these by schemes R–4 and R–5.

More formally, a mathematician would put this as follows:

1.3.1 Definition *The class of* **primitive recursive functions** *is the smallest class of numerical functions containing all functions of types R–1, R–2 and R–3 and closed under schemes R–4 and R–5.*

To say that a class of numerical functions is closed under scheme R–4 means that if $h_1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n)$ and $g(y_1, \ldots, y_k)$ are in the class, then so is $f(x_1, \ldots, x_n)$ as defined by R–4.

To say that a class of numerical functions is closed under scheme R–5 means that if $g(x_1, \ldots, x_{n-1})$ and $h(x_1, \ldots, x_{n-1}, y, z)$ are in the class, then so is $f(x_1, \ldots x_{n-1}, y)$ as defined by R–5.

for example, why is $x + y$ a primitive recursive function? We recall that the recursive definition of $x + y$ reads

$$\begin{cases} x + 0 = x \\ x + S(y) = S(x + y) \end{cases}$$

It will be convenient to denote the primitive recursive function defined by R–3 by p_k^n, thus

$$p_k^n(x_1, \ldots x_n) = x_k$$

We note that $x = p_1^1(x)$ and that $x + y = p_3^3(x, y, x + y)$, hence the recursive definition of $x + y$ may be written thus:

$$\begin{cases} x + 0 = p_1^1(x) \\ x + S(y) = S(p_3^3(x, y, x + y)) \end{cases}$$

Then $f(x,y) = x + y$ is primitive recursive by R–5, case $n = 2$, if we take $g(x) = p_1^1(x)$ and $h(x,y,z) = S(p_3^3(x,y,z))$. Here g is primitive recursive by R–3, and h is primitive recursive by R–5, with $n = 3$ and $k = 1$, since S is primitive recursive by R–1 and p_{33} is primitive recursive by R–3.

Similarly one shows that $x * y$ is primitive recursive. Now consider $x!$, this satisfies the recursive definition

$$\begin{cases} 0! = 1 \\ S(x)! = S(x) * x! \end{cases}$$

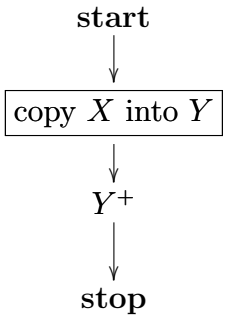
Then $f(x) = x!$ is primitive recursive by R–5, case $n = 1$, if we take $c = 1$, and $h(y,z) = S(y) * p_1^1(z) = S((p_1^2(y,z)) * p_2^2(y,z)$. Here h is primitive recursive by R–4, with $n = 1$, and $k = 2$, since multiplication is primitive recursive, by the above observation, and S and p_1^2 are primitive recursive by R–1 and R–3 respectively.

1.3.2 THEOREM *Every primitive recursive function is calculable.*

PROOF We must show that the class of calculable functions contains all numerical functions of types R–1, R–2, and R–3, and that it is closed under schemes R–4 and R–5.

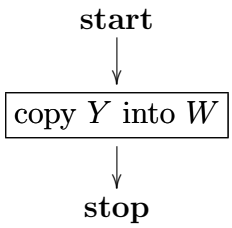
For concreteness, we take $n = 3$ and $k = 2$ in R–2, R–3, and R–4, and n=2 in R–5. The proof then consists in inspecting the following five programs.

R–1. Program for $y = S(x)$:

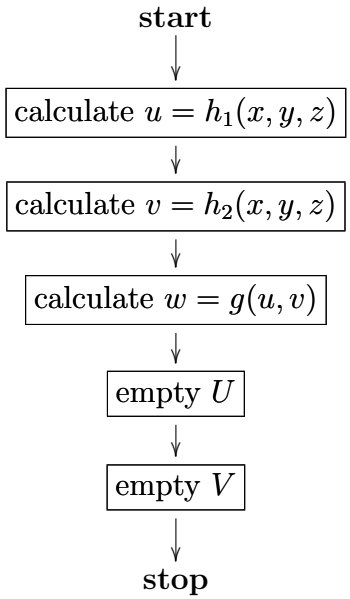


R–2. Program for $w = f(x,y,z) = 2$:

R–3. Program for $w = f(x, y, z) = y$:

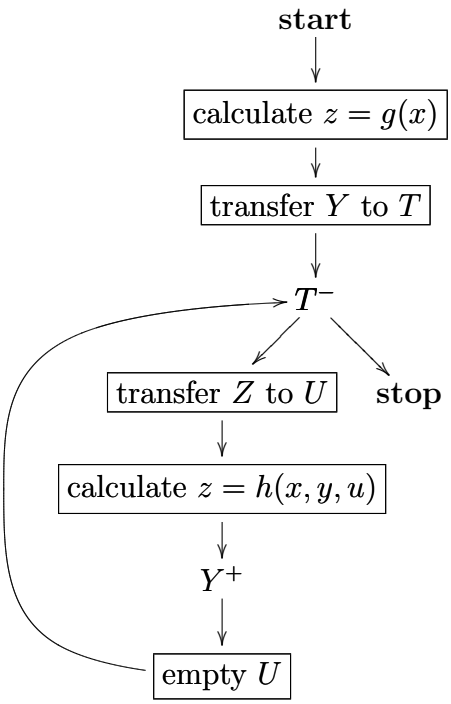


R–4. Program for $w = g(h_1(x, y, z), h_2(x, y, z))$:



R–5. Program to calculate $z = f(x, y)$, when

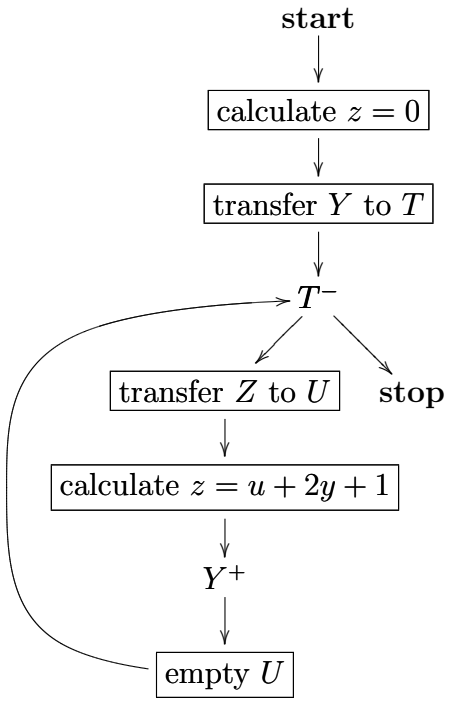
$$\begin{cases} f(x, 0) = g(x) \\ f(x, S(y)) = h(x, y, f(x, y)) \end{cases}$$



For example, take $f(x, y) = x + y$. Then we have seen that $g(x) = x$ and $h(x, y, u) = S(u)$. Hence the above program specializes to that given for $x + y$ in Section 1.2. The same goes for $x * y$ and xy.

Exercise

1. What function in one or two variables is calculated by the following program? Explain why this is so.



1.4 Some primitive recursive functions

The following list of primitive recursive functions is taken from the book by Kleene. We have already met (1) to (4). (5) deals with the "predecessor function", the predecessor of 0 being defined exceptionally as zero. (6) deals with the "naive difference" between x and y, which is considered to be 0 when $y \geq x$. (7) and (8) deal with the usual maximum and minimum of a pair of numbers. (9) is called the "delta function", (10) the "sign function", and (11) the "absolute difference".

(1) $x + y$ $\quad \begin{cases} x + 0 = x \\ x + S(y) = S(x + y) \end{cases}$

(2) $x * y$ $\quad \begin{cases} x * 0 = 0 \\ x * S(y) = (x * y) + x \end{cases}$

(3) x^y $\quad \begin{cases} x^0 = 1 \\ x^{S(y)} = x^y * x \end{cases}$

(4) $x!$ $\quad \begin{cases} 0! = 1 \\ S(x)! = x! * S(x) \end{cases}$

$$(5)\quad \mathtt{pre}(x) = \begin{cases} x-1 & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases} \qquad \begin{cases} \mathtt{pre}(0)=0 \\ \mathtt{pre}(S(x))=x \end{cases}$$

$$(6)\quad x \dot{-} y = \begin{cases} x-y & \text{if } x \geq y \\ 0 & \text{if } x<y \end{cases} \qquad \begin{cases} x \dot{-} 0 = x \\ x \dot{-} S(y) = \mathtt{pre}(x \dot{-} y \\ ) \end{cases}$$

$$(7)\quad \mathtt{min}(x,y) = x \dot{-} (x \dot{-} y)$$

$$(7')\quad \mathtt{min}(x,y,z) = \mathtt{min}(\mathtt{min}(x,y),z)$$

$$(8)\quad \mathtt{max}(x,y) = (x+y) \dot{-} \mathtt{min}(x,y) = x + (y \dot{-} x)$$

$$(8')\quad \mathtt{max}(x,y,z) = \mathtt{max}(\mathtt{max}(x,y),z)$$

$$(9)\quad \delta(x) = \begin{cases} 1 & \text{if } x=0 \\ 0 & \text{if } x>0 \end{cases} \qquad \begin{cases} \delta(0)=0 \\ \delta(S(x))=0 \end{cases}$$

Alternative definitions: $\delta(x) = 1 \dot{-} x = 0^x$

$$(10)\quad \mathtt{sgn}(x) = \begin{cases} 0 & \text{if } x=0 \\ 1 & \text{if } x>0 \end{cases} \qquad \begin{cases} \mathtt{sgn}(0)=0 \\ \mathtt{sgn}(S(x))=1 \end{cases}$$

Alternative definitions: $\mathtt{sgn}(x) = \delta(\delta(x)) = \mathtt{min}(x,1)$

$$(11)\quad |x-y| = (x \dot{-} y) + (y \dot{-} x)$$

The first three functions above suggest a whole sequence of functions $f_t(x,y)$, $t = 0$, 1, 2, 3, ..., where $f_0(x,y) = x+y$, $f_1(x,y) = x*y$, $f_2(x,y) = x^y$, and $f_3(x,y) = x \uparrow y$, which made an appearance in one of the exercises. We are led to define $f_{S(t)}(x,y)$ thus:

$$\begin{cases} f_{S(t)}(x,0) = \mathtt{sgn}(t) \\ f_{S(t)}(x,S(y)) = f_t(x, f_{S(t)}(x,y)) \end{cases}$$

It is easily seen, by mathematical induction on t, that each of the functions $f_t(x,y)$ of two variables is primitive recursive.

Let us now consider a function of three variables defined as follows:

$$f(t,x,y) = f_t(x,y)$$

In other words,

$$\begin{cases} f(0,x,y) = x+y \\ f(S(t),x,0) = \mathtt{sgn}(t) \\ f(S(t),x,S(y)) = f(t,x,f(S(t),x,y)) \end{cases}$$

It is clear that this function is defined recursively, in some sense of the word "recursive". In fact it can be shown that it is calculable on our abacus. Yet, it is not primitive recursive, as was shown by Ackermann. Putting $g(x) = f(x,x,x)$, he proved that, for every primitive recursive function $h(x)$, there exists a natural number x such that $g(x) > h(x)$. Thus $g(x)$ cannot be primitive recursive, hence $f(x;y,z)$ cannot be primitive recursive. (Actually, Ackermann considered not this function, but a very similar one.)

Exercise

1. Show how to calculate functions (5) to (11) above, by drawing flow diagrams of programs of the abacus. Make use of subroutines freely.

1.5 The minimization scheme

The reader will recall the substitution scheme R–4 and the recursive scheme R–5, which played a role in the definition of the class of primitive recursive functions. Another useful scheme is the following minimization scheme:

R–6. $f(x_1, \ldots, x_n) =$ smallest y such that $g(x_1, \ldots, x_n, y) = 0$.

Given a numerical function g of $n+1$ variables, this defines a new numerical function f of n variables provided the following condition is satisfied:

$$(*) \qquad \forall x_1 \ldots \forall x_n \exists y,\ g(x_1, \ldots, x_n, y) = 0$$

(Here "$\forall$" means "for all" and "$\exists$" means "there exists".)

If condition (*) is not satisfied then f is not a numerical function, but only a partial numerical function; its domain is a proper subset of $\mathbf{N} \times \ldots \times \mathbf{N}$, the set of all n-tuples of natural numbers.

1.5.1 PROPOSITION *In the minimization scheme, if g is calculable so is f.*

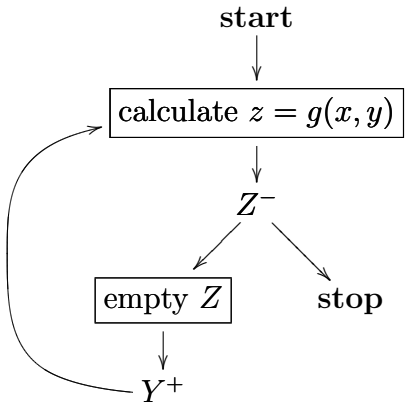
PROOF To make things concrete, let us consider the special case $n = 1$. Then R–6 becomes:

$$f(x) = \text{ smallest } y \text{ such that } g(x, y) = 0$$

and the condition reads

$$(*) \qquad \forall x\, \exists y,\ g(x, y) = 0$$

Now consider the following program:



X	Y	Z
x	0	0
x	0	$g(x,0) \neq 0$
x	1	$g(x,1) \neq 0$
.	.	.
x	y	$g(x,y) = 0$

We note that when $g(x, y) = 0$ for the first time, then $y = f(x)$; hence the program does calculate the function f at x. Suppose however that there does not exist a natural number y such that $g(x, y) = 0$, then the above calculation does not terminate, and f is not defined at x. We observe that Z is a temporary storage location.

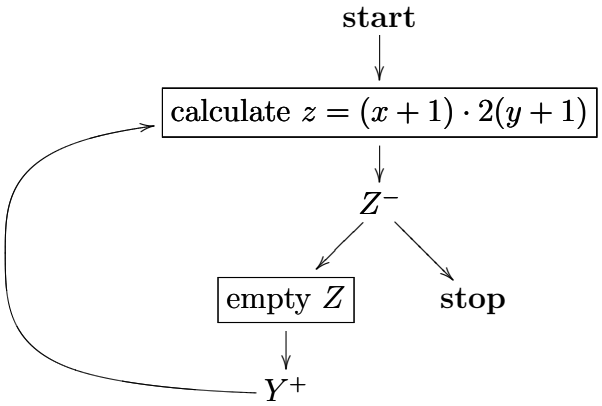
1.5.2 EXAMPLE Take $g(x, y) = x \dot{-} y$. Then $f(x) = x$. Thus f is the identity function, a special case of R–3. This example illustrates that a function defined by means of scheme R–6 may very well be definable without using scheme R–6.

1.5.3 EXAMPLE We recall that, for any real number ξ, $[\xi]$ denotes the greatest integer not exceeding ξ. Thus $[\xi] = n$ if and only if $n \leq \xi < n + 1$. $[\xi]$ may also be regarded as the smallest integer n such that $n + 1 > \xi$.

Let us calculate the function $y = [x/2]$ with the help of the minimization scheme. Indeed,

$$\begin{aligned} [x/2] &= \text{greatest } y \text{ such that } 2y \leq x \\ &= \text{smallest } y \text{ such that } 2(y+1) > x \\ &= \text{smallest } y \text{ such that } 2(y+1) \geq x + 1 \\ &= \text{smallest } y \text{ such that } (x+1) \dot{-} 2(y+1) = 0. \end{aligned}$$

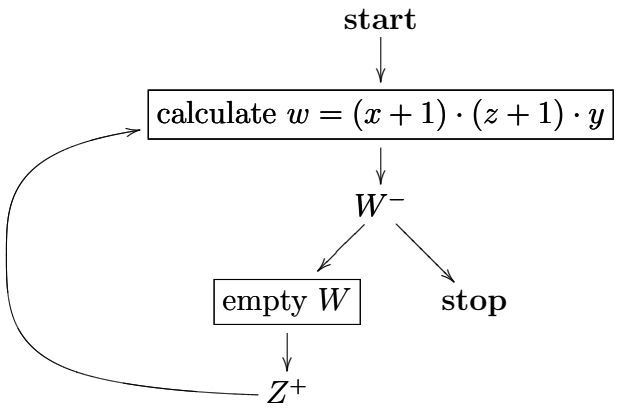
Clearly, condition $(*)$ is satisfied in this case. A program for calculating $y = [x/2]$ is thus given by the following flow diagram, where Z is a temporary storage location:



1.5.4 EXAMPLE More generally we may try to calculate

$$\begin{aligned} [x/y] &= \text{greatest } z \text{ such that } *\, y \leq x \\ &= \text{smallest } (\text{ such that } z+1) * y > x \\ &= \text{smallest } (\text{ such that } z+1) * y \geq x + 1 \\ &= \text{smallest } (\text{ such that } x+1)(z+1)y = 0 \end{aligned}$$

The program will be the following, using W as a temporary storage location:

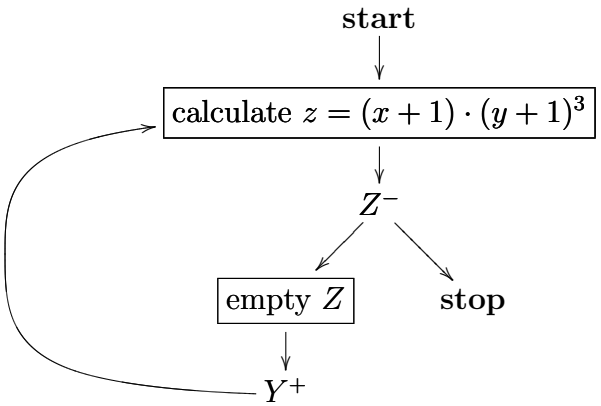


Of course, $[x/0]$ is undefined. In fact, it is easily seen that a calculation with this program does not terminate when $y = 0$.

Exercises

1. Using the minimization scheme, show how to calculate $y = [\sqrt{x}\,]$ and $y = [log_{10}(x+1)]$. Draw the flow diagrams.

2. What function is calculated by the following program? Explain how you got your answer.



1.6 Recursive functions and relations

We shall make a definition.

1.6.1 Definition *The class of* **recursive** *functions is the smallest class of numerical functions which contains all functions of types R–1, R–2, and R–3, and which is closed under schemes R–4, R–5, and R–6, where R–6 satisfies condition* $(*)$.

1.6.2 THEOREM *Every recursive function is calculable.*

PROOF This is an immediate consequence of Theorem 1.3.2, which asserts that every primitive recursive function is calculable, and of Proposition 1.5.1 which asserts that the class of calculable functions is closed under the minimization scheme. ∎

1.6.3 DEFINITION *The class of* **partial recursive** *functions is the smallest class of partial numerical functions which contains all functions of types R–1, R–2 and R–3, and which is closed under schemes R–4, R–5, and R–6, where R–6 need not satisfy condition (∗).*

For example, $z = [x/y]$ is only a partial recursive function, as it is undefined when $y = 0$. On the other hand, $z = [x/(y+1)]$ is always defined hence it is a recursive function. We can artificially introduce a function.

$$[x/y]' = \begin{cases} 0 & \text{when } y = 0 \\ [x/y] & \text{when } y > 0 \end{cases}$$

Then

$$[x/y]' = \text{ smallest } z \text{ such that } y = 0 \text{ or } (z+1) * y > x$$

It will become clear later that this equation can be written in the form

$$[x/y]' = \text{ smallest } z \text{ such that } g(x, y, z) = 0$$

where g is a suitable recursive function. Thus $z = [x/y]'$ will be a recursive function. (Actually, it is even a primitive recursive function, as is seen by looking at the book by Kleene.)

We write "$R(x, y)$" to say that x bears the relation R to y. "$R(x, y)$" is a statement capable of being true or false, and we might think of R as a function from $\mathbf{N} \times \mathbf{N}$ to $\{T, F\}$, the set of truth values. We regard R as a binary **relation**, and this is a special case of an n-ary relation, when $n = 2$. Unary relations, when $n = 1$, are also called **properties**.

If R and S are relations, binary for the sake of definiteness, we define new relations $R \wedge S$, $R \vee S$ and $\neg R$ by saying that $R \wedge S$ (read R **and** S) holds at (x, y) if and only if both $R(x, y)$ and $S(x, y)$, $R \vee S$ (read R **or** S) holds at (x, y) if and only if at least one of $R(x, y)$ or $S(x, y)$ does and $\neg R$ (read **not** R) holds at (x,y) if and only if $R(x, y)$ fails. We also say that $R \wedge S$ is the **conjunction** of R and S, that $R \vee S$ is the **disjunction** of R and S and that $\neg R$ is the **negation** of R.

We shall now define what we mean by a relation or property being recursive. For concreteness we restrict attention to the cases $n = 2$ and $n = 1$.

1.6.4 Definition *A binary relation R (property P) is said to be* **recursive** *if there is a recursive function g such that $R(x, y)$ if and only if $g(x, y) = 0$ ($P(x)$ if and only if $g(x) = 0$) for all natural numbers x and y.*

For example, $\geq$ is a recursive relation, since $x \geq y$ if and only if $y \dot{-} x = 0$, and equality is a recursive relation, since $x = y$ if and only if $|x - y| = 0$. The property of being even is recursive, since x is even if and only if $[x/2] * 2 = x$, that is, $|[x/2] * 2 - x| = 0$.

The usefulness of this definition arises from the fact that one can apply the minimization scheme to a recursive relation to produce a recursive function:

$$\begin{aligned} f(x) &= \text{smallest } y \text{ such that } R(x, y) \\ &= \text{smallest } y \text{ such that } g(x, y) = 0 \end{aligned}$$

The condition $(*)$ of Section 1.5 then becomes:

$$\forall x \exists y R(x, y)$$

1.6.5 Proposition *The class of recursive relations (properties) is closed under conjunction, disjunction, and negation.*

Proof Suppose

$$R(x, y) \text{ if and only if } g(x, y) = 0$$

and

$$S(x, y) \text{ if and only if } h(x, y) = 0$$

where g and h are recursive functions. Then

$$R(x, y) \wedge S(x, y) \text{ if and only if } g(x, y) + h(x, y) = 0$$

$$R(x, y) \vee S(x, y) \text{ if and only if } g(x, y) * h(x, y) = 0$$

and

$$\neg R(x, y) \text{ if and only if } 1 \dot{-} g(x, y) = 0$$

Since $+$, $*$ and $\dot{-}$ are all recursive functions, it follows that $g(x, y) + h(x, y)$, $g(x, y) * h(x, y)$ and $1 \dot{-} g(x, y)$ are all recursive. ∎

It is now clear that the function $z = [x/y]'$ defined earlier is recursive. In fact

$$\begin{aligned} [x/y]' &= \text{smallest } z \text{ such that } y = 0 \text{ or } (z + 1) * y \geq x + 1 \\ &= \text{smallest } z \text{ such that } y = 0 \text{ or } (x + 1) \dot{-} (z + 1) * y = 0 \\ &= \text{smallest } z \text{ such that } y * ((x + 1) \dot{-} (z + 1) * y) = 0. \end{aligned}$$

Exercise

Show that the following properties and relations are recursive, in each case producing a function g as in Definition 3.

1. x is odd.
2. x is a perfect square, that is, $x = [\sqrt{x}]^2$.
3. $x < y$.
4. $x \neq y$.
5. $x < y \leq z$ (ternary relation).

1.7 How to calculate the nth prime

One saxs that y **divides** x and writes $y|x$ when $\exists z, (y * z = x)$. When $y \neq 0$, this is equivalent to saying that $y * [x/y] = x$. On the other hand, $0|x$ only when $x = 0$. Thus, in any case, $y|x$ if and only if $y * [x/y]' = x$ if and only if $x \dot{-} (y * [x/y]') = 0$. Therefore the relation "divides" is recursive.

Let us now look at the property of being prime. A natural number x is called **prime**, and we write prim(x), provided $x > 1$ and x has no divisor $y > 1$ other than x. In other words, prim(x) if and only if

$$x > 1 \text{ and } x = \text{ smallest } y \text{ such that } y > 1 \text{ and } y|x$$

Unfortunately, when $x \leq 1$, there is no y such that $y > 1$ and $y|x$, hence the minimization scheme which appears in the above definition of prim(x) defines only a partial function. To make this into a function we write

$$\text{prim}(x) \text{ if and only if } x > 1 \wedge x = \phi(x)$$

where

$$\begin{aligned} \phi(x) &= \text{ smallest } y \text{ such that } x \leq 1 \vee (y > 1 \wedge y|x) \\ &= \text{ smallest } y \text{ such that } g(x, y) = 0 \end{aligned}$$

where

$$g(x, y) = (x \dot{-} 1) * ((2 \dot{-} y) + (x \dot{-} y * [x/y]'))$$

Once this has been checked, it follows that

$$\begin{aligned} \text{prim}(x) \quad &\text{if and only if } (2 \dot{-} x = 0) \wedge (x \dot{-} \phi(x)) = 0 \\ &\text{if and only if } \psi(x) = 0 \end{aligned}$$

where

$$\psi(x) = (2 \dot{-} x) + (x \dot{-} \phi(x))$$

The two functions $\phi(x)$ and $\psi(x)$ are of no importance except that they serve to show that prim is a recursive property.

The reader will be familiar with the sequence of prime numbers

$$2, 3, 5, 7, 11, \ldots$$

We wish to put $2 = p(0)$, $3 = p(1)$, etc. We thus write $p(x)$ for the $(x+1)$st prime. The function $y = p(x)$ may be defined recursively thus:

$$\begin{cases} p(0) = 2 \\ p(S(x)) = \text{ smallest } y \text{ such that } \text{prim}(y) \wedge (y > p(x)) \end{cases}$$

That is to say, $p(S(x)) = q(x, p(x))$, where

$$q(x, u) = \text{ smallest } y \text{ such that } \chi(u, y) = 0$$

where

$$\chi(u, y) = \psi(y) + (u + 1) \dot{-} y$$

We must check that $\forall u \, \exists y$, $\chi(u, y) = 0$. This amounts to showing that for each natural number x there exists a prime number y which is greater than $p(x)$. This fact was shown by Euclid, who observed that

$$1 + p(0) * p(1) * \cdots * p(x)$$

is not divisible by the first $x + 1$ prime numbers $p(0)$, $p(1)$, ..., $p(x)$ (since it leaves a remainder of 1 when divided by any of them). Knowing that it must be divisible by some prime number y (every integer > 1 is) he deduced that $y > p(x)$.

Therefore $y = p(x)$ is a recursive function. Since all recursive functions are calculable, it follows that there is a program to calculate $y = p(x)$ on the abacus. If the reader wishes to take the trouble, he could draw a flow diagram for this program, making use of known subroutines.

We can also calculate the function

$$\pi(x) = \text{ number of primes } < x$$

We note that this has the following recursive definition:

$$\begin{cases} \pi(0) = 0 \\ \pi(S(x)) = \pi(x) + \delta(\psi(x)) \end{cases}$$

where $\delta(x)$ was defined in Section 1.4. Indeed,

$$\delta(\psi(x)) \begin{cases} 1 & \text{if } \psi(x) = 0, \text{ that is, } x \text{ is a prime} \\ 0 & \text{otherwise} \end{cases}$$

One may also define $\pi(x)$ in terms of $p(x)$ by noting that

$$p(x) = \text{ smallest } y \text{ such that } x < \pi(y+1)$$

For technical reasons we shall later require a certain function in two variables $\exp(x, y)$, but we prefer to write $\exp_x(y)$. By this we mean the **exponent** of x in y, that is,the greatest z for which x^z divides y. For example, $\exp_5(100) = 2$, since 5^2 divides 100 but 5^3 does not. In fact, we may put

$$\exp_x(y) = \text{ smallest } z \text{ such that } \neg(x^{z+1}|y)$$

utilizing the minimization scheme, at least when $x \neq 1$ and $y \neq 0$. The only trouble is that Condition $(*)$ of Section 1.5 may be violated: x^{z+1} may divide y for all z. This will indeed be the case when $x = 1$ or $y = 0$. We arbitrarily put $\exp_x(y) = 0$ in these two exceptional cases. We now end up with the following definition:

$$\exp_x(y) = \text{ smallest } z \text{ such that } x = 1 \vee y = 0 \vee \neg(x^{z+1}|y)$$

Exercises

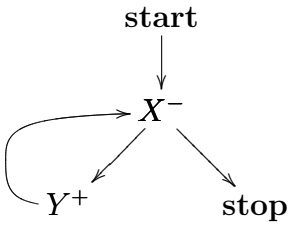
1. Draw a flow diagram for a program which will calculate the function $\psi(x)$ of the text, which has the property that x is a prime if and only if $\psi(x) = 0$.
2. Using the above exercise as a subroutine, exhibit the flow diagram of a program which will calculate $\pi(x)$, the number of primes less than x.
3. Using either of the above, draw the flow diagram of a program for calculating $p(x)$, the $(x+1)$st prime.

Chapter 2

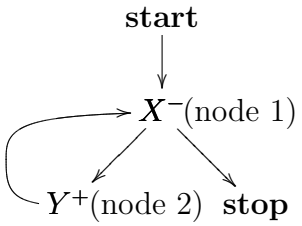
More about calculability

2.1 What is a program?

In Section 1.1 we described programs for an abacus informally with the help of flow diagrams. We now wish to give a precise definition. Let us begin by looking at an example. We recall the program "transfer content of X to Y":



Let us assign numerical labels to the **nodes** of this flow diagram (which correspond to the instructions of the program) other than those already labelled "start" and "stop".



The program can now be represented by the following table:

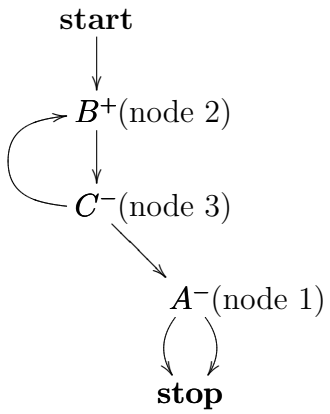
	start	1	2
yes	1	2	1
no	1	**stop**	1
location		X	Y
action		**minus**	**plus**

This table tells us the following: At node 1, carry out the operation X^-. Can it be done? Yes, then go to node 2; no, then go to node **stop**. At node 2, carry out the operation Y^+. Can it be done? Yes, then go to node 1; no, then also go to node 1. (Of course the negative alternative cannot really occur.) At node **start** do nothing to any location and go to node 1 in any case.

Suppose, for example, we are given the following table:

	start	1	2	3
yes	2	**stop**	3	2
no	2	**stop**	3	1
location		A	B	C
action		**minus**	**plus**	**minus**

It is clear that this can only correspond to the following flow diagram, probably representing a pretty stupid program:



Let us write "α" for "yes", "β" for "no", "γ" for "location" and "ϵ" for "action". Then α, β, γ and ϵ are clearly functions. To describe the source and target of each of these functions, we let L be the set of locations, and we assume that there is given a finite set N of **nodes**,

$$N = \{\mathbf{start}, 1, 2, \ldots, k, \mathbf{stop}\}$$

Then

$$\begin{aligned} \alpha: &\quad N - \{\mathbf{stop}\} \longrightarrow N - \{\mathbf{start}\} \\ \beta: &\quad N - \{\mathbf{stop}\} \longrightarrow N - \{\mathbf{start}\} \\ \gamma: &\quad N - \{\mathbf{stop}, \mathbf{start}\} \longrightarrow L \\ \epsilon: &\quad N - \{\mathbf{stop}, \mathbf{start}\} \longrightarrow \{\mathbf{plus}, \mathbf{minus}\}. \end{aligned}$$

(Here $\mathbf{N} - \{\mathbf{stop}\}$ is short for $\{\mathbf{start}, 1, 2, \ldots, k\}$, etc.)

A **program** may now be defined as a quadruple $(\alpha, \beta, \gamma, \epsilon)$ subject to the restriction that $\alpha(n) = \beta(n)$ if $n = \mathbf{start}$ or if $\epsilon(n) = \mathbf{plus}$.

Exercises

1. Make up a table for the program "copy A into B".
2. Draw the flow diagram corresponding to the following table:

	start	1	2	3	4
yes	1	2	**stop**	4	3
no	1	2	3	4	**stop**
location		A	B	C	B
action		**plus**	**minus**	**plus**	**minus**

2.2 Calculable functions are recursive

Let there be given a program $(\alpha, \beta, \gamma, \epsilon)$ for the abacus. We have talked about the finite set of nodes

$$N = \{\mathbf{start}, 1, 2, \ldots, k, \mathbf{stop}\}$$

and the infinite set of locations

$$L = \{A, B, C, \ldots\}$$

For our present purpose it will be more convenient if both nodes and locations are denoted by numbers, thus we let

$$N = \{0, 1, 2, \ldots, k, k+1\}$$

$$L = \{1, 2, 3, \ldots\}$$

A **stage** of a calculation is described by the **node** n and the **configuration** $(n_1, n_2, \ldots)$ where n_i is the number of pebbles at the location i. It is understood that all but a finite number of locations are empty, so that all but a finite number of the n_i are zero. With each stage we associate its so-called Gödel number

$$s = 2^n 3^{n_1} 5^{n_2} \cdots$$

where all but a finite number of factors are equal to 1.

Given any positive integer s, by factoring it into prime powers, we can read off the stage whose Gödel number it is (assuming the exponent of 2 does not exceed $k+1$). Note that the factorization of a positive integer into a prime powers is unique.

What does our program $(\alpha, \beta, \gamma, \epsilon)$ do to a stage of a calculation? It determines the next stage; in particular,

C–1. If $n = 0$, it changes node n to node $\alpha(n)$.

C–2. If $0 < n \leq k$ and $\epsilon(n) =$ **plus**, it adds one pebble to location $\gamma(n)$ and changes node n to node $\alpha(n)$.

C–3. If $0 < n \leq k$, $\epsilon(n) =$ **minus** and $\gamma(n)$ is nonempty, it takes one pebble from location $\gamma(n)$ and changes node n to node $\alpha(n)$.

C–4. If $0 < n \leq k$, $\epsilon(n) =$ **minus** and $g(n)$ is empty, it leaves the configuration unchanged, but changes node n to node $\beta(n)$.

C–5. If $n = k + 1$, there is no next stage.

Assuming that $n \leq k$, so that there is a next stage, let its Gödel number be $G(s)$. Then putting $n = exp_2(s)$, we obtain

$$G(s) = \begin{cases} s * 2^{\alpha(0)} & \text{if } n = 0 \\ s * 2^{\alpha(n)-n} * p(\gamma(n)) & \text{if } n > 0 \text{ and } \epsilon(n) = \textbf{plus} \\ s * 2^{\alpha(n)-n} / p(\gamma(n)) & \text{if } n > 0,\ \epsilon(n) = \textbf{minus} \text{ and } p(\gamma(n))|s \\ s * 2^{\beta(n)-n} & \text{if } n > 0,\ \epsilon(n) = \textbf{minus} \text{ and } \neg(p(\gamma(n))|s) \end{cases}$$

So far, $G(s)$ is undefined when $n \geq k + 1$ and also when $s = 0$. Let us arbitrarily put

$$G(s) = 0 \text{ if } s = 0 \text{ or } n \geq k + 1$$

With a little bit of extra work, one can rewrite the above as follows:

$$G(s) = \begin{cases} g_1(s) & \text{if } f_1(s) = 0 \\ g_2(s) & \text{if } f_2(s) = 0 \\ g_3(s) & \text{if } f_3(s) = 0 \\ g_4(s) & \text{if } f_4(s) = 0 \\ 0 & \text{if } s = 0 \text{ or } \exp_2(s) \geq k + 1 \end{cases}$$

where g_1, g_2, g_3, g_4 and f_1, f_2, f_3, f_4 are recursive functions. for example, $g_1(s) = s * 2^{\alpha(0)}$ and $f_1(s) = \exp_2(s)$. The others are a little harder. Now for each s, exactly one the five cases holds, hence

$$G(s) = g_1(s)\delta(f_1(s)) + \cdots + g_4(s)\delta(f_4(s))$$

and so $G(s)$ is seen to be a recursive function.

Iterating the function $G(s)$, we obtain the function $G^t(s)$ which may be regarded as a function of two variables and is defined recursively thus:

$$\begin{cases} G^0(s) = s \\ G^{S(t)}(s) = G(G^t(s)) \end{cases}$$

What do we mean by saying that the program $(\alpha, \beta, \gamma, \epsilon)$ calculates the function $y = f(x)$? Take $X = 1$ and $Y = 2$, then the initial and final configurations are given by

1	2
x	0
x	$f(x)$

and hence the Gödel numbers of the initial and final stages are

$$s = 2^0 3^x 5^0, \qquad G^{m(x)}(s) = 2^{k+1} 3^x 5^{f(x)}$$

where

$$m(x) = \text{ smallest } t \text{ such that } \exp_2 G^t(3^x) = k + 1$$

Therefore

$$f(x) = \exp_5 G^{m(x)}(3^x)$$

Since $f(x)$ is constructed from known recursive functions with the help of the minimization and substitution schemes, it follows that $f(x)$ is recursive. We have thus proved the following:

2.2.1 THEOREM *Every calculable numerical function is recursive.*

As we have already established the converse of this in Section 1.6, we conclude that "calculable" and "recursive" are synonymous terms.

Are there any numerical functions which are not calculable or recursive? We shall answer this question after a short digression.

Exercises

1. A calculation has arrived at node 5 of the flow diagram of a certain program. There are 3 pebbles at A, 1 pebble at C and no pebbles at any other location. What is the Gödel number of this stage of the calculation?
2. Assuming that a certain program contains enough instructions (nodes), describe the stage of a calculation corresponding to the Gödel number $90,000$?

2.3 Digression: enumerations

A set X is said to be **countable** (or **enumerable**) if it may be put into one-to-one correspondence with the set of natural numbers $\mathbf{N}$, that is, if there is a function $f : \mathbf{N} \longrightarrow\!\!\!\!\!\rightarrow X$ which is one-to-one and onto.

For example, the following sets are countable.

$$\begin{array}{lcl} \{0,1,2,3,\ldots\} & = & \text{set of natural numbers} \\ \{1,2,3,4,\ldots\} & = & \text{set of positive integers} \\ \{0,2,4,6,\ldots\} & = & \text{set of even natural numbers} \\ \{0,1,4,9,\ldots\} & = & \text{set of perfect squares} \\ \{2,3,5,7,\ldots\} & = & \text{set of prime numbers} \end{array}$$

The enumeration function f in these examples is given by

$$f(n) = n,\ n+1,\ 2n,\ n^2,\ p(n)$$

all of which are recursive functions. It is easily seen that every infinite subset X of $\mathbf{N}$ is countable, although the function $f : \mathbf{N} \longrightarrow X$ will not in general be recursive, as will be seen later.

A countable set X need not be a subset of $\mathbf{N}$. For example, the set of integers is countable:

$$\{0, 1, -1, 2, -2, \ldots\}$$

A more interesting example is the set $\mathbf{N} \times \mathbf{N}$ of all pairs of natural numbers. This is enumerated thus:

$$\begin{array}{l} (0,0), \\ (0,1),\ (1,0), \\ (0,2),\ (1,1),\ (2,0), \\ (0,3),\ (1,2),\ (2,1),\ (3,0), \\ \quad . \ . \ . \ . \ . \ . \ . \ . \ . \ . \end{array}$$

We first take all pairs with sum 0, then all pairs with sum 1, etc. Now the number of entries in the first n lines of the above array is $1+2+3+\cdots+n = n(n+1)/2$. It is easily checked that the function $f : \mathbf{N} \times \mathbf{N} \longrightarrow \mathbf{N}$ is given by $f(x,y) = ((x+y)(x+y+1)/2) + x$, clearly a recursive function in two variables.

As our next example, consider the set of all finite sequences of natural numbers. With each finite sequence $(a_0, a_1, \ldots, a_n)$ we associate the number $2^{a_0+1}3^{a_1+1}\cdots p(n)^{a_n+1}$. This gives a one-to- one correspondence between the set of finite sequences of natural numbers and a subset of $\mathbf{N}$.

An example of a set which is not countable is the set of all properties of natural numbers. Suppose this set can be enumerated, then all properties of natural numbers are contained in the sequence $P_0, P_1, P_2, \ldots$. Let us now defined a new property P as follows: $P(x)$ if and only if $\neg P_x(x)$. We don't care which natural numbers have property P, but we do know that P is not in the above sequence. For suppose $P = P_k$, then

$$\begin{array}{ll} P(k) & \text{if and only if } P_k(k) \\ & \text{if and only if } \neg P(k) \end{array}$$

clearly a contradiction. It follows that the set of all properties of natural numbers cannot be enumerated.

The above argument is due to Cantor, and it is known as Cantor's diagonal argument. Another application of this argument is to the set of all functions $\mathbf{N} \longrightarrow \mathbf{N}$. Suppose this set can be enumerated thus: f_0, f_1, f_2, ... Let us now define a function $f : \mathbf{N} \longrightarrow \mathbf{N}$ by putting $f(x) = f_x(x) + 1$. Suppose this function was a member of the sequence, then $f = f_k$, for some $k \in \mathbf{N}$. Hence

$$f_k(x) = f_x(x) + 1$$

for all $x \in \mathbf{N}$. Taking $x = k$, we get the contradiction

$$f_k(k) = f_k(k) + 1$$

Therefore the set of functions $\mathbf{N} \longrightarrow \mathbf{N}$ cannot be enumerated.

A third application of Cantor's diagonal argument deals with the set of all real numbers x such that $0 \leq x < 1$. Suppose this set can be enumerated thus:

$$\begin{aligned} a_1 &= 0.a_{11}a_{12}a_{13}\cdots \\ a_2 &= 0.a_{21}a_{22}a_{23}\cdots \\ a_3 &= 0.a_{31}a_{32}a_{33}\cdots \\ &\cdots\cdots\cdots \end{aligned}$$

where the a_{ij} are digits from 0 to 9. It will be convenient to assume that none of these decimal expansions involve an infinite sequence of nines, since every such expansion can be replaced by another one with an infinite sequence of zeros. (For example, $0.1999\cdots = 0.2000\cdots$. Let us now construct a number

$$b = 0.b_1b_2b_3\cdots$$

where b_1 is neither a_{11} nor 9, b_2 is neither a_{22} nor 9, and so on. Then $b \neq a_1$ since they differ in the first decimal place, $b \neq a_2$ since they differ in the second decimal place, $b \neq a_3$ since they differ in the third decimal place and so on. But $0 \leq b < 1$ so we must have $b = a_n$ for some n, a contradiction. It follows that the set of real numbers in the unit interval cannot be enumerated.

We shall meet yet another application of Cantor's diagonal argument in the next section.

Exercises

1. Explain why each of the following sets is countable:

a. the set of odd natural numbers,

b. the set of cubes of natural numbers,

c. the set of odd integers,

d. the set of rational numbers,

e. the set of 3-tuples of natural numbers.

2. Explain why the following sets are not countable:

a. the set of real numbers,

b. the set of complex numbers,

c. the set of irrational numbers.

2.4 Example of a noncalculable function

Every calculable numerical function $\mathbf{N} \longrightarrow \mathbf{N}$ can be calculated with the help of some program $(\alpha, \beta, \gamma, \epsilon)$. The program may be described explicitly as an array of numbers; provided we encode **plus** and **minus** as numbers, for example, as 1 and 0 respectively.

$$\begin{array}{ccccc} 0 & 1 & \cdots & k & k+1 \\ \alpha(0) & \alpha(1) & \cdots & \alpha(k) & \alpha(k+1) \\ \beta(0) & \beta(1) & \cdots & \beta(k) & \beta(k+1) \\ & \gamma(1) & \cdots & \gamma(k) & \gamma(k+1) \\ & \epsilon(1) & \cdots & \epsilon(k) & \epsilon(k+1) \end{array}$$

It is clear that these arrays can be enumerated, hence the set of programs is countable. Thus the set of calculable functions $\mathbf{N} \longrightarrow \mathbf{N}$ is countable. On the other hand, the set of all numerical functions $\mathbf{N} \longrightarrow \mathbf{N}$ is not countable, as we saw in the previous chapter. Therefore not all numerical functions are calculable. We shall actually construct a numerical function which is not calculable.

Let P_0, P_1, P_2, ... be an enumeration of all programs. Suppose an initial configuration of x pebbles at location 1 and 0 pebbles at location 2 is transformed by program P_n to a final configuration of x' pebbles at location 1 and y pebbles at location 2[1]. Then we shall write $f_n(x) = y$ and say the program P_n **calculates** f_n at x. If the calculation does not terminate, we shall say that $f_n(x)$ is undefined. Thus $f_n(x)$ is a partial recursive function in x. (Note that in the present context we do not insist that $x' = x$ as in an earlier chapter. However, it is easily seen that some slightly modified program would calculate $f_n(x)$ in the earlier sense.) Writing $f(x, n) = f_n(x)$, we obtain a partial recursive function in two variables.

Let us say that n has property G if the above calculation terminates for each x, that is, if $f_n(x)$ is a function, not just a partial function. Let $g(y)$ be a numerical function

[1]We do not care what happens at other locations.

which enumerates all numbers with property G. For example, we may write

$$\begin{cases} g(0) = \text{ smallest } n \text{ such that } G(n) \\ g(S(y)) = \text{ smallest } n \text{ such that } n > g(y) \wedge G(n) \end{cases}$$

The right hand side of this last equation is well defined, since there are evidently infinitely many natural numbers with property G, as there are infinitely many calculable functions. Thus $g(y)$ is a numerical function which enumerates all natural numbers with property G.

While $f(x, n)$ is only a partial function,

$$h(x, y) = f(x, g(y)) = f_{g(y)}(x)$$

is a genuine function of two variables, since $g(y)$ belongs to the set G. From this we obtain the function $h(x, x)$ of one variable by diagonalizing, that is, by putting $y = x$. We claim that the function $h(x, x)$ is not calculable.

Indeed, suppose it is calculable, then so is $h(x, x) + 1$. Let P_m be a program which calculates the latter function. Clearly, m has property G, hence we may write $m = g(k)$, for some natural number k. Therefore

$$h(x, x) + 1 = f_m(x) = f(x, m) = f(x, g(k)) = h(x, k)$$

Now, this is true for all $x \in \mathbf{N}$. Taking $x = k$, we get

$$h(k, k) + 1 = h(k, k)$$

a contradiction.

We have thus exhibited a function $h(x, x)$ which is not calculable on an abacus. We may squeeze a little more out of our argument.

It can be shown that $f(x, n)$ is a partial recursive function. The proof of this fact is not difficult, but a little tedious, and we shall skip it. Let us assume this result. We claim it then follows that $g(y)$ is a not a recursive function.

Indeed, suppose $g(y)$ is recursive. Then so is $h(x, y) = f(x, g(y))$, hence also $h(x, x)$. But we know that $h(x, x)$ is not calculable; in other words, it is not recursive. Thus $g(y)$ is not recursive. Since $g(y)$ was any numerical function which enumerates all natural numbers with property G, it follows that no recursive function can enumerate these numbers. We shall return to this statement in Section 2.6.

The program for calculating $f(x, y)$ may be used as a **super-program** for calculating all partial recursive functions in one variable, since every such partial function has the form $f_n(x) = f(x, n)$ for some n.

2.5 Turing programs

We have seen how to perform calculations on an abacus consisting of an arbitrary number of locations, each location being capable of storing arbitrarily many pebbles. One can overcome one's misgivings about an infinite number of locations by reassuring oneself that it is always possible to dig another hole in the ground when needed. But holes of infinite depth are less easy to accept.

Let us see what happens if we restrict the storage capacity of each location to be at most m pebbles. One has in mind the classical abacus with $m = 9$, to facilitate representations of numbers in the scale of 10. But we may as well go all the way and take $m = 1$. Thus each location can hold at most one pebble. Let us call this the **binary abacus**, because each location must be in one of two states: empty or full.

The elementary instruction $\downarrow X^+ \downarrow$ of the original abacus makes no sense for the binary abacus, as one cannot put a pebble into a full location. Instead we shall adopt a new elementary instruction $\downarrow X^+ \swarrow\ \searrow$. If X is empty, we are asked to put a pebble into location X and proceed along the left arrow. If X is full, we are asked to leave X alone and proceed along the right arrow. We may now enquire whether every calculation that can be performed on the original abacus can also be carried out on the binary abacus. Instead of arranging the locations in a single sequence

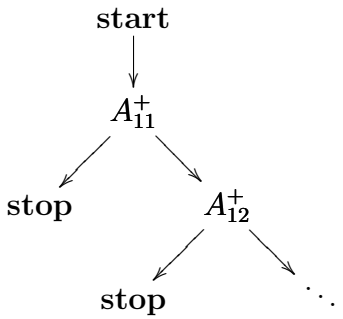
$$A_1 \quad A_2 \quad A_3 \quad \cdots$$

we may put them in double array as follows:

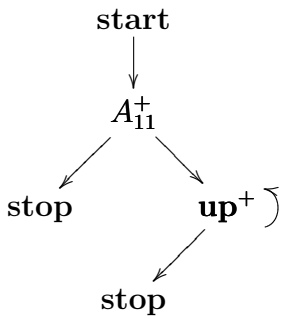
$$\begin{array}{cccc} \vdots & \vdots & \vdots & \iddots \\ A_{13} & A_{23} & A_{33} & \cdots \\ A_{12} & A_{22} & A_{32} & \cdots \\ A_{11} & A_{21} & A_{31} & \cdots \end{array}$$

Instead of having n pebbles at location A_1 we may have one pebble each at locations A_{11}, A_{12}, …, A_{1n}. (It is convenient to represent the number n in this way, and not in the scale of 2.) The old instruction A_1^+ may now be translated into a program for putting one pebble into the first empty location in the column A_{11}, A_{12}, …. What does such a

program look like?



Unfortunately, this will be an infinite flow diagram, unless we change our method of describing programs. Indeed, the following finite flow diagram will explain what we have in mind:



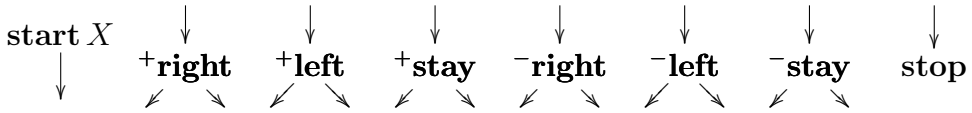
Here "up" means "one location higher".

Turing's idea is even more radical. He arranges the locations in one line extending infinitely in both directions. Thus the locations are in a natural one-to-one correspondence with the integers:

$$\ldots, A_{-2}, A_{-1}, A_1, A_2, \ldots$$

Moreover he never allows you to move from one location to another, unless the new location is adjacent to or coincides with the old location.

A **Turing program** is built up from the following elementary instructions:



The first instruction says: start at location X. The second instruction says: add one pebble to the location at which you are, if it doesn't have one already, then move to the location on the right. If this new location is full, go to the node indicated by the left arrow. If the new location is empty, go to the node indicated by the right arrow.

The other instructions are interpreted similarly.

One now represents the number x on the binary abacus by $x+1$ pebbles at consecutive locations. (Thus zero is represented by one pebble.) One says that a function $y = f(x)$ is **computed** by a Turing program, if the calculation set to start with a representation of x (the last pebble being at the starting location) ends with a representation of x followed by an empty location followed by a representation of $f(x)$ (the last pebble being at the terminal location).

It can be shown that a numerical function can be computed by a Turing program if and only if it is calculable, that is, recursive.

Actually, Turing did not talk about programs but about machines. We shall give a brief description of these.

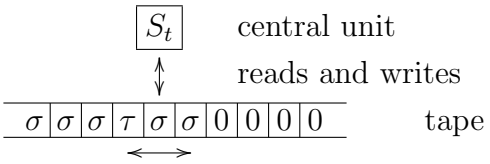
A **Turing machine** consists of a central unit which is scanning one square of an infinite tape. The central unit is in one of a finite number of states:

$$S_0 = \textbf{start},\ S_1,\ S_2,\ \ldots, S_m$$

Each square of the tape has written on it a word of the finite vocabulary Σ:

$$\sigma_1, \ldots, \sigma_n$$

or 0 = blank. For example, we may have the following situation



The scanned square may be read and written upon. The tape is capable of moving in either direction. The machine is capable of making the following kinds of moves:

(1) $$(\sigma, S_i) \longrightarrow (\tau, S_j)$$

erase σ, print τ, change state from S_i to S_j.

(2) $$(\sigma, S_i) \longrightarrow (\textbf{right}, S_j)$$

move tape one square to the right (or move scanner one square to the left), change state from S_i to S_j.

(3) $$(\sigma, S_i) \longrightarrow (\textbf{left}, S_j)$$

move tape to left and change state.

We assume the machine is **deterministic**, that is, each move is uniquely determined by (σ, S_i), $\sigma \in \Sigma$.

Without going into technical details we shall say this: if the machine is presented with a string of words on the tape when the central unit is in the state **start** and ultimately comes to a stop, then the string is said to have been **accepted** by the machine. One may ask, what sets of strings consist precisely of the acceptable strings of some Turing machine? Not surprisingly, the answer is: the recursively enumerable sets discussed in Section 2.6 below.

To compare Turing programs with Turing machines, we take $\Sigma = \{0, 1\}$ and adopt the following dictionary:

location = square on two-way infinite tape

pebble = printing of symbol "1"

program = machine

node = internal state

location at one stage of calculation = scanned square

The proof that a function is computable on a Turing machine if and only if it is recursive may be found in the book by Kleene. Alternatively, one may prove directly that a function is computable on a Turing machine if and only if it is calculable on the abacus; such a proof appears in the recent book by Boolos and Jeffrey.

2.6 Effective procedures

Mathematicians believe that a numerical function which can be effectively calculated or computed by any means whatsoever is calculable or computable in the technical sense of these words, hence recursive. This statement, known as the Church-Turing Thesis cannot be proved as we do not know what is meant by "effectively computable by any means whatsoever".

Given a set P of natural numbers, we may ask this: Is there an effective procedure for deciding whether any natural number n is an element of P? As a consequence of the Church-Turing Thesis we assert that there is such an effective procedure if and only if P is a **recursive set**, that is, there exists a calculable function $f : \mathbf{N} \longrightarrow \mathbf{N}$ such that $\forall x \in \mathbf{N}$, $x \in P$ if and only if $f(x) = 0$.

A related concept is embodied in the following.

2.6.1 DEFINITION *A set P of natural numbers is* **recursively enumerable** *if it is the image of a recursive function, that is, if there is a recursive function $f : \mathbf{N} \longrightarrow \mathbf{N}$ such that $P = \{f(0), f(1), f(2), \ldots\}$, with repetitions permitted.*

In particular, every finite set of natural numbers is recursively enumerable. For example, the set $\{a, b, c, d\}$ is the image of the function

$$f(x) = \begin{cases} a & \text{if } x = 0 \\ b & \text{if } x = 1 \\ c & \text{if } x = 2 \\ d & \text{if } x \geq 3 \end{cases}$$

This function is recursive, since

$$f(x) = a * \delta(x) + b * \delta(|x - 1|) + c * \delta(|x - 2|) + d * \delta(3 \dot{-} x)$$

2.6.2 THEOREM *A set P of natural numbers is recursive if and only if P and its complement $\mathbf{N} - P$ are each recursively enumerable or finite.*

PROOF Assume that P is recursive. If P is infinite, we may define a function g as follows:

$$\begin{cases} g(0) = \text{ smallest } y \text{ such that } y \in P \\ g(S(x)) = \text{ smallest } y \text{ such that } y \in P \wedge y > g(x) \end{cases}$$

Clearly $P = \{g(0), g(1), \ldots\}$, and so g affords a recursive enumeration of P.

If $\mathbf{N} - P$ is infinite, we may similarly define a function h which will recursively enumerate $\mathbf{N} - P$:

$$\begin{cases} h(0) = \text{ smallest } y \text{ such that } y \in P \\ h(S(x)) = \text{ smallest } y \text{ such that } y \in P \wedge y > h(x) \end{cases}$$

Conversely, let us assume first that P and $\mathbf{N} - P$ are both recursively enumerable, say g enumerates P and h enumerates $\mathbf{N} - P$. Since every natural number x is an element of P or of $\mathbf{N} - P$, there exists a number y such that $x = g(y)$ or $x = h(y)$. Put

$$e(x) = \text{ smallest } y \text{ such that } x = g(y) \vee x = h(y)$$

Then $x \in P$ if and only if $g(e(x)) = x$, that is, $f(x) = 0$, where $f(x) = |g(e(x)) - x|$. Next, let us assume that P is recursively enumerable and $\mathbf{N} - P$ is finite, say $\mathbf{N} - P = \{b_1, b_2, \ldots, b_m\}$. Then we define g as above and put

$$e(x) = \text{ smallest } y \text{ such that } x = g(x) \vee x = b_1 \vee x = b_2 \vee \ldots \vee x = b_m$$

Then we proceed as above. There remains the case when $\mathbf{N} - P$ is recursively enumerable and P is finite, and this case is treated quite similarly.

To illustrate the above notions, recall from Section 2.4 that a natural number n has property G if the calculation of $f_n(x)$ by the program P_n terminates. We may as well regard G as a set, then what we showed is that no recursive function can enumerate G.

In the present terminology, this means that G is not recursively enumerable. It follows from the above theorem that G is also not recursive.

In many applications of these ideas we are interested not in natural numbers, but in strings which are made up of symbols of a finite alphabet or words of a finite vocabulary. By putting these strings in alphabetic or lexicographic order, we obtain a one-to-one correspondence between strings and natural numbers. We may then speak of recursive and recursively enumerable sets of strings.

For example, let us consider the finite vocabulary $\{(,\ ),\ p,\ q,\ r,\ ',\ -,\ \Rightarrow,\ \vee\ ,\ \wedge,\ \Leftrightarrow\}$ which will allow us to deal with the propositional calculus, the propositional variables being $p,\ q,\ r,\ p',\ q',\ r',\ p''$, etc.

It is easily seen which strings are formulas (e.g. $(p \Rightarrow q) \wedge r)$) and which are not (for example $pq \Rightarrow (')$). Thus the set of formulas is recursive. So is the of tautologies, because the method of truth-tables allows us to decide effectively which formulas are tautologies.

To get a language adequate for all mathematics, we adjoin to the above vocabulary the set $\{x,\ y,\ z,\ \forall,\ \exists,\ =,\ \in\}$. Again, the set of formulas is recursive. However, it can be shown that the set of theorems is not recursive, that is, there is no effective method for deciding whether a formula is a theorem. This is just as well, or mathematicians could be replaced by computers. Still, the set of theorems is easily seen to be recursively enumerable. It then follows from the above theorem that the set of formulas which are not theorems is not recursively enumerable.

As another example, we may take as our vocabulary the set of all English words, say all the words listed in the Oxford dictionary. This set is quite large, but still finite. We are then interested in those strings which are grammatical sentences. Linguists are agreed that the set of grammatical sentences is recursively enumerable. One might even expect that this set is recursive.

Exercise

1. State whether each of the following sets is

a. recursive,

b. recursively enumerable:

(i) the set of theorems in mathematics,

(ii) the set of tautologies in the propositional calculus,

(iii) the set of statements in mathematics which are not theorems,

(iv) the set of formulas of the propositional calculus which are not tautologies,

(v) the set of programs for the abacus,

(vi) the set of programs which calculate a function $y = f(x)$.

Chapter 3

Syntactic types and semigroups

3.1 A simple grammar using types

It is the aim of grammar to describe the set of sentences in a language. There are several kinds of grammars that have been studied from a mathematical point of view. We shall first consider the method of **syntactic types**, also known as "categorial grammar". We shall illustrate this by looking at a small fragment of English.

We first define the set of types inductively

T–1. s is a type (s = sentence)

T–2. n is a type (n = name)

T–3. if a and b are types so is $a \cdot b$

T–4. if a and b are types so is a/b (read "a over b")

T–5. if a and b are types so is $a\backslash b$ (read "a under b").

For the moment we shall not accept any types other than those which may be constructed from T–1 to T–5. For example, the following expressions are types, where parentheses are used as ordinarily in mathematics:

$$s,\ s/n,\ (s/n)\cdot n,\ s/(n\cdot n),\ n\backslash(s/n)$$

We introduce a binary relation $\longrightarrow$ between types. It is our intention to interpret "$a \longrightarrow b$" to mean that every expression of type a is also of type b. The relation $\longrightarrow$ is subject to the following rules, for the time being:

Ar–1. $a \longrightarrow a$,

Ar–2. if $a \longrightarrow b$ and $b \longrightarrow c$, then $a \longrightarrow c$,

Ar–3. if $a \longrightarrow b$ and $c \longrightarrow d$, then $a \cdot c \longrightarrow b \cdot d$,

Ar–4. $(a/b) \cdot b \longrightarrow a$,

Ar–5. $a \cdot (a\backslash b) \longrightarrow b$.

Later we shall consider some additional rules.

We also use the arrow between words of the language studied and types. Thus John $\longrightarrow n$ is intended to mean that John has type n. For the sake of illustration we shall consider a small fragment of English consisting of the words John, Jane, milk, works, rests, likes, calls, poor, fresh, today, and, while, with, for.

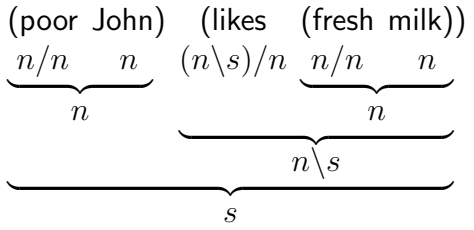
We begin by assigning types to these words. The first three words will be assigned the type n, as they are names. (Indeed, milk is the name of a substance.) If works is assigned type x, the sentence John works will have type $n \cdot x$. Rule T–5 suggests we take $x = n\backslash s$; then this sentence has type $n \cdot (n\backslash s) \longrightarrow s$. Since John likes Jane is a sentence, likes Jane should have the same type as works, that is, $n\backslash s$. Rule T–4 then suggests we let likes $\longrightarrow (n\backslash s)/n$, so we have the analysis

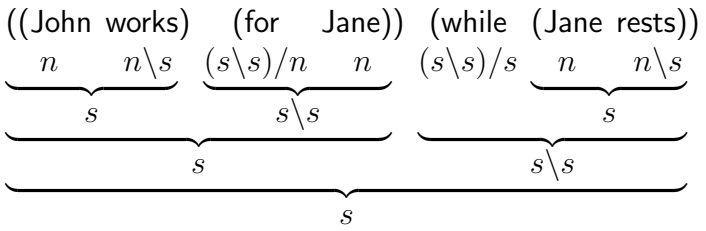
$$\underbrace{\underbrace{\text{John}}_{n}\quad \underbrace{\underbrace{\text{likes}}_{(n\backslash s)/n}\quad \underbrace{\text{Jane}}_{n}}_{n\backslash s}}_{s}$$

Continuing in this manner, we arrive at the following type assignment for our fragment of English:

John	$\longrightarrow$	n	(name)
Jane	$\longrightarrow$	n	
milk	$\longrightarrow$	n	
works	$\longrightarrow$	$n\backslash s$	(intransitive verb)
rests	$\longrightarrow$	$n\backslash s$	
likes	$\longrightarrow$	$(n\backslash s)/n$	(transitive verb)
calls	$\longrightarrow$	$n\backslash s, \quad (n\backslash s)/n$	
poor	$\longrightarrow$	n/n	(adjective)
fresh	$\longrightarrow$	n/n	
today	$\longrightarrow$	$s/s, \quad s\backslash s$	(adverb)
and	$\longrightarrow$	$(s\backslash s)/s$	(conjunction)
while	$\longrightarrow$	$(s\backslash s)/s$	
with	$\longrightarrow$	$(s\backslash s)/n$	(preposition)
for	$\longrightarrow$	$(s\backslash s)/s, \quad (s\backslash s)/n$	

A number of examples will show that one can now check the sentence-hood of strings that were not used in arriving at the above type assignments. Having learned a grammar, one should be able to use it to construct new sentences.





Of course, many meaningless sentences also turn out to be grammatical according to our rules, for example:

poor milk calls fresh John

milk rests for milk while milk works

On the other hand, some English sentences have escaped our net, for example,

Jane calls John fresh

This could be remedied by adding some further type assignments to the above list.

Let us try to adjoin the pronouns he and him to our fragment of English. Since we cannot say poor him works and Jane likes he, we must not assign the type n to he and him. Attempting to justify the sentences he works and Jane (likes him), we are led to the following assignments:

$$\begin{array}{lcl} \text{he} & \longrightarrow & s/(n\backslash s) \\ \text{him} & \longrightarrow & ((n\backslash s)/n)\backslash(n\backslash s) \end{array}$$

The type of him seems rather complicated. Had we allowed the constructions (Jane likes) John and (Jane likes) him, we should have arrived at different types for likes and him, namely

$$\begin{array}{lcl} \text{likes} & \longrightarrow & n\backslash(s/n) \\ \text{him} & \longrightarrow & (s/n)\backslash s \end{array}$$

Exercise

1. Using type assignments, check that the following are grammatical sentences:

a. poor poor John works

b. John works today today

c. he works for Jane today

d. he works, for Jane rests today

3.2 Analysis of the English verb phrase

It does not seem likely that we can push the method of syntactic types very far in applications to natural languages, as long as we insist that all types be built up from s and n alone. In this chapter we shall make use of five primitive types:

s	=	complete declarative sentence
n	=	name
i	=	infinitive of intransitive verb
p	=	present participle of intransitive verb
q	=	past participle of intransitive verb

Accordingly we adopt the type assignments:

$$\begin{array}{lcl} \mathsf{work} & \longrightarrow & i \\ \mathsf{working} & \longrightarrow & p \\ \mathsf{worked} & \longrightarrow & q \end{array}$$

Let us now look at the sentence

$$\begin{array}{ccc} \mathsf{John} & (\mathsf{must\ work}) \\ n & & i \end{array}$$

It is clear that must work should receive the same type as works, that is, $n\backslash s$. Since work has type i, we are led to assign
must $\longrightarrow (n\backslash s)/i$
The same type will be assigned to other "modal auxiliaries" such as will and may.

In the same way, by looking at the sentences

$$\begin{array}{ccc} \mathsf{John} & (\mathsf{is\ working}) \\ n & & p \end{array} \qquad \begin{array}{ccc} \mathsf{John} & (\mathsf{has\ worked}) \\ n & & q \end{array}$$

we are led to assign

$$\begin{array}{lcl} \mathsf{is} & \longrightarrow & (n\backslash s)/p \\ \mathsf{has} & \longrightarrow & (n\backslash s)/q \end{array}$$

Similarly we arrive at the following type assignments:

$$\begin{array}{cccc} \mathsf{John} & \mathsf{must} & \mathsf{have\ worked}, & \mathsf{have} \longrightarrow i/q \\ n & (n\backslash s)/t & q \end{array}$$

$$\underset{n}{\text{John}}\ \underset{(n\backslash s)/q}{\text{has}}\ \underset{p}{\text{been working}},\qquad \text{been} \longrightarrow q/p$$

$$\underset{n}{\text{John}}\ \underset{(n\backslash s)/p}{\text{is}}\ \underset{n}{\text{calling Jane}},\qquad \text{calling} \longrightarrow p/n$$

$$\underset{n}{\text{John}}\ \underset{(n\backslash s)/i}{\text{must}}\ \underset{n}{\text{call Jane}},\qquad \text{call} \longrightarrow i/n$$

$$\underset{n}{\text{John}}\ \underset{(n\backslash s)/q}{\text{has}}\ \underset{n}{\text{called Jane}},\qquad \text{called} \longrightarrow q/n$$

$$\underset{n}{\text{John}}\ \underset{q/n}{\text{is called}},\qquad \text{is} \longrightarrow (n\backslash s)/(q/n)$$

We summarize this information in the following table:

	infinitive	present participle	past participle	third person singular present tense
intransitive verb	work i	working p	worked q	works $n\backslash s$
transitive verb	call i/n	calling p/n	called q/n	calls $(n\backslash s)/n$
modal auxiliary				must $(n\backslash s)/i$
progressive auxiliary	be i/p		been q/p	is $(n\backslash s)/p$
perfect auxiliary	have i/q			has $(n\backslash s)/q$
passive auxiliary	be $i/(q/n)$	being $p/(q/n)$	been $q/(q/n)$	is $(n\backslash s)/(q/n)$

There are of course many other transitive and intransitive verbs, even have may occur as a transitive verb. On the other hand, it is difficult to think of many auxiliary verbs. Here is presumably almost a complete list:

modal auxiliaries: must, will, shall, can, may, would, should, could, might.

progressive auxiliaries: be, stop, start, keep.

passive auxiliaries: be, get.

The blank squares in the table indicate that there do not exist forms to fit those slots. In the column "modal auxiliaries" we are not surprised to note the absence of the nonexistent forms *musting, *musted, and *to must. However, there do exist forms being,

having, and had; so why do these not appear in the remaining three squares? Because we cannot say *John is being working, *John is having worked, *John has had worked. (An asterisk indicates that the expression to which it is attended does not appear in standard English.)

Our table could be enlarged by allowing for plural, past tense and 1st person forms. This project would require additional primitive types, and we shall refrain from carrying it out.

Using the above table, one can deduce that many strings of words are grammatical sentences, even if these strings were not used in constructing the table. Some such sentences turn out to be pretty unusual, for example

$$\begin{array}{ccccccc} \text{Jane} & \text{has} & \text{been} & \text{being} & \text{called} \\ n & (n\backslash s)/q & q/p & \underbrace{p/(q/n)} & \underbrace{q/n} \\ & & & \underbrace{\qquad\qquad p \qquad\qquad} & \\ & & \underbrace{\qquad\qquad\qquad q \qquad\qquad\qquad} & & \\ & \underbrace{\qquad\qquad\qquad\qquad n\backslash s \qquad\qquad\qquad\qquad} & & & \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} & & & & \\ s & & & & \end{array}$$

It would seem difficult to rule this out, without also ruling out

Jane has stopped being called

Jane has been getting called

Adverbs have been assigned the type $s\backslash s$, as in

$$\begin{array}{cccccc} (\text{John} & (\text{must} & \text{work})) & \text{today} \\ n & (n\backslash s)/i & i & s\backslash s \end{array}$$

Sometimes it may seem more natural to let an adverb have type $i\backslash i$, as in

$$\begin{array}{cccc} \text{John} & \text{must} & (\text{work well}) \\ n & (n\backslash s)/t & i & i\backslash i \end{array}$$

However, in John works well it would then be necessary to say that works has not only the simple type $n\backslash s$, but also the compound type $((n\backslash s)/i)\cdot i$. Similarly one can distinguish between "coordinate" conjunctions and "subordinate" conjunctions, as in the following sentences:

$$\begin{array}{cccccc} \text{John works} & \text{and} & \text{Jane rests} \\ n & n\backslash s & (s\backslash s)/s & n & n\backslash s \end{array}$$

$$\begin{array}{cccccc} \text{John} & \text{must} & \text{work} & \text{while} & \text{Jane rests} \\ n & (n\backslash s)/i & i & (i\backslash i)/s & n & n\backslash s \end{array}$$

Exercise

1. Using type assignments, show that the following are grammatical sentences.

a. John must have been working

b. John has stopped calling Jane

c. Jane may have stopped getting called

d. John is having milk

3.3 Syntactic calculus

We shall investigate a deductive system that will enable us to handle syntactic types in the same way as the propositional calculus enables us to handle propositions.

Primitive types are s, n, i, p, q, and perhaps others, according to need.

Types are defined inductively thus: primitive types are types, and if a and b are types so are $a \cdot b$, a/b, and $a\backslash b$.

A **formula** is an expression of the form $a \longrightarrow b$, where a and b are types.

We adopt one **axiom scheme**:

(1) $$a \longrightarrow a$$

and five **rules of inference**:

(2) $$\frac{a \longrightarrow b \qquad b \longrightarrow c}{a \longrightarrow c}$$

(3) $$\frac{a \cdot b \longrightarrow c}{a \longrightarrow c/b}$$

(4) $$\frac{a \longrightarrow c/b}{a \cdot b \longrightarrow c}$$

(5) $$\frac{a \cdot b \longrightarrow c}{b \longrightarrow a\backslash c}$$

(6) $$\frac{b \longrightarrow a\backslash c}{a \cdot b \longrightarrow c}$$

A formula is said to be a **theorem** if it can be deduced from the axioms and rules of inference by means of a **proof**. We shall illustrate what we mean by a proof by looking at some examples.

$$\frac{a/b \xrightarrow{1} a/b}{(a/b)\cdot b \longrightarrow a}\ 4 \qquad\qquad \frac{a\backslash b \xrightarrow{1} a\backslash b}{a\cdot(a\backslash b)\longrightarrow b}\ 6$$

These two proofs show that rules (4) and (5) of Section 3.1 are in fact theorems of the syntactic calculus. The theorems established by the following two proofs are new:

$$\frac{a\cdot b \xrightarrow{1} a\cdot b}{a\longrightarrow (a\cdot b)/b}\ 3 \qquad\qquad \frac{\dfrac{a\backslash b \xrightarrow{1} a\backslash b}{a\cdot(a\backslash b)\longrightarrow b}\ 6}{a\longrightarrow b/(a\backslash b)}\ 3$$

For example, the last proof establishes the theorem $n \longrightarrow s/(n\backslash s)$. This asserts that every name also has the type $s/(n\backslash s)$, the type of the pronoun he in Section 3.1. Indeed, in any context, he may be replaced by John, but not conversely. (It may also be replaced by Jane; distinctions of gender do not enter the very limited grammar discussed in Section 3.1.)

Rule (3) of Section 3.1 does not appear among our present rules of inference. However, it can be **derived**: given $a \longrightarrow b$ and $c \longrightarrow d$, there is a proof which leads from these assumptions to $a\cdot c \longrightarrow b\cdot d$. Here it is, a good example of a proof in tree form:

$$\dfrac{\dfrac{a \xrightarrow{\text{given}} b \qquad \dfrac{b\cdot c \xrightarrow{1} b\cdot c}{b\longrightarrow (b\cdot c)/c}\ 3}{a\longrightarrow (b\cdot c)/c}\ 2}{a\cdot c\longrightarrow b\cdot c}\ 4 \qquad \dfrac{\dfrac{c \xrightarrow{\text{given}} d \qquad \dfrac{b\cdot d \xrightarrow{1} b\cdot d}{d\longrightarrow b\backslash(b\cdot d)}\ 5}{c\longrightarrow b\backslash(b\cdot d)}\ 2}{b\cdot c\longrightarrow b\cdot d}\ 6$$

$$\dfrac{\qquad\qquad}{a\cdot c\longrightarrow b\cdot d}\ 2$$

We shall write this derived rule for future reference:

$$(\alpha)\qquad \frac{a\longrightarrow b \qquad\qquad c\longrightarrow d}{a\cdot c\longrightarrow b\cdot d}$$

Other derived rules of inference are:

$$(\beta)\qquad \frac{a\longrightarrow b \qquad\qquad c\longrightarrow d}{a/d\longrightarrow b/c}$$

$$(\gamma)\qquad \frac{a \longrightarrow b \qquad\qquad c \longrightarrow d}{d\backslash a \longrightarrow c\backslash b}$$

In applying the syntactic calculus to English, we assumed the English sentences are structured in a certain way. For example, we wrote John (likes Jane) but not *(John likes) Jane and thus we arrived at the type $(n\backslash s)/n$ for likes and not $n\backslash(s/n)$. As we shall see, these two types are equivalent in the associative syntactic calculus.

The **associative syntactic calculus** differs from the syntactic calculus by virtue of two additional axiom schemes:

$$(7)\qquad (a \cdot b) \cdot c \longrightarrow a \cdot (b \cdot c)$$

$$(8)\qquad a \cdot (b \cdot c) \longrightarrow (a \cdot b) \cdot c$$

We shall abbreviate $x \longrightarrow y$ and $y \longrightarrow x$, to $x \leftrightarrow y$ and say x is **equivalent** to y. Then (7) and (8) may be combined to

$$a \cdot (b \cdot c) \leftrightarrow (a \cdot b) \cdot c$$

The following are now theorems in the associative syntactic calculus:

$$\begin{array}{rcl} a\backslash(b/c) & \leftrightarrow & (a\backslash b)/c \\ a/(b \cdot c) & \leftrightarrow & (a/c)/b \\ (a \cdot b)\backslash c & \leftrightarrow & b\backslash(a\backslash c) \end{array}$$

For example, we prove $a\backslash(b/c) \longrightarrow (a\backslash b)/c$ by putting $d = a\backslash(b/c)$ proceeding as follows:

$$\dfrac{\dfrac{a \cdot (d \cdot c) \xrightarrow{8} (a \cdot d) \cdot c \qquad \dfrac{\dfrac{d \xrightarrow{1} a\backslash(b/c)}{a \cdot d \longrightarrow b/c}\ 6}{(a \cdot d) \cdot c \longrightarrow b}\ 4}{a \cdot (d \cdot c) \longrightarrow b}\ 2}{\dfrac{d \cdot c \longrightarrow a\backslash b}{d \longrightarrow (a\backslash b)/c}\ 3}\ 5$$

Another theorem is $(a/b) \cdot (b/c) \longrightarrow a/c$, which is proved as follows, making use of

the derived rule of inference (a):

$$\dfrac{\dfrac{((a/b)\cdot(b/c)\cdot c \xrightarrow{7} (a/b)\cdot((b/c)\cdot c) \qquad \dfrac{\dfrac{a/b \xrightarrow{1} a/b \qquad \dfrac{b/c \xrightarrow{1} b/c}{(b/c)\cdot c \longrightarrow b}\,4}{(a/b)\cdot((b/c)\cdot c) \longrightarrow (a/b)\cdot b}\,\alpha \qquad \dfrac{a/b \xrightarrow{1} a/b}{(a/b)/\cdot b \longrightarrow a}\,4}{(a/b)\cdot((b/c)\cdot c) \longrightarrow a}\,2}{((a/b)\cdot(b/c)\cdot c \longrightarrow a}\,2}{(a/b)\cdot(b/c) \longrightarrow (a/c)}\,3$$

Exercises

1. Show that the following is a derived rule of inference in the syntactic calculus: if $a \longrightarrow b$ and $c \longrightarrow d$ then $a/d \longrightarrow b/c$.

2. In the associative syntactic calculus prove the following:

a. $a/b \longrightarrow (a/c)/(b/c)$

b. $(a\backslash b)/c \longrightarrow a\backslash(b/c)$

c. $a/(b\cdot c) \longrightarrow (a/c)/b$

3.4 Multiplicative systems

A number of algebraic systems have been used, or could be used in applications to linguistics. We shall begin by considering a **multiplicative system** $(A, *)$, consisting of a set A and a binary operation $* : A \times A \longrightarrow A$. No associative law is postulated, so we must distinguish between $(a * b) * c$ and $a * (b * c)$.

Here is one illustration: $(\mathbf{N}, *)$, the set of natural numbers with operation $a * b = a^b$. Clearly, the associative law does not hold, as $(a^b)^c = a^{bc} \neq a^{(b^c)}$ in general. Of course, many multiplicative systems do satisfy the associative law, they are then called **semigroups**. Examples of semigroups are $(\mathbf{N}, +)$, $(\mathbf{N}, \cdot)$, $(\mathbf{N}, \mathsf{max})$, $(\mathbf{N}, \mathsf{min})$, $(\mathbf{N}, \mathtt{gcd})$, $(\mathbf{N}, \mathtt{lcm})$, $(\mathbf{N}, \mathsf{first})$, $(\mathbf{N}, \mathsf{last})$, where

$$\begin{array}{lcl} \mathtt{gcd}(a,b) & = & \text{greatest common divisor of } a \text{ and } b \\ \mathtt{lcm}(a,b) & = & \text{least common multiple of } a \text{ and } b \\ \mathsf{first}(a,b) & = & a \\ \mathsf{last}(a,b) & = & b \end{array}$$

The operations first and last are the projections we have met earlier. They are the only ones among the listed operations which are not commutative, for example,

$$\mathsf{first}(a, b) = a \neq b = \mathsf{first}(b, a)$$

in general. In all these examples, the associative law is established on the same principle, for example,

$$\mathsf{max}(\mathsf{max}(a, b), c) = \mathsf{max}(a, b, c) = \mathsf{max}(a, \mathsf{max}(b, c))$$

$$\mathsf{first}(\mathsf{first}(a, b), c) = \mathsf{first}(a, b, c) = \mathsf{first}(a, \mathsf{first}(b, c))$$

To give a motivation for the syntactic calculus studied in the preceding chapter, we shall define three operations on the subsets of a multiplicative system $(a, \cdot)$. Let X, Y, and Z be subsets of A, then we define

$$\begin{array}{rcl} X \cdot Y & = & \{x \cdot y \mid x \in X \wedge y \in Y\} \\ Z/Y & = & \{x \in A \mid \exists y{\in}Y,\, x \cdot y \in Z\} \\ X\backslash Z & = & \{y \in A \mid \exists x{\in}X,\, x \cdot y \in Z\} \end{array}$$

It is now easily seen that

$$\begin{array}{lll} X \cdot Y \subseteq Z & \text{if and only if} & X \subseteq Z/Y \\ & \text{if and only if} & Y \subseteq X\backslash Z \end{array}$$

and this should give some motivation for rules (3) to (6) in Section 3.3. Here is a proof of the first statement:

$$\begin{array}{lll} X \cdot Y \subseteq Z & \text{if and only if} & \forall x{\in}X\, \forall y{\in}Y,\, x \cdot y \in Z \\ & \text{if and only if} & \forall x{\in}X,\, x \in Z/Y \\ & \text{if and only if} & X \subseteq Z/Y \end{array}$$

The second statement is proved similarly.

Given a set Σ, we shall define a **bracketed string** in Σ as follows:

BS–1. all elements of Σ are bracketed strings in Σ,

BS–2. if A and B are bracketed strings in Σ, so is (AB),

BS–3. nothing is a bracketed string unless this follows from (1) and (2).

For example, if a, b and c are elements of Σ, then $((ab)c)$ and $(a(bc))$ are distinct bracketed strings of length 3. In practice, the outside parentheses are usually omitted. We may define an operation $\cdot$ between bracketed strings by putting $A \cdot B = (AB)$. Then the bracketed strings form a multiplicative system, called the **free multiplicative system generated by** Σ. Perhaps the syntactic calculus should really be considered as dealing

with subsets of the free multiplicative system generated by the vocabulary of the language we wish to study, these subsets being interpreted as types.

We shall conclude this chapter by looking at a purely mathematical problem. Given elements a_1, a_2, ..., a_n of Σ, how many distinct bracketed strings do there exist which contain exactly these elements in the given order? Let $f(n)$ be this number. When $n = 1$, 2, 3, and 4, we can easily calculate $f(n)$ directly.

n	bracketed strings	f(n)
1	a	1
2	(ab)	1
3	((ab)c), (a(bc))	2
4	(((ab)c)d), ((ab)(cd)), (a(b(cd))), (a((bc)d)), ((a(bc))d)	5

Can we obtain an explicit formula for $f(n)$? First we shall obtain a recursion formula. For $n > 1$, every bracketed string of length n has, by definition, the form (AB), where A and B are bracketed strings of lengths k and $n - k$, respectively, where $1 \leq k \leq n - 1$. Therefore

$$(*) \qquad \begin{aligned} f(n) &= \textstyle\sum_{k=1}^{n-1} f(k)f(n-k) \\ &= f(1)f(n-1) + f(2)f(n-2) + \cdots + f(n-1)f(1) \end{aligned}$$

This formula does allow us to calculate $f(n)$ inductively but we shall do better than that. Consider the formal infinite power series in the indeterminate x:

$$F(x) = \sum_{n=1}^{\infty} f(n)x^n = f(1)x + f(2)x^2 + \cdots$$

Then $F(0) = f(0) = 0$ and we have

$$\begin{aligned}
F(x)^2 &= \sum_{k=1}^{\infty} f(k)x^k \sum_{l=1}^{\infty} f(l)x^l \\
&= (f(1)x + f(2)x^2 + f(3)x^3 + \cdots)\,(f(1)x + f(2)x^2 + f(3)x^3 + \cdots) \\
&= (f(1)f(1))x^2 + (f(1)f(2) + f(2)f(1))x^3 + \\
&\qquad (f(1)f(3) + f(2)f(2) + f(3)f(1))x^4 + \cdots \\
&= f(2)x^2 + f(3)x^3 + f(4)x^4 + \cdots \\
&= \sum_{n=2}^{\infty} f(n)x^n \\
&= \sum_{n=1}^{\infty} f(n)x^n - f(1)x \\
&= F(x) - x
\end{aligned}$$

Thus

$$F(x)^2 - F(x) + x = 0$$

which may be solved using the quadratic formula (be careful, $F(x)$ is the unknown, while x is the constant term!) to yield

$$F(x) = \frac{1 \pm \sqrt{1 - 4x}}{2}$$

Since $F(0) = 0$ we must choose the minus sign. Thus

$$F(x) = \frac{1 - (1 - 4x)^{1/2}}{2}$$

The expression $(1 - 4x)^{1/2}$ can be expanded using the binomial theorem with exponent $1/2$. When this is done, we find that

$$F(x) = 1/2 - 1/2 \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n$$

and therefore the coefficient $f(n)$ of x^n is given by

$$(**) \qquad f(n) = 1/2 \binom{1/2}{n} (-1)^n 4^n$$

We do not wish to leave the answer in this form, but we shall use the fact that the binomial coefficient

$$\begin{aligned}\binom{1/2}{n} &= \frac{1/2(-1/2)(-3/2)\cdots(1/2-n-1)}{1*2*3*\cdots*n} \\ &= \frac{(-1)(-2)(-3)\cdots(-(2n-3))}{2^n n!}\end{aligned}$$

Multiplying top and bottom by $2*4*6*\cdots*(2n-2)$, we obtain

$$\binom{1/2}{n} = \frac{(-1)^{n-1}(2n-2)!}{2^n n! 2^{n-1}(n-1)!}$$

Substituting this into formula $(**)$, we get

$$\begin{aligned}f(n) &= \frac{1}{2}\frac{(-1)^{n-1}(2n-2)!}{2^n n! 2^{n-1}(n-1)!}(-1)^{n-1}4^n \\ &= \frac{2n-2)!}{n!(n-1)!} \\ &= \frac{1}{n}\binom{2n-2}{n-1}\end{aligned}$$

Writing $n-1=m$, we end up with the neater formula

$$(***) \qquad f(m+1) = \frac{1}{m+1}\binom{2m}{m} \qquad \text{for all } m>0$$

For example, taking $m=4$, we calculate

$$f(5) = \frac{1}{5}\binom{8}{4} = \frac{1}{5}\frac{8*7*6*5}{1*2*3*4} = 14$$

The formula for $f(n)$ was discovered by Catalan in 1838.

Exercises

1. Show that $(\mathbf{N}, \mathtt{gcd})$ is a semigroup.
2. Calculate $f(6)$ twice, once using the recursion formula $(*)$ and once using the explicit formula $(***)$.
3. For each $a \in \mathbf{N}$, let $(a) = \{a*x \mid x \in \mathbf{N}\}$ be the set of all multiples of a. In the multiplicative system $(\mathbf{N}, *)$, prove that $(a)*(b) = (a*b)$. Hint: $(a)*(b) = \{a*b*x*y \mid x, y \in \mathbf{N}\}$.

4. In the notation of this section and Exercise 3, show that $(c)/(a) = (c/d)$, where $d = \mathtt{gcd}(a, c)$. Hint: Check that

$$\begin{aligned}(c)/(a) &= \{y \in \mathbf{N} \mid \forall x{\in}(a),\ y * x \in (c)\} \\ &= \{y \in \mathbf{N} \mid \forall u{\in}\mathbf{N},\ y * a * u \in (c)\} \\ &= \{y \in \mathbf{N} \mid c \text{ divides } y * a\}\end{aligned}$$

Moreover, writing $c = c_1 d$ and $a = a_1 d$, use number theory to show that c divides $y * a$ if and only if c_1 divides y.

3.5 Semigroups and monoids

In the previous section, we defined a semigroup as a multiplicative system $(A, \cdot)$ satisfying the associative law:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c), \qquad \text{for all } a, b, c \in A$$

A **monoid** $(A, \cdot, 1)$ consists of a semigroup $(A, \cdot)$ together with an element $1 \in A$ such that

$$a \cdot 1 = a = 1 \cdot a, \qquad \text{for all } a \in A$$

We call 1 the **unity** element.

The unity element is unique, since if both 1 and $1'$ satisfied those equations, we would have

$$1 = 1 \cdot 1' = 1'$$

Examples of monoids are $(\mathbf{N}, +, 0)$, $(\mathbf{N}, *, 1)$, $(\mathbf{N}, \mathsf{max}, 0)$, $(\mathbf{N}, \mathtt{gcd}, 0)$, and $(\mathbf{N}, \mathtt{lcm} = 1)$. A little reflection will show that $(\mathbf{N}, \mathsf{min})$, $(\mathbf{N}, \mathsf{first})$, and $(\mathbf{N}, \mathsf{last})$ are not monoids.

A **group** $(A, \cdot, 1, {}^{-1})$ consists of a monoid $(A, \cdot, 1)$ together with an operation ${}^{-1} : A \longrightarrow A$ such that

$$a \cdot a^{-1} = 1 = a^{-1} \cdot a \qquad \text{for all } a \in A$$

A group is called **Abelian** if also

$$a \cdot b = b \cdot a \text{ for all } a, b \in A$$

Examples of Abelian groups are the integers under addition and the positive rationals under multiplication. An example of a non-Abelian group is the set of all matrices $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, where a, b, c and d are integers such that $ad - bc = 1$, under the usual multiplication of matrices.

It has been suggested to change the notation $n\backslash s$ to $n^{-1} \cdot s$, and to treat syntactic types as elements of a group. Indeed, the type of poor John works would then be computed as $(n \cdot n^{-1}) \cdot n \cdot (n^{-1} \cdot s) = s$. Unfortunately the type of *John poor works would also have type $n \cdot (n \cdot n^{-1}) \cdot (n^{-1} \cdot s) = s$, and so would many other unacceptable strings of English words.

One place where we can apply the theory of groups is to the language of mathematical physics. The grammatical sentences of this language are formulas or equations both sides of which have the same dimension. The types or dimensions form an Abelian group. One takes as basic dimensions $L(=$ length$)$, $T(=$ time$)$, and $M(=$ mass$)$. In a formula such as $Fl = mv^2$ one assigns dimensions

$$m \longrightarrow M$$
$$v \longrightarrow LT^{-1}$$
$$l \longrightarrow L$$
$$F \longrightarrow MLT^{-2}$$

Using the laws of an Abelian group, we find that both sides of the given equation have dimension ML^2T^{-2}, hence it is a grammatical sentence. This is not to say that the formula is true; the correct formula may very well read $Fl = 1/2mv^2$. On the other hand, the equation $Fl^2 = mv$ is not even a grammatical sentence.

In this manner it is customary to assign to every physical quantity a **dimension** $M^\alpha L^\beta T^\gamma$, where α, β and γ are rational numbers. Since this dimension is completely determined by the triple (α, β, γ) we see that the dimensions form an Abelian group, in fact a three-dimensional vector space over the field of rational numbers.

Most useful in applications to linguistics are the so-called free semigroup and monoid generated by a set Σ. In view of these applications, Σ will be called the **vocabulary** here, although it is often called the "alphabet".

A **string** in Σ of length $n \geq 1$ is an expression $a_1 a_2 \cdots a_n$, where $a_1, a_2, \ldots, a_n \in \Sigma$. Strings are multiplied by juxtaposition

$$(a_1 a_2 \cdots a_n) \cdot (b_1 b_2 \cdots b_k) = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_k$$

With this multiplication, the set of all strings in Σ is a semigroup, called the **free semigroup generated by** Σ.

For reasons of elegance, people often consider the **free monoid** Σ^* **generated by** Σ instead. To construct this, we only have to admit the empty string 1 of length 0, and put

$$A \cdot 1 = A = 1 \cdot A$$

for each string $A \in \Sigma^*$.

We shall consider three examples, when Σ has no element, one element, and two elements respectively.

When Σ is the empty set, then so is the free semigroup generated by Σ, but the free monoid $\Sigma^* = 1$ must contain the empty string as an element, hence is not empty.

When $\Sigma = \{\sigma\}$, the free semigroup consists of all strings $\sigma^n = \sigma \cdots \sigma$ (n times), where $n \geq 1$. The free monoid $\Sigma^* = \{1, \sigma, \sigma^2, \ldots\} = \{\sigma^n \mid n \in \mathbf{N}\}$, provided we define $\sigma^0 = 1$. Multiplication of σ^n and σ^k yields $\sigma^n \cdot \sigma^k = \sigma^{n+k}$; in fact the free monoid $(\Sigma^*, \cdot, 1)$ is isomorphic to $(\mathbf{N}, +, 0)$. (An isomorphism between two monoids is a one-to-one correspondence so that the binary operation and the unity element of one monoid correspond to those of the other.) Note that $\sigma^n \cdot \sigma^k = \sigma^k \cdot \sigma^n$, hence Σ^* is commutative.

When $\Sigma = \{\sigma, \tau\}$, where $\sigma \neq \tau$, the free semigroup is

$$\{\sigma, \tau, \sigma\sigma, \sigma\tau, \tau\sigma, \tau\tau, \sigma\sigma\sigma, \ldots\}$$

and the free monoid is the union of this and $\{1\}$. As $\sigma\tau$ and $\tau\sigma$ are different strings, we see that Σ^* does not satisfy the commutative law.

We may also consider free semigroups and monoids generated by more than two elements. Indeed, the vocabulary of English has a very large number of elements. However, in principle, for any vocabulary of more than two elements, this number can be reduced to two by coding as we shall see.

For instance, in English we may start by reducing the vocabulary to an alphabet of 27 elements (including the space between words), and these 27 elements can be further reduced to 3 symbols using the Morse code: $\cdot$, $-$, and $|$, the last being the divider between successive letters. Finally we may represent $\cdot = \tau\sigma\tau$, $- = \tau\sigma\sigma\tau$, and $| = \tau\tau$, where σ is a click and τ is a pause, as in telegraphy.

Exercises

1. If a semigroup can be made into a monoid by selecting a unity element, show that the unity element is uniquely determined.
2. If a monoid can be made into a group by specifying the operation $^{-1}$, show that x^{-1} is uniquely determined by x.

In the following exercises, two monoids are called isomorphic if they become identical when the elements and operations are suitably renamed.
3. Prove that the monoid $(\mathbf{N}, 1, \cdot)$ is not isomorphic to the free monoid generated by the empty set.
4. Prove that the monoid $(\mathbf{N}, 1, \cdot)$ is not isomorphic to the free monoid generated by $\{\sigma\}$, but that the latter is isomorphic to $(\mathbf{N}, 0, +)$.
5. Prove that the monoid $(\mathbf{N}, 1, \cdot)$ is not isomorphic to the free monoid generated by a set Σ with two or more elements.

Chapter 4

Context-free grammars

4.1 Grammars for some fragments of English

Given a vocabulary Σ, let Σ^* be the free monoid generated by Σ, that is, the set of all strings of elements of Σ, including the empty string 1. By a language in Σ we mean a subset $\mathcal{L} \subseteq \Sigma^*$. We want to describe $\mathcal{L}$ by a finite set of grammatical rules. Let us begin by looking at an example taken from English.

4.1.1 Example

$$\Sigma = \{\mathsf{the},\ \mathsf{a},\ \mathsf{man},\ \mathsf{ball},\ \mathsf{cat},\ \mathsf{hit},\ \mathsf{took}\}$$

$$\mathcal{L} = \left\{ \begin{array}{ccccc} \mathsf{the} & \mathsf{man} & \mathsf{hit} & \mathsf{the} & \mathsf{ball} \\ \mathsf{a} & \mathsf{ball} & \mathsf{took} & \mathsf{a} & \mathsf{man} \\ & \mathsf{cat} & & & \mathsf{cat} \end{array} \right\}$$

This language consists of a total of $2 * 3 * 2 * 2 * 3 = 72$ sentences. We could adjoin a symbol S(= sentence) to the vocabulary and list 72 grammatical rules of the form:

$$S \longrightarrow \mathsf{the\ man\ hit\ the\ ball}$$

A more customary method is to adjoin an auxiliary vocabulary

$$\{S,\ NP,\ VP,\ Art,\ N,\ V_t\}$$

of grammatical terms, where

S	=	sentence	Art	=	article
NP	=	noun phrase	N	=	noun
VP	=	verb phrase	V_t	=	transitive verb

Instead of the above 72 grammatical rules, only 10 are required, to

$$\begin{array}{lll} S & \rightarrow NP\ VP & \qquad Art \rightarrow \texttt{the, a} \\ NP & \rightarrow Art\ N & \qquad N \rightarrow \texttt{man, ball, cat} \\ VP & \rightarrow V_t\ NP & \qquad V_t \rightarrow \texttt{hit, took} \end{array}$$

Note that, compared with the categorial grammars studied earlier, the present arrow points in the reverse direction. This is because we now take the point of view of the speaker who generates the sentence, whereas previously we took the point of view of the listener who analyses the sentence. We subject the arrow to the usual reflexive, transitive, and substitution rules:

$$A \xrightarrow{(R)} A \qquad \frac{A \longrightarrow B \qquad B \longrightarrow C}{A \longrightarrow C}\ (\mathrm{T}) \qquad \frac{A \longrightarrow B \qquad C \longrightarrow D}{AC \longrightarrow BD}\ (\mathrm{S})$$

To show that the man hit the ball is a sentence, we could proceed quite formally thus:

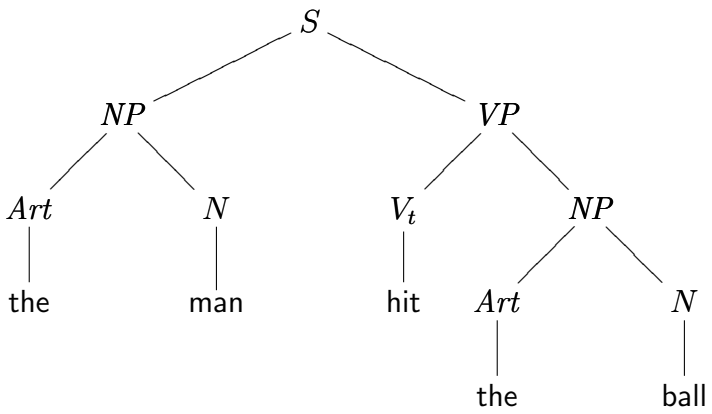
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Art \longrightarrow \text{the}} \qquad \dfrac{}{N \longrightarrow \text{ball}}}{NP \longrightarrow Art\ N} \qquad \dfrac{}{Art\ N \longrightarrow \text{the ball}}}{V_t \longrightarrow \text{hit}} \qquad \dfrac{}{NP \longrightarrow \text{the ball}}}{VP \longrightarrow V_t\ NP} \qquad \dfrac{}{V_t\ NP \longrightarrow \text{hit the ball}}}{VP \longrightarrow \text{hit the ball}}}{Art \longrightarrow \text{the}} \qquad \dfrac{}{N \longrightarrow \text{man}}}{NP \longrightarrow Art\ N} \qquad \dfrac{}{Art\ N \longrightarrow \text{the man}}}{NP \longrightarrow \text{the man}} \qquad \dfrac{}{VP \longrightarrow \text{hit the ball}}}{S \longrightarrow NP\ VP} \qquad \dfrac{}{NP\ VP \longrightarrow \text{the man hit the ball}}}{S \longrightarrow \text{the man hit the ball}}$$

(S) (T) (S) (T) (S) (T) (S) (T)

In practice, this proof in tree form may be abbreviated as a so-called parsing tree, shown as follows:

$$\begin{array}{ccccc} & & S & & \\ & \swarrow & & \searrow & \\ & NP & & & VP \\ & \swarrow & & \searrow & \\ Art & & N & V_t & NP \\ \text{the} & & \text{man} & \text{hit} & \swarrow \quad \searrow \\ & & & & Art \qquad N \\ & & & & \text{the} \qquad \text{ball} \end{array}$$

Frequently this parsing tree is pictured thus:



The reader may argue that there is too much theory just to replace 72 rules by 10. The next example will show that a language may quite easily consist of an infinite number of sentences (most languages do), and that grammar then becomes a matter of necessity rather than choice.

4.1.2 EXAMPLE Σ is as above with three new words added: and, or, but.

New grammatical term: $Conj$ = conjunction.
New grammatical rules: $S \rightarrow S\ Conj\ S$
$Conj \rightarrow$ and, or, but.

The total number of grammatical rules is now 14, but the total number of sentences is infinite, for example

the man hit the ball and the man hit the ball and $\cdots$

4.1.3 EXAMPLE One may argue that the conjunctions and and or can be placed not only between sentences but also between noun phrases, verb phrases and transitive verbs. This is not so for the conjunction but. To allow for this we rewrite the grammar of Example 4.1.2 in the following modified form

$$\begin{array}{lll} S & \rightarrow & S\ Conj\ S \\ NP & \rightarrow & NP\ Conj\ NP \\ VP & \rightarrow & VP\ Conj\ VP \end{array} \qquad \begin{array}{lll} Conj & \rightarrow & \text{and, or} \\ S & \rightarrow & S\ \text{but}\ S \\ V_t & \rightarrow & V_t\ Conj\ V_t \end{array}$$

The kind of grammar described in this section is known as a "context-free phrase-structure grammar" or just a "context-free grammar". A precise definition of this concept will be given in the next section.

Exercise

1. Give parsing trees for each of the following sentences, using the grammar of Example 4.1.3:

the man and the cat hit the ball

the man took and hit the ball

the man hit the ball and the cat

the man hit the cat and the cat hit the ball

4.2 Production grammars and deductive systems

Before we give a precise definition of what we mean by the term "context-free grammar" introduced in Section 4.1, we shall consider a more general concept.

4.2.1 DEFINITION *A* **production grammar** *or* **rewriting system** $\mathcal{G} = (\mathcal{V}, \Sigma, \Sigma', \mathcal{P})$ *consists of the following data:*

PG–1. *a finite set* $\mathcal{V}$ *called the* **vocabulary** *(its elements are called* **words***);*

PG–2. *two nonempty subsets* Σ *and* Σ' *of* $\mathcal{V}$ *called the* **terminal** *and* **initial** *vocabulary respectively;*

PG–3. *a finite subset* $\mathcal{P}$ *of* $\mathcal{V}^* \times \mathcal{V}^*$*, that is, a finite set of pairs of strings of words. These pairs are called* **productions***, and we write* $A \longrightarrow B$ *instead of saying that* (A, B) *is an element of* $\mathcal{P}$*.*

These data are usually subjected to the following restrictions:

PG–4. Σ *and* Σ' *have no elements in common.*

PG–5. Σ' *has only one element, that is,* $\Sigma' = S$*, where* S *is not in* Σ*.*

PG–6. A *is not a nonempty string of terminal words;*

PG–7. B *is not a nonempty string of initial words.*

COMMENTS.

(a) Sometimes $\mathcal{V}$ is called an "alphabet", its elements are then called "letters".

(b) We are particularly reluctant to adopt the restriction (PG–5). Let us see what we could do without this restriction. Σ' might consist of sentence types: S = statement, question, command, etc.

If Σ consists of all English words, $\mathcal{V} - \Sigma$ would consist of grammatical terms such as NP, VP etc., and the productions would be $S \longrightarrow NP\ VP$ etc. Thus $\mathcal{G}$ would be a grammar of English from the point of view of the speaker. However, we get another example if we take Σ' to be the vocabulary of English, Σ the set of sentence types, and if we reverse the arrows in all of the above productions, so that a new production would be $NP\ VP \longrightarrow S$, for example. Then the new grammar $\mathcal{G}$ would be a grammar of English from the point of view of the hearer.

(c) The reason for the restriction PG–4, for example, is that, in a grammar of English from the point of view of the speaker, a terminal string such as the man hit the cat should not be rewritten. Following Chomsky, a grammar is said to be

(0) of type 0, or a **generative grammar**, if it satisfies the above restrictions PG–4, PG–5, PG–6 and PG–7. Usually people are interested in grammars that satisfy further restrictions.

(1) of type 1, or a **context sensitive grammar**, if, in every production $A \longrightarrow B$, A is not longer (contains not more words) than B;

(2) of type 2, or a **context-free grammar**, if, in every production $A \longrightarrow B$, A is a nonterminal word and B is nonempty;

(3) of type 3, or a **finite state grammar**, if every production has the form $A \longrightarrow \sigma$ or $A \longrightarrow \sigma B$ where A is a nonterminal word, σ is a terminal word and B is any word.

Clearly, generative $\subseteq$ context sensitive $\subseteq$ context-free $\subseteq$ finite state. We shall not explain the words "context-sensitive" and "context-free". In fact, we shall disregard context sensitive grammars altogether and, for the time being, confine attention to context-free grammars.

With any grammar $\mathcal{G}$ we associate a **deductive system** $\mathcal{D}(\mathcal{G})$. Its **formulas** are expressions of the form $C \longrightarrow D$, where C and $D \in \mathcal{V}^*$. An **axiom** of $\mathcal{D}(\mathcal{G})$ is a production $A \longrightarrow B$ or an instance $A \longrightarrow A$ of the reflexive law (R). Moreover, there are two **rules of inference** the transitive law:

$$\frac{A \longrightarrow B \qquad\qquad B \longrightarrow C}{A \longrightarrow C} \quad \text{(T)}$$

and the substitution rule:

$$\frac{A \longrightarrow B \qquad\qquad C \longrightarrow D}{AC \longrightarrow BD} \quad \text{(S)}$$

Proofs are defined inductively: every axiom is a proof, and applying one of the rules of inference to two proofs we get a new proof. The last formula in a proof is the **theorem** which has been proved.

Proofs are usually written in tree form, see for example the proof of the formula

$$S \longrightarrow \text{ the man hit the ball}$$

in Section 4.1. This formula is therefore a theorem in the deductive system $\mathcal{D}(\mathcal{G})$, where $\mathcal{G}$ is the grammar of Section 4.1, Example 4.1.1. The substitution rule (S) is sometimes given a different formulation:

$$\frac{A \longrightarrow B}{CAD \longrightarrow CBD} \quad (\mathrm{S}')$$

It is easy to derive (S′) from (S) as follows:

$$\dfrac{\dfrac{C \xrightarrow{(\mathrm{R})} C \qquad A \longrightarrow B}{CA \longrightarrow CB}\ (\mathrm{S}) \qquad D \xrightarrow{(\mathrm{R})} D}{CAD \longrightarrow CBD}\ (\mathrm{S})$$

Conversely, one may derive (S) from (S′). Note that $C = 1AC$, $BC = BC1$.

$$\dfrac{\dfrac{A \longrightarrow B}{AC \longrightarrow BC}\ (\mathrm{S}') \qquad \dfrac{C \longrightarrow D}{BC \longrightarrow BD}\ (\mathrm{S}')}{AC \longrightarrow BD}\ (\mathrm{T})$$

Grammatical facts are expressed by formulas which can be proved in the deductive system $\mathcal{D}(\mathcal{G})$. We are particularly interested in theorems of the form $\mathrm{S} \longrightarrow \mathrm{E}$, where S is an initial word and $E \in \Sigma^*$ is a string in the terminal vocabulary. The language or set of sentences generated by the grammar $\mathcal{G}$ is the set $\mathcal{L}(\mathcal{G})$ consisting of all $E \in \Sigma^*$ for which $S \longrightarrow E$ is provable in $\mathcal{D}(\mathcal{G})$.

In the examples of Section 4.1, Σ is a fragment of the vocabulary of English, $\mathcal{V} - \Sigma$ consists of grammatical terms, $\Sigma' = \{S\}$, and the productions are the grammatical rules stated there. We shall now give two other illustrations of certain artificial languages, built up from a vocabulary consisting of two words, say σ and τ.

4.2.2 Example

$$\begin{aligned} \Sigma &= \{\sigma, \tau\}, \quad \sigma \neq \tau \\ \mathcal{L}_1 &= \{\sigma\tau, \sigma\sigma\tau\tau, \sigma\sigma\sigma\tau\tau\tau, \ldots\} \\ &= \{\sigma^n\tau^n \mid n \geq 1\} \end{aligned}$$

We wish to find a context-free grammar $\mathcal{G}_1$ so that $\mathcal{L}_1 = \mathcal{L}(\mathcal{G}_1)$. Take $\mathcal{V} = \{\sigma, \tau, S\}$ and let $\mathcal{P}$ consist of the following two productions:

P–1. $S \longrightarrow \sigma\tau$,

P–2. $S \longrightarrow \sigma S\tau$.

Here, for example, is a proof of the formula $S \longrightarrow \sigma\sigma\tau\tau$, showing that $\sigma\sigma\tau\tau \in \mathcal{L}_1$

$$\dfrac{\dfrac{S \xrightarrow{\text{P–2}} \sigma S\tau \qquad \dfrac{S \xrightarrow{\text{P–1}} \sigma\tau}{\sigma S\tau \longrightarrow \sigma\sigma\tau\tau}\ (\text{S}')}{S \longrightarrow \sigma\sigma\tau\tau}\ (\text{T})}$$

It can easily be seen that, for any string $E \in \Sigma^*$, $E \in \mathcal{L}_1$ if and only if $S \longrightarrow E$ is deducible from P–1 and P–2, that is, provable in the deductive system $\mathcal{D}(\mathcal{G}_1)$. While this fact ought to be clear intuitively, a proof will be given in Section 4.7, just to show that it is possible to give such a proof.

4.2.3 Example

$$\begin{aligned} \Sigma &= \{\sigma, \tau\}, \quad \sigma \neq \tau \\ \mathcal{L}_2 &= \{\sigma\sigma, \tau\tau, \sigma\tau\tau\sigma, \tau\sigma\sigma\tau, \ldots\} \\ &= \{AA^{\#} \mid A \in \Sigma^* \text{ and } A \neq 1\} \end{aligned}$$

where $A^{\#}$ is the mirror image of A. That is to say, if $A = a_1a_2 \ldots a_{n-1}a_n$, where the $a_i \in \Sigma$, then $A^{\#} = a_na_{n-1} \ldots a_2a_1$. We wish to find a context-free grammar $\mathcal{G}_2$ such that $\mathcal{L}_2 = \mathcal{L}(\mathcal{G}_2)$.

Again take $\mathcal{V} = \{\sigma, \tau, S\}$ and let $\mathcal{P}$ consist of the following productions:

P–3. $S \longrightarrow \sigma\sigma$,

P–4. $S \longrightarrow \tau\tau$,

P–5. $S \longrightarrow \sigma S\sigma$,

P–6. $S \longrightarrow \tau S\tau$.

It can now be easily seen that a string $E \in \Sigma^*$ is a sentence of $\mathcal{L}^2$ if and only if $S \longrightarrow E$ is deducible from P–3 to P–6.

REMARK. The language $\mathcal{L}'_1 = 1, \sigma\tau, \sigma\sigma\tau\tau, \ldots = \{\sigma^n\tau^n \mid n \geq 0\}$ can be described more elegantly by a grammar with productions $S \longrightarrow 1$ and $S \longrightarrow \sigma S\tau$. However, $\mathcal{L}'_1$ is not the same language as $\mathcal{L}_1$, it has one additional sentence, the empty string, and the grammar is not context-free. Similarly the language $\mathcal{L}'_2 = \{AA^{\#} \mid A \in \Sigma^*\}$ can be described more elegantly than $\mathcal{L}_2$ by a context-free grammar with productions $S \longrightarrow 1$, $S \longrightarrow \sigma S\sigma$ and $S \longrightarrow \tau S\tau$. This grammar is also not context-free.

Exercises

1. Deduce $S \longrightarrow \sigma\sigma\sigma\tau\tau\tau$ for $\mathcal{L}_1$.
2. Deduce $S \longrightarrow \tau\sigma\tau\tau\sigma\tau$ for $\mathcal{L}_2$.
3. Find a context-free grammar that will generate the language

$$\{\sigma^n\rho\tau^n \mid n \geq 1\}$$

for the terminal vocabulary $\Sigma = \{\rho,\ \sigma,\ \tau\}$
4. What language is generated by a context-free grammar

$$(\mathcal{V}, \{\sigma,\ \tau\},\ \{S\},\ \mathcal{P})$$

with vocabulary

$$\mathcal{V} = \{\sigma,\ \tau,\ S,\ U,\ V\}$$

and productions

$$\mathcal{P} = \{U \longrightarrow \sigma,\ U \longrightarrow \sigma U,\ V \longrightarrow \tau,\ V \longrightarrow \tau V,\ S \longrightarrow UV\}$$

4.3 Grammars of some formal languages

Context-free grammars are useful not only for describing natural languages such as English, but also for describing certain formal languages created by mathematicians. In particular, we shall discuss three such languages: the propositional calculus, the language of set theory, and a programming language for the abacus.

4.3.1 Example The propositional calculus. Its vocabulary is usually presented as follows:

Variables: $p, q, r, \ldots$
Connectives: $\neg, \Rightarrow, \wedge, \vee,$
Parentheses: $(,)$.
The set of sentences (= formulas) is usually described inductively as follows.

F–1. Every variable is a sentence.

F–2. If A is a sentence so is $\neg A$.

F–3. If A and B are sentences so are $(A \Rightarrow B)$, $(A \vee B)$, and $(A \wedge B)$.

F–4. Nothing else is a sentence.

This is to say, the set of sentences is the smallest set of strings in the given vocabulary which contains F–1 and is closed under F–2 and F–3. For example, we have the following two sentences with their construction:

$$\dfrac{\dfrac{\dfrac{\overset{(1)}{p} \qquad \overset{(1)}{q}}{p \Rightarrow q}\ (3) \qquad \overset{(1)}{r}}{((p \Rightarrow q) \vee r)}\ (3) \qquad\qquad \dfrac{\overset{(1)}{p} \qquad \dfrac{\overset{(1)}{q} \qquad \overset{(1)}{r}}{(q \vee r)}\ (3)}{(p \Rightarrow (q \vee r))}\ (3)$$

Usually people omit outside parentheses. Thus the above two sentences may be written without ambiguity:

$$(p \Rightarrow q) \vee r, \qquad p \Rightarrow (q \vee r)$$

However it won't do to omit parentheses altogether, as the expression $p \Rightarrow q \vee r$ is ambiguous.

Actually, the right parentheses are redundant, and one could still distinguish the following:

$$((p \Rightarrow q \vee r, \qquad (p \Rightarrow (q \vee r$$

Parentheses may be redundant, but they do facilitate reading. In English, ambiguity is avoided in a similar fashion. We say

if p then q	in place of $(p \Rightarrow q$
either p or q	in place of $(p \vee q$
both p and q	in place of $(p \wedge q$

The above two formulas are transcribed into English thus:

either if p then q or r, if p then either q or r

The second formation is more common than the first. The reason for this lies in the limited capacity of the short term memory in the human brain.

A word should be said about the bracketless or "Polish" notation. One may avoid parentheses altogether by writing connectives in front, thus $\Rightarrow pq$, $\vee pq$, $\wedge pq$ in place of $(p \Rightarrow q)$, $(p \vee q)$, $(p \wedge q)$. The above two formulas would become in Polish notation: $\vee \Rightarrow pqr$, $\Rightarrow p \vee qr$.

Economy of writing and ease of reading do not always seem to agree! However, it is not clear if the ease of reading is due to our lack of familiarity or this notation is inherently harder to understand. After all, we use a form of this in vertical mode when we learn arithmetic in elementary school.

After this digression, let us return to the propositional calculus and obtain a context-free grammar for it.

First, we must make the vocabulary finite, so let us replace p, q, r, ... by p, p', p'', ... Thus we have

$$\Sigma = \{p, ', \neg, \Rightarrow, \wedge, \vee, (,)\}$$
$$\mathcal{V} - \Sigma = \{S, V, C\}$$

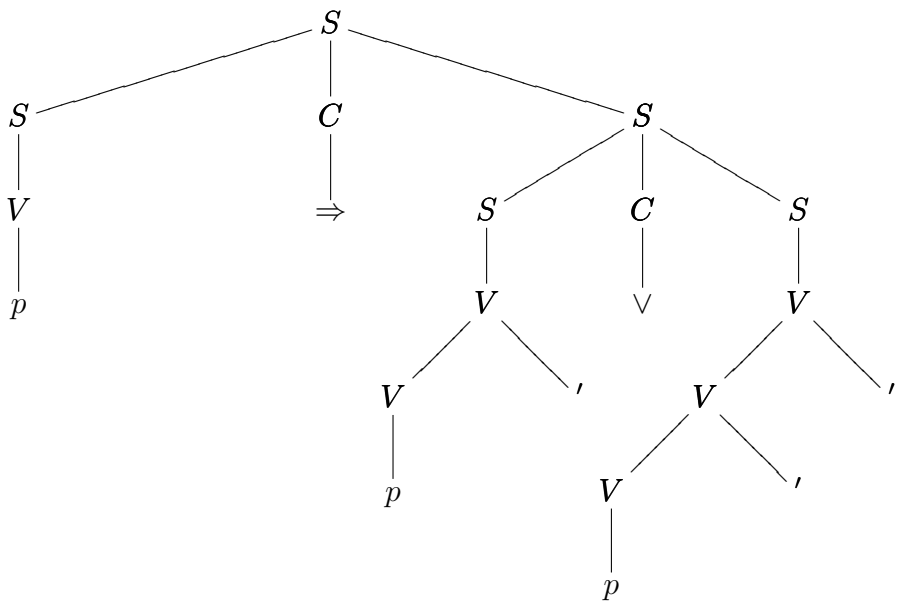
where

S = sentence
V = propositional variable
C = binary connective

The productions are the following:

$V \to p$ $\qquad$ $S \to V$
$V \to V'$ $\qquad$ $S \to \neg S$
$C \to \Rightarrow, \vee, \wedge$ $\qquad$ $S \to (S\ C\ S)$

Here, for example, is a parsing tree for the sentence $(p \Rightarrow (p' \vee p''))$:



We have not indicated the symbols "(" and ")" explicitly on the tree.

4.3.2 Example The language of set theory. As all mathematics can be reduced to set theory, it is now fashionable to regard this also as the language of mathematics. We present a context-free grammar for set theory.

$$\Sigma = \{x, ', =, \in, \neg, \Rightarrow, \wedge, \vee, \forall, \exists, (,)\}$$

$$\mathcal{V} - \Sigma = \{S, V, Q, C\}$$

where

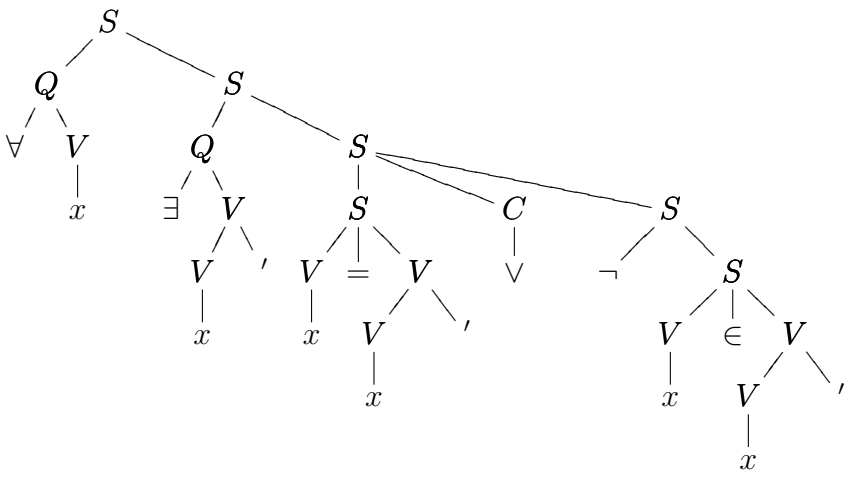
S = sentence
V = individual variable
Q = quantifier
C = binary connective

Productions are the following:

$V \to x$	$S \to V = V$
$V \to V'$	$S \to V \in V$
$Q \to \forall V$	$S \to \neg S$
$Q \to \exists V$	$S \to (S\ C\ S)$
$C \to \Rightarrow, \vee, \wedge$	$S \to Q\ S)$

We give an example of a parsing tree for the sentence:

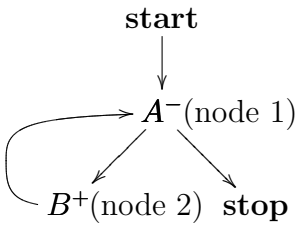
$$\forall x \exists x' \, (x = x') \vee \neg (x \in x')$$



We have not indicated the symbols "(" and ")" explicitly.

Of course, the above grammar does not tell us everything we want to know about the language of mathematics. It does not distinguish between open sentences, which contain free variables, and closed sentences, which do not. Above all, it does not generate the one set we are really interested in, the set of theorems.

4.3.3 EXAMPLE A programming language for the abacus. For purpose of illustration, recall the program: transfer content of A to B.



We may regard this program as consisting of the following instructions:

start: go to 1
1: A^- go to 2 else go to **stop**
2: B^+ go to 1.

We wish to consider these instructions as sentences of a language.

To ensure a finite vocabulary, we place A, B, C, ... by A, A', A'', ..., and 1, 2, 3, ... by 1, 1′, 1″, ...

$$\begin{aligned} \Sigma &= \{\textbf{start},\ \textbf{stop},\ +,\ ',\ \mathsf{goto},\ \mathsf{else},\ :\} \\ \mathcal{V} - \Sigma &= \{S,\ L,\ N,\ M\} \end{aligned}$$

where

S = sentence (= instruction)
L = location
N = node other than **start**or **stop**
M = node other than **start**.

Productions:

$L \to A$
$L \to L'$
$N \to 1$
$N \to N'$
$S \to N : L^-$ go to M else go to M
$M \to N$
$M \to$ **stop**
$S \to$ **start**: go to M
$S \to N : L^+$ go to M

Exercises

1. Write out a parsing tree for $((p \Rightarrow p') \vee p'')$.
2. Along the lines of Example 4.3.3 above, what instructions are needed for the program: copy content of A into B.

4.4 New languages from old

Given a terminal vocabulary Σ and a language $\mathcal{L} \subseteq \Sigma*$, we may form two new languages in the same vocabulary:

$$\begin{aligned} -\mathcal{L} &= \{A \in \Sigma^* \mid A \notin \mathcal{L}\} \\ \mathcal{L}^{\#} &= \{A^{\#} \in \Sigma^* \mid A \in \mathcal{L}\} \end{aligned}$$

$-\mathcal{L}$ is the **complement** of $\mathcal{L}$ and $\mathcal{L}^{\#}$ is the **mirror image** of $\mathcal{L}$. Given two languages $\mathcal{L}$ and $\mathcal{L}'$ in the same terminal vocabulary Σ, we may form three new languages as follows:

$$\begin{aligned} \mathcal{L} \cup \mathcal{L}' &= \{A \in \Sigma^* \mid A \in \mathcal{L} \vee A \in \mathcal{L}'\} \\ \mathcal{L} \cap \mathcal{L}' &= \{A \in \Sigma^* \mid A \in \mathcal{L} \wedge A \in \mathcal{L}'\} \\ \mathcal{L}\mathcal{L}' &= \{AA' \in \Sigma^* \mid A \in \mathcal{L} \wedge A' \in \mathcal{L}'\} \end{aligned}$$

$\mathcal{L} \cup \mathcal{L}'$ is the **union**, $\mathcal{L} \cap \mathcal{L}'$ is the **intersection**, and $\mathcal{L}\mathcal{L}'$ is the **product** of the given languages. As an example of the product of two or more languages we may form

$$\begin{aligned} \mathcal{L}^2 &= \{AB \in \Sigma^* \mid A, B \in \mathcal{L}\} \\ \mathcal{L}^3 &= \{ABC \in \Sigma^* \mid A, B, C \in \mathcal{L}\} \end{aligned}$$

and so on. Finally, we are able to construct

$$\mathcal{L}^* = \{A \in \Sigma^* \mid \exists n{\geq}1,\ A \in \mathcal{L}^n\}$$

Thus $\mathcal{L}^*$ consists of all strings which are made up by juxtaposing any number of strings of $\mathcal{L}$; that is, the sentences of $\mathcal{L}^*$ are the texts of $\mathcal{L}$. For example, if $\mathcal{L} = \{\sigma\}$, then $\mathcal{L}^2 = \{\sigma^2\}$, $\mathcal{L}^3 = \{\sigma^3\}$, and $\mathcal{L}^* = \{\sigma, \sigma^2, \sigma^3, \ldots\}$.

4.4.1 THEOREM *If $\mathcal{L}$ and $\mathcal{L}'$ have context-free grammars then so do $\mathcal{L}^{\#}$, $\mathcal{L}^*$, $\mathcal{L} \cup \mathcal{L}'$, and $\mathcal{L}\mathcal{L}'$.*

PROOF We shall assume that $\mathcal{L}$ has vocabulary $\mathcal{V}$, initial element $S \in \mathcal{V} - \Sigma$, and productions $\mathcal{P}$ We also assume that $\mathcal{L}'$ has vocabulary $\mathcal{V}'$, initial element $S' \in \mathcal{V}' - \Sigma$, and productions $\mathcal{P}$. Without loss in generality, we may also assume that $\mathcal{V} - \Sigma$ has no symbols in common with $\mathcal{V}' - \Sigma$; otherwise we could change the notation to make this so. Thus $\mathcal{V} \cap \mathcal{V}' = \Sigma$.

(1) For the grammar of $\mathcal{L} \cup \mathcal{L}'$ we take the vocabulary $\mathcal{V} \cup \mathcal{V}' \cup \{T\}$, assuming that T is a new symbol, not in $\mathcal{V}$ or $\mathcal{V}$'. T will be the initial element for $\mathcal{L} \cup \mathcal{L}'$. As productions we take $\mathcal{P} \cup \mathcal{P}' \cup \{T \longrightarrow S,\ T \longrightarrow S'\}$. It is easily seen that, for any string $E \in \Sigma^*$, $T \longrightarrow E$ is deducible from these productions if and only if $E \in \mathcal{L} \cup \mathcal{L}'$.

(2) For the grammar of $\mathcal{L}\mathcal{L}'$ we take the same vocabulary as in (1), and the set of productions $\mathcal{P} \cup \mathcal{P}' \cup \{T \longrightarrow SS'\}$. It is easily seen that, for any string $E \in \Sigma^*$, $T \longrightarrow E$ is deducible from these productions if and only if $E \in \mathcal{L}\mathcal{L}'$.

(3) $\mathcal{L}^*$ has sentences of the form $A_1A_2\cdots A_k$, where A_1, A_2, ..., $A_k \in \mathcal{L}$, and $k \geq 1$. We take as vocabulary $\mathcal{V} \cup \{T\}$, where T is a new symbol that will serve as the initial element. We take as productions the set $\mathcal{P} \cup \{T \longrightarrow S,\ T \longrightarrow ST\}$ It follows that $T \longrightarrow S^2$, $T \longrightarrow S^3$, etc. It is easily seen that for any string $E \in \Sigma^*$, $T \longrightarrow E$ is deducible from the above productions if and only if $E \in \mathcal{L}^*$.

(4) For the grammar of $\mathcal{L}^\#$ we take the same vocabulary as for $\mathcal{L}$ but replace the productions of $\mathcal{L}$ by $\mathcal{P}^\#$ defined so that the production $A^\# \longrightarrow B^\#$ for every production $A \longrightarrow B$ of $\mathcal{P}$. It is easily seen that, for any string $E \in \Sigma^*$, $S \longrightarrow E$ is deducible from these m productions if and only if $S = S^\# \longrightarrow E^\#$ was deducible from the old productions, that is, $E^\# \in \mathcal{L}$, that is, $E \in \mathcal{L}^\#$.

Up to verifying the statements about which it was said that they could easily be seen, our proof is now complete.

To illustrate this theorem and its proof, we recall the languages $\mathcal{L}_1$ and $\mathcal{L}_2$ of Section 4.2 as well as their grammars. To make sure that the auxiliary vocabularies have no symbols in common, we replace their initial elements by S_1 and S_2 respectively.

The productions for $\mathcal{L}_1$ were:

$$S_1 \longrightarrow \sigma\tau, \qquad S_1 \longrightarrow \sigma S_1 \tau.$$

The productions for $\mathcal{L}_2$ were: The productions for $\mathcal{L}_1 \cup \mathcal{L}_2$ were:

$$S_2 \longrightarrow \sigma\sigma, \qquad S_2 \longrightarrow \tau\tau, \qquad S_2 \longrightarrow \sigma S_2 \sigma, \qquad S_2 \longrightarrow \tau S_2 \tau.$$

The productions for $\mathcal{L}_1 \cup \mathcal{L}_2$ now are:

$$S_1 \longrightarrow \sigma\tau, \qquad S_1 \longrightarrow \sigma S_1 \tau, \qquad S_2 \longrightarrow \sigma\sigma, \qquad S_2 \longrightarrow \tau\tau,$$

$$S_2 \longrightarrow \sigma S_2 \sigma, \qquad S_2 \longrightarrow \tau S_2 \tau, \qquad T \longrightarrow S_1, \qquad T \longrightarrow S_2.$$

T serves as the initial symbol.

The productions for $\mathcal{L}_1\mathcal{L}_2$ are:

$$S_1 \longrightarrow \sigma\tau, \qquad S_1 \longrightarrow \sigma S_1 \tau, \qquad S_2 \longrightarrow \sigma\sigma, \qquad S_2 \longrightarrow \tau\tau$$

$$S_2 \longrightarrow \sigma S_2 \sigma, \qquad S_2 \longrightarrow \tau S_2 \tau, \qquad T \longrightarrow S_1 S_2$$

The productions for $\mathcal{L}_1^*$ are:

$$S_1 \longrightarrow \sigma\tau, \qquad S_1 \longrightarrow \sigma S_1 \tau, \qquad T \longrightarrow S_1, \qquad T \longrightarrow S_1 T$$

The productions for $\mathcal{L}_1^\#$ are:

$$S_1 \longrightarrow \tau\sigma, \qquad S_1 \longrightarrow \tau S_1 \sigma.$$

The initial symbol here is still S_1.

The reader may ask why we did not include in the theorem assertions that $-\mathcal{L}$ and $\mathcal{L} \cap \mathcal{L}'$ have context-free grammars. It will in fact be shown in Section 4.8 that there exist languages $\mathcal{L}$ and $\mathcal{L}'$ with context-free grammars for which $\mathcal{L} \cap \mathcal{L}'$ does not have a context-free grammar.

Assuming that, we shall now prove that the complement of a language with a context-free grammar does not necessarily have a context-free grammar. Indeed, suppose this were the case. Let $\mathcal{L}$ and $\mathcal{L}'$ be any two languages (with the same terminal vocabulary) which have context-free grammars. Then

$$\mathcal{L} \cap \mathcal{L}' = -(-\mathcal{L} \cup -\mathcal{L}')$$

would also have a context-free grammar, which is a contradiction.

Exercise

1. Find context-free grammar for the following languages:

$$\mathcal{L}_2^*,\ \mathcal{L}_2^\#,\ \mathcal{L}_1^* \cup \mathcal{L}_2, \mathcal{L}_1^\# \mathcal{L}_2,\ (\mathcal{L}_1\mathcal{L}_2)^*,\ (\mathcal{L}_1 \cup \mathcal{L}_2)^\#$$

Here $\mathcal{L}_1$ and $\mathcal{L}_2$ are the languages introduced in Section 4.2.

4.5 How are sentences produced and recognized?

It is highly unlikely that a person while engaged in uttering a sentence will construct a parsing tree in his mind or even carry out a proof in a deductive system. Let us look at an example, based on the grammar of Section 4.1.

			n
S	$\rightarrow$	$NP\ VP$	2
	$\rightarrow$	$Art\ N\ VP$	3
	$\rightarrow$	the $N\ VP$	2
	$\rightarrow$	the man VP	1
	$\rightarrow$	the man $VP\ Conj\ VP$	3
	$\rightarrow$	the man $V_t\ NP\ Conj\ VP$	4
	$\rightarrow$	the man took $NP\ Conj\ VP$	3
	$\rightarrow$	the man took $Art\ N\ Conj\ VP$	4
	$\rightarrow$	the man took the $N\ Conj\ VP$	3
	$\rightarrow$	the man took the ball $Conj\ VP$	2
	$\rightarrow$	the man took the ball and VP	1
	$\rightarrow$	the man took the ball and $V_t\ NP$	2

→	the man took the ball and hit NP	1
→	the man took the ball and hit $Art\ N$	2
→	the man took the ball and hit the N	1
→	the man took the ball and hit the cat	0

Note that in going from line 1 to line 2, we apply the production $NP \longrightarrow Art\ N$ rather than the production $VP \longrightarrow VP\ Conj\ VP$. Again, in going from line 2 to line 3, we apply the production $Art \longrightarrow$ the rather than $N \longrightarrow$ man or $VP \longrightarrow VP\ Conj\ VP$. In other words, we replace nonterminal words as soon as possible, starting from left to right.

We note that at each stage of the generation of the above sentence, part of the sentence has already been uttered, while the rest only exists in the form of a plan. The plan consists of a string of grammatical terms. The number n of these terms varies, as we go along, from 2 to 0, the maximum in the above example being 4. We may think of these grammatical terms as being stored temporarily in the short term memory of the human brain.

Psychologists have found that the short term memory is capable of storing at most seven "chunks" of information. They define one chunk as a nonsense word. However, adapting a proposal by Victor H. Yngve to our present terminology, we are tempted to say that each element of the vocabulary counts as one chunk. Yngve himself believed that the grammatical rules of a natural language would automatically rule out $n > 7$. This is not quite true, as the following example shows.

Let us look at the fragment of English consisting of the words:

John, Jane, Tom, works, rests, complains, if, then

together with the following productions:

$$\begin{aligned} \mathrm{S} &\rightarrow NP\ \ VP \\ NP &\rightarrow \text{John, Jane, Tom} \\ VP &\rightarrow \text{works, rests, complains} \\ \mathrm{S} &\rightarrow \text{if } S \text{ then } S. \end{aligned}$$

We shall now form two sentences, which in logical notation have the forms $p \Rightarrow (q \Rightarrow (r \Rightarrow s))$, $((p \Rightarrow q) \Rightarrow r) \Rightarrow s$.

(1) If John works then if Jane rests then if Tom works then Jane complains

(2) If if if John works then Jane rests then Tom works then Jane complains

One may conceivably utter (1), but uttering (2) is inconceivable, although grammatically they are both well-formed. Why is this so? Let us see how (1) would actually be uttered.

			n
S	$\rightarrow$	if S then S	3
	$\rightarrow$	if NP VP then S	4
	$\rightarrow$	if John VP then S	3
	$\rightarrow$	if John works then S	1
	$\rightarrow$	if John works then if S then S	3
	$\rightarrow$	$\ldots$	

We note that while uttering if, the three items "S then S" must be put into temporary storage. The word then, while belonging to the terminal vocabulary, cannot be uttered until after the word works has been uttered. We easily check that $n \leq 4$ throughout the entire process of generating the sentence (1).

The situation is different with (2).

			n
S	$\rightarrow$	if S then S	3
	$\rightarrow$	if if S then S then S	5
	$\rightarrow$	if if if S then S then S then S	7
	$\rightarrow$	if if if NP VP then S then S then S	8
	$\rightarrow$	if if if John VP then S then S then S	7
	$\rightarrow$	if if if John works then S then S then S	5
	$\rightarrow$	$\ldots$	

If our psychological hypothesis is correct, a speaker will get stuck as soon as $n \geq 8$.

One possible conclusion from this is that some grammatically possible sentences are never uttered, because they would overload the temporary storage capacity of the human brain. We shall drive home the point by considering another example. Let us add to the previous grammar the following productions:

$$\begin{aligned} Adj &\rightarrow \text{nice, known, true} \\ S &\rightarrow \text{it is } Adj \text{ that } S \\ S &\rightarrow \text{that } S \text{ is } Adj \end{aligned}$$

The following sentence causes no difficulty:

(1) It is true that it is known that it is true that Jane works

However, the following sentence is inconceivable:

(2) That that that Jane works is true is known is nice

Indeed, at one stage of its attempted generation we would have:

			n
S	$\rightarrow$	that that that NP VP is Adj is Adj is Adj	8

REMARK. We should point out the difference between "chunk of information" and "bit of information". The latter has a technical significance: bit = binary digit. If we think of a chunk as being a four letter word, each letter having 26 possibilities, say approximately 25, then 1 chunk is approximately $4 \cdot 5 = 20$ bits.

Commentary by MB

On the other hand, as the following example shows, an explanation in terms of the number of "chunks" to be stored is over-simplified. The following sentence is not a model of graceful style, but still can be processed, while in the course of processing the number of such chunks generated is as much as nine. The sentence uses yet another rule:

$S \rightarrow$ both S and S

which generates such sentences as

both John is late and Jane is early

and the one that interests us

If both that Jane works is true and that John works is false then Sue sleeps

			n
S	$\rightarrow$	if S then S	3
	$\rightarrow$	if both S and S then S	5
	$\rightarrow$	if both that S is S and S then S	7
	$\rightarrow$	if both that NP VP is S and S then S	8
	$\rightarrow$	...	

Consideration of this and sentences that are even worse from the point of view of chunks of information (using the production $S \longrightarrow$ either S or S and $S \longrightarrow$ neither S nor S) suggest quite a different interpretation of these facts; namely that there are specific processors in the brain that process such constructions and that these processors have no stack. Thus you cannot interrupt the if S then S processor *for another construction of the same type*, although there is no especial problem with using another type of complex construction that uses a different processor.

Having formed some idea how sentences are produced, we are led to ask how strings of words are recognized as sentences. Let us take the same example as before, now from the point of view of the hearer:

		n
	the man took the ball and hit the cat	0
$\leftarrow$	Art man ...	1
$\leftarrow$	Art N ...	2

$\longleftarrow$ *NP* took ...	1
$\longleftarrow$ *NP* V_t the ...	2
$\longleftarrow$ *NP* V_t *Art* ball ...	3
$\longleftarrow$ *NP* V_t *Art* *N* ...	4
$\longleftarrow$ *NP* V_t *NP* ...	3
$\longleftarrow$ *NP* *VP* and ...	2
$\longleftarrow$ *NP* *VP* *Conj* *hit* ...	3
$\longleftarrow$ *NP* *VP* *Conj* V_t the ...	4
$\longleftarrow$ *NP* *VP* *Conj* V_t *Art* cat	5
$\longleftarrow$ *NP* *VP* *Conj* V_t *Art* *N*	6
$\longleftarrow$ *NP* *VP* *Conj* V_t *NP*	5
$\longleftarrow$ *NP* *VP* *Conj* *VP*	4
$\longleftarrow$ *NP* *VP*	2
$\longleftarrow$ *S*	1

A number of comments are in order:

1. We have used the same grammar that had been used for producing the sentence. It might have been tempting to reverse the arrows, thus operating in the "dual" grammar with productions *Art* *N* $\longrightarrow$ *NP*, and so on. However, this dual grammar is not context-free (or even generative, as it has more than one initial word.)

2. After line 9 it would have been tempting to write $\leftarrow S$ and $\cdots$, but then we would have ended up with $\leftarrow S$ *Conj* *VP*, and there is no production which will justify our replacing this by *S*. Presumably a human hearer makes use of the other cues, for instance intonation, to see that the sentence is not yet complete.

3. The numeral on the right denotes the number of grammatical terms that are held in temporary storage at each stage. The number $n = 6$ in line 13 is dangerously close to the upper bound of 7. In fact, when we analyze the sentence

the man took the ball and hit the cat and the dog

we do get

	n
$\longleftarrow$ ...	
$\longleftarrow$ *NP* *VP* *Conj* V_t *NP* and	5
$\longleftarrow$ *NP* *VP* *Conj* V_t *NP* *Conj* the	6
$\longleftarrow$ *NP* *VP* *Conj* V_t *NP* *Conj* *Art* dog	7
$\longleftarrow$ *NP* *VP* *Conj* V_t *NP* *Conj* *Art* *N*	8
$\longleftarrow$ ...	

What is puzzling is that there are 8 chunks of information in storage, and yet there is no apparent difficulty in someone's recognizing the above as a sentence. One explanation comes by replacing the production $NP \longrightarrow NP\ Conj\ NP$ by two productions: $NP \longrightarrow NP^*\ NP$ and $NP^* \longrightarrow NP\ Conj$. Similarly we can replace $VP \longrightarrow VP\ Conj\ VP$ by two productions.

Exercises

In each of the following exercises, count the chunks of information in storage at each stage.

1. Show how a person might produce the sentence.

the man hit the cat and the dog

2. Show how a person might recognize the string

the man took the ball and hit the cat and the dog

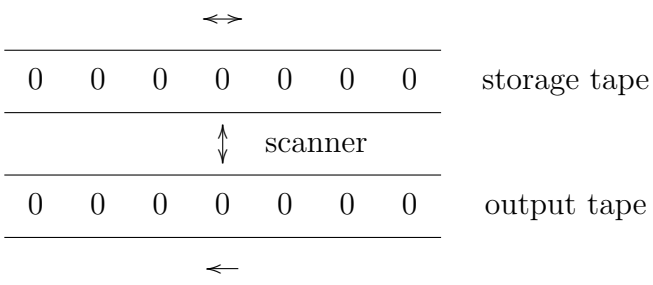
as a sentence, utilizing the suggestion made in the last paragraph above.

4.6 Machines for producing and recognizing sentences

We shall construct a machine that can produce the sentences of a language which is generated by a **normal** context-free grammar where all productions have one of two forms:

$$T \longrightarrow W \qquad T \longrightarrow UV$$

The machine we have in mind is a variation of what is usually called a "pushdown automaton". It consists of two tapes: a storage tape capable of moving in both directions and an output tape allowed only to move to the left. There is a scanner which can read and write on both tapes. Both tapes are inscribed with words of the same vocabulary $\mathcal{V}$, the total vocabulary of the grammar to be considered.



We have shown the machine at the beginning of an operation: all squares but one are blank; only the scanned square of the storage tape is imprinted with the symbol S, representing the fact that the machine plans to utter a sentence.

The machine is **non-deterministic**: a given situation may give rise to several possible moves; in fact, there is usually more than one way of completing a sentence. We now describe the moves of the machine, which depend on the given normal context-free grammar.

(1) $$\binom{T}{0} \longrightarrow \binom{0}{T}$$

This means: if the scanned square of the output tape is blank and the scanned square of the storage tape has T written on it, the machine prints T on the scanned square of the output tape and erases it on the storage tape.

(2) $$\binom{0}{0} \longrightarrow \binom{\mathsf{left}}{\mathsf{stay}}$$

This means: If both scanned squares are blank, the storage tape moves one square to the left.

(3) $$\binom{V}{U} \longrightarrow \binom{\mathsf{right}}{\mathsf{stay}}$$

This means: If both scanned squares have words written on them, the storage tape moves one square to the right.

(4) $$\binom{0}{T} \longrightarrow \binom{0}{W}$$

provided $T \longrightarrow W$ is a production of the grammar.

(5) $$\binom{0}{T} \longrightarrow \binom{V}{U}$$

provided $T \longrightarrow UV$ is a production of the grammar.

(6) $$\binom{0}{T} \longrightarrow \binom{\mathsf{stay}}{\mathsf{left}}$$

provided $T \in \Sigma$. Thus, if the scanned square of the output tape contains a word of the terminal vocabulary Σ, the output tape moves one square to the left.

We shall illustrate the operation of the machine for the fragment of English discussed in Section 4.1, Example 3. Unfortunately, a production such as

$$NP \longrightarrow NP\ \ Conj\ \ NP$$

does not belong into a normal grammar, so we replace it by two productions:

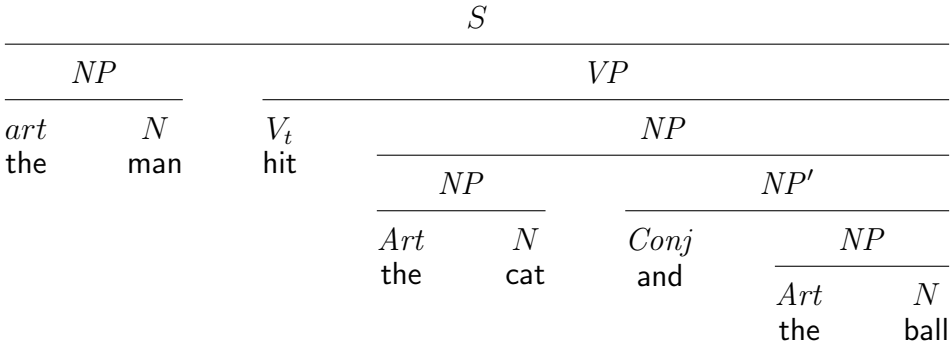
$$NP \longrightarrow NP\ \ NP'$$

$$NP' \longrightarrow Conj\ NP$$

We shall describe in detail how our machine would utter the sentence:

the man hit the cat and the ball

according to the following parsing tree:



In describing the successive situations of the machine, we shall only indicate what is written on the two scanned squares and on a few relevant squares to the right of the scanned square of the storage tape. In fact, all squares to the left on the scanned square of the storage tape are blank, and so are all squares to the right of the scanned square of the output tape. The squares of the output tape to the left of the scanned square contain nothing but past utterances. At no time need the machine remember what it has already uttered. Its memory (= storage tape) need only contain the plan of what will be uttered in future. Well, here are the successive situations of the machine:

$$\begin{pmatrix} S \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ S \end{pmatrix} \rightarrow \begin{pmatrix} VP \\ NP \end{pmatrix} \rightarrow \begin{pmatrix} 0 & VP \\ NP & \end{pmatrix} \rightarrow \begin{pmatrix} N & VP \\ Art & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N & VP \\ Art & & \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 0 & N & VP \\ \text{the} & & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N & VP \\ 0 & & \end{pmatrix} \rightarrow \begin{pmatrix} N & VP \\ 0 & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & VP \\ N & \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 0 & VP \\ \text{man} & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & VP \\ 0 & \end{pmatrix} \rightarrow \begin{pmatrix} VP \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ VP \end{pmatrix} \rightarrow \begin{pmatrix} NP \\ V_t \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP \\ V_t & \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 0 & NP \\ \text{hit} & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP \\ 0 & \end{pmatrix} \rightarrow \begin{pmatrix} NP \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ NP \end{pmatrix} \rightarrow \begin{pmatrix} NP' \\ NP \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP' \\ NP & \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} N & NP' \\ Art & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N & NP' \\ Art & & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N & NP' \\ \text{the} & & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N & NP' \\ 0 & & \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} N & NP' \\ 0 & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP' \\ N & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP' \\ \text{cat} & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP' \\ 0 & \end{pmatrix} \rightarrow \begin{pmatrix} NP' \\ 0 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 0 \\ NP' \end{pmatrix} \rightarrow \begin{pmatrix} NP \\ Conj \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP \\ Conj & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP \\ \mathsf{and} & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & NP \\ 0 & \end{pmatrix} \rightarrow$$
$$\begin{pmatrix} NP \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ NP \end{pmatrix} \rightarrow \begin{pmatrix} N \\ Art \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N \\ Art & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N \\ \mathsf{the} & \end{pmatrix} \rightarrow \begin{pmatrix} 0 & N \\ 0 & \end{pmatrix} \rightarrow$$
$$\begin{pmatrix} N \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ N \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ \mathsf{ball} \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The machine does not come to a stop, but the storage tape moves to the left and keeps on doing so, until a nonempty square is scanned, or until we come to the end of the tape.

It will have been noticed that the entire operation above required only three squares of the storage tape. One can easily check that the grammar of the fragment of English under consideration does not require more than a very small number of squares of the storage tape. For all practical purposes then, our machine is a finite automaton.

Let us see what happens if we adjoin to our grammar the production

$$S \longrightarrow \mathsf{if}\ S\ \mathsf{then}\ S$$

To make sure the grammar is normal, we must replace this by three productions

$$\begin{aligned} S &\longrightarrow \mathsf{if}\ S', \\ S' &\longrightarrow S\ S'', \\ S'' &\longrightarrow \mathsf{then}\ S. \end{aligned}$$

As we try to generate sentences that begin with if, if if, if if if, etc., we see that more and more squares of the storage tape are required. It is a matter of experience that a human speaker will get stuck when he tries to utter a sentence starting with if if if. This shows that whatever corresponds to the storage tape in the human brain has in fact a quite small capacity.

Surprisingly, the same machine can also be used for recognizing the sentences generated by a normal context-free grammar. By this we mean: if a terminal string is entered on the output tape (better called "input tape" now) to the right of the scanned square, the machine will erase this string and convert it to the initial symbol S on the scanned square of the storage tape, provided it adopts the right strategy.

We adopt the following moves:

$(1')$
$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 \\ \mathsf{left} \end{pmatrix}$$

$(2')$
$$\begin{pmatrix} 0 \\ T \end{pmatrix} \longrightarrow \begin{pmatrix} \mathsf{right} \\ W \end{pmatrix}$$

provided $W \longrightarrow T$ is a production of the grammar.

$$(3') \qquad \begin{pmatrix} 0 \\ T \end{pmatrix} \longrightarrow \begin{pmatrix} & T \\ 0 & \mathsf{left} \end{pmatrix}$$

$$(4') \qquad \begin{pmatrix} U \\ V \end{pmatrix} \longrightarrow \begin{pmatrix} \mathsf{right} & 0 \\ & W \end{pmatrix}$$

provided $W \longrightarrow UV$ is a production of the grammar.

$$(5') \qquad \begin{pmatrix} U \\ V \end{pmatrix} \longrightarrow \begin{pmatrix} \mathsf{left} \\ \mathsf{stay} \end{pmatrix}$$

The fourth move, for example, means: erase both scanned words, print W on the output tape, and move the storage tape one square to the right.

In the situations $\begin{pmatrix} 0 \\ T \end{pmatrix}$ and $\begin{pmatrix} U \\ V \end{pmatrix}$ the machine may find itself in a quandary. Suppose, for example, the machine is in the situation $\begin{pmatrix} U \\ V \end{pmatrix}$. When there is no production $W \longrightarrow UV$, only move $(5')$ is applicable. When there is a production $W \longrightarrow UV$, we prefer to make move $(4')$, but it may be a bad move, as we shall see in an example.

We assume the sentence $\mathsf{the\ man\ hit\ the\ cat\ and\ the\ ball}$ is entered on the input tape, to the right of the scanned square. We shall not show what is written on the input tape, except on the scanned square. Here are successive situations of the machine:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \to \begin{pmatrix} 0 \\ \mathsf{the} \end{pmatrix} \to \begin{pmatrix} 0 \\ Art \end{pmatrix} \to \begin{pmatrix} Art \\ \mathsf{man} \end{pmatrix} \to \begin{pmatrix} Art & 0 \\ & \mathsf{man} \end{pmatrix} \to \begin{pmatrix} Art \\ N \end{pmatrix} \to$$
$$\begin{pmatrix} 0 \\ NP \end{pmatrix} \to \begin{pmatrix} NP \\ \mathsf{hit} \end{pmatrix} \to \begin{pmatrix} NP & 0 \\ & \mathsf{hit} \end{pmatrix} \to \begin{pmatrix} NP \\ V_t \end{pmatrix} \to \begin{pmatrix} NP & 0 \\ & V_t \end{pmatrix} \to \begin{pmatrix} NP & V_t \\ & \mathsf{the} \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & 0 \\ & & \mathsf{the} \end{pmatrix} \to \begin{pmatrix} NP & V_t \\ & Art \end{pmatrix} \to \begin{pmatrix} NP & V_t & 0 \\ & & Art \end{pmatrix} \to \begin{pmatrix} NP & V_t & Art \\ & & \mathsf{cat} \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & Art & 0 \\ & & & \mathsf{cat} \end{pmatrix} \to \begin{pmatrix} NP & V_t & Art \\ & & N \end{pmatrix} \to \begin{pmatrix} NP & V_t \\ & NP \end{pmatrix}$$

At this stage we are tempted to make move $(4')$, but we shall make move $(5')$ instead (we abbreviate *Conj* by C):

$$\to \begin{pmatrix} NP & V_t & 0 \\ & & NP \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP \\ & & \mathsf{and} \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & NP & 0 \\ & & & \mathsf{and} \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP \\ & & C \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP & 0 \\ & & & C \end{pmatrix} \to$$

$$\begin{pmatrix} NP & V_t & NP & C \\ & & & \mathsf{the} \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP & C & 0 \\ & & & & \mathsf{the} \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP & C \\ & & & Art \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & NP & C & 0 \\ & & & & Art \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP & C & Art \\ & & & & \mathsf{ball} \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & NP & C & Art & 0 \\ & & & & & \mathsf{ball} \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP & C & Art \\ & & & & & N \end{pmatrix} \to$$
$$\begin{pmatrix} NP & V_t & NP & C \\ & & & NP \end{pmatrix} \to \begin{pmatrix} NP & V_t & NP \\ & & NP' \end{pmatrix} \to \begin{pmatrix} NP & V_t \\ & NP \end{pmatrix} \to \begin{pmatrix} NP \\ VP \end{pmatrix} \to \begin{pmatrix} 0 \\ S \end{pmatrix} \to$$
$$\begin{pmatrix} S \\ 0 \end{pmatrix}$$

What would have happened if we have followed our first inclination and made move $(4')$ instead of $(5')$?

$$\to \begin{pmatrix} NP \\ VP \end{pmatrix} \to \begin{pmatrix} 0 \\ S \end{pmatrix} \to \begin{pmatrix} S \\ \mathsf{and} \end{pmatrix} \to \begin{pmatrix} S & 0 \\ & \mathsf{and} \end{pmatrix} \to \begin{pmatrix} S \\ C \end{pmatrix} \to$$
$$\begin{pmatrix} S & 0 \\ & C \end{pmatrix} \to \begin{pmatrix} S & C \\ & \mathsf{the} \end{pmatrix} \to \begin{pmatrix} S & C & 0 \\ & & \mathsf{the} \end{pmatrix} \to \begin{pmatrix} S & C \\ & Art \end{pmatrix} \to \begin{pmatrix} S & C & 0 \\ & & Art \end{pmatrix} \to$$
$$\begin{pmatrix} S & C & Art \\ & & \mathsf{ball} \end{pmatrix} \to \begin{pmatrix} S & C & Art & 0 \\ & & & \mathsf{ball} \end{pmatrix} \to \begin{pmatrix} S & C & Art \\ & & N \end{pmatrix} \to \begin{pmatrix} S & C \\ & NP \end{pmatrix} \to$$
$$\begin{pmatrix} S \\ NP' \end{pmatrix} \to \begin{pmatrix} S & 0 \\ & NP' \end{pmatrix} \to \begin{pmatrix} S & NP' \\ & 0 \end{pmatrix}$$

and there's nowhere to go from here.

Exercise

1. Show how a machine would (a) produce, (b) recognize the sentence $\sigma^2\tau^2$ in the language $\mathcal{L}_1$ of Section 4.2, Example 4.2.2. (Hint: First replace the original context-free grammar of $\mathcal{L}_1$ by a normal one.

4.7 Derivations and parsing trees

For many purposes, proofs in tree form are better replaced by "derivations". A **derivation** of $C \longrightarrow D$ is a finite sequence of strings

$$C = C_0, C_1, \ldots, C_n = D$$

where

$$C_i = E_i A_i F_i, \qquad C_{i+1} = E_i B_i F_i$$

and $A_i \longrightarrow B_i$ is a production for each $i = 0, 1, \ldots, n-1$. We say that this derivation has n **steps**.

By a derivation of 0 steps we mean an application of the reflexive law: $C \longrightarrow C = D$. A derivation of 1 step has the form

$$C = EAF, \qquad EBF = D$$

where $A \longrightarrow B$ is a production. Note that $C \longrightarrow D$ can then be proved by the substitution rule (S′). We shall now give an example of a 2-step derivation. Suppose $A_0 \longrightarrow B_0$ and $A_1 \longrightarrow B_1$ are productions and $C = E_0A_0GA_1F_1$, $D = E_0B_0GB_1F_1$. Then we have the derivation

$$\begin{aligned} C = C_0 = E_0A_0\underbrace{(GA_1F_1)}_{F_0} &\longrightarrow C_1 = E_0B_0\underbrace{(GA_1F_1)}_{F_0} \\ = \underbrace{(E_0B_0G)}_{E_1}A_1F_1 &\longrightarrow C_2 = \underbrace{(E_0B_0G)}_{E_1}B_1F_1 \end{aligned}$$

In this example it is clear that we can prove $C \longrightarrow D$ as follows:

$$\dfrac{\dfrac{A_0 \longrightarrow B_0}{E_0A_0GA_1F_1 \longrightarrow E_0B_0GA_1F_1}\ (\text{S'}) \qquad \dfrac{A_1 \longrightarrow B_1}{E_0B_0GA_1F_1 \longrightarrow E_0B_0GB_1F_1}\ (\text{S'})}{E_0A_0GA_1F_1 \longrightarrow E_0B_0GB_1F_1}\ (\text{T})$$

Quite generally, if $C \longrightarrow D$ has a derivation, then it also has a proof in tree form. Indeed, in view of reflexivity and transitivity, we need only recall that this so for 1 step derivations.

We have thus established one half of the following:

4.7.1 THEOREM *A formula $C \longrightarrow D$ is provable in the deductive system $\mathcal{D}(\mathcal{G})$ if and only if it has a derivation.*

It remains to prove the implication the other way. Suppose $C' \longrightarrow D'$ is provable, that is, possesses a proof in tree form. We shall show by induction on the length of this proof that there is a derivation of $C' \longrightarrow D'$. If $C' = D'$, we have an instance of the reflexive law, hence a 0 step derivation.

If $C' \longrightarrow D'$ is a production, there is a one step derivation.

Suppose $C' \longrightarrow D'$ is obtained by substitution thus:

$$\dfrac{C \longrightarrow D}{C' = ECF \longrightarrow EDF = D'}\ (\text{S'})$$

By inductional assumption, $C \longrightarrow D$ has a derivation, say:

$$C = C_0, C_1, \ldots, C_n = D$$

Then

$$C' = EC_0F, EC_1F, \ldots, EC_nF = D'$$

is also a derivation, as is easily checked.

Finally, suppose $C' \longrightarrow D'$ is obtained by the transitive law thus:

$$\frac{C' \longrightarrow E \qquad\qquad E \longrightarrow D'}{C' \longrightarrow D'} \quad \text{(T)}$$

Juxtaposing derivations of $C' \longrightarrow E$ and $E \longrightarrow D'$, which may be assumed to exist by inductional assumption, we obtain a derivation of $C' \longrightarrow D'$. ∎

As an immediate consequence of the theorem we have the following:

4.7.2 COROLLARY *If $\mathcal{G} = (\mathcal{V}, \Sigma, \Sigma', \mathcal{P})$ is a grammar, then the language $\mathcal{L}(\mathcal{G})$ consists of all strings $E \in \Sigma^*$ for which there exists a derivation $S \longrightarrow E$ for some $S \in \Sigma'$.*

Note that in generative grammars, there is a unique $S \in \Sigma'$.

We may now return to Example 4.2.2 of Section 4.2. We wish to determine the language $\mathcal{L}(\mathcal{G}_1)$, where $\mathcal{G}_1$ is the grammar with productions

$$S \longrightarrow \sigma\tau \qquad\qquad S \longrightarrow \sigma S\tau$$

Thus we seek all possible derivations $S \longrightarrow E$ with $E \in \{\sigma, \tau\}^*$. There cannot be a 0 step derivation of $S \longrightarrow E$, since S is not an element of Σ^*. If there is a 1 step derivation, it must have the form

$$S = C_0 = E_0A_0F_0,\ C_1 = E_0B_0F_0 = E$$

where $A_0 \longrightarrow B_0$ is a production. Clearly, $A_0 = S$, $B_0 = \sigma\tau$ and $E_0 = F_0 = 1$. If there is a 2 step derivation $S = C_0, C_1, C_2 = E$, we require that

$$C_0 = E_0A_0F_0,\ C_1 = E_0B_0F_0 = E_1A_1F_1,\ C_2 = E_1B_1F_1$$

where $A_0 \longrightarrow B_0$ and $A_1 \longrightarrow B_1$ are productions. Clearly, then $A_0 = A_1 = S$, hence $E_0 = F_0 = 1$, and B_0, B_1 must be $\sigma\tau$ or $\sigma S\tau$. Now $B_0 = C_1 = E_1A_1F_1 = E_1SF_1$, hence $B_0 = \sigma S\tau$ and $E_1 = \sigma$, $F_1 = \tau$. Thus $C_2 = \sigma B_1\tau = E \in \{\sigma, \tau\}^*$, and so $B_1 = \sigma\tau$. Therefore, $E = C_2 = \sigma\sigma\tau\tau = \sigma^2\tau^2$.

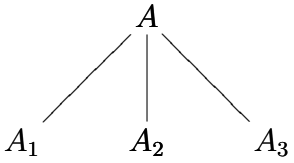
Continuing in the same way, we see that, if $S \longrightarrow E$ has an n step derivation, then $E = \sigma^n\tau^n$. If properly done, this proof should be by induction on n.

We shall now return to the notion of "parsing tree", which was informally introduced in Section 4.1. While proofs in tree form and derivations are valid techniques for all grammars, parsing trees are essentially confined to context-free grammars. (Their use has been extended to context sensitive grammars, but these will not be discussed here.)

In a context-free grammar, each production has the form

$$A \longrightarrow A_1 A_2 \dots A_n$$

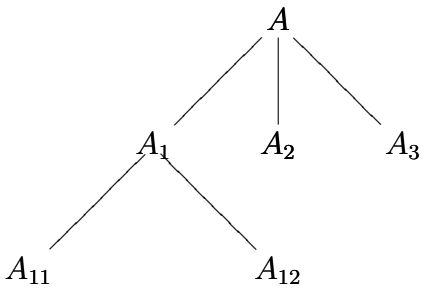
where A, A_1, A_2, …, A_n are words and $n \geq 1$. Take for instance the case $n = 3$, then we associate with this production the following picture:



Now suppose we are interested in a derivation $A \longrightarrow E$. Then either the string E starts with A_1 or else, at some stage of the derivation, A_1 is replaced with the help of a production

$$A_1 \longrightarrow A_{11} A_{12} \dots A_{1n_1}$$

Take, for instance, the case $n_1 = 2$, then we enlarge the picture to



Continuing in this fashion, we ultimately obtain a tree where all the words occurring in the string E appear at the end of the branches. Usually we are interested in the situation $A = S$, $E \in \Sigma^*$. The resulting tree is called a **parsing tree** of E.

For example, let us look at the grammar $\mathcal{G}_0 = (\mathcal{V}, \{\sigma, \tau\}, \{S\}, \mathcal{P})$ with vocabulary $\mathcal{V} = \{\sigma, \tau, S, U, V\}$ and productions $\mathcal{P} : U \longrightarrow \sigma$, $U \longrightarrow \sigma U$, $V \longrightarrow \tau$, $V \longrightarrow \tau V$, $S \longrightarrow UV$. We shall show that $\sigma^2\tau$ is a sentence generated by this grammar

(a) by giving a proof in tree form of the formula $S \longrightarrow \sigma^2\tau$,

(b) by presenting a derivation,

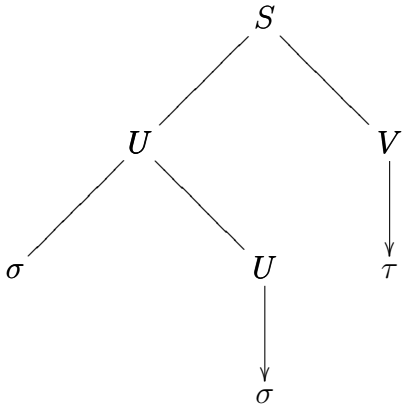
(c) by exhibiting a parsing tree.

(*a*)

$$\dfrac{\dfrac{U \xrightarrow{(P2)} \sigma U \qquad \dfrac{U \xrightarrow{(P1)} \sigma}{\sigma U \longrightarrow \sigma\sigma}\ (\text{S'})}{U \longrightarrow \sigma\sigma}\ (\text{T}) \qquad V \xrightarrow{(P3)} \tau}{UV \longrightarrow \sigma\sigma\tau}\ (\text{S}) \qquad S \xrightarrow{(P5)} UV}{S \longrightarrow \sigma\sigma\tau}\ (\text{T})$$

(*b*) S, UV, σUV, $\sigma U\tau$, $\sigma\sigma\tau$

(*c*)



Exercise

1. Show that $\sigma^3\tau^2 \in \mathcal{L}(\mathcal{G}_0)$

- **a.** by a proof in tree form,
- **b.** by a derivation,
- **c.** by a parsing tree.

4.8 A necessary condition for a language to be context-free

We asserted in Section 4.4 that the intersection of two context-free languages need not be context-free. Indeed, let us consider the terminal vocabulary $\Sigma = \{\sigma, \tau\}$ and the languages

$$\mathcal{L}_3 = \{\sigma^m\tau^m\sigma^p \mid m \geq 1,\ p \geq 1\}$$

$$\mathcal{L}_4 = \{\sigma^q\tau^n\sigma^n \mid q \geq 1,\ n \geq 1\}$$

We shall see that $\mathcal{L}_3$ and $\mathcal{L}_4$ have context-free grammars but that $\mathcal{L}_3 \cap \mathcal{L}_4$ does not.

First, observe that $\mathcal{L}_3 = \mathcal{L}_1\mathcal{L}_5$, where

$$\mathcal{L}_1 = \{\sigma^m\tau^m \mid m \geq 1\}$$

has already been met in Section 4.2, and

$$\mathcal{L}_5 = \{\sigma^p \mid p \geq 1\}$$

has the obvious grammar: $S_5 \longrightarrow \sigma$, $S_5 \longrightarrow \sigma S_5$. Therefore, $\mathcal{L}_3$ is the product of two context-free languages and thus also context-free, by Theorem 4.4.1. Now $\mathcal{L}_4$ is clearly the mirror-image of $\mathcal{L}_3$, hence also context-free, by the same theorem.

Now it is clear that

$$\mathcal{L}_3 \cap \mathcal{L}_4 = \{\sigma^k\tau^k\sigma^k \mid k \geq 1\}$$

We shall see that this language is not context-free. To show this, we require a result by Bar-Hillel, Perles and Shamir.

4.8.1 PROPOSITION *If $\mathcal{L} \subseteq \Sigma^*$ is an infinite language with a context-free grammar, then there exist strings A, B, C, D, $E \in \Sigma^*$ such that $AB^nCD^nE \in \mathcal{L}$ for all $n \geq 0$, where B and D are not both empty.*

For, example, in the fragment of English discussed in Section 4.1, we might take

$A =$ the man hit the cat
$B =$ and the man hit the cat
$C =$ and
$D =$ the man hit the cat and
$E =$ the man hit the cat

Then $ABCDE$, $ABBCDDE$, $ABBBCDDDE$, etc. are all grammatical sentences.

We shall not prove this result now, but apply it to the discussion of $\mathcal{L} = \mathcal{L}_3 \cap \mathcal{L}4$.

Let us suppose that $\mathcal{L} = \mathcal{L}_3 \cap \mathcal{L}_4$ is a context-free language. Then, by the proposition, there exist strings A, B, C, D, E such that $F_n = AB^nCD^nE \in \mathcal{L}$ for all $n \geq 0$, and B and D are not both empty. In particular, $F_1 = ABCDE = \sigma^k\tau^k\sigma^k$, $F_2 = ABBCDDE = \sigma^l\tau^l\sigma^l$, for certain k, $l \geq 1$. Moreover, since B and D are not both empty, l is strictly larger than k.

Now since A is an initial segment of F, we must have either that $A = \sigma^p$ for some $p \leq k$ or that A begins with the string $\sigma^k\tau$. But in the latter case, it follows that $F_2 = AB\cdots$ also begins with the string $\sigma^k\tau$ which is impossible since $F_2 = \sigma^l\cdots$ with $l > k$. Therefore $A = \sigma^p, p \leq k$, and, by the mirror image argument, $E = \sigma^q$, with $q \leq l$. We then get

$$BCD = \sigma^{k-p}\tau^k\sigma^{k-q}$$

from F_1, and

$$BBCDD = \sigma^{l-p}\tau^l\sigma^{l-q}$$

from F_2. If $B = \sigma\cdots\tau\cdots$, we would obtain

$$BBCDD = \sigma\cdots\tau\cdots\sigma\cdots\tau\cdots$$

contradicting the above. Therefore, B is a power of σ or a power of τ. Suppose $B = \tau^i$, $i \geq 1$, then the formulas for BCD and $BBCDD$ tell us that $k - p = 0$ and $l - p = 0$, again leading to the contradiction that $k = l$. Therefore, $B = \sigma^i$, and, similarly, $D = \sigma^j$. Thus, from the formula for BCD, we obtain

$$C = \sigma^{k-p-i}\tau^k\sigma^{k-q-j}$$

and, from the formula for $BBCDD$, we obtain

$$C = \sigma^{l-p-2i}\tau^l\sigma^{l-q-2j}$$

Thus, again we are led to the contradiction that $k = l$.

We have now exhausted all the possible cases, so that $\mathcal{L}_3 \cap \mathcal{L}_4$ cannot be a context-free language.

We shall prove Proposition 4.8.1 after some discussion.

Let us recall Example 4.1.1 of Section 4.1. The 72 sentences of that language may be summarized by the formula:

$$S \longrightarrow (\mathsf{the} \vee \mathsf{a})(\mathsf{man} \vee \mathsf{ball} \vee \mathsf{cat})(\mathsf{hit} \vee \mathsf{took})(\mathsf{the} \vee \mathsf{a})(\mathsf{man} \vee \mathsf{ball} \vee \mathsf{cat})$$

Indeed, if we regard multiplication as being distributive over $\vee$, we obtain:

$$S \longrightarrow (\mathsf{the\ man\ hit\ the\ cat}) \vee (\mathsf{the\ man\ hit\ the\ ball}) \vee \cdots$$

We have seen how adding the single word and will make the language infinite, provided we add only the following production: $S \longrightarrow S$ and S. Or we could add the single word poor and the production $N \longrightarrow$ poor N, and again we would get infinitely many sentences of the form

$$\mathrm{S} \longrightarrow \mathsf{the\ (poor)}^n \mathsf{\ man\ hit\ the\ cat}, \qquad \mathrm{n} = 0,1,2,\ldots$$

In both cases we have a production of the form $T \longrightarrow UTV$, where T belongs to the auxiliary vocabulary and U and V are not both empty. In the first case: $T = S$, $U = 1$, $V =$ and S; in the second case: $T = N$, $U =$ poor, $V = 1$. (Recall that $1 =$ the empty string.) This idea will be used in the proof of the proposition.

First, we shall establish a lemma.

4.8.2 LEMMA *Let $\mathcal{L}$ be a language with a context-free grammar $\mathcal{G}$. Then, without loss in generality, we may assume that, in any production $A \longrightarrow B$, either B is a terminal word or it is a string containing at least two words.*

PROOF For the purpose of this proof, let us call the nonterminal word B "bad" if it occurs on the right hand side of a production of $\mathcal{G}$. Let

$$A_1 \longrightarrow B,\ A_2 \longrightarrow B,\ \ldots,\ A_m \longrightarrow B$$

be a complete list of these productions. We also make a list of all productions which contain the bad word on the left hand side:

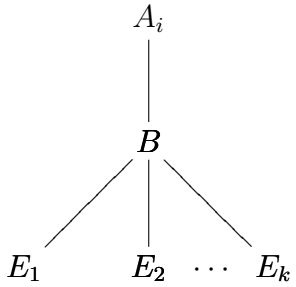
$$B \longrightarrow C_1,\ B \longrightarrow C_2,\ \ldots,\ B \longrightarrow C_n$$

where the C_j are strings. We may also assume that all $C_j \neq B$, as $B \longrightarrow B$ is an instance of the reflexive law and need not be given as a production.

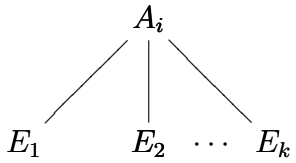
We now form a new grammar $\mathcal{G}'$. This differs from $\mathcal{G}$ in only one respect: the m productions $A_i \longrightarrow B$ and the n productions $B \longrightarrow C_j$ are replaced by the mn productions $A_i \longrightarrow C_j$.

We note first of all that $\mathcal{G}'$ has one less bad word than $\mathcal{G}$. For none of the C_j is B, and if C_j is bad in $\mathcal{G}'$, that is, a non terminal word, then it was already bad in $\mathcal{G}$.

We note secondly that $\mathcal{G}'$ generates the same language $\mathcal{L}$ as does $\mathcal{G}$. For suppose the production $A_i \longrightarrow C_j$ is used in showing that $S \longrightarrow D$ is a theorem in $\mathcal{D}(\mathcal{G}')$, for some terminal string D, then $A_i \longrightarrow B$ and $B \longrightarrow C_j$ may be used to establish the same theorem in $\mathcal{D}(\mathcal{G})$. Conversely, suppose the production $A_i \longrightarrow B$ is used in showing that $S \longrightarrow D$ is a theorem in $\mathcal{D}(\mathcal{G})$, D being a terminal string, then the parsing tree of D must contain as a part:



where $E_1E_2\cdots E_k = C_j$ for some j. Replacing this part by



we obtain a parsing tree for D in $\mathcal{G}'$, which utilizes the production $A_i \longrightarrow C_j$ in place of $A_i \longrightarrow B$.

Thus we have replaced the grammar $\mathcal{G}$ by another grammar $\mathcal{G}'$ which generates the same language as $\mathcal{G}$, but which contains one bad word less. Repeating this process, we ultimately get rid of all bad words without changing the language generated. ∎

We are finally able to give a proof of Proposition 4.8.1. By the **depth** of a parsing tree, we mean the length of the longest downward path you can follow along the tree. It corresponds to the number of steps in a production.

PROOF Let $\mathcal{L}$ be an infinite language with a context-free grammar that satisfies the conclusions of Lemma 4.8.2. In what follows, let

> m be the maximal length of all strings B which may appear on the right hand side of a production $A \longrightarrow B$ and
>
> p be the total number of words in the vocabulary $\mathcal{V}$.

We will now bound the number of parsing trees of depth at most l. The number of trees of depth 0 is 2 (including the empty tree) and the number of trees of depth at most 1 is $m+2$, since the only such trees are the 2 of depth 0 and the ones of depth 1 with the top node and a k-fold branch with $1 \leq k \leq m$. Let $f(l)$ denote the number of trees of depth at most l. Then from the fact that a tree of depth at most $l+1$ can be gotten (usually in more than one way) by replacing each of the bottom nodes of one of the $m+2$ trees of depth at most one by a tree of depth at most l and there are at most m such bottom nodes, the number of possibilities is at most $f(l)^m$. In particular, we have that $f(2) \leq (m+2)^m$, which gives $f(3) \leq f(2)^m \leq ((m+2)^m)^m = (m+2)^{2m}$, $\ldots$, $f(l) \leq (m+2)^{l-1}$.

A parsing tree is gotten by replacing each node in a tree by a word in the language. Since each node of a parsing tree may be occupied by at most p words, the total number of parsing trees of depth l cannot exceed $p^{f(l) \leq (m+2)^{l-1}}$; hence there cannot be more than that number of sentences in the language $\mathcal{L}$.

Given that the number of sentences in $\mathcal{L}$ is infinite, it follows that there must be parsing trees of arbitrary depth. Therefore, there exist parsing trees with branches of arbitrary length. In particular, there will exist a parsing tree with a path of length $q > p$. For convenience, we shall picture this path horizontally:

$$S = C_0 \text{ — } C_1 \text{ — } C_2 \text{ — } \cdots \text{ — } C_{q-1} \in \Sigma$$

a path in the tree with each $C_i \in \Sigma$. Since the total number of words in the vocabulary is $p < q$, the words appearing in this branch cannot all be different; say $C_i = C_j$, where $i < j < q-1$.

Now for each k such that $i \leq k < j$ there is a production

$$C_k \longrightarrow B_k C_{k+1} D_k$$

with $B_k, D_k \in \Sigma^*$. Since we have supposed that $\mathcal{G}$ satisfies the conclusions of Lemma 4.8.2 it follows that at least one of B_k or D_k is nonempty. In particular, either B_i or D_i is nonempty. We can therefore deduce a $(j-i)$-step production

$$C_i \longrightarrow B'C_jD'$$

where $B' = B_iB_{i+1}\dots B_{j-1}$ and $D' = D_{j-1}\dots D_i$, and at least one of B' and D' is nonempty. Putting $C' = C_i = C_j$, we obtain a theorem

$$C' \longrightarrow B'C'D' \tag{1}$$

Similarly we may obtain a theorem

$$S \longrightarrow A'C'E' \tag{2}$$

where A' and E' are strings that might be empty. From (1) and (2) we deduce

$$S \longrightarrow A'B'C'D'E' \longrightarrow A'B'B'C'D'D'E'$$

and so on, in fact,

$$S \longrightarrow A'(B')^nC'(D')^nE' \tag{3}$$

for all $n \geq 0$.

We have almost proved the proposition, except that there is no reason for A', B', C', D' and E' to be terminal strings. It remains to be shown that they can be replaced by terminal strings in (3).

Take for instance $B' = B_iB_{i+1}\dots B_{j-1}$. Now B_i occurs in a parsing tree, hence there is a theorem

$$B_i \longrightarrow \beta_1\beta_2\cdots\beta_{n_i}$$

in our deductive system, where the β_r are the terminal words appearing at endpoints of branches emanating from B_i. Applying the same reasoning to B_{i+1}, …, B_j, we obtain a theorem

$$B' \longrightarrow \beta_1\beta_2\cdots\beta_n = B$$

Thus there are theorems

$$A' \longrightarrow A, \quad B' \longrightarrow B, \quad C' \longrightarrow C, \quad D' \longrightarrow D, \quad E' \longrightarrow E \tag{4}$$

where A, B, C, D and E are terminal strings. Since our grammar is context-free, B contains at least as many words as B' and D at least as many as D'. Since B' or D' is known to be nonempty, it follows that B or D is nonempty. Finally, from (3) and (4), we obtain the theorem

$$S \longrightarrow AB^nCD^nE$$

for all $n \geq 0$, and our proof is complete. ∎

Chapter 5

Generative and transformational grammars

5.1 Inflection rules conjugation

Traditional grammar distinguishes between syntax and morphology, the former dealing with the formation of sentences from words, while the latter deals with the formation of words themselves. By now the reader should be convinced that context-free grammars constitute a useful tool for dealing with syntax. Let us now look at such rules as:

```
Past work → worked
Plur man  → men
Part be   → being
```

If we count `Past`, `Plur` and `Part` among the words of the auxiliary vocabulary, these rules clearly do not fit into a context-free grammar, or even a context-sensitive grammar, as the left hand side of these productions contains more words than the right hand side. Unfortunately, English is still an inflected language, and any grammar of English must incorporate such inflection rules, or "morphographemic" rules, as it may be more fashionable to call them.

Let us begin by looking at the inflected forms of the English verb. We note that for each verb V there are 6 forms $C_{ki}(V)$ where $k = 1, 2$ refers to the tenses present and past, and $i = 1, 2, 3$ refers to the three persons usually associated with the pronouns "I", "you" and "he" or "she". These 6 forms are conveniently arranged in the form of a matrix:

$$C(V) = \begin{pmatrix} C_{11}(V) & C_{12}(V) & C_{13}(V) \\ C_{21}(V) & C_{22}(V) & C_{23}(V) \end{pmatrix}$$

Here are some examples

$$C(\mathsf{work}) = \begin{pmatrix} \mathsf{work} & \mathsf{work} & \mathsf{works} \\ \mathsf{worked} & \mathsf{worked} & \mathsf{worked} \end{pmatrix}$$
$$C(\mathsf{go}) = \begin{pmatrix} \mathsf{go} & \mathsf{go} & \mathsf{goes} \\ \mathsf{went} & \mathsf{went} & \mathsf{gone} \end{pmatrix}$$
$$C(\mathsf{can}) = \begin{pmatrix} \mathsf{can} & \mathsf{can} & \mathsf{can} \\ \mathsf{could} & \mathsf{could} & \mathsf{could} \end{pmatrix}$$
$$C(\mathsf{must}) = \begin{pmatrix} \mathsf{must} & \mathsf{must} & \mathsf{must} \\ \mathsf{must} & \mathsf{must} & \mathsf{must} \end{pmatrix}$$
$$C(\mathsf{be}) = \begin{pmatrix} \mathsf{am} & \mathsf{are} & \mathsf{is} \\ \mathsf{was} & \mathsf{were} & \mathsf{was} \end{pmatrix}$$

A few comments are in order.

1. English verbs are usually presented in the infinitive, except modal auxiliaries such as can and must, which do not have an infinitive.

2. We have mentioned two simple tenses: present and past. Compound tenses will be dealt with in the next section. There are two more simple tenses in English: the present and past subjunctive. However, these occur so rarely that we have chosen to ignore them here.

3. We have mentioned three persons of the singular. What about the three persons of the plural? We may safely identify these with the second person of the singular at this stage of our investigation, as the verb forms of the plural always coincide with those of the second person singular.

4. The conjugation matrix for work is regular and serves as a model for most English verbs. Any departure from this model ought to be listed in the dictionary.

5. Actually, at most five of the six forms mentioned are distinct, as always $C_{21}(V) = C_{23}(V)$. The maximum, diversity is achieved for $V = \mathsf{be}$, and for $V = \mathsf{must}$ all six forms coincide.

After this preliminary discussion we are able to state productions which will generate some elementary English sentences, due attention being paid to the morphology.

(1) $$S \longrightarrow Subj\ \ Pred$$
(2) $$Subj \longrightarrow NP_i\ P_i,\ \ (i = 1, 2, 3)$$
(3) $$NP_1 \longrightarrow \mathsf{I}$$

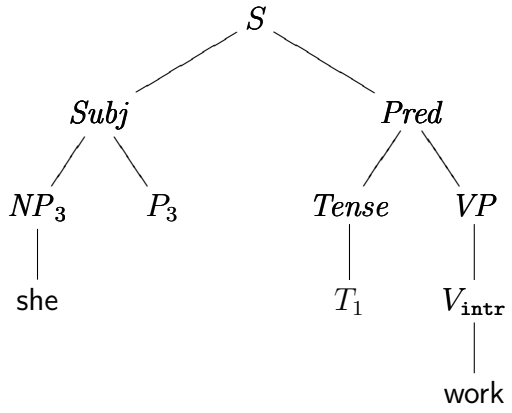
$$NP_2 \longrightarrow \text{you, we, they}$$
$$NP_3 \longrightarrow \text{he, she, it, one, } Name$$
$$Name \longrightarrow \text{John, Jane, } \ldots$$
(4) $$Pred \longrightarrow Tense\ VP$$
(5) $$Tense \longrightarrow T_1,\ T_2$$
(6) $$VP \longrightarrow V_{\texttt{intr}},\ V_{\texttt{tr}}\ Obj, \ldots$$
(7) $$V_{\texttt{intr}} \longrightarrow \text{work, rest, go, } \ldots$$
$$V_{\texttt{tr}} \longrightarrow \text{like, call, hit, take, } \ldots$$
(8) $$Obj \longrightarrow \texttt{Acc}\ NP_i (i = 1, 2, 3)$$
(9) $$P_i\ T_k\ V \longrightarrow C_{ki}(V),\ (i = 1, 2, 3, k = 1, 2),\ \text{for any verb } V$$
(10) $$\texttt{Acc I} \longrightarrow \text{me}$$
$$\texttt{Acc he} \longrightarrow \text{him}$$
$$\texttt{Acc she} \longrightarrow \text{her}$$
$$\texttt{Acc we} \longrightarrow \text{us}$$
$$\texttt{Acc they} \longrightarrow \text{them}$$
$$\texttt{Acc}\ X \longrightarrow X \text{ otherwise}$$

The auxiliary vocabulary consists, aside from *Name* and *Tense*, of the following abbreviations:

S = statement
$Subj$ = subject
$Pred$ = predicate
NP_i = ith person noun phrase
P_i = ith person
T_1 = present
T_2 = past
VP = verb phrase
$V_{\texttt{intr}}$ = intransitive verb
$V_{\texttt{tr}}$ = transitive verb
Obj = object
Acc = accusative case

For the moment, V is called a "verb" if there is a production $V_{\texttt{intr}} \longrightarrow V$ or $V_{\texttt{tr}} \longrightarrow V$. Other verbs will be discussed later. The distinction between transitive and intransitive verbs is complicated by the fact that some intransitive verbs can be used transitively, as in I work him hard, and some transitive verbs permit object deletion, as in she called today, but we shall ignore these complications here.

We note that productions (1) to (8) satisfy the definition of a context-free grammar, only productions (9) and (10) do not. For example, let us see how we can generate the sentence she works. Staying within the context-free part of the grammar, we may construct the following parsing tree:



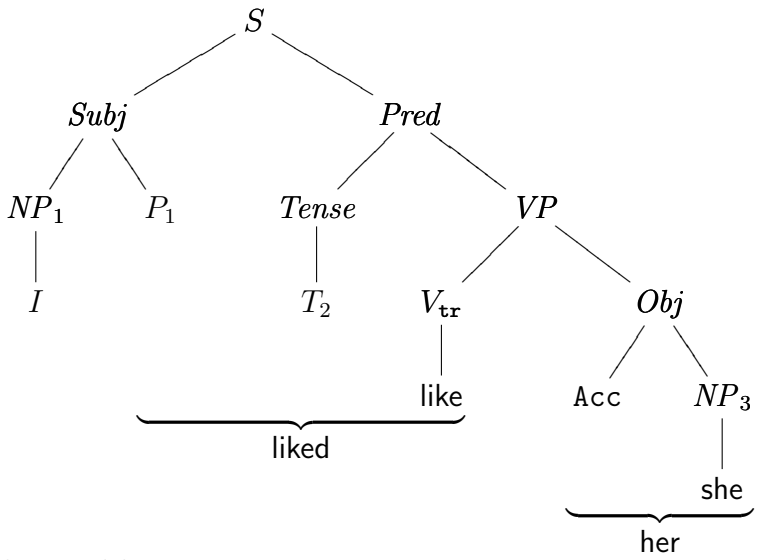
This parsing tree yields only the pseudo-sentence:

$$\text{she } P_3\ T_1\ \text{work}$$

To convert this into a terminal string, we apply rule (9):

$$P_3\ T_1\ \text{work} \longrightarrow C_{13}(\text{work}) = \text{works}$$

Here is another example:



The inflection rules used here are:

$$P_1\ T_2\ \text{like} \longrightarrow C_{21}(\text{like}) = \text{liked}, \qquad \texttt{Acc she} \longrightarrow \text{her}$$

Exercises

1. Consider the following sentences:

a. I like her

b. she likes me

c. we worked

d. John took it

Construct the underlying parsing trees and show which productions yield the terminal strings.

2. Suppose we add to the above productions the following:

$$NP_2 \longrightarrow NP_i \text{ and } NP_j$$

Show that the following four strings then become sentences.:

a. John and I work

b. I and John work

c. she likes me and John

d. she likes John and I

3. Suppose, in place of the production in Exercise 2, we postulate the following two productions:

$$Subj \longrightarrow NP_i \text{ and } NP_j \ P_2$$

$$Obj \longrightarrow \texttt{Acc}\ NP_i \text{ and } \texttt{Acc}\ NP_j$$

Show that (a), (b) and (c) in Exercise 2 remain sentences, but that (d) will be replaced by:

(d') she likes John and me

5.2 More about tense and verb phrase

While Section 5.1 has served to illustrate some general principles, only the simplest sentences can be generated by the productions considered so far. In particular, we have only looked at two simple tenses: present and past. However, even aside from the subjunctive, which we have chosen to ignore, there are lots of compound tenses. We shall now amplify production (5) of Section 5.1:

(5.1) $$Tense \longrightarrow T_i \ Mod \ Asp_1 \ Asp2, \quad (i = 1, 2)$$

(5.2) $$Mod \longrightarrow 1, \ V_{\texttt{mod}}$$

(5.3) $$Asp_1 \longrightarrow 1, \ \text{have } \texttt{Perf}$$

(5.4) $$Asp_2 \longrightarrow 1, \ \text{be } \texttt{Part}$$

(5.1) to (5.4) are usually summarized as follows:

$$\mathit{Tense} \longrightarrow T_i\ (V_{\texttt{mod}})(\textsf{have}\ \texttt{Perf})(\textsf{be}\ \texttt{Part})$$

We require the following further productions:

(5.5) $$V_{\texttt{mod}} \longrightarrow \textsf{must, may, can, shall, will}$$

(5.6)
$$\begin{aligned}
\texttt{Part}\ \textsf{work} &\longrightarrow \textsf{working}\\
\texttt{Part}\ \textsf{take} &\longrightarrow \textsf{taking}\\
&\vdots\\
\texttt{Perf}\ \textsf{work} &\longrightarrow \textsf{worked}\\
\texttt{Perf}\ \textsf{take} &\longrightarrow \textsf{taken}\\
\texttt{Perf}\ \textsf{be} &\longrightarrow \textsf{been}\\
&\vdots
\end{aligned}$$

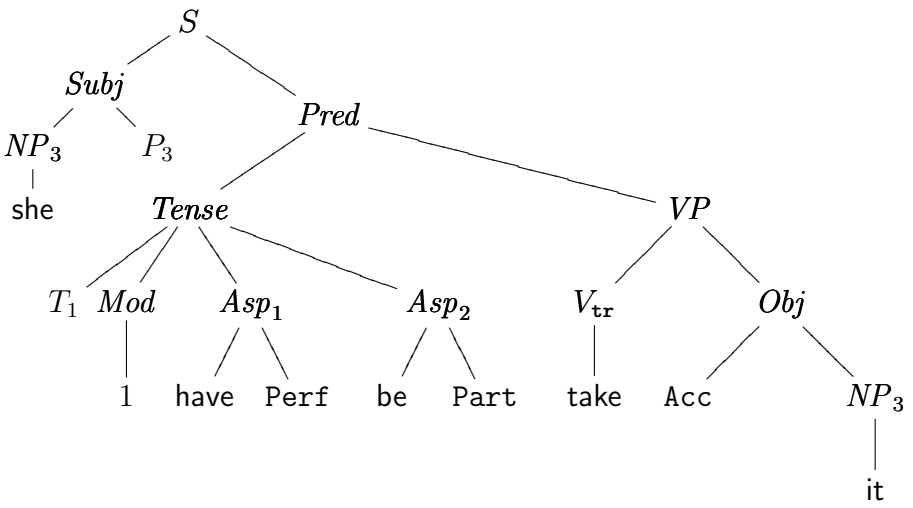
We have used a number of new auxiliary words:

$$\begin{aligned}
\mathit{Mod} &= \text{modality}\\
V_{\texttt{mod}} &= \text{modal auxiliary verb}\\
\mathit{Asp}_i &= \text{aspect of type } i,\ (i = 1, 2)\\
\texttt{Perf} &= \text{past participle (= perfect)}\\
\texttt{Part} &= \text{present participle}
\end{aligned}$$

(The word "perfect" is borrowed from Latin grammar, where it stands for completed action. Curiously, in English it is the simple past that is perfective in that sense, while it is the compound past, whose meaning is generally imperfective that is called the perfect. This illustrates the follity of attempting to impose the grammar of one language on another.)

The class of verbs has now been enlarged by $V = \textsf{have}$ and $V = \textsf{be}$, and by such V for which there are productions $V_{\texttt{mod}} \longrightarrow V$. It should be pointed out that some verbs, such as like, are not supposed to admit a progressive form, although one hears such sentences as how are you liking your new dress?

Here is an example:



This parsing tree yields a pseudo-sentence, which is converted into a terminal string as follows:

$$\text{she } \underbrace{P_3\ T_1 \text{ have}}_{\text{has}}\ \underbrace{\texttt{Perf be}}_{\text{been}}\ \underbrace{\texttt{Part take}}_{\text{taking}}\ \underbrace{\texttt{Acc it}}_{\text{it}}$$

Now let us turn attention to the "verb phrase" introduced in Section 5.1. (It should be pointed out that we have used this term more narrowly than other authors, whose *VP* comprises tense and person.) The following expansion of rule (6) of Section 5.1 gives only a very partial account of the possible verb phrase constructions in English:

(6) $V_P \rightarrow V_{\texttt{intr}},\ V_{\texttt{tr}}\ Obj,\ V_{\texttt{cop}}\ Adj,\ V_{\text{give}}\ Obj\ Obj,$
$V_{\text{give}}\ Obj$ to Obj, $V_{\text{make}}\ Obj\ Adj$

(6.1) $V_{\texttt{cop}} \rightarrow$ be, stay, seem, look, ...

(6.2) $Adj \rightarrow$ poor, old, good, green, ...

(6.3) $V_{\texttt{make}} \rightarrow$ make, render, leave, ...

(6.4) $V_{\texttt{give}} \rightarrow$ give, teach, tell, show, ...

Here

$V_{\texttt{cop}}$ = copula
Adj = adjective
$V_{\texttt{make}}$ = verb like "make"
$V_{\texttt{give}}$ = verb like "give"

The class of verbs has been enlarged, and we might summarize all the verbs introduced hitherto by a production:

$$Verb \longrightarrow V_{\texttt{intr}},\ V_{\texttt{tr}},\ \text{have},\ \text{be},\ V_{\texttt{mod}},\ V_{\texttt{cop}},\ V_{\text{make}},\ V_{\text{give}}, \ldots$$

There is some duplication in this list, as for example: $V_{\texttt{tr}} \longrightarrow$ have, $V_{\texttt{make}}$ and $V_{\texttt{give}}$ seem to be causally related to $V_{\texttt{cop}}$ and $V_{\texttt{tr}}$. For example, make means cause to be, give means cause to have, teach means cause to know, etc.

As usual, we have neglected to specify what types of objects can be taken by these verbs. For example, in

$$X \text{ teaches } Y \text{ to } Z$$

which is equivalent to

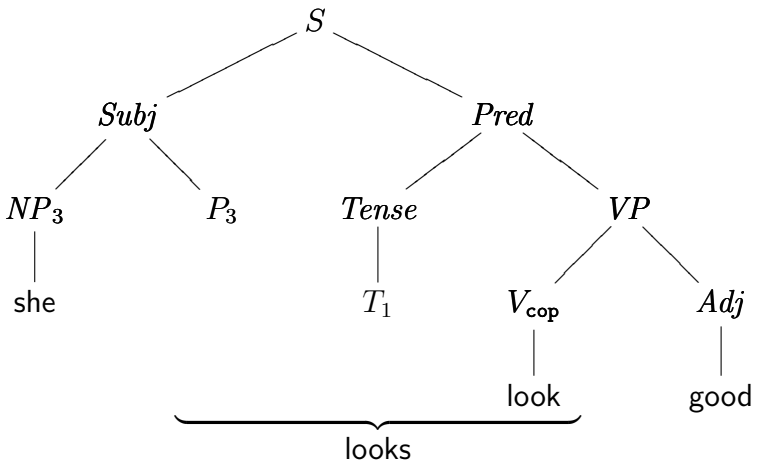
$$X \text{ teaches } Z\ Y$$

Z normally denotes a person or an animal, and Y denotes something but not somebody, while in

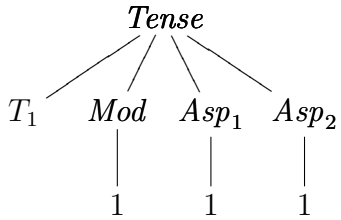
$$Z \text{ knows } Y$$

Y may denote somebody as well as something. Many grammarians try to incorporate such restrictions into the grammar. However, it is difficult to be consistent. One author recently asserted that the subject of eat must be animate. Unfortunately, he went on to argue his case by saying that stones do not eat, thus inadvertently using the construction which he claimed is ungrammatical.

Here is one illustration of the expanded version of (6):



Here we did not make use of the expanded version of (5), or we should have replaced the Tense part of the parsing tree by:



Exercises

1. For the following sentences construct the underlying parsing trees and show how the terminal strings are derived:

- **a.** he may have hit her
- **b.** she had hit him
- **c.** she would hit him
- **d.** I was resting
- **e.** you should have been working

2. Suppose you amplify (5.4) by introducing the productions:

$$Asp_2 \rightarrow V_{\texttt{prog}}\texttt{Part}$$
$$V_{\texttt{prog}} \rightarrow \text{be, start, stop, keep}$$

Show that the following are sentences:

- **a.** he has stopped hitting her
- **b.** she kept calling him

3. Construct the underlying parsing trees of the following:

- **a.** they left her poor
- **b.** I teach it to her
- **c.** she could have shown me it

5.3 More about the noun phrase

The only noun phrases we had admitted in Section 5.1 were names and pronouns. This is of course inadequate: we must also consider noun phrases built up from nouns. The problem is that some nouns must and other nouns may be preceded by an article or another determiner.

Aside from pronouns, and names, there are in fact several different kinds of nouns:

1. Count nouns ($= N_{\texttt{count}}$) such as man, cat, ball, etc.

   They require an article or another determiner and admit plural formation.

2. Plurals ($= N_{\texttt{plur}}$) may be plurals of count nouns, e.g. men, cats, balls, etc., or may not possess a singular at all, e.g. scissors. They admit the definite article the, but not the indefinite article a(n).

3. Substance nouns ($= N_{\texttt{subst}}$) such as milk, soap, bread, etc. They admit the definite but not the indefinite article.

If a short digression is permitted, we shall make some remarks of a historical and philosophical nature on the difference between count nouns and names of substances.

One can say one man, two cats, three balls, etc., but one must say one glass of milk, two cakes of soap, three loaves of bread, etc. Some similar linguistic phenomenon in ancient Greek presumably induced Aristotle to develop his theory of substance and form. The idea is simply this: when talking about a loaf of bread, bread denotes the substance and loaf the form. Other ancient philosophers were led to ask: how many substances are there in nature? The traditional answer was: four, namely, earth, water, air and fire; but each of these had its champions who proclaimed it to be the only substance. Then there was the theory that substance=mat(t)er is female, while the form is male, but this takes us too far afield.

Whether something is denoted by a count noun or is regarded as a substance is often a question of size. The English language decrees that beans can be counted, but that rice cannot, unless in the form of grains. At one time peas were regarded too small for counting and there was an old substances noun pease, as in pease porridge. However, later peas(e) was regarded as the plural of the newly coined word pea. Perhaps some day rice will be considered as the plural of rouse.

Articles are not the only kind of determiners. In fact we shall distinguish several types of determiners:

$D_{\texttt{count}}$ = determiners which may be used for count nouns
$D_{\texttt{subst}}$ = determiners which may be used for substance nouns
$D_{\texttt{plur}}$ = plural determiners, which may be used for plurals
$D_{\texttt{sing}}$ = singular determiners, which may be used for count and substance nouns
$D_{\texttt{univ}}$ = universal determiners, which may be used for count nouns, substance nouns and plurals.

Rule (3) of Section 5.1 should now be supplemented by the following new productions:

(3.1) $NP_2 \longrightarrow D_{\texttt{plur}}\ N_{\texttt{plur}}$

(3.2) $NP_3 \longrightarrow D_{\texttt{count}}\ N_{\texttt{count}},\ D_{\texttt{subst}}\ N_{\texttt{subst}},$ somebody, something ...

(3.3) $D_{\texttt{plur}} \longrightarrow D_{\texttt{univ}},$ these, those, 1

(3.4) $D_{\texttt{univ}} \longrightarrow$ the, my, his, her, your, ..., some, no, ...

(3.5) $D_{\texttt{count}} \longrightarrow D_{\texttt{sing}},$ a(n), one, each, every

(3.6) $D_{\texttt{sing}} \longrightarrow D_{\texttt{univ}},$ this, that

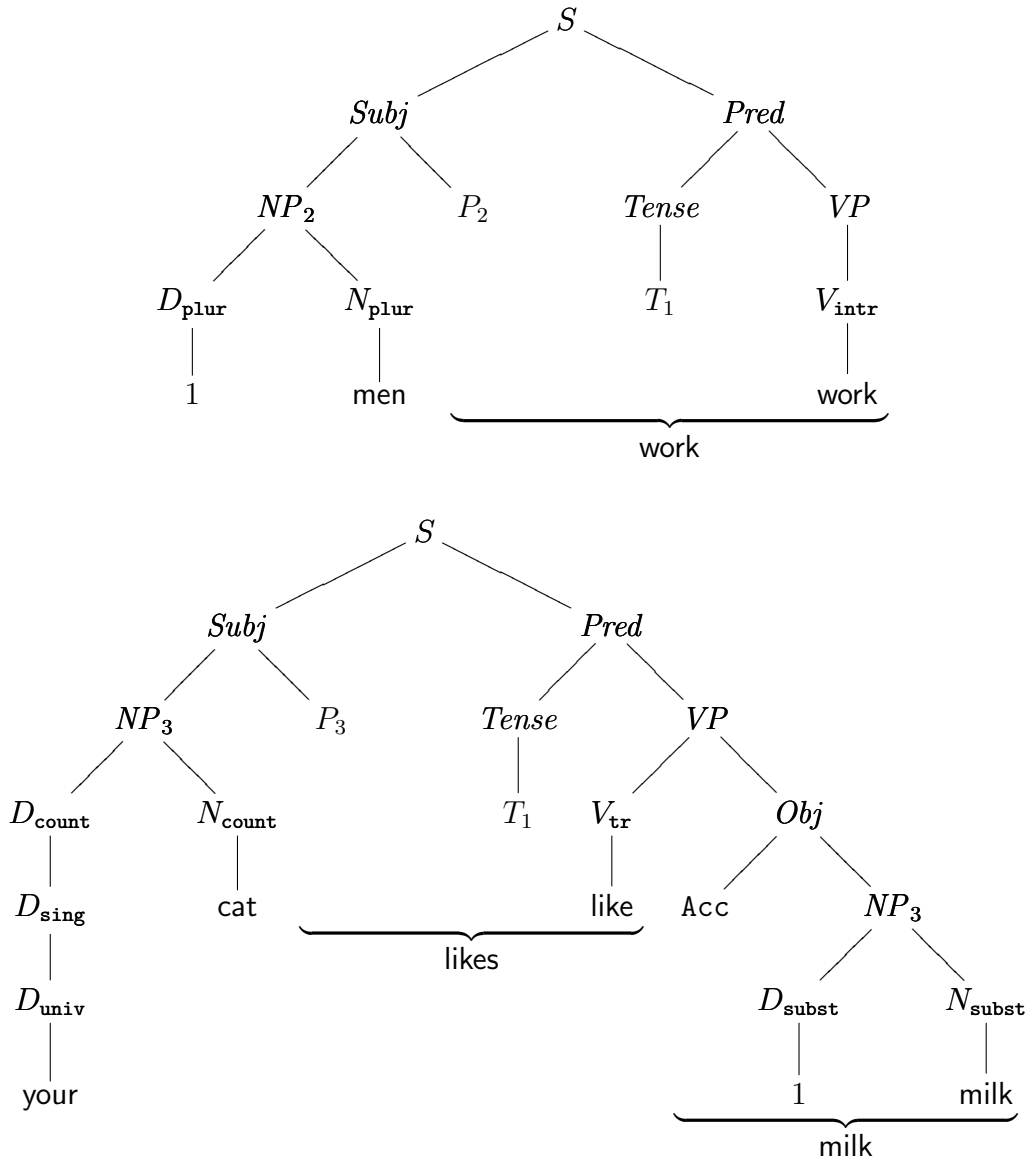
(3.7) $D_{\texttt{subst}} \longrightarrow D_{\texttt{sing}}, 1$

(3.5) must be supplemented by the following rule:

(3.8) a(n) $X \longrightarrow \begin{cases} \text{an } X & \text{if } X \text{ starts with a vowel} \\ \text{a } X & \text{if } X \text{ starts with a consonant} \end{cases}$

However, (3.8) refers to the pronunciation of X and not its spelling.

In the following examples, we make use of the grammar of Section 5.1, but not of the grammar of Section 5.2.



While we have distinguished between substance nouns and count nouns, we have not distinguished between masculine and feminine, animate and inanimate, etc. Thus we cannot rule out questionable sentences such as milk likes your cat.

There are many other questions that we have left open. Where do the numerals and the word all belong? What about your two cats, all her cats, etc.?

Exercises

1. Utilizing the grammar of Sections 5.1 and 5.3, construct the underlying parsing trees for the following sentences and show how the terminal strings are derived:

a. each man works

b. the man hit my cat

2. Utilizing also the grammar of Section 5.2, do the same for

a. your cat was resting

3. What additional productions are required to generate the following sentences:

a. this is good

b. I like that

5.4 The passive construction and transformational grammar

We shall discuss the English passive construction, first within generative grammar, and then from the point of view of transformational grammar, it being our intention to introduce the latter concept in this section.

Everyone knows that the sentences

$$\text{the man} \begin{cases} \text{takes} \\ \text{took} \\ \text{will take} \end{cases} \text{the ball}$$

give rise to the passive forms

$$\text{the ball} \begin{cases} \text{is taken} \\ \text{was taken} \\ \text{will be taken} \end{cases} \text{by the man}$$

the propositional phrase by the man being optional. It is clear from these examples that the passive of a sentence depends on its structure, that is, on its underlying parsing tree.

(1) $$S \longrightarrow NP_i \; P_i \; \textit{Tense} \; V_{\texttt{tr}} \; \texttt{Acc} \; NP_j$$

where i, $j = 1, 2, 3$, we may form quite generally

(2) $$S \longrightarrow NP_j \; P_j \; \textit{Tense} \; \text{be} \; \texttt{Perf} \; V_{\texttt{tr}} \; (\text{by} \; \texttt{Acc} \; NP_i)$$

Terminal strings derived from (2) may be regarded as passives of terminal strings derived from (1). From the viewpoint of generative grammar, we are not interested in the relationship between these two terminal strings, but only in the grammaticalness of the former.

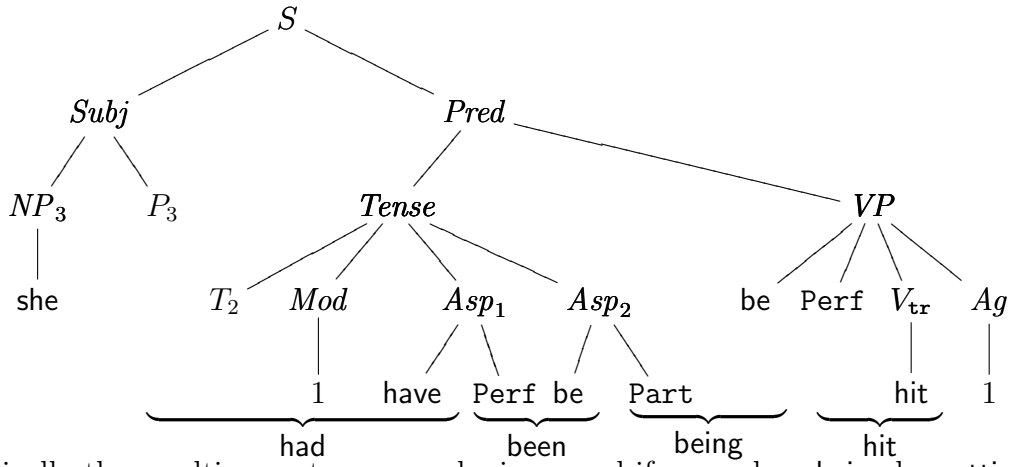
The easiest way to assure that (2) and all consequences of (2) are theorems in the deductive system $\mathcal{D}(\mathcal{G})$ is to postulate the following new productions, augmenting rule (6) of Section 5.2:

$$(6.5)\quad \begin{array}{l} VP \rightarrow \textsf{be}\ \texttt{Perf}\ V_{\texttt{tr}}\ Ag \\ Ag \rightarrow 1,\ \textsf{by}\ Obj \end{array}$$

where

$$Ag = \text{agent}$$

Here is an example of a passive construction based on (6.5):



(Stylistically the resulting sentence may be improved if we replace being by getting, as in Exercise 2 at the end of this section.)

While (6.5) only deals with transitive verbs, verbs like make and give also admit passives, for example

he was rendered poor by her

she is shown it

it is shown to her

We are therefore led to supplement (6.5) as follows (see Section 5.2 for the verbs involved):

$$(6.6)\quad \begin{array}{l} VP \rightarrow \textsf{be}\ \texttt{Perf}\ V_{\texttt{make}}\ Adj\ Ag \\ \quad \rightarrow \textsf{be}\ \ \texttt{Perf}\ V_{\texttt{give}}\ Obj\ Ag \\ \quad \rightarrow \textsf{be}\ \ \texttt{Perf}\ V_{\texttt{give}}\ \textsf{to}\ Obj\ Ag \end{array}$$

One could go on in this way, but perhaps a more radical approach is called for, which will account for all these rules at once. Here is one suggestion:

$$(6.5*)\quad \begin{array}{l} VP \quad \rightarrow \texttt{Pass}\ VP \\ \texttt{Pass}\ V \rightarrow\ \textsf{be}\ \texttt{Perf}\ V\ \Psi \\ \Psi\ Obj\ \rightarrow 1 \end{array}$$

Here `Pass` = passive, Ψ is an auxiliary symbol, and V is assumed to be any verb which possesses a passive. This last proviso could be used to prevent us from forming the passive of the following sentences:

the play lasts three hours

this colour becomes you

he went the whole way

This approach could be extended to account for sentences such as

a new idea is called for

but we shall not pursue it any further.

We have discussed the construction of passive sentences within the framework of generative grammar. However, this approach does not explain what sentence a passive sentence is the passive of. It does not explain why it is that in

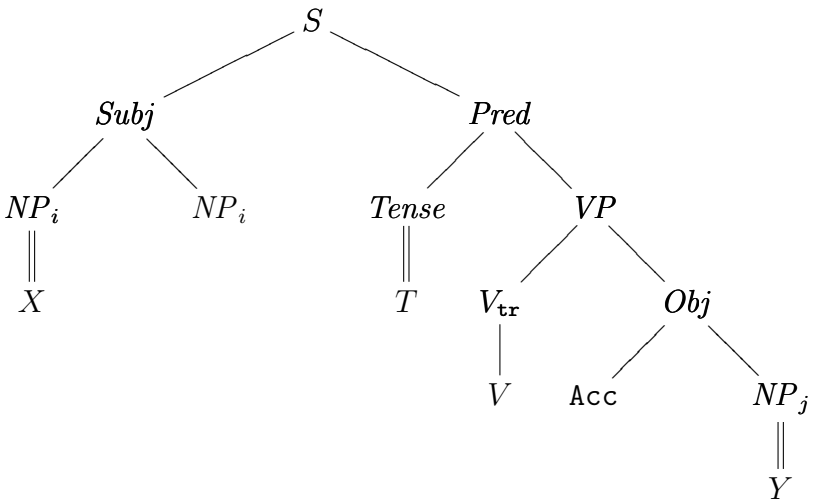
$$Y \text{ is eaten by } X$$

X normally denotes an animal or a person just as in

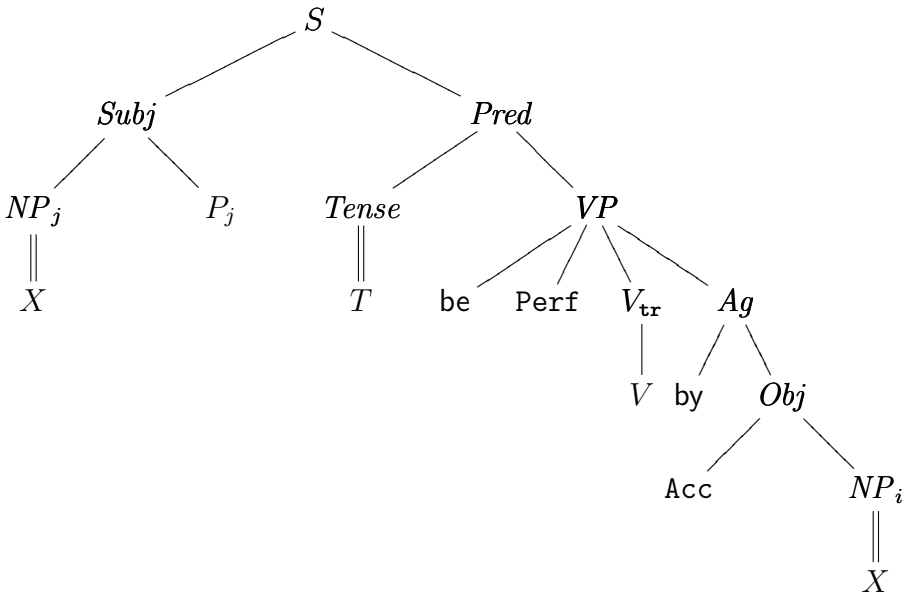
$$X \text{ eats } Y$$

It does not explain why in both sentences Y may be soup in Canada and tea in England.

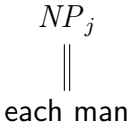
For these and other reasons, Chomsky has attempted to view the passive construction as a "transformation". According to this view, a transformation transforms one parsing tree into another. Thus, the passive transformation corresponding to (6.5) would transform the parsing tree



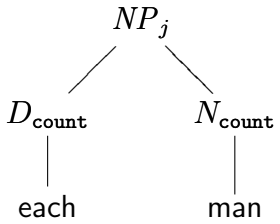
into the parsing tree



Here X and Y are terminal strings, V is a terminal word and T is a string made up of terminal words and inflections (T_k, `Perf` and `Part`). The branches to those strings are really abbreviations. Thus



stands for



How can we interpret this process in more familiar mathematical terms? One way, which is close to Chomsky's original view, would be to regard it as a rule of inference as follows:

$$\frac{S \longrightarrow X\ P_i\ T\ V\ \texttt{Acc}\ Y,\ \ NP_i \longrightarrow X,\ \ Tense \longrightarrow T,\ \ V_{\texttt{tr}} \longrightarrow V,\ \ NP_j \longrightarrow Y}{S \longrightarrow Y\ P_j\ T\ \texttt{be}\ \texttt{Perf}\ V\ (\texttt{by}\ \texttt{Acc}\ X)}$$

A **rule of inference** in a deductive system is a rule that says that if you have derived a certain set of strings, then you may derive another string. The rules (S) and (T) of Section 4.2 are rules of inference.

In the present instance the new rule says that if there are productions that derive the string $X\ P_i\ T\ V$ `Acc` Y from S, the string X from NP_i, the string T from $Tense$, the string $V_{\texttt{tr}} \longrightarrow V$ from V (which means that V is a transitive verb) and the string Y from NP_j, then derive the string $Y\ P_j\ T$ be `Perf` V (by `Acc` X) from S.

Recall that a production $A \longrightarrow B$ of the generative grammar $\mathcal{G}$ was regarded as an axiom in the deductive system $\mathcal{D}(\mathcal{G})$ in Chapter 4. We may now define a **transformational grammar** $\mathcal{G}'$ as stipulating not only axioms (productions) but also rules of inference (= grammatical transformations) for a deductive system $\mathcal{D}(\mathcal{G}')$. We replaced (6.5) by a rule of inference, and we can do the same for each of the productions (6.6). Once is tempted to combine all the resulting rules of inference into one, as follows:

$$\frac{S \longrightarrow X\ P_i\ T\ V\ \texttt{Acc}\ Y\ Z,\ \ NP_i \longrightarrow X,\ \ Tense \longrightarrow T,\ \ V_{\texttt{pass}} \longrightarrow V,\ \ NP_j \longrightarrow Y}{S \longrightarrow Y\ P_j\ T\ \text{be}\ \texttt{Perf}\ V\ Z\ (\text{by}\ \texttt{Acc}\ X)}$$

where

$$V_{pass} = \text{verb possessing a passive}$$

There are however some difficulties concerning Z, the structure of which should be specified in some way. We do not want to transform

the cat chased John and me

into

*John was chased and me

or

I saw the father of my friend

into

*the father was seen of my friend

Apparently it is not easy to formulate the correct restrictions on Z, and so perhaps the pedestrian approach of listing all the rules separately is to be preferred.

Here is another approach to the generation of passives. From such examples as

I found the door closed

there is a well motivated need for a rule of the form

$$Adj \longrightarrow \texttt{Perf}\ VP^-$$

where VP^- stands for a verb phrase with its direct object (or its first object if it takes more than one) deleted. Assuming such a rule then we have productions such as

$$\begin{aligned} S &\longrightarrow NP\ V_{\text{cop}} Adj \\ &\longrightarrow Art\ N\ V_{\text{cop}}\ \texttt{Perf}\ VP^- \\ &\longrightarrow \text{the book was given to Mary} \end{aligned}$$

where since we have a production

$$VP \longrightarrow \text{give book to Mary}$$

then we have a production

$$VP^{-} \longrightarrow \text{book to Mary}$$

If we apply the same production to

$$VP \longrightarrow \text{give Mary a book}$$

we derive instead the sentence

Mary was given a book

This proposal is also most simply formulated as a rule of inference. We leave the details to the reader. The point of this proposal is that it does not add to language any rule that is not required in any case. It derives the passive as a special case of two rules; the one that gives predicative adjective sentences and the one that turns a past participle into an adjective. The agent phrase, if any, is derived by the rule that says

$$S \longrightarrow S\ PP$$

where PP stand for prepositional phrase. This explains the optional nature of the agent clause.

Commentary by MB

I am not entirely convinced by this analysis of the passive. First, it is well motivated that English has a rule of the form $Adj \longrightarrow Part$ since so many participles function as pure adjectives (however notice the contrast between, "The door was open" and "The door was opened"; but there are not two words in the case of "closed"). At any rate, it is not necessary to posit a passive transformation since a sentence such as the door was closed by him is generated by the production

$$S \longrightarrow S\ PP \longrightarrow NP\ VP\ PP \longrightarrow Art\ N\ V_{\text{cop}}\ Adj\ PrepPron$$

$$\longrightarrow \text{the door was closed by him}$$

It would be harder, however, to fit the negative and interrogative into the same mold as the declarative. And many languages would have to be studied to determine if this kind of analysis of the passive is general. It is interesting to note that this passive "transformation" was one of the main motivations for transformational grammar, but has now been largely abandoned. Another point in favor of the transformational approach is that it gives a ready explanation for the semantic relation of the active and passive modes, which my account does not. The jury is still out.

Exercises

1. Using (6.5), exhibit the underlying parsing trees of the following sentences:

the ball may have been taken by her

she is being hit by the ball

2. Discuss a modification of (6.5) which will allow two passive auxiliaries

$$V_{\mathsf{pass}} = \mathsf{be},\ \mathsf{get}$$

the latter as in she got hit.

3. Show that the proposal of the last paragraphs allows even more passives, such as

the door stayed closed

the book seemed damaged by the rain

The last is perhaps marginal, but not obviously bad.

5.5 Some other transformations

English abounds with constructions that are usually viewed as grammatical transformations. We shall content ourselves with a few examples. However, we shall not insist on treating a construction by the methods of transformational grammar if generative grammar will do.

5.5.1 Example The **negative** transformation assigns to any English sentence its "grammatical" negation. This should be distinguished from the logical negation, with which it usually coincides, but not always. For example, the grammatical negation of

John must work

is

John must not work

while its logical negation is

John need not work

We find it most convenient to formulate the negative transformation as a set of productions. Writing

$$\mathtt{Neg} = \text{negative}$$

we postulate

(11) $Mod \rightarrow$ Neg Mod
(12) Neg have Perf $\rightarrow$ have not Perf
(13) Neg have $Obj \rightarrow$ do not have Obj
(14) Neg $V \rightarrow V$ not, if $V =$ be, must, may, can, shall, will
(15) Neg $V \rightarrow$ do not V, if $V \neq$ have, be, must, ...

These productions are supplemented by certain optional **contraction** rules:

(16)
is not $\rightarrow$ isn't
was not $\rightarrow$ wasn't
can not $\rightarrow$ cannot, can't
⋮

which also involve transfer of stress.

Note that Neg have cannot be evaluated until we know whether have is used as an auxiliary or as a transitive verb. Admittedly, our rules do not explain

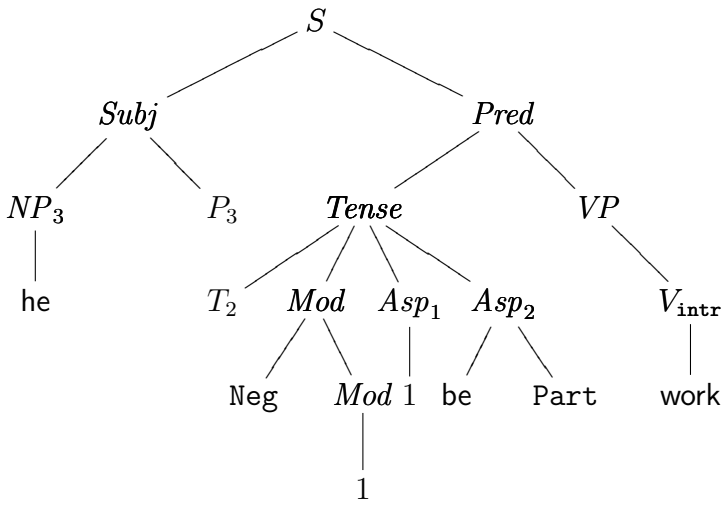
I haven't the time

she hasn't any wool

but it would be difficult to admit these sentences without also admitting

*the cat hasn't the ball

For example, in view of (11), we have the parsing tree



This yields the pseudo-sentence

he P_3 T_2 Neg be Part work

In view of (14), this becomes

$$\text{he } P_3\ T_2 \text{ be not Part work}$$

By the usual inflection rules, this reduces to

$$\text{he was not working}$$

An optional contraction rule (16) converts this to

$$\text{he wasn't working}$$

5.5.2 EXAMPLE The **emphatic** transformation is similar to the negative transformation. In place of Neg it has Emph and in place of not it has stress, to be placed on the preceding syllable. The stress is not shown in written English (but see the last problem of this section).

5.5.3 EXAMPLE The **interrogative** transformation forms a question from a statement, asking if the statement is true. For technical reasons, we begin by negating the statement and express the interrogative transformation as a rule of inference:

$$\frac{S \longrightarrow X\ C_{ki}(V)\ \text{not}\ Y,\ NP_i \longrightarrow X,\ V_x \longrightarrow V}{Q \longrightarrow C_{ki}(V)\ X\ Y?}$$

Here V_x stands for one of the various classes of verbs and Q = question. The latter is a sentence type that ought to be distinguished from S, the type of statements; as, for instance, one cannot put and between a question and a statement. The rule above refers to written English and ignores the rising intonation.

As an illustration, let us look at the pseudo-sentence

$$\text{he } P_3\ T_2 \text{ be not Part work}$$

that was discussed in Example 5.5.1. Recalling that

$$P_i\ T_k\ V \longrightarrow C_{ki}(V)$$

we obtain

$$S \longrightarrow \text{he } C_{23}(\text{be}) \text{ not working}$$

Since we have

$$NP_3 \longrightarrow \text{be}, \quad V_{\text{cop}} \longrightarrow \text{be}$$

the proposed rule of inference will allow us to deduce

$$Q \longrightarrow C_{23}(\text{be}) \text{ he working?}$$

that is,

$$Q \longrightarrow \text{was he working?}$$

Our rule explains the question

do you have the time?

but not

have you the time?

If it is desirable to include the second question as grammatical, the rule could be modified.

In case one of the rules (16) allows a contraction

$$C_{ki}(V) \text{ not} \longrightarrow C_{ki}(V)\text{n't}$$

our rule should be supplemented by the following

$$\frac{S \longrightarrow X\ C_{ki}(V)\text{n't}\ Y,\ NP_i \longrightarrow X,\ V_x \longrightarrow V}{Q \longrightarrow C_{ki}(V)\text{n't}\ X\ Y?}$$

This yields, for instance,

$$Q \longrightarrow \text{wasn't he working?}$$

5.5.4 Example Wh-question. Given a sentence containing he (or she) one may form a question by changing the pronoun to who and moving it to the beginning of the sentence, unless it was there already. However, if who is moved, the interrogative transformation of Example 5.5.3 must be applied. For example, from (the underlying parsing tree of)

he works

one obtains

who works?

and from

this is he

one obtains

who is this?

and from

you say he drinks

one obtains

who do you say drinks?

We may try to summarize these facts with the help of two rules of inference:

(Wh–1) $$\frac{S \longrightarrow \textsf{he } P_3\ X,\ \mathit{Pred} \longrightarrow X}{Q \longrightarrow \textsf{who } P_3\ X?}$$

(Wh–2) $$\frac{S \longrightarrow X\ C_{ki}(V)\ \textsf{not}\ Y\ \textsf{he}\ Z,\ NP_i \longrightarrow X,\ V_x \longrightarrow V}{Q \longrightarrow \textsf{who } C_{ki}(V)\ X\ Y\ Z}$$

(As in Example 5.5.3, it is convenient to start with the negative.)

There are some difficulties with (Wh–2), unless the structures of Y and Z are specified. For example, (Wh–2) would allow us to infer from the negative sentence

$$\begin{array}{cccc} \textsf{I} & \textsf{do} & \textsf{not } \underbrace{\textsf{know what}}_{Y} & \textsf{he } \underbrace{\textsf{will say}}_{Z} \\ X\ C_{11}(V) & & & \end{array}$$

the question

*who do I know what will say?

and from the negative sentence

this is not John and I

the question

*who is this John and?

Supposing these difficulties can be overcome (and various suggestions have been made for doing so), one may treat in a similar way questions starting with whom, what, where, etc., which are obtained by replacement of him (or her), it, here, etc., respectively.

Exercises

1. Exhibit the underlying parsing trees of the following sentences:

I cannot call you
she does not work
he has not been resting
you were not called by me
she is not clever

2. Show how the interrogative transformation may be used to obtain the following questions:

can I call you?
does she work?
has he been resting?
does he have the ball?
were you called by me?
is she clever?

3. Examine the possibility of formulating the rules of emphasis and contraction in such a way that the contraction rule obligatory—unless its operation is blocked by the not being emphasized. There is evidence in this suggestion in that whenever contraction is not applied, the not *is* emphasized in normal speech.

References

Y. Bar-Hillel, M. Perles and E. Shamir, On formal properties of simple phrase structure grammars. Z. Phonetik Sprachwiss, Kommunikat. **14** (1961), 143–172. Reprinted in Luce *et al.*, Vol. 3.

L. S. Bobrow and M. A. Arbib, Discrete Mathematics. Saunders, Philadelphia, London, Toronto, 1974.

G. S. Boolos and R. C. Jeffrey, Computability and Logic. Cambridge University Press, London, New York, 1974.

S. Brainerd, Introduction to the Mathematics of Language Study. American Elsevier, New York, 1971.

N. Chomsky, Syntactic Structures. Mouton, 'S-Gravenhage, 1957.

N. Chomsky, On the formal properties of grammars. Information and control, **2** (1959), 137–167.

N. Chomsky, Aspects of the Theory of Syntax. M.I.T. Press, Cambridge, MA, 1964.

N. Chomsky, Language and Mind. Harcourt, Brace, Jovanovich, New York, San Francisco, Atlanta, 1972.

P. M. Cohn, Algebra and language theory. Bull. London Math. Soc. **7** (1975), 1–29.

M. Davis, Computability and Unsolvability. McGraw-Hill, New York, Toronto, London, 1958.

S. Eilenberg, Automata, Languages and Machines, Vol. A. Academic Press, New York, 1974.

E. Engeler, Introduction to the Theory of Computation. Academic Press, New York, London, 1973.

R. Fowler, An Introduction to Transformational Syntax. Routledge and Kegan Paul, London, 1971.

S. Ginsburg, The mathematical Theory of Context-Free Languages. McGraw-Hill, New York, 1966.

A Ginzburg, Algebraic Theory of Automata. Academic Press, New York, London, 1968.

M. Gross, Mathematical Models in Linguistics. Prentice Hall, Englewood Cliffs, NJ, 1972.

M. Gross and A. Lentin, Introduction to Formal Grammars. Springer-Verlag, Berlin, Heidelberg, New York, 1970.

Z. Harris, Mathematical Structures of Language. Interscience, Wiley, New York, London, Sydney, Toronto, 1968.

J. E. Hopcroft and J. D. Ullman, Formal Languages and their Relation to Automata. Addison-Wesley, Reading, MA, 1969.

S. C. Kleene, Introduction to Metamathematics. Van Nostrand, New York, 1952

S. C. Kleene, Representations of events in nerve nets. In Shannon, 3–40.

R. R. Korfhage, Logic and Algorithms. Wiley, New York, London, Sydney, 1966.

J. Lambek, How to program an infinite abacus? Canad. Math. Bull. **4** (1961), 295–302

J. Lambek, The mathematics of Sentence structure. Amer. Math. Monthly, **65** (1958), 154–170.

J. Lambek, Contributions to the mathematical analysis of the English verb phrase. J. Canad. Linguistic Assoc. **5** (1959), 83–89.

J. Lambek, On the calculus of syntactic types. Proc. Symposium Appl. Math. **12** (1961), 166–178, Amer. Math. Soc., Providence, RI.

J. Lambek, A mathematician looks at French conjugation. Theoretical Linguistics, **2** (1975), 203–214

D. Luce, E. Bush and E. Galanter, eds., Handbook of mathematical psychology, 3 Vols. Wiley, New York, London, 1963, 1965.

J. Lyons, Introduction to Theoretical Linguistics. Cambridge University Press, London, New York, 1968.

A. I. Mal'cev, Algorithms and recursive functions. Wolters-Noordhoff, Groningen, 1970.

S. Marcus, Algebraic Linguistics, Analytic Models. Academic Press, New York, London, 1967.

Z. A. Melzak, An informal introduction to computability and computation. Canad. Math. Bull. **4** (1961), 279–293.

F. Palmer, Grammar. Penguin Books, Harmondsworth, Middlesex, 1971.

P. C. Rosenbloom, The Elements of Mathematical Logic. Dover Publications, New York, 1950.

D. Scott, Some definitional suggestions for automata theory. J. Computer Systems Sciences **1** (1967), 187–212.

C. E. Shannon and J. McCarthy, eds., Automata Studies. Princeton University Press, Princeton NJ, 1956.

A. M. Turing, On computable numbers with an application to the Entscheidungsproblem. J. London Math. Soc. (Series 2), **42** (1933-37), 230–265, **43** (1937), 544–546.

V. H. Yngve, The depth hypothesis. Amer. Math. Soc., Proc. Symposium in Appl. Math. **12** (1961), 130–138, Amer. Math. Soc., Providence, RI.

Appendix: A mathematician looks at the French verb

J. Lambek

6 Introduction

The penalty one has to pay for the enjoyment of French culture is that one must face the problem of conjugating the French verb. The following considerations are offered to the mathematically included reader in the hope that they may help him to bypass some of the struggles the author went through in trying to cope with this problem. However, it is only fair to point out that the author still does not manage use the "simple" past and the two subjunctives without the help of his pocket calculator.

The problem is briefly this: even forgetting about imperatives and participles, one finds associated with French verb V an array of 42 forms $C_{ik}(V)$, where i ranges from 1 to 7 and k from 1 to 6. Here i denotes the "simple tenses": present, imperfect, future, conditional, present subjunctive, simple past and past subjunctive. No purpose is served for the moment in subdividing these into two "moods": indicative and subjunctive. The subscript k represents the six "persons" associated with the pronouns: je, tu, il, (or elle), nous, vous, ils (or elles). Again, we have chosen not to partition these persons into two "numbers": singular and plural.

How is the matrix $C(V)$ constructed? It would be too bad if one had to learn 42 forms for each verb V. In fact, a superficial investigation shows that the matrix $C(V)$ depends only on 5 parameters, which may be restricted to 3 on closer examination. Moreover, for most verbs in the dictionary and for all newly formed verbs, these parameters are predictable from the ending of the infinite. It turns out that the conjugation matrix $C(V)$ may be obtained by rewriting rules from a product $A(V)B$ where $A(V)$ is a 7 by 3 matrix depending on the verb V, while B is a 3 by 6 matrix not depending on the verb. It so happens that B has only 6 nonempty entries, one in each column. Therefore, multiplication of the matrices $A(V)$ and B is possible, even though French is only a semigroup and not a ring.

To make this article accessible to as many readers as possible, the present investigation will be concerned with written french. However, one concession has been made to spoken French: stressed vowels or diphthongs have been underlined, as an understanding of stress makes it easier to see what is going on. It has been assumed that each verb form has exactly one stressed vowel or diphthong. Regrettably it has not been found convenient to distinguish between ai pronounced è and ai pronounced é.

A very much simpler treatment could have been achieved by ignoring 10 troublesome "exceptional" verbs:

aller, avoir, falloir, pouvoir, savoir

valoi1r, vouloir, être, faire, dire

However, these verbs occur so frequently that it was decided to include them in our considerations.

7 The present tense

Surprisingly, about half the effort in leaning French conjugation is required for mastering the present tense. We observe that each verb form in the present tense is made up of two components: stem plus ending. We write x for the stem and provisionally let the number $i = 1, 2, \ldots, 6$ stand for the ending. The actual nature of the ending depends on the grapheme preceding it.

How the actual ending is obtained is shown by the following six rewriting rules:

(1)
$$X1 \longrightarrow \begin{cases} X\text{i} & \text{if } X \text{ ends in a} \\ X & \text{if } X \text{ ends in e} \\ X\text{x} & \text{if } X \text{ ends in au or eu} \\ X\text{s} & \text{otherwise} \end{cases}$$

Examples: j'ai, j'aime, je peux, je finis.

(2)
$$X2 \longrightarrow \begin{cases} X & \text{if } X \text{ ends in s} \\ X\text{x} & \text{if } X \text{ ends in au or eu} \\ X\text{s} & \text{otherwise} \end{cases}$$

Examples: tu es, tu vaux, tu aimes.

(3)
$$X3 \longrightarrow \begin{cases} X & \text{if } X \text{ ends in a, e, c,d, or t} \\ X\text{t} & \text{otherwise} \end{cases}$$

Examples: il a, il aime, il vainc, il vend, il bat, il finis.

(4)
$$X4 \longrightarrow \begin{cases} X\text{mes} & \text{if } X \text{ is stressed} \\ X\text{ons} & \text{if } X \text{ is unstressed} \end{cases}$$

Examples: nous s̲ommes, nous aimo̲ns.

(5) $$X5 \longrightarrow \begin{cases} X\text{tes} & \text{if } X \text{ is stressed} \\ X\underline{\text{e}}\text{z} & \text{if } X \text{ is unstressed} \end{cases}$$

Examples: nous fa̲ites, nous aime̲z.

(6) $$X6 \longrightarrow \begin{cases} X\text{nt} & \text{if } X \text{ ends in e or } \underline{\text{o}} \\ X\text{ent} & \text{otherwise} \end{cases}$$

Examples: ils a̲iment, ils o̲nt, ils fini̲ssent.

We hasten to point out that these same rewriting rules will be used in other tenses later. For instance, while fa̲ites might be discounted as an exception, this method of forming the fifth person ending will become the rule in the simple past.

The partition into stem plus ending often allows some leeway and may be based on an arbitrary decision. For example, we have analyzed a̲iment as a̲ime+nt rather than a̲im+ent, but we could equally well have adopted the second alternative. Rule (6) yields a̲iment in either case.

In general, there are three present stems. Consider, for example, the present stem of devo̲ir: je do̲is, tu do̲is, il do̲it, nous devo̲ns, vous deve̲z, ils do̲ivent. To all appearances, the first three persons are derived from do̲i, the fourth and fifth persons from the stem dev and the sixth person from the stem do̲iv. This is the general situation, although in many cases two of these stems coincide and some exceptional verbs such as ê̲tre appear to have more than three stems in the present tense.

We shall denote the three present stems of the verb V as follows:

$$\underline{\alpha}(V), \qquad \beta(V), \qquad \underline{\gamma}(V)$$

where the underlining again indicates stress. The full present tense is then normally obtained by the rewriting rules (1) to (6):

$$\underline{\alpha}(V)1, \quad \underline{\alpha}(V)2, \quad \underline{\alpha}(V)3, \quad \beta(V)4, \quad \beta(V)5, \quad \underline{\gamma}(V)6$$

For example,

$$\underline{\alpha}(dev\underline{oi}r) \longrightarrow d\underline{oi}$$

and

$$\underline{\alpha}(\text{devo̲ir})1 \longrightarrow \text{do̲i}1 \longrightarrow \text{do̲is}$$

The present stems of many French verbs may be read off from Tables I and II which will be found in Sections 10 and 11, respectively. These tables also contain some other data that will be discussed later.

8 The other tenses

Where the present tense depends on the stems

$$\underline{\alpha}(V), \qquad \beta(V), \qquad \underline{\gamma}(V)$$

the imperfect tense uses instead the stems

$$\beta(V)\underline{\mathsf{ai}}, \qquad \beta(V)\mathsf{i}, \qquad \beta(V)\underline{\mathsf{ai}}$$

In other words, the full imperfect tense is obtained by rules (1) to (6) from

$$\beta(V)\underline{\mathsf{ai}}1, \quad \beta(V)\underline{\mathsf{ai}}2, \quad \beta(V)\underline{\mathsf{ai}}3, \quad \beta(V)\mathsf{i}4, \quad \beta(V)\mathsf{i}5, \quad \beta(V)\underline{\mathsf{ai}}6,$$

Thus the imperfect reads:

$$\beta(V)\underline{\mathsf{ai}}\mathsf{s}, \quad \beta(V)\underline{\mathsf{ai}}\mathsf{s}, \quad \beta(V)\underline{\mathsf{ai}}\mathsf{t}, \quad \beta(V)\mathsf{i}\underline{\mathsf{o}}\mathsf{ns}, \quad \beta(V)\mathsf{i}\underline{\mathsf{e}}\mathsf{z}, \quad \beta(V)\underline{\mathsf{ai}}\mathsf{ent}$$

The future and the conditional have the stems

$$\phi(V)\mathsf{r}\underline{\mathsf{a}}, \qquad \phi(V)\mathsf{r} \qquad \phi(V)\mathsf{r}\underline{\mathsf{o}}$$

and

$$\phi(V)\mathsf{r}\underline{\mathsf{ai}}, \qquad \phi(V)\mathsf{ri} \qquad \phi(V)\mathsf{r}\underline{\mathsf{ai}}$$

respectively. Here $\phi(V)$ is a new morpheme, which is closely related to the infinitive V. The reader will observe that rule (1) yields

$$\phi(V)\mathsf{r}\underline{\mathsf{a}}1 \longrightarrow \phi(V)\mathsf{r}\underline{\mathsf{ai}}$$

in the future, but

$$\phi(V)\mathsf{r}\underline{\mathsf{ai}}1 \longrightarrow \phi(V)\mathsf{r}\underline{\mathsf{ai}}\mathsf{s}$$

for the conditional.

The present subjunctive makes use of the stems

$$\underline{\gamma}'(V)\epsilon, \qquad \beta'(V)\iota, \qquad \gamma'(V)\epsilon$$

Here $\underline{\gamma}'(V) = \underline{\gamma}(V)$ and $\beta'(V) = \beta(V)$ for all but nine of the ten exceptional verbs. Moreover, the Greek letters ϵ and ι are subject to the following rewriting rules:

(ι) $$X\iota \longrightarrow \begin{cases} X\mathsf{y} & \text{if } X \text{ ends in } \mathsf{a} \text{ or } \mathsf{o} \\ X\mathsf{i} & \text{otherwise} \end{cases}$$

Examples: que nous ay$\underline{\mathsf{o}}$ns, que nous soy$\underline{\mathsf{o}}$ns, que vous aimi$\underline{\mathsf{e}}$z, que nous voyi$\underline{\mathsf{o}}$ns

(ϵ) $$X\epsilon \longrightarrow \begin{cases} X & \text{if } X \text{ ends in } \mathsf{e} \\ X\underline{\mathsf{i}} & \text{if } X \text{ ends in } \underline{\mathsf{a}} \text{ or } \underline{\mathsf{o}} \\ X\mathsf{e} & \text{otherwise} \end{cases}$$

Examples: que tu aimes, qu'il ait, qu'il soit, que je meure

The simple past has the stems

$$\underline{\psi}(V), \qquad \underline{\psi}(V)\hat{}, \qquad \underline{\psi}(V)r$$

where $\underline{\psi}(V)$ is another morpheme, one that is rapidly disappearing in spoken French. Moreover the circumflex is subject to the following rewrite rule

$$X\hat{} \longrightarrow \begin{cases} X & \text{if the stressed vowel of } X \text{ has a circumflex} \\ \hat{X} & \text{with the circumflex placed on the stressed} \\ & \text{vowel otherwise} \end{cases}$$

Examples: nous aimâmes, vous vîntes, nous crûmes (from croître as well as croire).

Finally, the past subjunctive has six persons

$$\underline{\psi}(V)\mathsf{sse}, \quad \underline{\psi}(V)\mathsf{sses}, \quad \underline{\psi}(V)\hat{}\mathsf{t}, \quad \psi(V)\mathsf{ssions}, \quad \psi(V)\mathsf{ssiez}, \quad \underline{\psi}(V)\mathsf{ssent}$$

where $\psi(V)$ is the same as $\underline{\psi}(V)$, only without stress. With a little bit of cheating, we may reduce these six persons to three stems as follows:

$$\underline{\psi}(V)[\mathsf{sse}], \qquad \psi(V)\mathsf{ssi}, \qquad \underline{\psi}(V)\mathsf{sse}$$

provided [sse] is subject to the following rewriting rules:

$$[\mathsf{sse}]k \longrightarrow \begin{cases} \hat{}k & \text{if } k = 3 \\ \mathsf{sse}k & \text{if } k = 1 \text{ or } 2 \end{cases}$$

Note that we rewrite

$$\mathsf{aima}[\mathsf{sse}]3 \longrightarrow \mathsf{aima}\hat{}3 \longrightarrow \mathsf{aim\hat{a}}3 \longrightarrow \mathsf{aim\hat{a}t}$$

Had there been no circumflex on the â, rule (3) would have eliminated the final t.

9 Factorizing the conjugation matrix

Summarizing our considerations up to this point, we find that the conjugation matrix $C(V)$ may be obtained by various rewriting rules from the following matrix $C'(V)$:

$$\begin{pmatrix} \underline{\alpha}(V)1 & \underline{\alpha}(V)2 & \underline{\alpha}(V)3 & \beta(V)4 & \beta(V)5 & \underline{\gamma}(V)6 \\ \beta(V)\underline{\mathsf{ai}}1 & \beta(V)\underline{\mathsf{ai}}2 & \beta(V)\underline{\mathsf{ai}}3 & \beta(V)\mathsf{i}4 & \beta(V)\mathsf{i}5 & \beta(V)\underline{\mathsf{ai}}6 \\ \phi(V)\mathsf{r}\underline{\mathsf{a}}1 & \phi(V)\mathsf{r}\underline{\mathsf{a}}2 & \phi(V)\mathsf{r}\underline{\mathsf{a}}3 & \phi(V)\mathsf{r}4 & \phi(V)\mathsf{r}5 & \phi(V)\mathsf{r}\underline{\mathsf{o}}6 \\ \phi(V)\mathsf{r}\underline{\mathsf{ai}}1 & \phi(V)\mathsf{r}\underline{\mathsf{ai}}2 & \phi(V)\mathsf{r}\underline{\mathsf{ai}}3 & \phi(V)\mathsf{ri}4 & \phi(V)\mathsf{ri}5 & \phi(V)\mathsf{r}\underline{\mathsf{ai}}6 \\ \underline{\gamma'}(V)\epsilon 1 & \underline{\gamma'}(V)\epsilon 2 & \underline{\gamma'}(V)\epsilon 3 & \beta'(V)\iota 4 & \beta'(V)\iota 5 & \underline{\gamma'}(V)\epsilon 6 \\ \underline{\psi}(V)1 & \underline{\psi}(V)2 & \underline{\psi}(V)3 & \underline{\psi}(V)\hat{}4 & \underline{\psi}(V)\hat{}5 & \underline{\psi}(V)r6 \\ \underline{\psi}(V)[\mathsf{sse}]1 & \underline{\psi}(V)[\mathsf{sse}]2 & \underline{\psi}(V)[\mathsf{sse}]3 & \underline{\psi}(V)\mathsf{ssi}4 & \underline{\psi}(V)\mathsf{ssi}5 & \underline{\psi}(V)\mathsf{sse}6 \end{pmatrix}$$

Thus we have $C'_{ik}(V) \longrightarrow C_{ik}(V)$

To a mathematician it is apparent that the matrix $C'(V)$ may be factored as follows:

$$\begin{pmatrix} \underline{\alpha}(V) & \beta(V) & \underline{\gamma}(V) \\ \beta(V)\underline{\mathsf{ai}} & \beta(V)\mathsf{i} & \beta(V)\underline{\mathsf{ai}} \\ \phi(V)\mathsf{r}\underline{\mathsf{a}} & \phi(V)\mathsf{r} & \phi(V)\mathsf{r}\underline{\mathsf{o}} \\ \phi(V)\mathsf{r}\underline{\mathsf{ai}} & \phi(V)\mathsf{ri} & \phi(V)\mathsf{r}\underline{\mathsf{ai}} \\ \underline{\gamma}'(V)\epsilon & \beta'(V)\iota & \underline{\gamma}'(V)\epsilon \\ \underline{\psi}(V) & \underline{\psi}(V)\hat{} & \underline{\psi}(V)r \\ \underline{\psi}(V)[\mathsf{sse}] & \underline{\psi}(V)\mathsf{ssi} & \underline{\psi}(V)\mathsf{sse} \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & & & \\ & & & 4 & 5 & \\ & & & & & 6 \end{pmatrix}$$

We note that, even though we are in a semigroup and addition has not been defined, it is possible to multiply two matrices if, as in the present case, each row of the first meets each column of the second in at most one non-empty place.

Whether this factorization is indicative of the internal logic of the French language, or whether it is a mere fluke, as linguists would tend to say, there is no harm if the mathematically inclined reader makes use of it as an aid to memory.

Let us write

$$P_k = k\text{th person}$$
$$T_i = i\text{th tense}$$

then the importance of the matrix $C'(V)$ derives from the 42 rewriting rules

$$P_k T_i V \longrightarrow C'_{ik}(V)$$

The significance of these rules will only become apparent when one wishes to construct sentences as in Section 7 below.

A more radical factorization of the matrix $C'(V)$ into a column vector and a row vector is possible. The column vector is the middle column of the above 7 by 3 matrix, and the row vector has the form

$$(\Sigma 1 \quad \Sigma 2 \quad \Sigma 3 \quad 4 \quad 5 \quad \Sigma' 6)$$

where Σ and Σ' are certain "stress operators". But this would necessitate introducing a whole new lot of new rewriting rules, and we shall desist from such a project here.

10 Radicals of ordinary verbs

We observe that the matrix $C'(V)$ discussed in Section 4 depends on 5 parameters called "radicals":

$$\underline{\alpha}(V),\ \beta(V),\ \underline{\gamma}(V),\ \phi(V),\ \underline{\psi}(V)$$

We are not counting $\psi(V)$ as different from $\underline{\psi}(V)$ only by the absence of stress. We are also not counting $\beta'(V)$ and $\underline{\gamma}'(V)$, which differ from $\beta(V)$ and $\underline{\gamma}(V)$ only for nine exceptional verbs.

A reduction in the number of parameters to 3 is possible in view of the following observations:

(i) With very few exceptions, $\phi(V)$ is obtained from the infinitive V by removing the final r or re.

(ii) With very few exceptions, $\underline{\gamma}(V)$ has the same consonants as $\beta(V)$ and the same stress and vowels as $\underline{\alpha}(V)$.

In this connection, one may also observe that very often $\underline{\alpha}(V)$ can be obtained from $\phi(V)$ by adding stress.

We shall refrain from giving a precise formulation of these two observations, so we are left with 5 radicals. The five radicals of a number of ordinary verbs are listed in Table I below. It is understood that the verbs listed serve as models for many verbs not listed. In particular, this is so for aimer and finir, which represent the so called "living conjugations". Table I does not include the 10 exceptional verbs.

Table I

V	$\underline{\alpha}(V)$	$\beta(V)$ $= \beta'(V)$	$\underline{\gamma}(V)$ $= \underline{\gamma}'(V)$	$\phi(V)$	$\underline{\psi}(V)$
aimer	aime	aim	aime	aime	aima*
appeler	appelle	appel	appelle	appelle	appela*
manger	mange	mang*	mange	mange	mang*a*
placer	place	plac*	place	place	plac*a*
finir	fini	finiss	finiss	fini	fini
haïr	hai	haïss	haïss	haï	haï
mourir	meur	mour	meur	mour	mouru
sortir	sor	sort	sort	sorti	sorti
venir	vien	ven	vienn	viend	vin
devoir	doi	dev	doiv	dev	du
voir	voi	voy	voi	ver	vi
croitre	croi	croiss	croiss	croît	cru
naître	nai*	naiss	naiss	naît	naqui
prendre	prend	pren	prenn	prend	pri
vendre	vend	vend	vend	vend	vendi
vaincre	vainc	vainqu	vainqu	vainc	vainqui
croire	croi	croy	croi	croi	cru
écrire	écri	écriv	écriv	écri	écrivi
lire	li	lis	lis	li	lu
plaire	plai*	plais	plais	plai	plu

Table I should be thought of as a condensed list of rewriting rules. For example, the first entry represents the rule

$$\underline{\alpha}(\mathsf{aim\underline{e}r}) \longrightarrow \mathsf{\underline{a}ime}$$

The starred letters in Table I signal some minor irregularities, mainly of an orthographic nature:

a*	becomes è	before r
g*	becomes ge	before a or o
c*	becomes ç	before a or o
i*	becomes î	before t

Examples: ils aim**è**rent, nous mange**o**ns, je pla**ç**ais, il pla**î**t.

Of course, these remarks can be phrased as rewriting rules. For example, the first one becomes

$$\mathsf{\underline{a}}^{*}\ Y \longrightarrow \begin{cases} \mathsf{\underline{è}}Y & \text{if } Y \text{ starts with r} \\ \mathsf{\underline{a}}Y & \text{otherwise} \end{cases}$$

It should be pointed out that, when applying rules (1) to (6), the stem X may contain a star. Thus we have

$$\beta(\mathsf{mang\underline{e}r})4 \longrightarrow \mathsf{mang}^{*}4 \longrightarrow \mathsf{mang}^{*}\mathsf{\underline{o}ns} \longrightarrow \mathsf{mange\underline{o}ns}$$

11 Radicals of exceptional verbs

The radicals of the 10 exceptional verbs are listed in Table II, with separate listing for $\beta'(V)$ and $\underline{\gamma}'(V)$. Note that fall**oi**r is incomplete, as it only possesses forms in the third person.

TABLE II

V	$\underline{\alpha}(V)$	$\beta(V)$	$\beta'(V)$	$\underline{\gamma}(V)$	$\underline{\gamma}'(V)$	$\phi(V)$	$\underline{\psi}(V)$
$all\underline{e}r$	$[v\underline{a}]$	all	all	$v\underline{o}$	$\underline{ai}ll$	i	$all\underline{a}*$
$av\underline{oi}r$	$\underline{a}$	av	a	$\underline{o}$	$[\underline{ai}]$	au	$e\underline{u}$
$fall\underline{oi}r$	$f\underline{au}$	$fall$	$-$	$-$	$f\underline{ai}ll$	$faud$	$fall\underline{u}$
$pouv\underline{oi}r$	$p\underline{eu}$	$pouv$	$puiss$	$p\underline{eu}v$	$p\underline{ui}ss$	$pour$	$p\underline{u}$
$sav\underline{oi}r$	$s\underline{ai}$	sav	$sach$	$s\underline{a}v$	$s\underline{a}ch$	sau	$s\underline{u}$
$val\underline{oi}r$	$v\underline{au}$	val	val	$v\underline{a}l$	$v\underline{ai}ll$	$vaud$	$val\underline{u}$
$voul\underline{oi}r$	$v\underline{eu}$	$voul$	$voul$	$v\underline{eu}l$	$v\underline{eui}ll$	$voud$	$voul\underline{u}$
$\underline{ê}tre$	$[\underline{e}s]$	$[ét]$	so	$s\underline{o}$	$s\underline{o}$	se	$f\underline{u}$
$f\underline{a}ire$	$f\underline{ai}$	$[fais]$	$fass$	$f\underline{o}$	$f\underline{a}ss$	fe	$f\underline{i}$
$d\underline{i}re$	$d\underline{i}$	$[dis]$	dis	$d\underline{i}s$	$d\underline{i}s$	di	$d\underline{i}$

We may regard $\beta'(V)$ and $\underline{\gamma}'(V)$ as alternative forms (allomorphs) of $\beta(V)$ and $\underline{\gamma}(V)$ respectively. Other morphemes may possess alternative forms too, and when this is the

case we have places them in square brackets. The square brackets may be removed according to the following instructions, with a change of form in the specified context. The understanding is that, in contexts not mentioned explicitly, the square brackets may be removed without change.

[va]	becomes	vai	before 1
[es]	becomes	sui	before 1
[ét]	becomes	som	before 4
[ét]	becomes	ê	before 5
[fais]	becomes	fai	before 5
[dis]	becomes	di	before 5
[ai]	becomes	a	before $\epsilon 3$

All together there are 7 exceptional forms: je vais, je suis, nous sommes, vous êtes, vous faites, nous dites, qu'il ait.

We may of course rephrase the above instructions for removing square brackets as rewriting rules. For example, the first one becomes:

$$[\mathsf{va}]k \longrightarrow \begin{cases} \mathsf{vaik} & \text{if } k = 1 \\ \mathsf{vak} & \text{otherwise} \end{cases}$$

It should be pointed out that we are not allowed to apply rules (1) to (6) to an expression containing square brackets, and the same goes for rule (ϵ). Thus

$$\underline{\alpha}(\mathsf{aller}) \longrightarrow [v\underline{a}]1 \not\longrightarrow [\mathsf{va}]1 \longrightarrow \mathsf{vai}1 \longrightarrow \mathsf{vais}$$

Similarly

$$\underline{\gamma}'(\mathsf{avoir})\epsilon 3 \longrightarrow [\mathsf{ai}]\epsilon 3 \not\longrightarrow [\mathsf{ai}]$$

but

$$\longrightarrow \mathsf{a}\epsilon 3 \longrightarrow \mathsf{ai}3 \longrightarrow \mathsf{ait}$$

12 A production grammar for a fragment of French

Our investigation has been conducted in the language of "production grammars". Actually, we are considering the free semigroups generated by an alphabet made up of two parts:

(a) the terminal alphabet consisting of French graphemes in *sans serif*, sometimes with underlining to indicate stress

(b) an auxiliary alphabet consisting of grammatical terms, in the present case the following symbols:

$$1, \ldots, 6, \underline{\alpha}, \ldots, \psi, \iota, \epsilon, [,], *, P_1, \ldots, P_6, T_1, \ldots, T_7$$

The free semigroup is equipped with a binary relation $\longrightarrow$ which satisfies the usual reflexive, transitive and substitution laws, as well as all the rewriting rules that have been introduced in Sections 2 to 6, specifically the following rules:

(i) for 1 to 6 in Section 2,

(ii) for ϵ, ι, $\hat{}$ and [sse] in Section 3,

(iii) for P_k T_i in Section 4,

(iv) for $\underline{\alpha}$ to $\underline{\psi}$ in Table I and Table II,

(v) for the star in Section 5

(vi) for the square brackets in Section 6.

What one is really interested in is the formation of sentences. For expository purposes, we shall here confine attention to a small part of the French language which contains a few declarative sentences made up out of personal pronouns followed by inflected forms of intransitive verbs and also of the corresponding subjunctive clauses.

We need some additional symbols:

$$\begin{array}{ll} S & = \text{sentence} \\ S_i & = \text{subjunctive clause in } i\text{th tense } (i = 5, 7) \\ Subj_k & = \text{subject in } k\text{th person} \\ Pred_i & = \text{predicate in } i\text{th tense} \\ V_{\texttt{intr}} & = \text{intransitive verb} \end{array}$$

We also postulate the following new rewriting rules

$$\begin{array}{ll} S & \rightarrow Subj_k\ Pred_i, \quad (i = 1, 2, 3, 4, 6) \\ S_i & \rightarrow \mathsf{que*}\ Subj_k\ Pred_i \quad (i = 5, 7) \\ Subj_1 & \rightarrow \mathsf{je*}\ P_1 \\ Subj_2 & \rightarrow \mathsf{tu}\ P_2 \\ Subj_3 & \rightarrow \mathsf{il}\ P_3,\ \mathsf{elle}\ P_3,\ \mathsf{on}\ P_3 \\ Subj_4 & \rightarrow \mathsf{nous}\ P_4 \\ Subj_5 & \rightarrow \mathsf{vous}\ P_5 \\ Subj_6 & \rightarrow \mathsf{ils}\ P_6,\ \mathsf{elles}\ P_6 \\ V_{\texttt{intr}} & \rightarrow \mathsf{all\underline{e}r},\ \mathsf{sort\underline{i}r},\ \mathsf{veni\underline{r}},\ldots \\ Pred_i & \rightarrow T_i\ V_{\texttt{intr}} \end{array}$$

as well as the following:

$$\mathsf{e*}X \longrightarrow \begin{cases} \mathsf{e}X & \text{if } X \text{ starts with a consonant} \\ \mathsf{'}X & \text{if } X \text{ starts with a vowel} \end{cases}$$

Unless otherwise specified, i ranges from 1 to 7 and k from 1 to 6.

A few examples will make it clear how declarative sentences and subjunctive clauses are generated by the above production grammar.

(1) $S \rightarrow Subj_1\ Pred_1$
$\rightarrow$ je* $P_1\ T_1\ V_{\texttt{intr}}$
$\rightarrow$ je* $P_1\ T_1$ all̲er
$\rightarrow$ je* $\underline{\alpha}$(all̲er)1
$\rightarrow$ je* [v̲a]1
$\rightarrow$ je* v̲ai1
$\rightarrow$ je* v̲ais

(2) $S \rightarrow Subj_5\ Pred_5$
$\rightarrow$ vous $P_5\ T_2\ V_{\texttt{intr}}$
$\rightarrow$ vous $P_5\ T_2$ sort̲ir
$\rightarrow$ vous β(sort̲ir)i5
$\rightarrow$ vous sorti5
$\rightarrow$ vous sorti̲ez

(3) $S \rightarrow Subj_4\ Pred_3$
$\rightarrow$ nous $P_4\ T_3\ V_{\texttt{intr}}$
$\rightarrow$ nous $P_4\ T_3$ mour̲ir
$\rightarrow$ nous ϕ(mour̲ir)r4
$\rightarrow$ nous mourr4
$\rightarrow$ nous mourr̲ons

(4) $S_5 \rightarrow$ que* $Subj_2\ Pred_5$
$\rightarrow$ que* tu $P_2\ T_5\ V_{\texttt{intr}}$
$\rightarrow$ que tu $P_2\ T_5$ all̲er
$\rightarrow$ que tu γ'(all̲er)ϵ2
$\rightarrow$ que tu a̲illϵ2
$\rightarrow$ que tu a̲ille2
$\rightarrow$ que tu a̲illes

(5) $S \rightarrow Subj_6\ Pred_6$
$\rightarrow$ ils $P_6\ T_6\ V_{\texttt{intr}}$
$\rightarrow$ ils $P_6\ T_6$ veni̲r
$\rightarrow$ ils $\underline{\psi}$(veni̲r)r6
$\rightarrow$ ils vi̲nr6
$\rightarrow$ ils vi̲nrent

(6) $S_7 \rightarrow$ que* $Subj_3\ Pred_7$
$\rightarrow$ que* il $P_3\ T_7\ V_{\texttt{intr}}$
$\rightarrow$ qu'il $P_3\ T_7$ veni̲r
$\rightarrow$ qu'il $\underline{\psi}$(veni̲r)^3
$\rightarrow$ qu'il vi̲n^3
$\rightarrow$ qu'il vî̲n

REFERENCES

1. N. Chomsky, Syntactic structures. Mouton, 'S Gravenhage, 1957.

2. J. Dubois and R. Lagane, La nouvelle grammaire du Français. Larousse, Paris, 1973.

3. M. Gross, Mathematical Models in Linguistics. Prentice Hall, Englewood Cliffs, NJ, 1972.

4. A. Simond, Les verbes français. Payot, Lausanne, 1973.