as complex numbers. If they were real numbers, then $\Re(x + i\,y)$ would simplify to x.

```
> abznorm( x+y*I );
```

$$\sqrt{\Re(x + I\,y)^2 + \Im(x + I\,y)^2}$$

Many Maple commands return unevaluated in such cases. Thus, you might alter `abznorm` to return `abznorm(x+y*I)` in the above example. Later examples in this book show how to give your own procedures this behavior.

1.2 Basic Programming Constructs

This section describes the programming constructs you require to get started with real programming tasks. It covers assignment statements, `for` loops and `while` loops, conditional statements (`if` statements), and the use of local and global variables.

The Assignment Statement

Use assignment statements to associate names with computed values. They have the following form.

```
variable := value ;
```

This syntax assigns the name on the left-hand side of `:=` to the computed value on the right-hand side. You have seen this statement used in many of the earlier examples.

The use of `:=` here is similar to the assignment statement in programming languages, such as Pascal. Other programming languages, such as C and Fortran, use `=` for assignments. Maple does not use `=` for assignments, since it is such a natural choice for representing mathematical equations.

If you want to write a procedure called `plotdiff` which plots an expression $f(x)$ together with its derivative $f'(x)$ on the interval $[a, b]$, you can accomplish this task by computing the derivative of $f(x)$ with the `diff` command and then plotting both $f(x)$ and $f'(x)$ on the same interval with the `plot` command.

```
> y := x^3 - 2*x + 1;
```

$$y := x^3 - 2\,x + 1$$

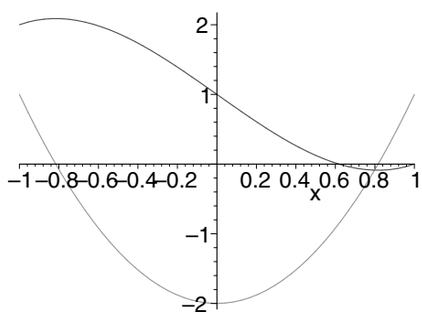
Find the derivative of y with respect to x.

```
> yp := diff(y, x);
```

$$yp := 3\,x^2 - 2$$

Plot y and yp together.

```
> plot( [y, yp], x=-1..1 );
```



The following procedure combines this sequence of steps.

```
> plotdiff := proc(y,x,a,b)
>     local yp;
>     yp := diff(y,x);
>     plot( [y, yp], x=a..b );
> end proc;
```

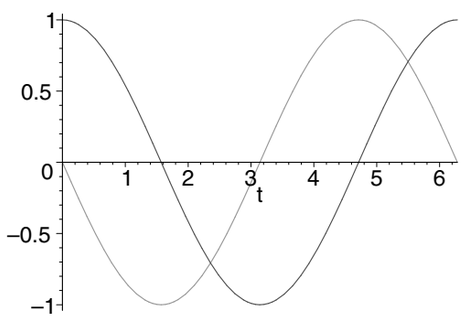
$$\begin{aligned}&\mathit{plotdiff} := \mathbf{proc}(y,\ x,\ a,\ b)\\ &\mathbf{local}\ \mathit{yp};\\ &\quad \mathit{yp} := \mathrm{diff}(y,\ x)\,;\ \mathrm{plot}([y,\ \mathit{yp}],\ x = a..b)\\ &\mathbf{end\ proc}\end{aligned}$$

The procedure name is `plotdiff`. It has four parameters: y, the expression it differentiates; x, the name of the variable it uses to define the expression; and a and b, the beginning and the end of the interval over which it generates the plot. The procedure returns a Maple plot object which you can either display, or use in further plotting routines.

By specifying that yp is a local variable, you ensure that its usage in the procedure does not clash with any other usage of the variable that you may have made elsewhere in the current session.

To use the procedure, simply invoke it with appropriate arguments. Plot $\cos(t)$ and its derivative, for t running from 0 to 2π.

```
> plotdiff( cos(t), t, 0, 2*Pi );
```



The `for` Loop

Use looping constructs, such as the `for` loop, to repeat similar actions a number of times. For example, you can calculate the sum of the first five natural numbers in the following way.

```
> total := 0;
```

```
> total := total + 1;
```

```
> total := total + 2;
```

```
> total := total + 3;
```

```
> total := total + 4;
```

```
> total := total + 5;
```

You may instead perform the same calculations by using a `for` loop.

```
> total := 0:
> for i from 1 to 5 do
>     total := total + i;
> end do;
```

$$total := 1$$

$$total := 3$$

$$total := 6$$

$$total := 10$$

$$total := 15$$

For each cycle through the loop, Maple increments the value of `i` by one and checks whether `i` is greater than 5. If it is not, then Maple executes the body of the loop again. When the execution of the loop finishes, the value of `total` is 15.

```
> total;
```

$$15$$

The following procedure uses a `for` loop to calculate the sum of the first n natural numbers.

```
> SUM := proc(n)
>     local i, total;
>     total := 0;
>     for i from 1 to n do
>        total := total+i;
>     end do;
>     total;
> end proc:
```

The purpose of the `total` statement at the end of `SUM` is to ensure that `SUM` returns the value `total`. Calculate the sum of the first 100 numbers.

```
> SUM(100);
```

$$5050$$

The `for` statement is an important part of the Maple language, but the language also provides many more succinct and efficient looping constructs. For example, the command `add`.

```
> add(n, n=1..100);
```

$$5050$$

The Conditional Statement

The loop is one of the two most basic constructs in programming. The other basic construct is the `if` or *conditional statement*. It arises in many contexts. For example, you can use the `if` statement to implement an absolute value function.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0. \end{cases}$$

Below is a first implementation of `ABS`. Maple executes the `if` statement as follows: If $x < 0$, then Maple calculates $-x$; otherwise it calculates x. In either case, the absolute value of x is the last result that Maple computes and so is the value that `ABS` returns.

The closing words `end if` completes the `if` statement.

```
> ABS := proc(x)
>     if x<0 then
>         -x;
>     else
>          x;
>     end if;
> end proc;
```

$$ABS := \mathbf{proc}(x)\,\mathbf{if}\,x < 0\ \mathbf{then}\ -x\,\mathbf{else}\,x\,\mathbf{end\ if\ end\ proc}$$

```
> ABS(3); ABS(-2.3);
```

$$3$$

$$2.3$$

Returning Unevaluated The `ABS` procedure above cannot handle non-numeric input.

```
> ABS( a );
```

```
Error, (in ABS) cannot evaluate boolean: a < 0
```

The problem is that since Maple knows nothing about `a`, it cannot determine whether `a` is less than zero. In such cases, your procedure should *return unevaluated*; that is, `ABS` should return `ABS(a)`. To achieve this result, consider the following example.

```
> 'ABS'(a);
```

$$\mathrm{ABS}(a)$$

The single quotes tell Maple not to evaluate `ABS`. You can modify the `ABS` procedure by using the `type(..., numeric)` command to test whether x is a number.

```
> ABS := proc(x)
>   if type(x,numeric) then
>       if x<0 then -x else x end if;
>   else
>       'ABS'(x);
>   end if;
> end proc:
```

The above `ABS` procedure contains an example of a *nested* `if` statement, that is, one `if` statement appearing within another. You need an even more complicated nested `if` statement to implement the function

$$\mathrm{hat}(x) = \begin{cases} 0 & \text{if } x \le 0 \\ x & \text{if } 0 < x \le 1 \\ 2 - x & \text{if } 1 < x \le 2 \\ 0 & \text{if } x > 2. \end{cases}$$

Here is a first version of `HAT`.

```
> HAT := proc(x)
>    if type(x, numeric) then
>       if x<=0 then
>          0;
>       else
>          if x<=1 then
>             x;
>          else
>             if x<=2 then
>                2-x;
>             else
>                0;
>             end if;
>          end if;
>       end if;
>    else
>       'HAT'(x);
>    end if;
> end proc:
```

The indentations make it easier to identify which statements belong to which `if` conditions.

A better implementation uses the optional `elif` clause (else if) in the second-level `if` statement.

```
> HAT := proc(x)
>    if type(x, numeric) then
>       if x<=0 then 0;
>       elif x<=1 then x;
>       elif x<=2 then 2-x;
```

```
>       else 0;
>       end if;
>    else
>       'HAT'(x);
>    end if;
> end proc:
```

You may use as many `elif` branches as you need.

Symbolic Transformations You can improve the `ABS` procedure from the last section even further. Consider the product ab. Since ab is an unknown, `ABS` returns unevaluated.

```
> ABS( a*b );
```

$$\mathrm{ABS}(a\,b)$$

However, the absolute value of a product is the product of the absolute values.

$$|ab| \to |a||b|$$

That is, `ABS` should map over products.

```
> map( ABS, a*b );
```

$$\mathrm{ABS}(a)\,\mathrm{ABS}(b)$$

You can use the `type(..., '*')` command to test whether an expression is a product and use the `map` command to apply `ABS` to each operand of the product.

```
> ABS := proc(x)
>    if type(x, numeric) then
>       if x<0 then -x else x end if;
>    elif type(x, '*') then
>       map(ABS, x);
>    else
>       'ABS'(x);
>    end if;
> end proc:
> ABS( a*b );
```

$$\mathrm{ABS}(a)\,\mathrm{ABS}(b)$$

This feature is especially useful if some of the factors are numbers.

```
> ABS( -2*a );
```

$$2\,\mathrm{ABS}(a)$$

You may want to improve `ABS` further so that it can calculate the absolute value of a complex number.

Parameter Type Checking Sometimes when you write a procedure, you intend it to handle only a certain type of input. Calling the procedure with a different type of input may not make any sense. You can use type checking to verify that the inputs to your procedure are of the correct type. Type checking is especially important for complicated procedures as it helps you to identify mistakes early .

Consider the original implementation of `SUM`.

```
> SUM := proc(n)
>     local i, total;
>     total := 0;
>     for i from 1 to n do
>        total := total+i;
>     end do;
>     total;
> end proc:
```

Clearly, n should be an integer. If you try to use the procedure on symbolic data, it breaks.

```
> SUM("hello world");

Error, (in SUM) final value in for loop must be numeric
or character
```

The error message indicates what went wrong inside the `for` statement while trying to execute the procedure. The test in the `for` loop failed because `"hello world"` is a string, not a number, and Maple could not determine whether to execute the loop. The following implementation of `SUM` provides a much more informative error message. The `type(...,integer)` command determines whether n is an integer.

```
> SUM := proc(n)
>    local i,total;
>    if not type(n, integer) then
>       error("input must be an integer");
>    end if;
>    total := 0;
>    for i from 1 to n do  total := total+i  end do;
>    total;
> end proc:
```

Now the error message is more helpful.

```
> SUM("hello world");

Error, (in SUM) input must be an integer
```

Using `type` to check inputs is such a common task that Maple provides a simple means of declaring the type of an argument to a procedure. For example, you can rewrite the `SUM` procedure in the following manner. An informative error message helps you to find and correct a mistake quickly.

```
> SUM := proc(n::integer)
>    local i, total;
>    total := 0;
>    for i from 1 to n do  total := total+i  end do;
>    total;
> end proc:


> SUM("hello world");

Error, invalid input: SUM expects its 1st argument, n,
to be of type integer, but received hello world
```

Maple understands a large number of types. In addition, you can combine existing types algebraically to form new types, or you can define entirely new types. See `?type`.

The `while` Loop

The `while` loop is an important type of structure. It has the following structure.

```
while condition do commands end do;
```

Maple tests the *condition* and executes the *commands* inside the loop over and over again until the *condition* fails.

You can use the `while` loop to write a procedure that divides an integer n by two as many times as is possible. The `iquo` and `irem` commands calculate the quotient and remainder, respectively, using integer division.

```
> iquo( 7, 3 );
```

2

```
> irem( 7, 3 );
```

1

Thus, you can write a `divideby2` procedure in the following manner.

```
> divideby2 := proc(n::posint)
>    local q;
>    q := n;
>    while irem(q, 2) = 0 do
>       q := iquo(q, 2);
>    end do;
>    q;
> end proc:
```

Apply `divideby2` to 32 and 48.

```
> divideby2(32);
```

1

```
> divideby2(48);
```

3

The `while` and `for` loops are both special cases of a more general repetition statement; see section 4.3.

Modularization

When you write procedures, identifying subtasks and writing these as separate procedures is a good idea. Doing so makes your procedures easier to read, and you may be able to reuse some of the subtask procedures in another application.

Consider the following mathematical problem. Suppose you have a positive integer, in this case, forty.

```
> 40;
```

40

Divide the integer by two, as many times as possible; the `divideby2` procedure above does just that for you.

```
> divideby2( % );
```

5

Multiply the result by three and add one.

```
> 3*% + 1;
```

16

Again, divide by two.

```
> divideby2( % );
```

1

Multiply by three and add one.

```
> 3*% + 1;
```

4

Divide.

```
> divideby2( % );
```

1

The result is 1 again, so from now on you will get 4, 1, 4, 1, ... Mathematicians have conjectured that you always reach the number 1 in this way, no matter with which positive integer you begin. You can study this conjecture, known as *the* $3n + 1$ *conjecture*, by writing a procedure which calculates how many iterations you need to get to the number 1. The following procedure makes a single iteration.

```
> iteration := proc(n::posint)
>    local a;
>    a := 3*n + 1;
>    divideby2( a );
> end proc:
```

The `checkconjecture` procedure counts the number of iterations.

```
> checkconjecture := proc(x::posint)
>    local count, n;
>    count := 0;
>    n := divideby2(x);
>    while n>1 do
>       n := iteration(n);
>       count := count + 1;
>    end do;
>    count;
```

```
> end proc:
```

You can now check the conjecture for different values of x.

```
> checkconjecture( 40 );
```

$$1$$

```
> checkconjecture( 4387 );
```

$$49$$

You could write `checkconjecture` as one self-contained procedure without references to `iteration` or `divideby2`. But then, you would have to use nested `while` statements, thus making the procedure much harder to read.

Recursive Procedures

Just as you can write procedures that call other procedures, you can also write a procedure that calls itself. This is called *recursive programming*. As an example, consider the Fibonacci numbers, which are defined in the following procedure.

$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2,$$

where $f_0 = 0$, and $f_1 = 1$. The following procedure calculates f_n for any n.

```
> Fibonacci := proc(n::nonnegint)
>    if n<2 then
>       n;
>    else
>       Fibonacci(n-1)+Fibonacci(n-2);
>    end if;
> end proc:
```

Here is a sequence of the first sixteen Fibonacci numbers.

```
> seq( Fibonacci(i), i=0..15 );
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610$$

The `time` command tells you the number of seconds a procedure takes to execute. `Fibonacci` is not very efficient.

```
> time( Fibonacci(20) );
```

.450

The reason is that `Fibonacci` recalculates the same results over and over again. To find f_{20}, it must find f_{19} and f_{18}; to find f_{19}, it must find f_{18} again and f_{17}; and so on. One solution to this efficiency problem is to tell `Fibonacci` to remember its results. That way, `Fibonacci` only has to calculate f_{18} once. The `remember` option makes a procedure store its results in a *remember table*. Section 2.5 further discusses remember tables.

```
> Fibonacci := proc(n::nonnegint)
>    option remember;
>    if n<2 then
>       n;
>    else
>       Fibonacci(n-1)+Fibonacci(n-2);
>    end if;
> end proc:
```

This version of `Fibonacci` is much faster.

```
> time( Fibonacci(20) );
```

0.

```
> time( Fibonacci(2000) );
```

.133

If you use remember tables indiscriminately, Maple may run out of memory. You can often rewrite recursive procedures by using a loop, but recursive procedures are often easier to read. On the other hand, iterative procedures are more efficient. The procedure below is a loop version of `Fibonacci`.

```
> Fibonacci := proc(n::nonnegint)
>    local temp, fnew, fold, i;
>    if n<2 then
>       n;
>    else
>       fold := 0;
>       fnew := 1;
>       for i from 2 to n do
>          temp := fnew + fold;
>          fold := fnew;
>          fnew := temp;
>       end do;
```

```
>       fnew;
>    end if;
> end proc:


> time( Fibonacci(2000) );
```

.133

When you write recursive procedures, you must weigh the benefits of remember tables against their use of memory. Also, you must make sure that your recursion stops.

The `return` Statement A Maple procedure by default returns the result of the last computation within the procedure. You can use the `return` statement to override this behavior. In the version of `Fibonacci` below, if $n < 2$ then the procedure returns n and Maple does not execute the rest of the procedure.

```
> Fibonacci := proc(n::nonnegint)
>    option remember;
>    if n<2 then
>       return n;
>    end if;
>    Fibonacci(n-1)+Fibonacci(n-2);
> end proc:
```

Using the `return` statement can make your recursive procedures easier to read; the usually complicated code that handles the general step of the recursion does not end up inside a nested `if` statement.

Exercise

1. The Fibonacci numbers satisfy the following recurrence.

$$F(2n) = 2F(n-1)F(n) + F(n)^2 \quad \text{where } n > 1$$

and

$$F(2n+1) = F(n+1)^2 + F(n)^2 \quad \text{where } n > 1$$

Use these new relations to write a recursive Maple procedure which computes the Fibonacci numbers. How much recomputation does this procedure do?