Maple Tips, Tricks and Hacks

Paul Vrbik
University of Western Ontario

November 15, 2010

Developing Large Codes Sets In Maple

- Develop in text files.
- Read these files into maple in one of two ways.

.mapleinit

A file (that you should put in your home directory) that Maple will read and execute when you run.
Useful things to put in it:

1. `kernelopts(printbytes=false):`
2. `interface(verboseproc=2):`
3. `plotsetup(maplet);` or `plotsetup(x11);`
4. `interface(imaginaryunit=I):`

List Extension

```
1  BadList := proc(n)
2  local i, LL;
3
4      LL := [];
5
6      for i from 1 to n do
7          LL := [ op(LL) ,i ];
8      end do;
9
10     return LL;
11
12 end proc:
```

```
[> TIME( BadList( 10^5 ) );
[  66.401
```

```
1  TableList := proc(n)
2  local i, LL;
3
4      LL := table();
5
6      for i from 1 to n do
7          LL[i] := i;
8      end do;
9
10     return [ seq( LL[i], i=1..n ) ];
11
12 end proc:
```

```
[> TIME( TableList( 10^5 ) );
[  0.215
```

(308 times faster).

A MAPLE list is an immutable array. So, each new list costs $O(i)$ where i is the length of the list. Put this in a loop you get

$$\sum_{i=1}^{n} O(i) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

and thus: bad list extension is quadratic.

Good list extension is linear.

Useful list shortcuts

```
[> L := [ seq( i, i=1..10) ];
[  L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[> nops( L );
[  10
[> L[-1];
[  10
[> L[3..5];
[  [3, 4, 5]
[> L[2..-1];
[  [2, 3, 4, 5, 6, 7, 8, 9, 10]
[> L[1..-2];
[  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Plus all the extra commands in `ListTools`.

Using The Right Function

```
1  BadFloor := proc( N )
2  local i, x;
3
4      for i from 1 to N do
5          x := rand();
6          floor( x );
7      end do;
8
9  end proc:
```

```
[> TIME( BadFloor( 10^7 ) );
[  73.836
```

```
1  GoodFloor := proc( N )
2  local i, x;
3      for i from 1 to N do
4          x := rand();
5          trunc( x );
6      end do;
7
8  end proc:
```

```
[> TIME( GoodFloor( 10^7 ) );
[  38.308
```

(1.9274303 times faster).

`floor` will work (or try to work) on virtually anything you give to it. `trunc` is a simple C instruction.

Others

1. `sum` versus `add`
2. $\div$ versus `quo` or `rem`

types

```
1 FooBar := proc( f::list(polynom(posint)),
2                 n::nonnegint )
3     return 0;
4 end proc:
```

```
[> FooBar( [2-x], 3);
[  Error, invalid input:  FooBar expects its 1st
argument, f, to be of type list(polynom(posint)), but
received [2-x]
[> FooBar( [2+x], -3);
[  Error, invalid input:  FooBar expects its 2nd
argument, n, to be of type nonnegint, but received -3
[> ?  type;
```

```
> f := Matrix( [[x^3 + 2*x - 4],[x - 2]] );
> type( f, Matrix[polynom] );
  true
> type( f, Matrix[polynom(integer)] );
  true
> type( f, Matrix[polynom(posint)] );
  false
```

Remember Tables

```
1  FooBar := proc( n::posint )
2  local i;
3
4      for i from 1 to 10^5 do
5          i^i;
6      end do;
7
8      FooBar(n) := n;
9
10     return n;
11
12 end proc:
```

```
[> Time( FooBar(3) );
[  22.243
[> Time( FooBar(3) );
[  0.
```

Remember Tables

```
[> FACT := n -> n * FACT(n-1);
[> FACT(5);
[  Error, (in FACT) too many levels of recursion
[> FACT(0):=1:
[> FACT(5);
[  120
```

Recursive Functions

Don't do:

```
1  pow := proc( x::integer, n::nonnegint )
2
3      if n = 0 then
4          return 1;
5      end if;
6
7      return x * pow(x, n-1);
8
9  end proc:
```

Recursive Functions

Do:

```
1  pow := proc( x::integer, n::nonnegint )
2
3      if n = 0 then
4          return 1;
5      end if;
6
7      return x * procname(x, n-1);
8
9  end proc:
```

Loop Tricks

```
for i from 1 to 10 by 3 do
    ...
end do;
```

```
for i to 10 by 3 do
    ...
end do;
```

```
for i by 3 while i<10 do
    ...
end do;
```

```
L := [1,2,3,4];
for i in L do
    ...
end do;
```

Assertions

```
1  FooBar := proc( x::integer, y::integer )
2  local a;
3
4      a := x/y;
5
6      ASSERT(type(a,integer),"y must divide x");
7
8      return a;
9  end proc:
```

```
[> FooBar(2,3):
[> kernelopts(assertlevel=1);
[> FooBar(2,3):
[  Error, (in FooBar) assertion failed, assumed y|x
```

Local Procedures

```
1  FooBar := proc()
2
3      rcMatMult:=proc(A::Matrix,B::Matrix)
4          return RegularChains:-MatrixTools
5                 :-MatrixMultiply(A,B,rc,R);
6      end proc;
7
8  end proc:
```

Get Current Memory Usage

```
1 MemUsage:=proc()
2 #returns mem usage in MB
3     return kernelopts( bytesalloc )/1024^2.;
4 end proc;
```

```
[> MemUsage();
[   1.374748230
```

Argument-less Procedures

```
1  FooBar := proc()
2      if nargs > 0 then
3          return args[1] + FooBar(args[2..-1]);
4      end if;
5
6      return 0;
7  end proc:
```

```
[> FooBar(1,2,3,4,5,6);
[  21
```

Man Page Shortcuts

`?`	`gcd`	Brings you to man page for gcd.
`??`	`gcd`	Jumps to gcd description.
`???`	`gcd`	Jumps to gcd examples.

Debugging help

`printf`

Useful for printing `printf("%a",x)` will print any "algebraic object" (really the only thing you should print).

Inline if

```
[>  IsZero := x -> `if`( x=0, true, false);
[>  IsZero(2);
[   false
[>  IsZero(0);
[   true
```

Switch to symmetric mod

```
[>  5 mod 7;
[   5
[>  `mod` := mods;
[>  5 mod 7;
[   -2
```

Questions? Challenges?