

Maple Lecture 5. Assignment and Unassignment

In this lecture we see the effects of the assignment (`:=`) and how to undo the assignment. Just as important it is in the low level programming language C to make the distinction between the value and the address of a variable, we will see how Maple distinguishes between a value of a variable and its name. The material of this lecture is inspired on [1, Section 3.1].

5.1 Verifying symbolic formulas

As example we start with solving a general cubic polynomial, where the coefficients are left as parameters:

```
[> p := x^3 + a*x^2 + b*x + c;    # a general cubic polynomial
[> sols := solve(p,x);           # have to tell Maple what we are solving for
```

Let us verify these formulas for a general choice of the parameters for which we know the roots. How could we do this? Well, we can choose three random numbers to be the roots of a polynomial. If we know the roots of a polynomial, we know its coefficients. We assign the known coefficients to the general ones and see what the formulas give us.

```
[> x1 := rand(); x2 := rand(); x3 := rand();
[> sp := (x-x1)*(x-x2)*(x-x3);    # our special polynomial
[> spe := expand(sp);             # we expand it to compute the coefficients
[> a1 := coeff(spe,x,2); b1 := coeff(spe,x,1); c1 := coeff(spe,x,0);
[> a := a1; b := b1; c := c1;
[> p;                             # this assignment has changed p
[> sols;                           # sols is sequence of three solutions
[> simplify(sols[1]);             # try to simplify first solution
```

Here we run into a limitation of the symbolic simplifier. Let us turn to numerical calculations for rescue.

```
[> evalf(sols[1]); evalf(sols[2]); evalf(sols[3]);
[> x1;x2;x3;                       # compare with the chosen roots
```

With some goodwill, we recognize something of the chosen roots in the ones predicted by the exact formulas. To increase our faith in the computations, let us increase the precision:

```
[> evalf(sols[1],30); evalf(sols[2],30); evalf(sols[3],30);
```

We see that the imaginary parts of the solutions predicted by the formulas shrink and we recognize all digits in the chosen roots. With the specialization of the coefficients, we lost our general polynomial `p`:

```
[> p;
[> a := 'a'; b := 'b'; c := 'c'; # unassign with right quotes
[> p;                             # check if we have the general polynomial back
```

The above commands illustrate the use of right quotes to unassign a variable. Another use of the right quotes will be given in the next subsection.

Notice that multiple assignments are possible:

```
[> a,b,c := 1,2,3;                 # a multiple assignment
[> a; b; c;                       # verify
```

The sequence at the right of the assignment must have the same length as the sequence at the left.

5.2 Sharing values and the side effects of the assignment

Sometimes we want variables to share the same value, and sometimes we don't. When a variable appears at the right hand side of an assignment, it is evaluated:

```
[> x := 10; y := x;
> x := 11; y;
```

On Figure 1, we see that, as x was evaluated to 10, y is independent of x .

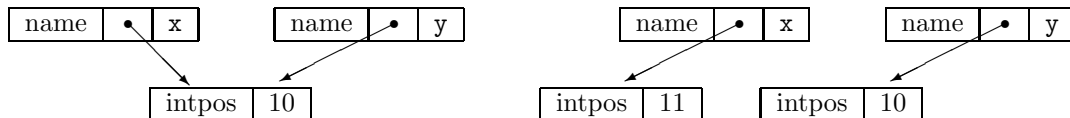


Figure 1: After $x := 10; y := x;$ the internal representation of links between variables, before and after the assignment $x := 11;$ depicted respectively at the left and the right.

Suppose we want z to be just another name for x , then we can use the right quotes (as in the unassignment):

```
[> z := 'x'; z;           # right quotes prevent evaluation
> x := 12; y; z;         # z shares the same value with x
```

As x changes, also the value of z changes, so z acts as a pointer to x , see Figure 2.

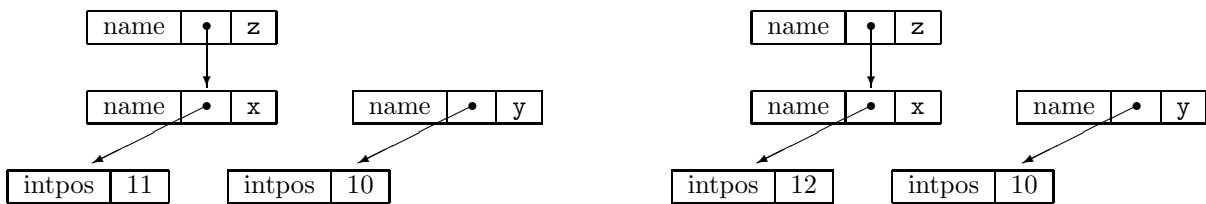


Figure 2: After $z := 'x';$ the internal representation of links between variables, before and after the assignment $x := 12;$ depicted respectively at the left and the right.

After the assignment $z := 'x';$ we can verify that z is not really a variable like x , but more like an alias to x . We can verify this with the command **addressof**, which can be seen as the analogue to the address operator $\&$ in C. The Maple analogue to dereferencing operator $*$ in C is the command **pointto**. For example:

```
[> a := addressof(x);   # shows the address of the variable x, a = &x in C
> addressof(z);         # is the same as a, after z := 'x'
> pointto(a);           # gives the expression pointed to by a, *a in C
```

The commands **addressof** and **pointto** are part of the “hackware package” in Maple. Unlike low level programming in C, you will almost never need these commands in normal usage of Maple.

In the next lecture we will see the more important **eval** command which allows us to trace the links between variables more easily.

5.3 The commands `assign`, `unassign`, and `evaln`

The `assign` and `unassign` commands have broader use than the constructions we have seen before. We can first store an equation and then turn the equation into an assignment, when all values for the

```
[> formula := balance = deposit*(1+interest_rate);
[> deposit := 1024;
[> interest_rate := 0.04;
[> formula;                               # shows what the balance will be
[> balance;                               # but balance still has no value
```

If we are satisfied with our balance, we can assign:

```
[> assign(formula);                       # assign rhs(formula) to balance
[> balance;                               # shows the value of balance
```

One example of the `assign` is to give formal parameters a solution value:

```
[> eqs := {y1+y2 = p1, y1 + p1*y2 = p2}; # linear system with parameters
[> vars := {y1,y2};
[> sols := solve(eqs,vars);              # solve for y1 and y2
```

Note that again we have to be cautious when using these formulas. But let us move on:

```
[> assign(sols);                          # turns the "=" into "=="
[> y1; y2;
[> unassign('y1','y2');                   # observe the right quotes!
[> y1; y2;
```

The `assign` and `unassign` come in handy when the number of variables is huge, or not a priori known when used in Maple loops. The third possibility to undo an assignment is to work with `evaln` (evaluate to a name):

```
[> assign(sols);
[> y1 := evaln(y1); y2 := evaln(y2);
[> y1; y2;
```

To prevent programming mistakes like forgetting to initialize a variable, or overwriting the current value of a variable, Maple has the command `assigned` which returns true if the variable has been assigned a value.

In case you forgot which names that are currently assigned a value, we can use the command `anames`:

```
[> anames(user);      # variables currently assigned by user
[> anames(integer);  # variables assigned to integer values
[> anames();         # all assigned variables
```

The command `unames()` returns all unassigned variables currently in use in a Maple session.

5.4 Assignments

1. Suppose we have a guess for the solution of an equation, and we first want to try our guess, before asking Maple to solve the equation:

```
[> restart;
[> equation := x^2 - 2 = 0;
[> x := 1.4;
[> equation;
[> solve(equation);
```

Explain what went wrong. What is the command we must execute just before `solve(equation)` to see the exact solutions to this equation?

2. Consider the following sequence of commands:

```
[> restart;
[> x := y;
[> y := x + 1;
```

Maple will complain about this recursive assignment. How can you modify the last statement to avoid the error? Explain the solution.

3. Do the following experiment:

```
[> restart;
[> a := b; a := 3; a; b;
[> x := y; assign(x,3); x; y;
```

Compare the values of `a` with `b` and `x` with `y`. Do you observe a difference?

Describe the difference in the outcome of `assign(x,3)`, compared to `x := 3`.

Consult the online help on `assign`, i.e.: do `?assign` and read the description. Does the online help give you an explanation for what happened?

4. Execute the following commands:

```
[> restart;
[> v[1] := 1: v[2] := 2: v[3] := 3:
[> for i from 1 to 3 do v[i] := 'v[i]'; end do;
[> v[1]; v[2]; v[3];
```

Explain the outcome of these commands.

Why does this loop not unassign the variables `v[1]`, `v[2]`, and `v[3]`?

Find one alternative *loop* to unassign the variables `v[1]`, `v[2]`, and `v[3]`.

5. Explain the difference between `f := sin;` and `alias(f=sin)`. Which instruction is a sin? To make the word pun explicit, indicate which instruction is the preferred way to express in Maple the mathematical statement “Let f be the sine function.”
6. Show how to evaluate all variables which the user has assigned to floating-point numbers with 3 decimal places. Illustrate beginning with `restart; a,b,c := Pi,exp(1),gamma;`. You must use `anames`.

References

- [1] A. Heck. *Introduction to Maple*. Springer-Verlag, third edition, 2003.