Local Variables in *Mathematica*

Bruce Smith, Wolfram Research, 29 May 92

The Purpose of Local Variables: Modularity

It is desirable for separate parts of the program to be independently understandable. This requires preventing name interference.

Contexts can prevent the names in different pieces of code from interfering, but it is not convenient to use a new context for every function, and contexts don't provide correct lexical scoping in all cases (for example, when a function is used to process its own code; examples are shown below).

Two Kinds of Locality: Lexical and Dynamic

Locality to a piece of code: "lexical scoping", or local names

Example using `Module` to separate a local variable and a mathematical variable

```
sqm[ym_] := Module[{x = 2}, ym ^ x]
```

```
x + sqm[x]
```

```
x + x^2
```

Lexical scoping is used to avoid interference between different pieces of code;

Had we used **`Block`** instead, we would get the wrong answer because the *x*'s would interfere:

```
sqb[yb_] := Block[{x = 2}, yb ^ x]
```

```
x + sqb[x]
```

```
4 + x
```

... and to avoid interference between different uses of the same code; for example, when the code must create independent arbitrary constants:

```
Attributes[AntiDerivative] = HoldAll;
AntiDerivative[form_, var_] :=
    Module[{constant},
        Integrate[form, var] + constant
    ]
```

```
AntiDerivative[x, x] + AntiDerivative[x^2,x]
```

$$\text{constant\$11} + \text{constant\$12} + \frac{x^2}{2} + \frac{x^3}{3}$$

(There is a more practical example of this technique in The *Mathematica* Book.)

☐ **Pure functions use lexical scoping to allow correct substitution within them:**

```
adder = Function[{y}, Function[{x}, x + y]]
```

```
Function[{y}, Function[{x}, x + y]]
```

```
adder[2]
```

```
Function[{x$}, x$ + 2]
```

```
adder[x]
```

```
Function[{x$}, x$ + x]
```

☐ **Other lexical scoping constructs:**

○ **With**

For local constants. Uses replacement. In some languages, called "**let**".

○ **Replacement rules**

○ **Condition (/;) used as a pattern object**

○ **Definitions (Set, Unset, etc.)**

☐ **Implemented with lexicalized variable names and $ModuleNumber.**

```
$ModuleNumber
```

```
14
```

```
Module[{x},x]
```

```
x$14
```

```
Module[{x},x]
```

```
x$15
```

```
$ModuleNumber = 345432463254687631618
```

```
345432463254687631618
```

```
Module[{x},x]
```

```
x$345432463254687631618
```

```
Module[{x},x]
```
```
x$345432463254687631619
```

☐ **Related objects: `$SessionID`, `Unique`.**

```
$SessionID
```
```
19111471782301817326
```

```
Unique[x]
```
```
x$345432463254687631621
```

```
Unique["yy"]
```
```
yy2
```

☐ **(Exact scope of local symbols in each scoping construct:)**

```
  With[{sym1 = val1, sym2 = val2, ...}, body]
Module[{sym1 = val1, sym2 = val2, ...}, body]

(* scope of sym2: body (not symi or vali) *)

(* scope of an external copy of sym2:
   vali, but not vari or body *)


f[lhs1, x_, lhs2] -> rhs
f[lhs1, x_, lhs2] :> rhs
(* and similarly for Set, Unset, all relatives *)

(* scope of x: rhs, but not lhs1 or lhs2 (or f) *)

(* scope of an external copy of x:
   lhs1 or lhs2 (or f), but not rhs *)


f[lhs1, x_, lhs2] /; cond[x]  (* when a pattern *)

(* scope of x: cond[x], but not lhs1 or lhs2 (or f)*)

(* scope of an external copy of x:
   lhs1 or lhs2 (or f), but not cond[x] *)
```

☐ **(Efficiency)**

`Module` is slightly slower than **`Block`**, since it does complicated substitution on its body.

`With` is faster than **`Module`** but slower than **`Block`**.

These effects are usually less important than the reduced likelihood of bugs compared with the indiscriminate use of **Block**.

■ Locality to a portion of the execution (*i.e.*, to a certain period of time in one process): "dynamic scoping", or local values

☐ **Used to provide some form of "environment" to executing code:**

☐ **... either a global parameter,**

```
Block[{$RecursionLimit = 20},
    Nest[f,x,21] /. f -> Plus ]
```

```
$RecursionLimit::reclim: Recursion depth of 20 exceeded.
```

```
Hold[Plus[Plus[Plus[x]]]]
```

```
$RecursionLimit
```

```
256
```

```
z = 2 w;
Table[z, {w,1,4}]
```

```
{2, 4, 6, 8}
```

☐ **... or a place to collect information. (unfinished example)**

```
TraceCount::usage =
"TraceCount[expr_] gives the number of intermediate
expressions used to evaluate expr, not including any
that are used in enclosed TraceCount[] constructs.";
```

```
TraceCountEnclosed = 0;
Attributes[TraceCount] = HoldAll;
TraceCount[expr_] :=
  Block[{TraceCountEnclosed = 0},
    {TraceScan[TraceCountEnclosed += 1 &, expr],
     TraceCountEnclosed}
  ]
```

☐ **(Iterators are dynamic, even when they're not simple symbols)**

```
i1to3 = {i[1], i[2], i[3]}
Table[i1to3, {i[2],10,12}]
```

```
{i[1], i[2], i[3]}
```

```
{{i[1], 10, i[3]}, {i[1], 11, i[3]}, {i[1], 12, i[3]}}
```

Copyright

Mathematica is a registered trademark of Wolfram Research, Inc.

All other product names mentioned are trademarks of their producers.

Copyright 1992 by Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.