Reprint from the *Mathematica* Conference

June, 1992 Boston, MA

744.9.10.1992

Mathematica Internals

David Withoff

Mathematica systems typically consist of a “front end”, which handles interaction with the user, and a “kernel’, which processes input, performs computations, and sends results back to the front end. The kernel and the front end are separate programs, and each can be used without of the other. For practical reasons the front end is a relatively machine-dependent program, designed to take advantage of user-interface features available on particular computers. In contrast, most of the functions handled by the kernel, such as mathematical computations, are by nature independent of the computer on which the kernel is being run.

These notes focus on the *Mathematica* kernel.

Section 1 summarizes kernel initialization and the internal main loop, which controls input, output, and evaluation. Section 2 describes the internal structure of expressions, the basic data objects used throughout the program. Section 3 describes the evaluation process. Input and output is described in section 4.

1 Initialization and the Main Loop

■ 1.1 Kernel Initialization

When the *Mathematica* kernel is first started, a number of initialization tasks are performed before the program enters the main loop to begin handling regular input.

Although most initialization tasks are functionally identical on all computers, some depend on installation options, or otherwise vary from one system to the next. As a result of these differences, it is likely that not all of the initialization steps in the list below will apply to your system. For example, if command line options are not part of the operating system on the host computer, an equivalent effect is achieved by other means when needed. Initialization is also somewhat different if the kernel is be run from a “notebook” front end or through *MathLink*.

- Create the symbols for `List`, `String`, and `Symbol`.
- Create the main symbol table, which is itself a *Mathematica* list.
- Create the standard contexts ``System` `` and ``Global` ``.
- Insert the symbols for `List`, `String`, and `Symbol` into the symbol table.
- Create special contexts, such as ``Integrate` ``.
- Create built-in symbols, and attach any initial attributes and internally defined transformation rules to each symbol.
- Initialize dispatch tables, global variables, and other special structures used by internal algorithms.
- Open the streams `$Output`, `$Messages`, and `$Urgent`.
- Check the password.
- Evaluate expressions from the `-run` command line option.
- If the kernel was created using `Dump`, evaluate the appropriate startup expression (second argument of `Dump`). Otherwise, load `sysinit.m`.
- Load `init.m`.
- Enter the main loop and wait for input.

1.2 Package Autoloading

Packages that define *Mathematica* functions can be loaded either during initialization or when the corresponding functions are needed. Loading during initialization has a number of advantages, such as making the corresponding function immediately available, but has the disadvantage of consuming a great deal of memory. A common solution to this dilemma is arrange for appropriate packages to be loaded automatically when particular expressions are first evaluated.

Package autoloading is typically set up during initialization. A simple way to do this is to introduce rules that erase themselves after loading specified packages. When an expression matching one of these rules is evaluated, application of the rule has the side effect of loading additional rules. The new rules are then used in evaluating the original expression.

The so-called "thin" kernel uses just such a system as a memory saving device. In the thin kernel, automatic loading of packages is implemented through the function `AutoLoad`.

`AutoLoad[`*expr*`, ` *sym*`, ` *file*`]`	evaluate *expr* after clearing *sym* and loading *file*
`SystemStub[`*sym*`, ` *file*`]`	create a rule to load *file* automatically when *sym*`[… ]` is evaluated.

Automatic package loading in the thin kernel.

```
SetAttributes[ AutoLoad, HoldFirst ]

AutoLoad[expr_, {sym__}, file_]:=
    Module[{protected},
        protected = Unprotect[sym] ;
        Clear[sym]; (* clear the AutoLoad definition itself *)
        Protect[Evaluate[protected]];
        If[ $AutoLoad === False,
                Message[ General::autoload, Head[expr], file],
                AbortProtect[Get[file]]];
        expr
    ]
```

Definition of the `AutoLoad` function.

The function `SystemStub` is provided as a utility for attaching `AutoLoad` rules to symbols.

```
In[1]:= SystemStub[f, "pkg.m"]
```

Clear the `ReadProtected` attribute from the new symbol.

```
In[2]:= Unprotect[f]; ClearAttributes[f, ReadProtected]
```

Here is the `AutoLoad` rule attached to `f`. The package `pkg.m` will be loaded by evaluation of any expression that matches the pattern `f[___]`. The context of the symbol `AutoLoad` is `` System`Private` ``.

```
In[3]:= ??f

Global`f

System`Private`t$:f[___] :=
  System`Private`AutoLoad[System`Private`t$, {f}, "pkg.m"]
```

The thin kernel is established by loading a file that evaluates `SystemStub[`*sym*`, ` *file*`]` for the desired symbols and packages, rather than by loading the packages themselves. Infinite recursion during evaluation of `AutoLoad[`*expr*`, ` *sym*`, ` *file*`]` is prevented by clearing rules attached to *sym*, and by the `HoldFirst` attribute. Since *expr* is normally the original expression, it is important that it not be evaluated again as a function argument until after the `AutoLoad` rule is cleared.

1.3 Kernels Created Using Dump

The most significant difference between initialization on different systems is related to the Dump facility, which allows you to save the current state of *Mathematica* as an executable binary file. If the kernel was created using Dump, many of the initialization steps can be skipped entirely, since these steps will already have been completed. The Dump command is not available on all computers.

1.4 The Main Loop

The main loop is entered after initialization, and is executed repeatedly with each new input until the program terminates. The three basic operations controlled from the main loop are input processing, evaluation of expressions, and output formatting. The main loop also handles a number of utility tasks, such as incrementing the line number.

The list below summarizes the operations that are controlled from the default main loop in the kernel.

- Set $MessageList to the empty list.
- Initialize $Line to zero.
- Increment the line number.
- Construct the input prompt.
- Evaluate the stream $Output.
- On some front ends, request input by writing the input prompt to $Output.
- Get an input expression.
- Determine if the input represents the end of a file, and terminate if appropriate.
- Evaluate $Echo and write the input string to the corresponding stream.
- Assign the input string to InString[*n*] and the input expression to In[*n*].
- Determine if the input expression is Exit or Quit, and terminate if appropriate.
- Apply the value of $Pre to the input expression.
- Call the main evaluator.
- Apply the value of $Post, if any.
- Save the evaluated expression in a temporary variable, for later assignment to Out[*n*].
- Evaluate $MessageList, assign the result to MessageList[*n*], and clear $MessageList.
- Apply the value of $PrePrint, if any.
- Construct the output prompt, if needed, and report the output.
- Assign the previously saved result as the value of Out[*n*].
- Wait for more input.

2 Expressions

2.1 Information Contained in Expressions

The basic data objects used throughout *Mathematica* are called "expressions". Expressions can be classifed as either "raw" expressions or "normal" expressions. Symbols, numbers, and strings are raw expressions. Normal expressions, such as `f[x]` and `Plus[2, 2]`, are constructed from raw expressions.

Expressions are represented internally using `struct` and `union` in the C programming language. The exact internal structure of an expression depends on whether the expression is a normal expression, a symbol, a number, or a string. The internal structure of numbers further depends on type of number being represented. The *Mathematica* expression for a machine-size integer, for example, contains a C representation of the same integer, as a C `int`, and the *Mathematica* expression for a machine-size real number contains a C `double`. Normal expressions are represented as C arrays, in which each element is another expression.

All expressions include a "reference count", which is used for memory management. The reference count of an expression keeps track of the number of references to the expression. When the number of references drops to zero, the expression is removed and the memory associated with the expression is made available for use by the rest of the system.

The internal structure of symbols and normal expressions also includes information that keeps track of the time of the last evaluation. An expression that has been evaluated once will not normally be evaluated again unless some part of the expression is modified. The system that implements this behavior is based on marking each expression with the time of the last evaluation. Strings and numbers do not require this information since they always evaluate to themselves.

2.2 Symbols

The general strategy for evaluating expressions is to iteratively apply transformation rules until the expression no longer changes. Of particular interest are attributes and transformation rules "attached" to symbols. Symbols are the only expressions to which general transformation rules can be attached. As a result, most of the transformation rules used in evaluating an expression are determined by looking at the symbols in the expression.

`DownValues[`*sym*`]`	rules for evaluating expressions of the form *sym*`[{...}]`
`SubValues[`*sym*`]`	rules for evaluating expressions such as *sym*`[... ][... ]` with a "symbolic head" of *sym*
`OwnValues[`*sym*`]`	a rule for evaluating *sym* itself
`UpValues[`*sym*`]`	rules for evaluating expressions such as *f*`[... `*sym*` ... ]`, where *sym* appears as an argument or the head of an argument
`FormatValues[`*sym*`]`	printing rules for *sym*
`NValues[`*sym*`]`	rules used in evaluating `N[... `*sym*` ... ]`
`DefaultValues[`*sym*`]`	default values for arguments in *sym*`[... ]`
`Options[`*sym*`]`	default options attached to *sym*
`Messages[`*sym*`]`	messages attached to *sym*
`Attributes[`*sym*`]`	attributes associated with *sym*

Visible information attached to symbols.

down code	internal rules analogous to `DownValues`
up code	internal rules analogous to `UpValues`
sub code	internal rules analogous to `SubValues`
num code	internal rules analogous to `NValues`
format code	internal rules analogous to `FormatValues`

Internal information attached to symbols.

Transformation rules can be defined in internal code or stored as *Mathematica* expressions. Rules stored as *Mathematica* expressions are attached to symbols as a side effect of evaluating assignments. The assignment `x = 5`, for example, attaches the rule `{Literal[x] :> 5}` to the symbol `x`. This particular rule would be used during evaluation of the symbol `x`. Rules defined in internal code are attached to symbols at initialization time, and cannot normally be modified by the user.

Several classes of transformation rules are stored as part of the internal structure of each symbol. The rule `{Literal[x] :> 5}` generated by the assignment `x = 5` is called the "own value" of the symbol `x`. The assignment `f[x_] := x^2` attaches the rule `Literal[f[x_]] :> x^2`, called a "down value", to the symbol `f`. A corresponding rule defined in internal code is called the "down code" of the symbol. Other classes of transformation rules include `UpValues`, `SubValues`, `NValues`, and `FormatValues`. The internal code corresponding to `UpValues` is called "up code", the internal code corresponding to `SubValues` is called "sub code", the internal code corresponding to `NValues` is called "num code", and the internal code corresponding to `FormatValues` is called "format code".

In addition to transformation rules, the internal structure of each symbol includes `Messages`, `DefaultValues`, `Options`, and `Attributes`. `Messages`, `DefaultValues`, and `Options` are stored as *Mathematica* expressions, much like transformation rules. `Messages[`*sym*`]` gives the list of messages attached to *sym*. `DefaultValues[`*sym*`]` gives default values to use for arguments in an expression of the form *sym*`[... ]`. `Options[`*sym*`]` gives a list of rules that is used by convention as a list of named function arguments. `Attributes` are flags used by the main evaluator and other parts of the program to determine how certain expressions containing the symbol should be handled.

3 Evaluation

3.1 The Evaluation Process

The list below summarizes the steps used in evaluating expressions. If the expression changes at any point, the process usually starts again from the beginning with the new expression.

Raw expressions are handled in the first few steps of evaluation. The remaining steps are meaningful only for normal expressions.

If the expression being evaluated is a normal expression, evaluation is controlled in part by the head of the expression and the heads of normal subexpressions. If the head of a normal expression is a symbol, transformation rules attached to that symbol are used in evaluating the expression. If the head is not a symbol, but rather a normal expression, the head of the head is used, or the head of the head of the head, and so forth until the result is a symbol. For example the "symbolic head" used in evaluating `f[1][x]` would be the symbol `f`.

- If the expression is a string or a number, return the expression.
- If the expression is a symbol with no `OwnValues`, return the expression.
- Evaluate symbols with `OwnValues`.
- If no part of the expression has changed since the last evaluation, return the expression.
- Evaluate the head h of the expression. Evaluate elements e_i of the expression as appropriate.
- If h has attributes `HoldFirst`, `HoldRest`, or `HoldAll`, leave the corresponding e_i unchanged. If h is not a symbol, no attributes will be found.
- Evaluate arguments with a head of `Evaluate`.
- If an element e_i has a head of `Unevaluated`, replace it with the arguments of the element, and keep a record of the original expression.
- If h has attribute `Flat`, flatten out nested expressions with head h.
- Flatten out nested expressions with head `Sequence`.
- If h has attribute `Listable`, then thread over any e_i that are lists.
- If h has attribute `Orderless`, then sort the e_i.
- Apply user-defined `UpValues` attached to the symbolic heads of the e_i.
- Apply internally defined `UpValues` (up code) attached to the symbolic heads of the e_i.
- Apply user-defined `DownValues` if the head of the expression is a symbol. If the head is not a symbol, apply user-defined `SubValues` attached to the symbolic head of the expression.
- Apply internally defined `DownValues` (down code) if the head of the expression is a symbol. If the head is not a symbol, apply sub code attached to the symbolic head of the expression.
- Restore the head `Unevaluated` if no applicable rules were found.
- Discard the head `Return`, if present, for expressions generated through application of user-defined rules.

4 Input and Output

4.1 Input to the Main Loop

The input processing code is accessed from the main loop to translate input from keyboards, front ends, "batch" files, and *MathLink* connections into the corresponding *Mathematica* expressions. The same code is also called through the main evaluator if the expression being evaluated contains functions such as `Get`, `Input`, or `ToExpression`.

Input to the kernel is generally given as a sequence of characters, with the exception of input from a *MathLink* connection, which is sent as a tokenized expression. The items in the list below summarize the steps used in processing character input. Most of these steps are simplified or eliminated if the input is received as a tokenized *MathLink* expression.

- Display an input prompt, or otherwise request input.
- Read the input into an array of characters.
- Handle information requests (`?`*symbol* and `??`*symbol*) and external commands (`!`*command* and `!!`*file*).
- Detect simple syntax errors, such as misplaced commas. If an error is found, generate a warning message and apply `$SyntaxHandler`.
- Read additional input until there are no non-matching parentheses, brackets, double quote marks (`"`), or other delimiters. For mismatched delimiters, generate a message and apply `$SyntaxHandler`.
- Apply the value of `$PreRead`, if any, to the input string.
- Try to construct a *Mathematica* expression from the input.
- If a syntax error is found, generate a warning message and apply `$SyntaxHandler`.
- If the input represents an incomplete expression, request additional input.
- If the input represents a complete expression, return the expression.

4.2 Input from a Text-Based Interface

The text-based interface provided with certain versions of *Mathematica* is basically a primitive front-end. Input is entered from a keyboard and read into the kernel using functions provided by the operating system. Each input is treated as a single expression. A new input prompt is displayed when *Mathematica* is ready for more input.

The initial input prompt is generated in the main loop and displayed through the stream `$Output`, which normally corresponds to the computer screen. The input prompt is also passed to the input processing code where it is used in requesting additional input.

Requests for additional input are indicated by redisplaying the input prompt, by indentation equal to the width of the input prompt, or by displaying partial input. All such requests are displayed through the stream `$Urgent`.

4.3 Input from a Notebook Interface

Input from a notebook front end is transmitted to the kernel in units called "cells". Each cell can contain one or more expressions to be evaluated. If a cell contains more than one expression, the individual expressions are read and evaluated sequentially. A number of details, such as generation of input prompts, are handled by the front end.

Messages associated with syntax errors are sent to the front end just like any other messages. If the input represents an incomplete expression, the kernel reports the position at which the expression was found to be incomplete, and the front end normally moves the cursor to this position.

Information requests entered as `?`*symbol* or `??`*symbol* give information messages that are sent back to the front end. The input characters associated with the request are subsequently processed as an equivalent number of spaces.

■ 4.4 Batch Input

On some computers *Mathematica* can be used in "batch" mode, with input fed directly to the kernel from a text file.

In batch mode, input and output prompts are suppressed. As with input from a front end, information requests entered using `?`*symbol* or `??`*symbol* are treated as an equivalent number of spaces after the request has been processed. The program terminates when it reaches the end of the input file.

■ 4.5 Functions That Use Input Processing Code

Packages are loaded into *Mathematica* using `Get`, `Needs`, `BeginPackage`, and related functions, all of which call `Get`. A package loaded using `Get` is read into an array of characters and scanned to find the first complete expression. After the first expression is evaluated, the next expression is read and evaluated, and the process is repeated until the end of the file. If package loading terminates due to a syntax error, an interrupt signal, or for some other reason, all remaining input is discarded.

The functions `ToExpression` and `ToHeldExpression` use the input processing code to translate strings into equivalent expressions. In contrast with reading of packages, each argument string is read and converted to an expression before any part of the expression is evaluated. Complete expressions separated by newline characters are treated as compound expressions.

`SyntaxLength` performs exactly the same operations as `ToExpression`, except that it returns the position of the first syntax error rather than the expression. `SyntaxQ` also uses the same internal code, and returns `True` or `False` depending on whether or not the string contains a syntax error.

■ 4.6 Types of Output

Probably the most familiar type of *Mathematica* output is the textual output displayed on a computer screen or written to an external file. As with input, textual output is handled through objects known as "streams". General characteristics of the output, such as page size, are taken from the properties of the stream.

Graphics is another common type of output. Graphical output is produced as a side-effect of evaluating expressions such as plotting functions.

A third type of output is "structured" output, most notably the tokenized expressions associated with *MathLink*. Unlike "unstructured" textual output, structured output is optimized for communication with other computer programs, rather than for reading by the user.

4.7 Formatting Textual Output

The output formatting code is called with both the expression to be formatted and the output stream to which formatted output will be written. After setting various parameters derived from the options of the output stream, the internal formatting functions are called recursively to format the expression.

As with evaluation, output formatting is controlled by rules attached to symbols. If the expression being formatted is a symbol, the `FormatValues` for the symbol, if present, are applied first, followed by internally defined formatting rules. For all other raw expressions and for normal expressions, the rules attached to the symbolic head of the expression are used.

The steps listed below are used in formatting textual output.

- Set page characteristics based on the `PageWidth`, `PageHeight`, `TotalWidth`, and `TotalHeight` options of the output stream.
- Set the global variable `$StringConversion` to the value of the `StringConversion` option of the output stream.
- Determine the format type. The initial format type is taken from the `FormatType` option of the output stream, and is required to be an element of the `$PrintForms` list.
- Apply any user-defined formatting rules (`FormatValues`) attached to the symbolic head of the expression, and format the result.
- If there are no applicable `FormatValues`, use internally defined formatting rules (format code).
- Apply the value of `$StringConversion`, if any, in each internal formatting function.
- Perform any required line-breaking on the formatted result.
- Send the result to the output stream.