Chapter 14

Tail recursion

It is often convenient to do a lot of reasoning in Miranda because the language has a more elegant notation that is more directly related to mathematical ideas. For instance, the properties of list functions such as append and reverse came out fairly simply in Miranda. However, in practice, you will often want to use an imperative language for its greater efficiency and so it would be nice somehow to reuse that reasoning in the context of Modula-2. We saw an example in Chapter 12. While on the subject of efficiency, it is worth mentioning that efficiency is usually less important than clarity. This is because *any* unclear piece of program can hide a fatal error, while it is only in frequently used parts that inefficiencies make a significant difference.

The feature that we now address is the transfer from the *recursive* definitions of Miranda to the *iterative* (looping) definitions of Modula-2. Of course, one can also give recursive definitions in Modula-2, but it is generally less efficient to do so.

There is a general method by which a particular special kind of definition in Miranda, the so-called *tail recursive* definition, can be converted automatically into a `WHILE` loop implementation in Modula-2; and even though not all recursive definitions are tail recursive, there is still a chance of finding equivalent tail recursive definitions — ones that define the same function.

14.1 Tail recursion

A definition of a function f is *tail recursive* iff the results of any recursive calls of f are used immediately as the result of f, without any further calculation. Therefore in a tail recursive definition, the recursion is used simply to call the same function but with different arguments.

The reason for this name is that the recursion occurs right at the end, the tail, of the calculation, and there is no more to do afterwards. For instance, the following definition of `isin` (to test whether a list *t* contains an element *x*) is tail recursive. The result of the recursive call, (`isin` *x ys*), is used directly as the result of what was being defined (`isin` *x* (*y* :*ys*)).

```
isin x []= False
isin x(y:ys)= True,     if y=x
            = isin x xs, otherwise
```

The following example, on the other hand, is *not* a tail recursive definition. The result of the recursive call (`append` *xs ys*) is used in a further calculation: it has *x cons*-ed on the front.

```
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

Figure 14.1 contains some function definitions. Which are tail recursive? ANSWERS: the definitions of `rev1`, `gcd`, `f1` and `listcomp` are tail recursive. What is (`f1` *a n*) for general *a*, not necessarily 1?

```
reverse [] = []
reverse (x:xs) = (reverse xs)++[x]

||reverse xs = rev1 [] xs
rev1 as []    = as
rev1 as (x:xs)  = rev1 (x:as) xs

gcd x y = x,                 if y=0
        = gcd y(x mod y),    otherwise

fact n = 1,                  if n=0
       = n*(fact (n-1)),     otherwise

||fact n=f1 1 n
f1 a n = a,                if n=0
       = f1 (a*n) (n-1),   otherwise

order ::= Before | Same | After
listcomp [ ] [ ]  = Same
listcomp [ ] (y:t) = Before
listcomp (x:s) [ ] = After
listcomp (x:s) (y:t) = Before,       if x < y
                     = After,       if x > y
                     = listcomp s t, otherwise
```

Figure 14.1 Assorted Miranda definitions

Tail recursion and `WHILE` loops

Think of the tail recursion as meaning 'do the same computation again, but with new arguments'. In Modula-2, you could keep variables for the arguments, and then tail recursion means 'update the variables, and repeat'. This is just looping.

To express this more precisely, we use the method of loop invariants:

> The loop invariant says: the answer you originally wanted is the same as if you calculated it starting with the variables you have got now.

For instance, for `isin` the loop invariant would be

`isin` $x\ ys_0$ (`isin` calculated with original ys)
$=$ `isin` $x\ ys$ (`isin` calculated with current ys)

14.2 Example: `gcd`

It is easy to imagine Euclid's algorithm set out in a table. For instance, to calculate the gcd of 26 and 30, you could write

x	y
26	30
30	26
26	4
4	2
2	0 answer is 2

At each stage, you replace x and y by y and x `mod` y, because the method says that (`gcd` $x\ y$) = (`gcd` y (x `mod` y)) if $y \neq 0$. The crucial property is that in each line, (`gcd` $x\ y$) = (`gcd` $x_0\ y_0$), where x_0 and y_0 are the original values of x and y (26 and 30 here). This is our loop invariant. Note also that the loop variant y is the same as the recursion variant for `gcd` $x\ y$.

```
PROCEDURE GCD(x,y: CARDINAL):CARDINAL;
(*pre: none
 *post: result = (gcd x_0 y_0) where gcd is as defined in Miranda.
 *)
VAR z: CARDINAL ;
BEGIN
(* loop invariant: (gcd x y)=(gcd x_0 y_0)
 * variant = y
 *)
  WHILE y#0 DO z := x MOD y ; x := y ; y := z END ;
  RETURN x
END GCD ;
```

Justification

initialization: initially by definition $x = x_0$ and $y = y_0$, so the invariant holds without any initialization being necessary.

loop test and finalization: we stop looping when $y = 0$, for then the first clause in the Miranda definition tells us that $(\texttt{gcd}\ x\ y) = x$, and by the loop invariant this is the answer we want. So we just return it.

reestablishing the invariant: when $y \neq 0$, then

$$(\texttt{gcd}\ x\ y) = (\texttt{gcd}\ y\ (x\ \texttt{mod}\ y)).$$

Hence by replacing x and y by y and $x\ \texttt{mod}\ y$ (which is what the sequence of assignments does), we leave $(\texttt{gcd}\ x\ y)$ unchanged and hence reestablish the invariant. Also, we have decreased the variant, y. (NOTE: $(x\ \texttt{mod}\ y)$ has a pre-condition, namely that $y \neq 0$. This holds in this part of the program.)

To be slightly more formal, let x_1 and y_1 be the values of x and y at the start of the iteration. The invariant tells us that $\texttt{gcd}\ x_1\ y_1\ = \texttt{gcd}\ x_0\ y_0$. It is easy to see that after the loop body we have $x = y_1,\ y = x_1\ mod\ y_1$ (EXERCISE: prove this with mid-conditions). Thus we have reestablished the invariant for

$$\texttt{gcd}\ x\ y\ \ =\ \ \texttt{gcd}\ y_1\ (x_1\ \texttt{mod}\ y_1)\ \ =\ \ \texttt{gcd}\ x_1\ y_1\ \ =\ \ \texttt{gcd}\ x_0\ y_0.$$

Recall that in general we resolved not to assign to variables that were called by value. This was to make the reasoning easier. However, with this method it is particularly convenient and natural to break this resolution — after all, the informal justification was that we change the arguments of the function. Therefore, we put in an explicit disclaimer to say that the call-by-value parameters might change. In this example, of course, the only effects of this are local to the procedure — the change cannot be detected in the outside world.

14.3 General scheme

In general, a tail recursive definition in Miranda looks as follows:

```
f x = a1,   if c1
    = a2,   if c2
    = ...          || more non-recursive cases
    = an,   if cn
    = f x1, if d1
    = f x2, if d2
    =  ...         || more recursive cases
```

$a_1, a_2, \ldots, a_n$ are expressions giving the answers in the non-recursive cases. $x_1, x_2, \ldots$ are the new parameters used in the tail recursive cases. $a_1, a_2, \ldots, a_n, x_1, x_2, \ldots$, as well as the guards $c_1, c_2, \ldots, c_n, d_1, d_2, \ldots$, are all calculated simply, without recursion. There is no difficulty in making this work when `f` has more than one parameter.

Translation using `WHILE` loop

```
PROCEDURE f(x: ... ): ...;
(* NB Value parameter x  may be changed
 *pre: any pre-conditions needed for f
 *post: result = (f x_0) where f is as defined above in Miranda
 *)
BEGIN
(* loop invariant: (f x) = (f x_0)
 * variant: recursion variant for Miranda f
 *)
  WHILE NOT c1 AND NOT c2 AND ... NOT cn  DO
    IF d1 THEN x:=x1
    ELSIF d2 THEN x:=x2
    ELSIF ...
    END
  END;
  IF c1 THEN RETURN a1
  ELSIF c2 THEN RETURN a2
  ELSIF ...
  END
END f;
```

Exercise: how does `gcd` fit this pattern? Note that the invariant and the variant come automatically.

14.4 Example: `factorial`

The following is the obvious recursive definition of the factorial function, but it is not tail recursive:

```
fact :: num -> num
||pre: nat(n)
||post: fact n = n!
fact n = 1,               if n=0
       = n*(fact (n-1)),  otherwise
```

After the recursive call (`fact`(n—1)), there is still a residual computation ($n * \ldots$). However, these can be 'accumulated' into a single variable:

```
f1 a n = a,             if n=0
      = f1(a*n)(n-1),   otherwise
```

and then (`fact` n) = (`f1` *1* n) (but we shall have to prove this). a is the *accumulator parameter* in `f1`. `f1` *is* tail recursive, so you can convert it into a `WHILE` loop. But in fact, we do not need to implement `f1` separately in Modula-2; we can put its `WHILE` loop into the implementation for `fact`, with an extra local variable for the accumulator parameter:

```
PROCEDURE fact(n: CARDINAL):CARDINAL;
(* NB may change n
 *pre: none
 *post: result = (fact n_0)
 *      where fact is as defined in Miranda
 *)
VAR a: CARDINAL;
BEGIN
  a := 1;
(*loop invariant: (fact n_0) = (f1 a n) where f1 as defined in Miranda
 *variant = n
 *)
  WHILE n#0 DO a := a*n; n := n-1 END;
  RETURN a
END fact;
```

Justification

initialization: this relies on the property, promised but not yet proved, that $(fact\ \ n) = (f1\ \ 1\ \ n)$.

loop test and finalization: when $n = 0$, we know that $(f1\ \ a\ \ n)$ is just a; but this is the answer we require, so we can just return a as the result.

reestablishing the invariant: when $n \neq 0$ then $(f1\ \ a\ \ n) = (f1\ \ (a*n)(n-1))$, so we reestablish the invariant by replacing a and n by $a*n$ and $n-1$.

It still remains to be shown that $fact\ \ n = f1\ \ 1\ \ n$. The method to use is induction, but some care is needed. Suppose we try to use simple induction on n to prove $\forall n : nat.\ P(n)$, where

$$P(n) \equiv fact\ \ n = f1\ \ 1\ \ n$$

For the induction step we assume $P(n)$, and prove $P(n+1)$:

$$fact\ \ (n+1) = (n+1) \times (fact\ \ n) = (n+1) \times (f1\ \ 1\ \ n)$$
$$= f1\ \ 1\ \ (n+1) = f1\ \ (n+1)\ \ n$$

How can we bridge the gap and prove $(n+1) \times (f1\ \ 1\ \ n) = f1\ \ (n+1)\ \ n$? The answer is that we cannot. The inductive hypothesis only tells us about the behaviour of $f1$ when its accumulator parameter is 1. We actually have to prove something more general and this involves understanding what $(f1\ \ a\ \ n)$ calculates for the general a: it is $a \times n!$, so we want to prove it equal to $a \times (fact\ \ n)$.

Proposition 14.1 : $\forall n : nat.\ fact\ \ n = f1\ \ 1\ \ n$

Proof We first prove by induction on n that $\forall n : nat.\ P(n)$ where

$$P(n) \equiv \forall a : nat.\ a * (fact\ \ n) = f1\ \ a\ \ n$$

base case: $f1\ \ a\ \ 0 = a = a * 1 = a * (fact\ \ 0)$

induction step: Assume $P(n)$, and prove $P(n+1)$. Let a be a natural number. Then

$$\begin{aligned} f1\ \ a\ \ (n+1) &= f1(a*(n+1))n \\ &= a*(n+1)*(fact\ \ n) \text{ by induction} \\ &= a*(fact\ \ (n+1)) \\ \text{Hence } 1*(fact\ \ n) &= fact\ \ n = f1\ \ 1\ \ n. \end{aligned}$$

□

For functions with accumulating parameters, you may need to first understand how the accumulator works, and then formulate a stronger statement to prove.

14.5 Summary

- A recursive function is said to be *tail* recursive if in each recursive clause of the definition the entire right-hand side of its equation consists of a call to the function itself. A tail recursive function is similar to a loop.
- A general technique for transforming recursive Miranda definitions into `WHILE` loop Modula-2 definitions is as follows:

  1. Find an obvious solution in Miranda.
  2. Find a (perhaps less obvious) tail recursive solution in Miranda.
  3. Prove that they both give the same answers.
  4. Translate the tail recursive version into Modula-2 with `WHILE` loops.
  5. Write down the loop invariant in terms of the Miranda function.
  6. The loop variant is the recursion variant.

14.6 Exercises

1. Write Modula-2 code for the tail recursive Miranda functions in Section 14.1. Prove that `reverse` *xs*=`rev1` *xs* as claimed.

2. One way of viewing integer division `x div y` is that the result is how many times you can subtract y from x (and the remainder `x mod y` is what is left). The following is an implementation of that idea:

```
divmod::num->num->(num,num)
||pre: nat(m) & nat(n) & n >= 1
||post: divmod m n = (m div n, m mod n)
||i.e. nat(q) & nat(r) & r < n
||      & m = q*n + r
||          where (q,r) = divmod m n
divmod = f 0
           where f a m n = (a,m),     if m < n
                         = f (a+1) (m-n) n, otherwise
```

   How does this work? (HINT: using n as recursion variant for $f\ a\ m\ n$, show that if a, m and n are natural numbers with $n \geq 1$, then $f\ a\ m\ n$ satisfies the post-condition for `divmod (m + a*n) n`.)

   Use the fact that f is tail recursive to implement the method iteratively in Modula-2.

3. Define a recursive function `add` in Miranda for the addition of two *diynat* (as defined in Chapter 7) natural numbers. Rewrite your function in tail recursive style.

4. The Fibonacci sequence is

   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

   Each number is the sum of the preceding two, and this can be defined in Miranda by

```
fib :: num -> num
||pre: nat(n)
||post: fib n is the nth Fibonacci number
||      (starting with "zeroth"=0, "first"=1)
fib n=0,                        if n=0
     = 1,                       if n=1
     = (fib (n-2))+(fib (n-1)), otherwise
```

   This is *terribly* inefficient. Try `fib 25`. Why does it take so long?

   A more efficient method is to calculate the *pair* $(fib\ n,\ fib\ (n+1))$:

```
twofib :: num -> (num,num)
||pre: nat(n)
||post: twofib n = (fib n, fib(n+1))
twofib n=(0,1),        if n=0
        =(y,x+y),      otherwise
          where (x,y) = twofib (n-1)
fib1 n=x
       where (x,y) = twofib n
```

Prove (by induction on n) that

$$\forall n : nat.\ twofib\ \ n = (fib\ \ n\ \ fib\ \ (n+1))$$

5. Let us define the *generalized* Fibonacci numbers ($gfib\ \ x\ \ y\ \ n$) by

```
gfib :: num -> num -> num -> num
||pre:nat(n)
||post: g fib x y n is the nth generalized Fibonacci number
||      (starting with "zeroth"=x, "first"=y)
gfib x y n=x,  if n=0
          =y,  if n=1
          =(gfib x y(n-2))+(gfib x y(n-1)), otherwise
```

They are generated by the same recurrence relation (the 'otherwise' alternative) as the ordinary Fibonaccis, but starting off with x and y instead of 0 and 1.

(a) Prove by induction on n that

$$\forall n : nat.\ fib\ \ n = gfib\ \ 0\ \ 1\ \ n$$

(b) Now the sequence ($gfib\ \ y\ \ (x+y)$) (as n varies) is the same as the sequence ($gfib\ \ x\ \ y$) except that the first term x is omitted: $(gfib\ \ x\ \ y\ \ (n+1)) = (gfib\ \ y\ \ (x+y)\ \ n)$. Prove this by induction on n.

Let us therefore define

```
gfib1 x y n = x,                 if n=0
            = gfib1 y(x+y)(n-1), otherwise
```

(c) Prove by induction on n that

$$\forall n : nat.\ \forall x, y : num.\ gfib1\ \ x\ \ y\ \ n = gfib\ \ x\ \ y\ \ n$$

Part II

Logic

Chapter 15

An introduction to logic

15.1 Logic

In this part of *Reasoned Programming* we investigate *mathematical logic*, which provides the formal underpinnings for reasoning about programming and is all about formalizing and justifying arguments. It uses the same rules of deduction which we all use in drawing conclusions from premisses, that is, in reasoning from assumptions to a conclusion. The rules used in this book are *deductive* — if the premisses are believed to be true, then the conclusions are bound to be true; acceptance of the premisses forces acceptance of the conclusion.

A program's specification can be used as the premiss for a logical argument and various properties of the program may be deduced from it. These are the conclusions about the program that we are forced to accept given that the specification is true.

```
A :: num -> num
|| pre: none
||post: returns (x+1)^2
```

For example, in the program above, it can be deduced from the specification of A that, for whatever argument (input) x that A is applied to, it delivers a result ≥ 0. We cannot deduce, however, that it will always deliver a result $\geq x^2$ unless the pre-condition is strengthened, for example to $x \geq 0$.

Examples of applications of correct, or valid, reasoning are 'I wrote both program A and program B so I wrote program A', 'if the machine is working I run my programs; the machine is working so I run my programs', 'if my programs are running the machine is working; the machine is not working so my programs are not running', etc.

It is not difficult to spot examples of the use of invalid reasoning; political debates are usually a good source. Some examples are 'if wages increase too fast then inflation will get worse; inflation does get worse so wages are increasing too fast', 'some people manage to support their elderly relatives; so all people can'.

In this Chapter we introduce the language in which such deductions can be expressed.

15.2 The propositional language

An example

In order to see clearly the logical structure of an English sentence we translate it into a special logical notation which is unambiguous. This is what we mean by 'translating into logic'. For example, consider the sentence

> If Humphrey is over 21 and either he has previously been sentenced to imprisonment or non-imprisonment is not appropriate then a custodial sentence is possible.

We can translate this into logical notation in stages, by teasing out the logical structure layer by layer. First, we may write

> If Humphrey is over 21 $\wedge$ (he has previously been sentenced to imprisonment $\vee$ non-imprisonment is not appropriate) then a custodial sentence is possible.

Next,

> (Humphrey is over 21 $\wedge$ (he has previously been sentenced to imprisonment $\vee$ non-imprisonment is not appropriate)) $\rightarrow$ a custodial sentence is possible.

Then,

$$\begin{pmatrix} \textit{over21}(\textit{Humphrey})\wedge \\ (\textit{already-sentenced}(\textit{Humphrey}\ ) \vee \text{non-imprisonment-is-not-appropriate}) \end{pmatrix} \\ \rightarrow \textit{possible-custodial-sentence}(\textit{Humphrey})$$

and, finally,

$$\begin{pmatrix} \textit{over21}(\textit{Humphrey})\wedge \\ (\textit{already-sentenced}(\textit{Humphrey}\ ) \vee \neg \textit{non-imprisonment-is-appropriate}) \end{pmatrix} \\ \rightarrow \textit{possible-custodial-sentence}(\textit{Humphrey}).$$

In this example we have introduced the connectives $\vee$ (*or* or *disjunction*), $\wedge$ (*and* or *conjunction*), $\rightarrow$ (*implies* or *if* $\cdots$ *then*), and $\neg$ (*not*). We also used parentheses to disambiguate sentences. Without parentheses we cannot tell whether $A \wedge B \rightarrow C$ is really $A \wedge (B \rightarrow C)$ or $(A \wedge B) \rightarrow C$.

Eventually, the analysis reaches statements, or propositions, such as 'Humphrey is over 21', where we do not wish to analyze the logical structure any further. These are called *atoms* (atomic means indivisible), that is, not made up using connectives. The connectives then connect atoms to make *sentences*. We have also introduced a structure for the atoms. Propositions usually have a subject (a thing) and then describe a property about that thing. For example, 'Humphrey' is a thing and 'over 21' is a property, or predicate, about it. Atoms are usually written as *predicate*(*thing*). We distinguish between *terms*, which are things, and *predicates*, which are the properties.

As another example, consider 'Jane likes logic and (she likes) programming'. The logical meaning is two sentences connected by 'and'. In each one, Jane is the subject, so the translation is

$$likes(Jane, logic) \wedge likes(Jane, programming)$$

Notice how Jane appears twice in the logical structure, although only once in English. The English 'and' is more flexible because it can conjoin noun phrases (logic and programming) as well as sentences (Jane likes logic and Jane likes programming).

The use of parentheses to express priority can sometimes be avoided by a convention analogous to that used in algebraic expressions:

$\rightarrow$ binds less closely than $\wedge$ or $\vee$ and $\neg$ binds the closest of all.

Thus $P \wedge Q \rightarrow R$ is shorthand for $(P \wedge Q) \rightarrow R$, not for $P \wedge (Q \rightarrow R)$ and $\neg A \wedge B$ is not the same as $\neg(A \wedge B)$. Also, (as in English) we do not need parentheses for $P \wedge Q \wedge R \wedge \ldots$ or $P \vee Q \vee R \vee \ldots$, but we do need them if the $\wedge$ and $\vee$ are mixed, as in $(P \wedge Q) \vee R$.

The language of atoms and connectives is called *propositional logic*.

Atoms

An atom, or a proposition, is just a statement or a fact expressing that a property holds for some individual or that a relationship holds between several individuals, for example 'Steve travels to work by train'. Sometimes, the atoms are represented by single symbols such as *Steve-goes-by-train*. More usually, the syntactic form is more complex. For instance, 'Steve goes by train' might be expressed as *goesbytrain*(*Steve*) or as *travels*(*Steve, train*).

The *predicate* symbol *travels*(,) requires two *arguments* in order to become an *atom*. *Steve-goes-by-train* (or *SGT* for short) is called a *proposition symbol*, or a predicate symbol that needs no arguments. The predicate symbol *goesbytrain* needs one argument to become an atom. The two arguments of *travels* used here are *Steve* and *train* and the argument of *goesbytrain* is *Steve*. Adjectives are translated into predicate symbols and nouns into arguments, which is why, for example 'programming is fun' is translated into *fun*(*programming*) rather than into *programming*(*fun*).

You may come across the word *arity*, which is the number of arguments a predicate symbol has. Predicate symbols with no arguments are called propositional, predicate symbols of arity one express properties of individuals and predicate symbols of arity two or more express relations between individuals.

In English, predicates often involve several words which are distributed around the nouns, or in front of or behind the nouns, but when translating, a convention is used that puts the predicate symbol first followed by the arguments in parentheses and separated by commas. In case the predicate has just two arguments it is sometimes written between the arguments in *infix* form. Whenever a predicate symbol is introduced a description of the property or relation it represents should be given. For example, $travels(x, y)$ is read as 'x travels by y'.

The arguments of predicate symbols are called *terms*. Terms can be simple constants, names for particular individuals, but you can also build up more complex ones using a structured or *functional term* which is a function symbol with one or more arguments. For example, whereas an empty list may be denoted by the constant [ ], a non-empty list is usually denoted by a functional term of the form (*head* : *tail*), where *head* is the first element and *tail* is the list consisting of the rest of the elements. Thus the list [*cat*,*dog*] is represented by the term (*cat* : (*dog* : [ ])). Here : is an infix function symbol.

An example of the use of a prefix function symbol is $s(0)$. Just as predicates may have arities of any value ≥ 0, so can function symbols and each argument of a functional term can also be a functional term. So functional terms can be nested, as in $mum(mum(Krysia))$ or $+(*(2, 2), 3)$.

15.3 Meanings of the connectives

In English, words such as 'or' may have several slightly different meanings, but the logical connectives $\vee$, $\wedge$, etc., have a fixed unambiguous meaning.

$A \wedge B$ means A and B are both true.
$A \vee B$ means at least one of A and B is true.
$A \rightarrow B$ means if A then B (or A implies B, or B if A)
$\neg A$ means not A (or it is not the case that A is true).
$A \leftrightarrow B$ means A implies B and B implies A (or either both A and B are true or both A and B are not true).

Figure 15.1 Meanings of the connectives

The meanings can be described using a *truth table*, shown in Figure 15.2. It is possible for each atom to be either true (*tt*) or false (*ff*) so for two atoms there are exactly four possibilities: $\{tt, tt\}$, $\{tt, ff\}$, $\{ff, tt\}$, $\{ff, ff\}$. Each row of the truth table gives the meaning of each connective in one situation.

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
tt	*tt*	*tt*	*tt*	*tt*	*tt*
tt	*ff*	*ff*	*tt*	*ff*	*ff*
ff	*tt*	*ff*	*tt*	*tt*	*ff*
ff	*ff*	*ff*	*ff*	*tt*	*tt*

Figure 15.2 A truth table

From this truth table it can be seen that $A \wedge B$ is only true when both A and B are true.

Determining whether a sentence is true or not in some situation is analogous to calculating the value of an arithmetic expression. To find the value of the expression $2 + (x * y)$ when x and y have the values 3 and 6, respectively, you calculate $2 + (3 * 6) = 20$. Similarly, to find the value of $A \vee (B \wedge C)$ when A, B and C are *ff*, *tt* and *ff*, respectively, you calculate $ff \vee (tt \wedge ff) = ff$.

So, in order to decide if a complex sentence is true you need to look at its atoms, decide if they are true, and then use the unambiguous meanings of the connectives to decide whether the sentence is true. For example, consider again the sentence

> If Humphrey is over 21 and either he has previously been sentenced to imprisonment or non-imprisonment is not appropriate then a custodial sentence is possible.

which was written in logic as

$$\left(\begin{array}{l} \textit{over21}(\textit{Humphrey}) \wedge \\ (\textit{already-sentenced}(\textit{Humphrey}) \vee \neg \textit{non-imprisonment-is-appropriate}) \end{array}\right) \\ \rightarrow \textit{possible-custodial-sentence}(\textit{Humphrey}).$$

Suppose that Humphrey is over 21, that he has not been sentenced to imprisonment before and that non-imprisonment is appropriate, then the condition of the implication is false — although the first conjunct is true the second is not as each of the disjuncts is false. In this case, then, the whole sentence is true, for an implication is true if its condition is false. You can use this method for any other situation.

Some comments on the meanings of connectives

The truth tables give the connectives a meaning that is quite precise, more precise in fact than that of their natural language counterparts, so care is sometimes needed in translation.

The meaning of $\wedge$ is just like the meaning of 'and' but notice that any involvement of time is lost. Thus A and (then) B is simply $A \wedge B$ and, for example, both 'Krysia fell ill and had an operation' and 'Krysia had an operation and fell ill' are translated the same way. 'A but B' is also translated as $A \wedge B$, even though in general it implies that B is not usually the case, as in 'Krysia fell ill but carried on working'. To properly express these sentences you need to use the quantifier language of Section 15.4.

$A \vee B$ means 'A or B *or both*'. The stronger, 'A or B but not both', can be captured by the sentence $(A \vee B) \wedge \neg(A \wedge B)$. The stronger meaning is called *exclusive or*. For example, consider 'donations to the cause will be accepted in cash or by cheque' and 'you can have either coffee or tea after dinner'. (Which of these is using the stronger, exclusive or?)

Consider the meaning of

$$\mathit{diets}(\mathit{Jack}) \rightarrow \mathit{lose\text{-}weight}(\mathit{Jack})$$

that is,

'If Jack diets then he will lose weight'.

The only circumstance under which one can definitely say the statement is false is when

Jack diets but stays fat.

In other circumstances, for example

Jack carries on eating, but gets thin
Jack carries on eating, and stays fat

there is no reason to doubt the original statement as the condition of that statement is not true in these situations.

Natural language also uses other connectives, such as 'only if' and 'unless', which can be translated using the connectives given already.

A unless B is usually translated as 'A if $\neg B$' (that is, $\neg B \rightarrow A$), in which B occurs rather like an escape clause. A unless B can also can be translated as $B \vee A$. All of the sentences 'Jack will not slim unless he diets', 'either Jack diets or he will not slim' and 'Jack will not slim if he does not diet' can be translated in the same way as $\mathit{diets}(\mathit{Jack}) \vee\neg\ \mathit{slims}(\mathit{Jack})$.

'A only if B' is usually translated as $A \to B$, as in 'you can enter only if you have clean shoes', which would be 'if you enter then you (must) have clean shoes'. The temptation to translate A only if B as $B \to A$ instead of $A \to B$ is very strong. To see the problem, consider

> I shall go only if I am invited (A only if B)

Logically, it is $A \to B$ — if you start from knowledge about A then you can go on to deduce B (or that B must have happened). Temporally it is the other way around — B (the invitation) comes first and results in A. *But* A is not inevitable (I might fall ill and be unable to go) so there is *no* logical $B \to A$.

The sentence $A \leftrightarrow B$, is often defined as $A \to B \wedge B \to A$, which is A only if B and A if B, or A if and only if B, which is often shortened to A iff B.

15.4 The quantifier language

The logic language covered so far is not sufficiently expressive to fully analyze sentences such as 'all students enjoy themselves' or 'Jack will always be fat' — we need the use of generalizations.

Consider the sentence 'the cat is striped', or, in logical notation, $\mathit{striped}(\mathit{cat})$. Before you can understand this sentence or consider whether it is true or not cat needs to be defined so that you know exactly which cat is meant.

Now compare this with 'something is striped'. 'Something' here is rather different from 'cat'. To test the truth of this sentence you do not need to know beforehand exactly what 'something' is; you just need to know the range of acceptable possibilities and then you go through them one by one to find at least one that is striped. If you succeed then 'something is striped' is true.

In line with this distinction, we do not write $\mathit{striped}(\mathit{something})$ in logic, but, instead, write $\exists x.\ \mathit{striped}(x)$. This is read literally as 'for some x, x is striped', but we are sure that you can see this is equivalent to 'something is striped'.

The meaning of this sentence is

> there is some value, which when substituted for x in $\mathit{striped}(x)$, yields a true statement.

This is even more clear if you consider 'the cat is striped and hungry', $\mathit{striped}(\mathit{cat}) \wedge \mathit{hungry}(\mathit{cat})$; since the meaning of cat is fixed beforehand both occurrences of 'the cat' refer to the same thing. On the other hand, in 'something is striped and something is hungry', $\exists x.\ \mathit{striped}(x) \wedge \exists y.\ \mathit{hungry}(y)$, the two somethings could be different.

It is also possible to say 'something is both striped and hungry', as in $\exists x.\ [\mathit{striped}(x) \wedge \mathit{hungry}(x)]$. This time there is only one something referred to and, whatever it is, it is hungry as well as striped.

Now, unlike *cat*, which was a *constant*, x has the potential to vary and is called a *variable*. The x in $\exists x$ announces that x is a variable and applies to all of the following formula that follows the '.'. For non-atomic formulas parentheses (square or round) are needed to show the scope of the x. For example, $\exists x.\ [P(x) \vee Q(x)]$. The occurrence of $\exists$ is said to *bind* the occurrences of x in that formula.

$\exists$ is called a *quantifier* (and $\exists x$ is called a *quantification*). Another quantifier is $\forall$, the *universal quantifier*, which can be read as 'for all' or 'every'. For example, in 'Fred likes everyone', we do not write $likes(Fred, everyone)$, but $\forall x.\ likes(Fred, x)$. To see if this sentence is true you need to check that Fred likes all values in a specified range. The meaning of this sentence is

for all values substituted for x, $likes(Fred, x)$ is a true statement.

Something that is rather important is that when you have two occurrences of the same variable bound by the same quantification they must denote the same value. For instance, the xs in $\exists x.\ [striped(x) \wedge hungry(x)]$ must denote the same value; to make the sentence true you must find a value for x that is both striped and hungry. On the other hand, in $\exists x.\ striped(x) \wedge \exists x.\ hungry(x)$ the two xs are bound by different quantifications and you just need to find something that is striped and something that is hungry — the same or different, it does not matter — for the sentence to be true.

$\forall x.\ [likes(x, Fred) \vee likes(x, Mary)]$ is true if every value tried for x makes either $likes(x, Fred)$ or $likes(x, Mary)$ true. Compare this with $\forall x.\ likes(x, Fred) \vee \forall x.\ likes(x, Mary)$, which is true if *either* $\forall x.\ likes(x, Fred)$ is true, or $\forall x.\ likes(x, Mary)$ is true. In the second case the two xs are bound by different quantifications and again are really two different variables.

In the sentence 'someone likes everyone', which is $\exists x.\ \forall y.\ likes(x, y)$ the two variables of the nested quantifiers are different. It would be asking for trouble if they were the same and so we shall forbid it.

Quantifiers which are of the *same* sort can be placed *in any order*. For example,

$\forall x.\ \forall y.\ [mother(x, y) \rightarrow parent(x, y)]$

is no different from

$\forall y.\ \forall x.\ [mother(x, y) \rightarrow parent(x, y)]$.

They both mean that, for any x and y, if x is a mother of y then x is a parent of y. Similarly, $\exists x.\ \exists y.\ \cdots$ means the same as $\exists y.\ \exists x.\ \cdots$.

For quantifiers of *different* sorts the order is important. For instance,

$\forall x.\ \exists y.\ mother(y, x)$

does not mean the same as

$\exists y.\ \forall x.\ mother(y, x)$

The first says that everyone has a mother, literally, for all x there is some y such that y is the mother of x and you know this is true when x and y vary over people. The second says that there is *one single person* who is the mother of everyone; literally, there is some y such that for all x, y is the mother of x. This is a much stronger statement which you know is not true when x and y can vary over people.

15.5 Translation from English

You have already seen how to translate from English to logic in the propositional case, teasing out the logical structure connective by connective. The same principles apply when you have quantifiers and variables, but there are also some specific new issues to consider.

There are several useful rules of thumb which aid the process of translation.

Pronouns

Pronouns, words such as 'he', 'it' or 'nothing', do not in themselves refer to any specific thing but gain their meaning from their context. You have already seen how the words 'something' and 'everyone' are translated using quantifiers and 'nothing' is similar — 'nothing is striped' becomes $\neg\exists x.\ \mathit{striped}(x)$.

Words such as 'she' or 'it' are used specifically as a reference to someone or something that has already been mentioned, so they inevitably correspond to two or more references to the same value. When you come across a pronoun such as this you must work out exactly what it does refer to. If that is a constant then you replace it by the constant: so 'Chris adores Pat who adores her' becomes $\mathit{adores}(\mathit{Chris},\ \mathit{Pat}) \wedge \mathit{adores}(\mathit{Pat},\ \mathit{Chris})$.

That is easy, but when the pronoun refers to a variable you must first set up a quantification and ensure that it applies to them both.

For instance, consider 'something is spotted and it is hungry'. An *erroneous* approach is to translate the two phrases 'something is spotted' and 'it is hungry' separately. This is wrong because the 'something' and 'it' are linked across the connective 'and' and you must set up a variable to deal with this linkage:

$$\exists x.\ [x \text{ is spotted and } x \text{ is hungry}]$$

and *then* deal with the 'and':

$$\exists x.\ [\mathit{spotted}(x) \wedge \mathit{hungry}(x)]$$

The rule of thumb is:

If pronouns are linked across a connective, deal with the pronouns before the connective.