Mathematica Programming

Dennis Silverman
Mathematical Physics 212 B
U . C . Irvine

■ Built−in Programming

- **Mathematica already has several important built – in programming capabilities over and above standard programming languages .**
- **Graphics is immediately done without having to first output data and then entering a graphics program .**
- **Matrix and vector algebra are built in, saving the usual multiplicative loops .**
- **Functions are easily defined in place .**
- **Standard functions are automatically calculated without adding subroutine links or encoding interpolating functions .**
- **Complicated algebra and complex numbers are automatically handled .**
- **Variables do not always have to be typed or dimensioned .**
- **Differential equations are directly solved numerically without direct programming .**
- **Tables of functions can be directly calculated without having to write loops .**

■ Input and Output of Data

```
? OpenWrite
```

```
OpenWrite["file"] opens a file to write output to it, and returns an OutputStream object.
```

```
? OpenAppend
```

```
OpenAppend["file"] opens a file to append output to it, and returns an OutputStream object.
```

```
outstream = OpenWrite["temp"]
```

```
OutputStream[temp, 21]
```

```
? Write
```

```
Write[channel, expr1, expr2, ... ] writes the expressions expri in sequence, followed
  by a newline, to the specified output channel.
```

```
Write[outstream, 1]
```

```
Write[outstream, 2]
```

```
Write[outstream, 3]

Write[outstream, 4]

Close[outstream]

temp

!!temp

1
2
3
4

readin = OpenRead["temp"]

InputStream[temp, 22]

?Read

Read[stream] reads one expression from an input stream, and returns the expression.
  Read[stream, type] reads one object of the specified type. Read[stream, {type1,
  type2, ... }] reads a sequence of objects of the specified types.

Read[readin, Number]

1

Read[readin, Number]

2

a = ReadList["temp", Table[Number, {2}]]

{{1, 2}, {3, 4}}

Close[readin]

temp

Close["temp"]

temp
```

Type assignments for data are : Byte, Character, Real, Number, Word, Record, String, Expresssion, and Hold[Expression] .

■ Assignments in Loops

```
i++  increment i by 1
i--  decrement i by 1
++i  pre - increment i
--i  pre - decrement i
i += di  add di to i
i -= di  subtract di from i
x *= c   multiply x by c
x /= c   divide x by c
```

■ Loops

```
?Do
```

```
Do[expr, {imax}] evaluates expr imax times. Do[expr, {i, imax}] evaluates expr
  with the variable i successively taking on the values 1 through imax (in steps
  of 1). Do[expr, {i, imin, imax}] starts with i = imin. Do[expr, {i, imin, imax,
  di}] uses steps di. Do[expr, {i, imin, imax}, {j, jmin, jmax}, ... ] evaluates
  expr looping over different values of j, etc. for each i.
```

```
Do[Print[i], {i, 0, 6, 2}]
```

```
0

2

4

6
```

For a nested loop :

```
Do[Print[{i, j}], {i, 3}, {j, 3}]
```

```
{1, 1}

{1, 2}

{1, 3}

{2, 1}

{2, 2}

{2, 3}

{3, 1}

{3, 2}

{3, 3}
```

■ Testing Loops

```
?While
```

```
While[test, body] evaluates test, then body, repetitively, until test first fails to give True.
```

```
n = 10; While[(n = n - 1) > 5, Print[n]]
```

```
9

8

7

6
```

? For

```
For[start, test, incr, body] executes start, then repeatedly evaluates body and
  incr until test fails to give True.
```

```
For[i = 1, i < 4, i++, Print[i^2]]
```

```
1

4

9
```

start and body can be multiple statements separated by semicolons. Semicolons separate statements that are executed without displaying the results.

■ Transfers

? Label

```
Label[tag] represents a point in a compound expression to which control can be
  transferred using Goto.
```

? Goto

```
Goto[tag] scans for Label[tag], and transfers control to that point.
```

```
q = 2; Label[begin]; Print[q]; q += 1; If[q < 6, Goto[begin]]
```

```
General::spell1 :
 Possible spelling error: new symbol name "begin" is similar to existing symbol "Begin".

2

3

4

5
```

? Break

```
Break[ ] exits the nearest enclosing Do, For or While.
```

? Continue

```
Continue[ ] exits to the nearest enclosing Do, For or While in a procedural program.
```

? Return

```
Return[expr] returns the value expr from a function. Return[ ] returns the value Null.
```

■ If Statements

? If

```
If[condition, t, f] gives t if condition evaluates to True, and f if it evaluates
  to False. If[condition, t, f, u] gives u if condition evaluates to neither True nor False.
```

Or you can regard this in Fortran as :
If[test, then, else] .

```
Do[Print[i]; If[i > 5, Break[], Continue[]], {i, 10}]
```

```
1

2

3

4

5

6
```

For nested if statements :

```
Do[If[i > 2, If[i == 3, Print[i], Print[10 i]], Print[-i]], {i, 5}]
```

```
-1

-2

3

40

50
```

The multiple if transfer statement is :

```
? Which
```

```
Which[test1, value1, test2, value2, ... ] evaluates each of the testi in turn,
  returning the value of the valuei corresponding to the first one that yields True.
```

```
i = 4
```

```
4
```

```
Which[i < 1, out = 0, i < 4, out = 1, i < 10, out = 2]; out
```

```
2
```

■ Compile

```
? Compile
```

```
Compile[{x1, x2, ... }, expr] creates a compiled function which evaluates expr assuming
  numerical values of the xi. Compile[{{x1, t1}, ... }, expr] assumes that xi is of a type
  which matches ti. Compile[{{x1, t1, n1}, ... }, expr] assumes that xi is a rank
  ni array of objects each of a type which matches ti. Compile[vars, expr, {{p1,
  pt1}, ... }] assumes that subexpressions in expr which match pi are of types which match pti.
```

■ Modules

```
?Module
```

```
Module[{x, y, ... }, expr] specifies that occurrences of the symbols x, y, ...  in expr
  should be treated as local. Module[{x = x0, ... }, expr] defines initial values for x, ... .
```