Maple Lecture 16. Maple Procedures and Recursion

Maple procedures can take procedures as input and give procedures on return. We will also see how to work with indexed procedures. With a remember table we can make recursive procedures to run efficiently.

The material in this lecture in inspired on [2, Section 8.4]. The first example below is taken from [3, pages 75-77], see also [1, Section 3.5] for recursion and remember tables. The most recent information can be found in the Maple 9 manuals [4] and [5].

16.1 Procedures returning Procedures

Newton’s method is one of the most fundamental algorithms for approximating solutions of \( f(x) = 0 \), where the approximations are generated as follows:

\[
  x(k + 1) = x(k) - \frac{f(x(k))}{f'(x(k))}, \quad \text{for } k = 0, 1, \ldots
\]

where \( f'(x) \) is the derivative of the function \( f \).

We will make a procedure that returns the right hand side of the iteration above. First of all, we must note the difference between \( x \) and \( x \rightarrow x \): the first \( x \) is just the name \( x \), while \( x \rightarrow x \) is the function \( x \).

```maple
> newtonstep := proc(f::procedure)
>     description `returns one step with Newton's method on f`:
>     local ix:
>     ix := x -> x: # identity function
>     ix - eval(f)/D(eval(f)); # implicit return
> end proc;
```

Note that we use the \texttt{eval} in the procedure to force Maple to evaluate, because for efficiency, Maple would otherwise delay the evaluation. Let us apply this to approximate a root of \( \cos(x) = 1/2 \). First we must make a function \( g(x) = \cos(x) - 1/2 \).

\[
> g := x -> \cos(x) - 1/2; # compute root of g(x) = 0
> gstep := newtonstep(g); # create a procedure
> gstep(a); # symbolic execution
> gstep(1.4); # numerical execution
> y := 0.4: # starting value
> Digits := 32: # working precision
> for i from 1 to 7 do
>     y := gstep(y);
> end do;
\]

We know that \( \cos(\pi/3) = 1/2 \), let us thus check how accurate our result is:

\[
> \text{evalf}(y - \text{Pi}/3);
\]

16.2 Indexed Procedures

An example of an indexed procedure is the logarithm, where the base can be given as an index.

\[
> \text{interface(verbatimproc=3);}
> \text{print(log)};
\]

By default, we get the natural logarithm:

\[
> \text{log(10.0); log(exp(1));}
\]
To get the decimal logarithm, we need to provide the base 10 of the logarithm as index to the function call:

```maple
> log[10](10.0);
```

An index is just like an index in an array:

```maple
> a := A[3];
> type(a, 'indexed');
> op(a);
```

We see that we can check on whether a name is indexed or not via type and get access to the index with op.

As example, suppose \( f(t) = b + (70 - b) \cdot \exp(-0.2 \cdot t) \) models temperature in function of time with \( b \) as index. Initially, at \( t = 0 \), the temperature is 70. As \( t \) goes to infinity, the final temperature is \( b \). If \( b \) is not provided as index, take \( b = 32 \) as default.

```maple
> cool := proc(t)
> description `model of cooling temperature with index`:
> local b:
> if type(procname, 'indexed') # test if procedure has an index
> then b := op(procname): # take index as base
> else b := 32: # default value of base
> end if:
> return b + (70-b)*\exp(-0.2*t): # the general formula
> end proc;
> cool[20](1.4); cool(1.4); # test for different values of base
> cool[20](0); cool(0); # initially we are inside
> cool[20](100); cool(100); # close to outside temperature
```

We use indexed procedures to implement functions with parameters for which good default values are known. The default values may correspond to cases for which a very efficient implementation exists, whereas for other values, a general recipe needs to be applied.

### 16.3 Recursive Procedure Definitions

Many functions are defined recursively. We see how Maple has a nice mechanism to avoid superfluous recursive calls. One classical example of a recursive sequence are the Fibonacci numbers:

\[
F(0) = 0, \quad F(1) = 1, \quad \text{and} \quad F(n) = F(n-2) + F(n-1), \quad \text{for } n \geq 2.
\]

The direct way to implement this goes as follows:

```maple
> fib := proc(n::nonnegint)
> description `returns the nth Fibonacci number`:
> if n = 0 then
> return 0:
> elif n = 1 then
> return 1:
> else
> return fib(n-2)+fib(n-1):
> end if:
> end proc;
> for i from 1 to 10 do # first ten Fibonacci numbers
> fib(i);
> end do;
```

This is a very expensive way to compute the Fibonacci numbers, because of too many repetitive calls.
In Figure 1 we see the tree of procedure calls to compute $F(4)$. In general, to compute the $n$th Fibonacci number, $2^n$ calls are needed.

We will slightly modify the definition of the procedure to compute the Fibonacci numbers:

```maple
definedfib := proc(n::nonnegint)
    description 'Fibonacci with remember table':
    option remember:
    if n = 0 then
        return 0;
    elif n = 1 then
        return 1;
    else
        return definedfib(n-2) + definedfib(n-1);
    end if:
end proc;
definedfib(20);
definedfib(21);
definedfib := 1;  # introduce error in the table
end proc;
```

With the option remember, Maple has built a “remember table” for the procedure. This remember table stores the results of all calls of the procedure. Here is how we can consult this table:

```maple
eval(definedfib);
T := op(4,eval(definedfib));
eval(definedfib);
eval(T);
```

If you are curious about the “4”, do `?proc` to see where the other operands are used for. With calls to definedfib for higher numbers, we add values to the table:

```maple
definedfib(21);
eval(T);
```

Once we selected the remember table and assigned it to a variable, we can modify the table:

```maple
definedfib(20) := 1;  # introduce error in the table
eval(T);
```

We can also unassign values in the table:

```maple
eval(definedfib);
eval(T);
```
As the computation of the the 22nd Fibonacci number required the 20th, the 20th element has been recomputed and stored in the remember table:

As the computation of the the 22nd Fibonacci number required the 20th, the 20th element has been recomputed and stored in the remember table:

The command forget is used to clear the remember table of a Maple procedure. For example:

16.4 Assignments

1. Write a procedure fractional_power which returns $x^{1/n}$ for one argument $x$ and index $n$. If the index is omitted, fractional_power($x$) = $\sqrt{x}$.

2. Indices can be sequences. Write a procedure line which has one argument $x$ and up to two indices. The output of line is as follows: line[$a; b$]($x$) = $a + bx$, line[$a$]($x$) = $a_1 + a_2x$, and line($x$) = $x$.

3. The secant method to find a solution of $f(x) = 0$ is defined by

$$x_n = x_{n-1} - \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} f(x_{n-1}), \quad \text{for } n \geq 2.$$ 

While the secant method requires no derivatives, we need two points ($x_0$ and $x_1$) to start the iteration. For simplicity we will take for $x_0$ and $x_1$ a random float generated by evalf(rand()/10^12).

(a) Write a Maple procedure to implement the formula above, to execute one step of the secant method. Use the following prototype:

```
secantstep := proc(f::procedure,x0::float,x1::float);
```

Test your implementation on $f(x) = \cos(x) - 1/2 = 0$.

(b) Use secantstep to define the Maple procedure with prototype

```
secant1 := proc(f::procedure,n::nonnegint);
```

which returns $x_n$, starting from random values for $x_0$ and $x_1$.

Also here, test your implementation on $f(x) = \cos(x) - 1/2 = 0$.

(c) Write a recursive implementation for the secant method, using the prototype

```
secant2 := proc(f::procedure,n::nonnegint);
```

which also returns $x_n$, starting from random values for $x_0$ and $x_1$.

Make sure this recursive implementation is as efficient as the iterative version.

4. Execute diff(sin(x),x); and change the remember table of diff so that next time we execute diff(sin(x),x); we get sin(x) on return.

5. The Bell numbers $B(n)$ are defined by $B(0) = 1$ and $B(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} B(i)$, for $n > 0$. They count the number of partitions of a set of $n$ elements.

Write a recursive procedure to compute the Bell numbers. The binomial coefficient $\binom{n-1}{i}$ is computed by binomial(n-1,i). Make sure your procedure is efficient enough to compute $B(50)$.
6. The \( n \)-th Chebychev polynomial is also often defined as \( \cos(n \arccos(x)) \).

Give the definition of the procedure \( C \) which takes on input \( x \) and has index \( n \).

Thus \( C[n](x) \) returns \( \cos(n \arccos(x)) \) while \( C[10](0.5) \) returns the value of the 10-th Chebychev polynomial at 0.5. Compare this value with \( \text{orthopoly}[T](10, 0.5) \).

7. Let \( L[n](x) \) denote a special kind of the Laguerre polynomial of degree \( n \) in the variable \( x \).

We define \( L[n](x) \) by \( L[0](x) = 1, L[1](x) = x \), and

for any degree \( n > 1 \) : \( n*L[n](x) = (2*n-1-x)*L[n-1](x) - (n-1)*L[n-2](x) \).

Write a Maple procedure \( \text{Laguerre} \) that returns \( L[n](x) \).

Use an index for the degree \( n \) and take \( x \) as parameter in the procedure.

Make sure your procedure can compute the 50-th Laguerre polynomial.

References


