You can prevent a startup or shutdown macro from executing when you start or quit ParaSheet by renaming it.

Setting a Local Startup or Shutdown Macro

You can also tell ParaSheet to execute one macro automatically each time a particular spreadsheet is opened or closed. This kind of startup or closing macro must be a local macro. To make a local macro the startup or shutdown macro for a spreadsheet, just change its name to STARTUP or SHUTDOWN (note that you must use all uppercase letters), following the steps listed in the "Setting a Global Startup or Shutdown Macro" section above.

ParaSheet runs startup and shutdown macros in the following order: global startup, local startup, local shutdown, global shutdown.

After you delete a macro, its name no longer appears in the list of available macros.

Deleting a macro cannot be undone.

Changing the Macro Type

Once you've made a macro global, or local to a particular spreadsheet, you can still change your mind. For example, suppose you've created a local macro for one spreadsheet that consists of some formatting commands. As you work with it, you realize that it might be useful in other spreadsheets as well. Using the *Copy* and *Paste* commands in the Edit menu, you can easily copy a macro from one macro window to another and change its type from local to global or global to local and then delete the original. For more information, see the "Copying a Macro" section page 42.

Setting a Global Startup or Shutdown Macro

You might want a particular macro to execute automatically every time you start or quit ParaSheet. Such a macros are called *global startup* or *global shutdown* macros.

To set a global startup or shutdown macro:

- Open the macro, as described in the "Opening a Macro" section on page 37
- If necessary, choose *Macro Panel* from the Macro menu
- Choose *Global* from the pop-up list
- Select the macro
- In the Macro name field, replace the macro's name with the name STARTUP if you want the macro to execute automatically when you start ParaSheet, or SHUTDOWN if you want the macro to execute automatically when you quit ParaSheet. (Note that you must use all uppercase letters.)
- Click *OK*

Now when you start ParaSheet, the sequence of commands in the global macro named STARTUP will execute automatically; when you quit, the sequence of commands in the global macro named SHUTDOWN will execute automatically.

- Decide whether you want the copy to be a local or global macro, then choose *Local* or *Global* from the pop-up list, as shown below.

- Create a new macro using the *New* button
- If you want, give the copy a new name as shown below

- Click *Open* to open the new macro window
- Select the macro window containing the macro you want to copy, then select the macro instructions to copy. (Don't select the macro opening =**Macro{** or macro terminator **}**.)
- Choose *Copy* from the Edit menu
- Select the macro window to hold the copy, place the insertion point after the macro opening =**Macro{** then choose *Paste* from the Edit menu.

Deleting a Macro

You might want to delete a macro. Perhaps you've found that you no longer need it or that you have an unnecessary extra copy of it.

If you delete a spreadsheet, any local macros associated with that spreadsheet will also be deleted. To save those macros, copy them to other spreadsheets or make them global. For more information, see the "Copying a Macro" section on page 42, and the "Changing the Macro Type" section on page 44.

To delete a macro:

- If you're deleting a local macro, make sure its window or its parent spreadsheet's window is the key window
- If necessary, choose *Macro Panel* from the Macro menu
- Choose *Local* or *Global* from the pop-up list, as shown below

- Select the macro you want to delete
- Click *Delete*

To run a macro from a menu:

- If you're running a local macro, make sure the macro's spreadsheet is the key window
- From the Macro menu, choose *Run Local* or *Run Global*
- Choose the name of the macro you want to run

WILMA completes the instructions stored in that macro.

To remove a macro from the Run Local or Run Global menus:

- If you're removing a local macro from the Run Local menu, make sure its window or its parent spreadsheet's window is the key window.
- If necessary, choose *Macro Panel* from the Macro menu
- Choose *Local* or *Global* from the pop-up list.
- If you choose *Local*, the macro will be removed from the Run Local menu; if you choose *Global*, it will be removed from the Run Global menu
- Select the macro you want to remove
- Click the *Remove From Menu* button
- The macro is removed from the menu.

Copying a Macro

You might want to copy a local macro from one spreadsheet to another. Or, you might want to create a new macro that's slightly different from an existing one, without having to retype the entire macro. Instead of creating the macro from scratch, you can duplicate it using the *Copy* and *Paste* commands in the Edit menu to copy macro instructions from one window to another.

To copy a macro:

- If necessary, choose *Macro Panel* from the Macro menu
- Open the macro you want to copy
- If you're copying a macro from one spreadsheet to another, open the spreadsheet in which you want to copy the macro

- !UDF: A user-defined function failed to run. ParaSheet displays the error !UDF in the cell from which the UDF was run and in any cells referencing that cell.

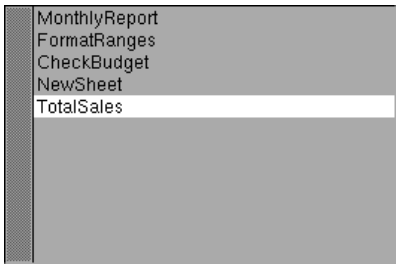
Using the Run Local and Run Global Menus

The Run Local and Run Global menus provide a fast way to run macros. To use one of these menus, you must first add a macro to the menu.

To add a macro to the Run Local or Run Global menu:

- If you're adding a local macro to the Run Local menu, make sure its window or its parent spreadsheet's window is active.
- If necessary, choose *Macro Panel* from the Macro menu.
- Choose *Local* or *Global* from the pop-up list, as shown below.

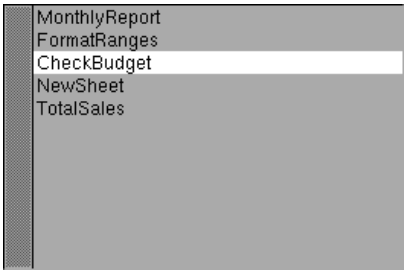
- If you choose *Local*, the macro will be added to the Run Local menu; if you choose *Global*, it will be added to the Run Global menu.
- Select the macro you want to add, as shown below.



- Click the *Add Macro to Menu* button.
- The macro is added to the menu, as shown below.

After you've added a macro to a menu, you can run it from that menu. The contents of the Run Local menu will vary, depending on which spreadsheet is active: the menu contains any local macros you've added for the spreadsheet that is the key window. The Run Global menu contains any global macros you've added to it, regardless of which spreadsheet is open.

- Select the macro you want to run, as shown below.



- Click *Run*

WILMA executes the instructions stored in the macro.

Recovering from Macro Errors

When a macro does not run successfully, you need to find out which macro instructions are causing the problem. If you're sure all of the necessary instructions are in your macro, your macro may have one of the following types of errors:

- Syntax: One or more instructions do not conform to the syntax rules of WILMA. WILMA detects syntax errors before the macro actually starts executing. For more information on syntax errors, see the "Following Syntax Rules" on page 13 and "Checking Syntax" section on page 14.
- Run-time: An instruction, though syntactically correct, cannot be evaluated by WILMA—the macro stops executing and a run-time error alert panel appears. Typically, run-time errors occur when a macro instruction contains an invalid argument or refers to something that does not exist. Some examples are:

```
Select ( A2); NextCell ( -1, -2 );
```

This macro attempts to select a cell one cell to the left of and two cells above cell A2—a cell that doesn't exist in the spreadsheet.

```
Enter ( Limit, E27 );
```

This instruction causes the macro to stop executing if Limit has not been declared as a named range.

Note *Any instructions following the instruction causing the run-time error will not be executed.*

- From the Run Local or Run Global menus (For more information, see the "Using the Run Local and Run Global Menus" section on page 41.)
- From a cell (For more information, refer to the "Working with User-Defined Functions" section on page 16.)

A local macro only runs when its parent spreadsheet is open. A global macro can always run when you're working in ParaSheet, regardless of whether any spreadsheets are open. When a macro is running, the text Running Macro: *MacroName* appears at the top of the macro's parent spreadsheet window.

After you run a macro, you cannot undo any of its actions by choosing the Undo command.

To run a macro from its window:

- Make sure the macro's window is the key window
- Click *Run*

If you press ***Command-period*** *(.) while a macro is running, the macro will immediately stop. Any instructions that have not yet been executed will be ignored. However, instructions that executed before you stopped the macro may have changed your spreadsheet. To get rid of those changes, use the Revert to Saved command in the Document menu to return to the last saved version of your spreadsheet. (If you use Revert to Saved, any unsaved changes from before you ran the macro will also be lost.)*

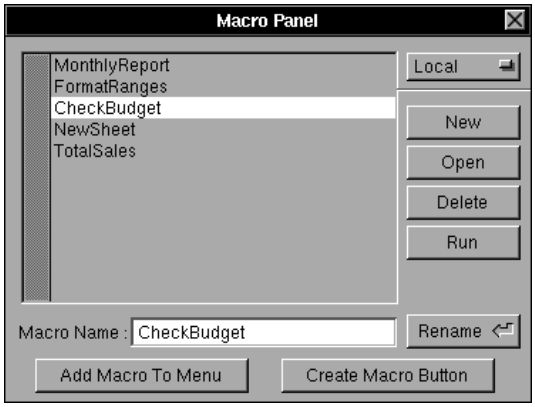
To run a macro from the macro panel:

- If you're running a local macro, make sure the macro's parent spreadsheet is the key window.
- If necessary, choose *Macro Panel* from the Macro menu.
- Choose *Local* or *Global* from the pop-up list, as shown below.

To open a macro:

- If you want to open a local macro, make sure the macro's parent spreadsheet is the main window
- Choose *Macro Panel* from the Macro menu
- Choose *Local* or *Global* from the pop-up list

- Select the macro you want to open, as shown below



- Click *Open*

Note *To open a miniaturized macro window, you can either use the method described above, or simply double-click its icon. The macro window opens and becomes the key window.*

When a macro is open and is the key window, it's ready to be edited, copied, or run. Editing and copying macros are explained in other sections of this chapter.

Running a Macro

Running a macro means having WILMA execute the commands in that macro. There are several ways to run a macro:

- By clicking *Run* from the macro's window
- From the Macro panel, if the macro was created in a macro window

The variable Result is assigned a 1 or a 0, depending on the user's choice. If the user chooses to delete the results, a second macro named Delete runs a deletion routine.

Returning Results

With the Return statement, a macro can evaluate an action and return a user-specified result to another part of the macro (or spreadsheet, if the macro is a user-defined function), depending on the results. Return has two capabilities. First, it causes the currently running macro to stop. If that macro was called from another macro, Return brings a value back to the calling macro or user-defined function. If the currently running macro was *not* called from another macro, Return stops macro execution and returns control to the user. The format of the Return statement is:

```
Return ( [ Value ] )
```

Value can be a text, logical, or numeric value, or a combination of values that produces a single value when calculated. If you don't specify a value and the macro containing the Return statement was called, WILMA returns a logical True.

When combined with calling additional macros from the main macro, Return is especially useful for detecting the success or failure of instructions in a called macro. Suppose you have a macro that, under certain conditions, gives control to a different macro that consolidates results from a set of data, then returns to the original macro. Using Return, you can detect whether the call- ed macro successfully completes the consolidation procedure. For example:

```
Result := Consolidation ( );
If ( Result = 1 )
           Return ( "Consolidation finished without error." );
Else
           Return ( "Consolidation aborted." );
```

Managing Macros

Once you've created a macro, you can do more than just run it. The following sections describe some common macro-handling operations.

Opening a Macro

Opening a macro gives you access to the WILMA instructions that make up the macro. Once you open a macro, you can read, edit, and copy all or portions of it to other macros.

exceeded 40 hours. Suppose you also have a database, in which the named ranges StartRow, EndRow, StartCol, and EndCol mark the boundaries of the data.

```
While ( Row <= EndRow #and# Col <= EndCol )
{
           HoursWorked := Cell ( Col, Row );
           If ( HoursWorked > 8 )
           Alert ( "Warning", "Employee has exceeded standard
work week hours." );
           }
```

Prompt is similar to Alert, but has customizable buttons. The format of the Prompt statement is:

```
Prompt ( Title, Message, Button1 [ , Button2 [ , Button3 ] ] )
```

Title is a title for the panel. The title must be less than or equal to 15 characters, surrounded by quotation marks. If you don't want a title, you must specify two double quotes ("") with no intervening characters.

Message can be any text string to be displayed in the panel, with a maximum of 255 characters, surrounded by quotation marks. If you don't want a message, you must specify two double quotes ("") with no intervening characters.

Button1, *Button2*, and *Button3* specify the custom text that appears on up to three buttons in the Prompt panel.

Button1 is the default button. Each button text string must be less than or equal to 15 characters. If you don't want text on the button, you must specify two double quotes ("") with no intervening characters.

Prompt is especially useful for interpreting a user's response to a situation before taking further action. Suppose you are in charge of buying supplies for a small company. You have a macro that consolidates information on the suppliers that offer the lowest prices. If the prices are not within your budget, you need to send the results in a cost report to your supervisor requesting more money. Otherwise, you can order what you need and delete the consolidation results from your spreadsheet. For example:

```
Result := Prompt ( "Delete", "Should the results of the
consolidation be deleted?", "Yes", "No" );
If ( Result = 1 )
Delete ( );
```

```
For ( Arg_1 := 10, Arg_2 := -33, Arg_3 := TheNamedRng;
Arg_3 = Sum ( A1 * B3 / Arg_4 ) #or# Arg_1 = 3;
Arg_1 := Arg_1 - 2, Arg_5 := Arg_5 + 1 )
```

Making Macros Interactive

Interactive macros can pause while executing, and prompt you for information. Interactive statements include Pause, Alert, and Prompt.

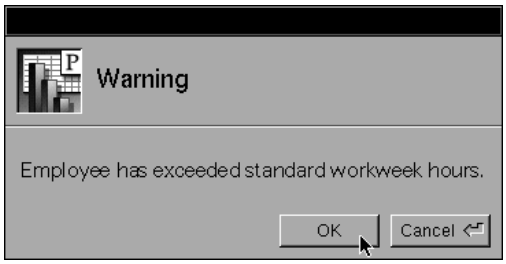
Pause stops a running macro until you tell it to continue. Each time Pause occurs in a running macro, a panel containing the standard text **Click Continue to continue running the macro** is displayed. The Pause panel remains until you click Continue to continue running the macro.

Alert causes a macro to stop running until a decision is made. When Alert occurs in a running macro, a panel containing a message you specify is displayed. The Alert panel has two standard buttons: OK and Cancel. The format of the Alert statement is:

```
Alert ( Title, Message )
```

Title is a text string less than 20 characters to be used as a title for the panel. The title text must be surrounded by quotation marks. If you don't want a title, enter two double quote marks ("") with no intervening characters for *Title*.

Message can be any text string to be displayed in the panel, with a maximum of 255 characters. The message text must be surrounded by quotation marks.



Clicking *OK* confirms the Alert message and WILMA returns the logical value True (1) to the macro. Clicking *Cancel* returns the logical value False (0) to the macro.

Alert is particularly useful for trapping data errors in a spreadsheet. For example, suppose you have a macro that calculates daily payment for hourly workers who should not work overtime. The macro can check for time beyond 40 hours and display an Alert panel if an employee has

- You can use different variables for the control test (*Condition*) than for incrementing or modifying (*ControlList*). For example:

```
For ( Row := 1; End = True ( ); Row := Row + 1 )
```

- You can use other methods than simple incrementing and decrementing (like multiplication) to change the value of the variable that determines how many repetitions the loop makes. For example:

```
For ( Row := 1; Row = 100; Row := Row * Value )
```

- You can omit one or more portions of the For structure. If you omit all three, you will create an infinite loop, as in the following example. If you find it necessary to create an infinite loop, remember to include a way to exit the loop within the For clause. This is the only way to break out of the loop from within the macro. For example:

```
For ( Row := 1; ; Row := Row + 1 )
{
Select ( Col, Row ); Enter ( Row + Col );
If ( Row = 25 )
            Return ( );
}
```

If you create a For loop which has no terminating condition that can be satisfied, the only way to halt the loop is to stop the macro. To stop the macro, press Command-period (.). *Any instructions that have not yet been executed will be ignored. However, instructions that executed before you stopped the macro may have changed your spreadsheet. To get rid of these changes, use the Revert to Saved command in the Document menu to return to the last saved version of your spreadsheet. (If you use Revert to Saved, any unsaved changes from before you ran the macro will also be lost.)*

The *Condition* is not included in this For statement, but the loop is still controlled because the instructions that form the body of the For loop contain If and Return statements that test the value of the Row variable.

- You can initialize multiple variables in the For statement. For example:

```
For ( Arg_1 := 10, Arg_2 := -33, Arg_3 := TheNamedRng . . . )
```

- You can use any expression (including a number, formula, or named range) to initialize a variable or act as the test case for terminating a loop. For example:

For

The For structure causes its associated instructions to repeat for a set number of times. After the last repetition, the macro continues executing, following the last instruction associated with the For loop.

The format of the For structure is:

```
For ( [ AssignList ]; [ Condition ]; [ ControlList ] )
          { Instructions }
```

AssignList consists of one or more expressions (values or formulas), each assigned to a variable. You must first declare variables (using the Local or Global statements) before using For. Multiple expressions must be separated by commas.

Condition is the test case for *AssignList*. When the test fails, the macro skips instructions within the For structure and continues executing the instructions that follow. *Condition* can be single or multiple expressions (values or formulas, each assigned to a variable) that evaluate to a single value. Multiple expressions must be separated by logical operators (for example, #and#, #or#, and #not#).

ControlList consists of one or more expressions that increment or decrement the variables in *AssignList*. Multiple expressions must be separated by commas.

Instructions are statements to be repeated until *Condition* is no longer true.

The For structure first assigns the specified values to each variable in *AssignList*. Then, it begins executing the instructions following the For statement (the For clause). When it reaches the end of the For clause, it increments or decrements variables in *ControlList*, as appropriate, then tests each condition specified in *Condition*. If all conditions in *Condition* are satisfied, the For clause is repeated. This process continues until one or more of the conditions specified in *Condition* are no longer satisfied. Then, macro execution resumes with the instructions following those in the For clause.

This statement is most valuable for its ability to centralize initialization, control, and incrementing or decrementing variables and conditions into a single statement. The following rules apply when using For:

- Multiple instructions to be executed within the For clause must be enclosed in braces ({ }). Otherwise, braces are optional.

While

The While statement causes its associated instructions to repeat as long as a certain condition is true. When the condition is no longer true, the macro continues executing, following the last instruction in the While clause.

The format of the While structure is:

```
While ( Condition )
WhileClause
```

Condition can be any condition that, as long as it's true, controls how many times an associated *WhileClause* executes.

WhileClause specifies an instruction or group of instructions that WILMA executes as long as the While *Condition* is true.

In a While structure, WILMA first evaluates whether the While *Condition* is true. If it is true, WILMA executes instructions in the *WhileClause*. If the While *Condition* is false, WILMA continues executing at the first instruction following the While structure.

The following rules apply when using While:

- Multiple instructions to be executed in the While clause must be enclosed in braces ({ }).
- If the While condition starts out false, none of the statements in the While clause are executed.
- When using While, be sure to adjust the condition (by incrementing or decrementing the counter) within the loop instructions, or you'll create an infinite loop. The example below shows an incrementing counter:

```
While (Row < 100 )
           {
           Select ( Col, Row );
           Enter (Col + Row );
           Row := Row + 1;
           }
```

This While loop continues executing as long as the test value Row is less than 100. The instruction Row := Row + 1 increments the test value, causing the loop to stop executing when Row equals 100.

```
Local ( UserChoice, Result, Success := "OK" );
UserChoice := Prompt ( "EntryForm", "Which type of data do you
want to enter?", "Numeric", "Text" );
EntryPoint:
If ( UserChoice = 1 )
{
Result := NumHandler ( );
If ( Result <> Success )
          {
          ErrorRecovery ( );
          Branch ( "EntryPoint" );
          }
Else
          {
          DataEntry ( );
          Result := AnyMoreData ( );
          If ( Result = Success )
                    Branch ( "EntryPoint" );
          Else
                    Branch ( "NextStep" );
          }
          }
Else
          {
          Result := TextHandler ( );
          If ( Result <> Success )
                    ErrorRecovery ( );
          Else
                    TextEntry ( );
          }
NextStep:
…
```

This macro sends the user to one of six macro modules (NumHandler, ErrorRecovery, DataEntry, AnyMoreData, TextHandler, and TextEntry), depending on the user's choice of actions or possible errors. As each module finishes executing, WILMA returns control to the main macro. Branch statements control the flow of execution in the main macro, letting the user repeat actions.

By calling additional macros, you can create macros that are:

- Easy to manage and understand
- Organized into modules that can be maintained independently without affecting the main (calling) macro
- Smaller since they use fewer instructions but contain the same functionality as macros in which all instructions are sequential

Calling additional macros is a good way to handle non-fatal run-time errors (ones that will not cause the macro to stop executing) by transferring control to another set of instructions and then returning to the original set.

For example, here is the first macro:

```
\\ Macro name: CheckBudget
...
If ( Budget > Limit )
{
Result := Alert ( "Over Budget", "Archive analysis anyway?" );
If ( Result = 1 )
           GraphData ( );
}
```

Here is the second macro:

```
\\ Macro name: GraphData
New ( );
SaveAs ( "AuxiliaryAnalysis" );
Select ( "Analysis" );\\ Selects the original spreadsheet
Copy ( BudgetData );
Select ( "AuxiliaryAnalysis" );\\ Selects the new spreadsheet
Paste ( B2 );
ChartNew ( "SBar" );
```

Suppose you have a macro called *CheckBudget* that analyzes data in two named ranges called Budget and Limit in a spreadsheet called Analysis. If the value in the named range called Budget exceeds the value in the named range called Limit, the *CheckBudget* macro lets you choose whether to archive the budget analysis in a separate spreadsheet (to be named AuxiliaryAnalysis). If you choose to archive the data, the *CheckBudget* macro calls the *GraphData* macro, which makes a copy of data in the original spreadsheet, graphs it, and saves it. Otherwise, the analysis is not graphed or saved.

Another common use for calling additional macros is to invoke a subroutine to handle a specific action. For example, suppose you have a custom form giving users different ways to enter and edit different types of data. Instead of trying to check and process every possible error in one macro, you could write several small macros that can be called from the main macro, each capable of efficiently handling a specific type of data entry error. For example:

Once WILMA finds the label in the current macro, it continues executing instructions from the label on. If no label is found, an alert panel appears.

The most common use of Branch is to handle fatal run-time errors (ones that will cause the macro to stop executing) by transferring control to another set of instructions. For example, suppose you have an order entry application that can check for data entry errors:

```
Local ( ErrorTrap );

…

IfErr ( ErrorTrap = 1, Branch ( "FoundError" ),
Branch ( "EnterData" ) );

…

FoundErr:

Alert ( "Warning", "The data you just entered is invalid." );
```

If the user makes a data entry error, the macro then branches to a label called FoundError. Otherwise, the user can continue entering data.

Calling Another Macro

Sometimes you might want to execute instructions in a different macro. To call another macro from the current macro, you can use the same procedure as when calling a user-defined function from a cell: type the macro's name followed by opening and closing parentheses. Unlike using the Branch statement, after the second macro finishes executing this calling syntax transfers control back to the instructions following the point where the second macro was called. (For more information on user-defined functions, see page 18.)

The format to call another macro is:

```
MacroName ( );
```

MacroName specifies the macro to which you want to transfer control. Remember, macro names are case sensitive.

When you call another macro, WILMA first searches the local macros associated with the current spreadsheet, then the global macros for ParaSheet until a match is found. WILMA opens and executes instructions in the called macro until reaching a Return statement or the end of the called macro. It then resumes at the first instruction after the call in the original macro. If no macro with the specified name is found, a run-time error alert panel appears.

```
If ( Grade >= 90 #and# Grade <= 100)
            {
            Enter ( "A", FinalGrade );
            If ( Total >= 90 #and# Total <= 100 )
                        Enter ( "Great Job!", FinalResult );
            }
Else If ( Grade >= 80 #and# Grade < 90 )
            Enter ( "B", FinalGrade );
…
```

- The If statement can be used independently of the Else statement. For example:

```
If (Sales < 1000)
Alert ( "Inventory exceeds storage." );
```

- Multiple If statements can be nested. For example:

```
If ( Grade >= 90 #and# Grade <= 100)
Enter ( "A", FinalGrade );
Else
If (Grade >= 80 #and# Grade < 90 )
            Enter ( "B", FinalGrade );
Else
If ( Grade >= 70 #and# Grade < 80 )
                        Enter ( "C", FinalGrade );
            Else
                        Enter ( "D", FinalGrade );
```

- If the If condition is a string, it is case sensitive.

Branch

The Branch statement is an *unconditional* control structure that causes WILMA to stop executing instructions, go to a specified label in the current macro, and continue executing at the new location. Once the branch instructions are executed, macro execution stops. Branch does not return to the line from which it was executed.

The format of the Branch statement is:

```
Branch ( Label )
```

Label is a text string specifying the location at which to continue executing. The label must occur twice in the macro: once in the Branch statement, surrounded by quotation marks, and once at the branch destination, followed by a colon (:). The branch destination label can be before or after the Branch statement. Branch labels are case sensitive.

```
If (Sales > 1000)
          Alert ( "Warning", "Inventory is low." );
Else
          {
          Alert ( "Warning", "Inventory exceeds storage." );
          Enter ( "OK", Z1 );
          }
```

Since the If statement only has one instruction, surrounding braces are unnecessary. The Else statement has multiple instructions, so surrounding braces are necessary.

If you omit the braces surrounding the Else instructions in the above example, the instruction Enter ("OK", Z1); will always execute, even when Sales are greater than 1000. This is because after WILMA executes the If instructions, it searches for the next semicolon (which terminates an instruction), then executes whatever instructions follow. When the If condition is true, surrounding the Else instructions with braces causes WILMA to skip them and execute whatever instructions follow the closing brace.

- If … Else constructs may be nested. For example:

```
If ( Condition )
          {
          If ( Condition )
                    {
                    If Instructions;
                    }
          Else
                    {
                    Alternative ;
                    }
          }
Else
          {
          Alternative ;
          }
```

- When nesting If … Else statements, an Else clause is grouped with the closest preceding If statement that doesn't already have an Else clause. (The Else clause consists of the Else statement and any associated instructions.) To override the natural order in which nested If … Else statements execute, use braces. For example:

Loop structures cause *iteration*—repetition of a sequence of actions to achieve a result. You can create loop structures using the While and For statements.

Placing a semicolon immediately after an If or Else condition, While statement, or For statement causes instructions that follow not to repeat and might produce unexpected results. See examples in the sections that follow for correct semicolon placement.

If … Else

The If … Else statement is a *conditional* control structure that provides an either-or situation. If a certain condition exists, the instructions associated with If execute; otherwise, the instructions associated with Else execute.

The format of the If … Else structure is:

```
If ( Condition )
IfClause
[ Else
ElseClause ]
```

Condition can be any non-zero value that evaluates to True (1) or any value that evaluates to False (0).

IfClause can contain single or multiple instructions.

When using the optional Else, *ElseClause* can contain single or multiple instructions, including nested If … Else statements.

In an If … Else structure, WILMA first evaluates whether the If condition is true. When it is true, the instructions in the IfClause are executed. When the If condition is false and there is an Else statement in the structure, WILMA executes the instructions in the ElseClause. When the If condition is false and there is no Else statement, WILMA continues executing at the first instruction following the If structure.

The following rules apply when using If … Else:

- Multiple instructions to be executed in the If … Else structure must be enclosed in braces ({ }). For example:

Operator	Meaning	Precedence
<, >	Less than, greater than	7
<=, >=	Less than or equal, greater than or equal	7
#not# or ~	Logical not	8
#and# or &&	Logical and	9
#or# or ||	Logical or	10
#cat# or &	Concatenation	11
:=	Assignment (one per formula)	12
,	Comma (function separator)	13

If your macro compares two text strings, such as Dog and dog, remember that ParaSheet is sensitive to the character's case. Uppercase characters have a higher priority than lowercase characters. For example:

- "Dog" = "dog" is false
- "Dog" <> "dog" is true
- "DOG" > "dog" is true
- "DOG" < "dog" is false

Using Control and Loop Structures

Typically, macro instructions execute in the order in which they occur in the macro. However, there might be times when you want to have the macro choose which action to execute based on certain data (a control structure), or repeat the same action several times (a loop structure).

There are two types of control structures in WILMA: conditional and unconditional.

Conditional control structures cause WILMA to decide what instructions to execute based on the presence of a specified condition—for example, one value being greater than another. You can create a conditional control structure using If … Else.

Unconditional control structures cause WILMA to stop executing instructions, go to a different location in the current macro or to another macro, and continue executing at the new location. You can create an unconditional control structure using the Branch statement or by calling another macro.

easier to read and understand, and protect your data if you insert or delete columns or rows in the spreadsheet.

For more information on assigning named ranges, refer to page 84 in the Tutorial & Documentation manual, or to the "NamedRngAdd command" section on page 63 in this manual.

Before you use a range name in a macro, be sure to assign the range name in the spreadsheet. Otherwise, a run-time error alert panel appears.

WILMA doesn't require that you declare the value type of a variable (text, numeric, and so on), as some other languages do; WILMA assumes a variable's type is that of the value assigned to it. Furthermore, you can change a variable's type after it has been assigned in a macro. For example:

```
Local ( Variable3 := 27 );
Enter ( Variable3, A1);
Variable3 := "Over Budget";
```

First, the number 27 is assigned to Variable3 and the value is displayed in cell A1. Then, the text "Over Budget" replaces the value 27 for Variable3, changing its type from numeric to text.

Precedence of Operators

Operators are symbols representing certain arithmetic or logical actions to be performed in a ParaSheet formula or macro. The table below lists the ParaSheet operators and their precedence levels. 1 is ranked the highest, and is calculated first; 13 is ranked the lowest and is calculated last. Operators with equal precedence are calculated in the order listed.

Operator	Meaning	Precedence
( )	Left and right parentheses	1
–	Unary minus (denotes negative value)	2
^ or **	Exponentiation	3
*, /	Multiplication, division	4
%	Percent	5
+, –	Addition, subtraction	6
=, <>	Equal, not equal	7

data to be passed. Then, you can access that data directly from a macro in another spreadsheet by using the following syntax:

```
VariableName := "Filename"!RangeReference;
```

Remember to declare the variable before assigning the spreadsheet data to it.

With both local and global variables, you can declare and initialize multiple variables with a single statement, and mix data types within the same statement. For example:

```
Local ( Sales, LastYearVal := 2000, Author := "J.S.B.", Confirm
 );
```

The variables Sales and Confirm in this example are both initialized to 0, because they were not assigned a value.

You can assign an expression to a variable, but it must evaluate to a single result.

The following types of values can be assigned to local or global variables, as shown below:

Value Type	**Examples**
Numbers	Local (MonthlySales := 550);
	Global (Area := 11 * 5);
	Local (Budget := 150);
Text	Local (Name := "Sandy");
	Global (TotalSales := "This month's sales were:");
Cell references	Local (Variable1 := A1:C3);
Named ranges	Local (Variable2 := ANamedRange);
Logical	Global (Profit := True ( ));

If your macro uses spreadsheet data, try to assign descriptive range names to the groups of data used by the macro. (Use the *Named Ranges* command in the Sheet menu, or the *NamedRngAdd* command in a macro.) Then you can use the range names in your variables (rather than the cell coordinates) to access the data. This will make the macro much

used as a variable, WILMA will interpret it as a function or command and you will get unexpected results.

Declaring and Assigning Variables

Before you can use a variable in a macro, you must *declare* it; that is, let ParaSheet know the variable exists. Declaring a variable involves the following steps:

- Deciding on the variable's level (local or global)
- Naming the variable

Variables are declared using one of the following formats:

```
Local ( VariableName ); \\ for a local variable
Global ( VariableName ); \\ for a global variable
```

VariableName represents a descriptive name for the variable.

When you declare a variable, its initial value is zero. You can assign the variable a value other than zero when you declare it, or later in the macro. To declare and assign a variable in one step, use one of the following formats:

```
Local ( VariableName := Value ); \\ for local variable
Global ( VariableName := Value ); \\ for global variables
```

The colon and equals characters (:=) represent the assignment operator in WILMA. (Note that there is no space between these two characters.) The variable name must be to the left of the assignment operator, and the value must be to the right. *Value* represents a data value such as a text string, number, range reference, or another variable.

To assign a variable later in the macro, after it has been declared, use the following formats:

```
LocalVariableName := Value; \\ for local variables
"GlobalMacroName"@GlobalVariableName :=Value; \\ for globals
```

To use a global variable declared in another macro, use the following format:

```
"/PathName/Filename/MacroName"@GlobalVarName
```

If you prefer to use local variables but need to pass data between macros, try designating a portion of one of your spreadsheets as a holding area for

if the spreadsheet location for certain data changes in your macro, variables and named ranges are always associated with the data they represent.

- The macro will be easier to understand (for yourself and anyone else using it).
- The macro will be easier to maintain, edit, and debug.

Deciding on a Variable Level

There are two levels of variables in WILMA: local and global. A *local* variable can only be used in the macro in which it's created. The value it represents cannot be used or changed by other macros. Because local variables only require storage space while a macro is running, they are faster and more memory efficient than global variables.

A *global* variable can be used by any macro in any spreadsheet. Because global variables are stored with the ParaSheet application, they are not as memory efficient as local variables. The space occupied by a global variable is not freed until you delete the global variable.

Naming Variables

Just as when naming spreadsheets or ranges, you should try to give variables descriptive names—names you'll easily remember after some time has passed. The following rules apply when naming variables:

- A variable name and the value assigned to it must be in parentheses when declared, preceded by the word *local* or *global*.
- A variable name can include up to 255 characters. For speed and clarity of a macro, however, try to limit variable names to 31 characters each.
- A variable name must begin with a letter. Subsequent characters may be letters, numbers, or underscores. Variable names cannot be the same as cell references.
- Variable names are case sensitive and must be used consistently.
- The following words are reserved in WILMA and cannot be used as variable names: Branch, Else, For, If, Return, and While.

Function and command names are not reserved words and can be legally used as variable names. However, this is not advisable. If the syntax of your macro instruction includes a left parenthesis after a function or command name being

data is graphed and the graph is moved 64 pixels below its data source, the named range called BudgetData; otherwise, no graph is created. In either case, control is then returned back to Trigger.

Note how using small, modular UDFs enables increasingly complex tasks to be programmed easily and efficiently. The above example could be expanded into a more thorough analysis, where each column, row, or range of cells is archived into a different location and available for cell-by-cell comparison as data is being entered. Nested UDFs could be used to treat results in different columns differently.

Running a User-Defined Function from a Cell

You can run a UDF from a cell just like any other ParaSheet function. If you're running the UDF by itself from a cell, precede its name with an equal (=), plus (+), or minus (–) sign, just like any other formula. If you're running the UDF as part of a formula, type the UDF's name into the formula follow- ed by opening and closing parentheses. Remember, macro names are case sensitive and must be entered exactly as they appear in the Macro panel.

To ensure that absolute or mixed cell references behave as expected in a UDF run from a cell, precede column references, row references, or both with a dollar sign ($). For more information on absolute, relative, and mixed cell references, refer to page 82 in the Tutorial & Documentation manual.

Programming in WILMA

Using WILMA, the ParaSheet programming language, you can build *smart* macros—macros capable of making decisions and executing alternative instructions based on those decisions. The following sections describe how to put WILMA statements to work in your macros.

Using Variables

Variables are names for places in which you can store data values. Each variable keeps track of any changes to the value it represents, as that value is processed in the macro. Assigning values to variables instead of entering them directly into a macro has the following advantages:

- Keeping track of how you're processing data in the macro is easier since each piece of data has a meaningful name.
- When the macro is run, complex calculations are more likely to be accurate, especially when you use named ranges with variables. Even

To create a user-defined function to be run from a cell:

- Create a new macro window and enter the necessary functions, commands, and statements, as described above in the sections "Creating a New Macro" and "Writing a Macro." Remember to exclude the illegal commands listed earlier in this chapter. UDFs can be local or global (for more information, see the "The Types of Macros" section on page 4).
- Give the UDF a unique name. It will not run from a cell under its default name.
- Check the UDF's syntax and make any necessary corrections.
- Close the macro window.

The UDF is now ready to be run from a cell.

Nesting User-Defined Functions

UDFs can be nested (a UDF can contain another UDF, and so on), when you need to create more complex automated tasks. The following example builds on the previous example using an alert trigger. Suppose that even if a hypothetical budget exceeds a set limit, you want to graph all the data in a named range called BudgetData.

```
\\ UDF: Trigger:
\\ Revised to nest GraphData UDF
Local ( TotalDollars );
TotalDollars := Sum ( Budget );
If ( TotalDollars > Limit )
            {
            Alert("ALERT!", "You're over budget!");
            GraphData ( );\\ New UDF added — see below
            }
\\ UDF name: GraphData
Local ( Result );
Result := Prompt ( "Warning", "Data is over budget. Make a
graph anyway?", "Yes", "No" );
If ( Result = 1 )
            {
            ChartNew ( "Bar", BudgetData );
            ObjMove ( 0, 64, BudgetData );
        }
```

GraphData issues a prompt asking whether to graph the data even though it is over budget. (This could be useful for illustrating different scenarios in a cash-flow analysis.) If the user responds "Yes", then the

simulate the Lotus 1-2-3 and SuperCalc CTERM function (similar to the Excel NPER function).

```
\\ UDF: CTERM

Return ( Ln ( FutureValue / PresentValue ) /
( Ln ( 1 + IntRate ) ) );
```

(If you want to use this UDF with other named ranges, remember to substitute the names FutureValue, PresentValue, and IntRate for the range names you're using.)

Writing a User-Defined Function

You can write a UDF in a macro window just like a regular macro. However, if you intend to run the UDF from a cell, the following commands should not be included:

Activate	Clear	ClearSpecial
Close	Cut	Delete
Enter	Fill	Hide
Import	IncFill	Insert
Maximize	Miniaturize	Move
NamedRngAdd	NamedRngDel	NamedRngEdit
New	Open	OpenLinks
Paste	PasteSpecial	Prefs
Recalc	RecalcAll	RecalcOptions
RevertToSaved	Save	SaveAs
SaveTo	SheetSizeSet	Sort

If the above commands are used in a UDF run from a cell, ParaSheet ignores them—so you might get unexpected results.

You should *not* write a UDF to be run from a cell when you want to:

- Use one or more of the commands in the above list
- Run the macro only when you choose to run it

There are two places from which you can run a UDF: from a cell and from a regular macro. When you run a UDF directly from a cell or from within another UDF running from a cell, that UDF will execute every time the spreadsheet recalculates.

When you run a UDF from a regular macro, it doesn't execute at every recalculation. This distinction is important because there are certain limitations that apply to UDFs run from cells. The following sections discuss these limitations. For more information on running UDFs from regular macros, see the section "Calling Another Macro" on page 29.

When to Use a User-Defined Function in a Cell

A UDF can be used in a cell when you want to see immediately how changes to the spreadsheet affect existing data. UDFs in cells run automatically as part of spreadsheet recalculation whenever the spreadsheet is recalculated. UDFs in cells are especially useful for:

- Special formatting and alignment routines based on changes to spreadsheet data. For example, a UDF can reformat a chart to accommodate more data or longer labels.
- Alerts. For example, changes in data can trigger a panel to appear.

Suppose that you have a large financial consolidation application, and want to work with some hypothetical budget information. You have a named range called Budget, and you want to check it against a predetermined limit stored in another named range called Limit. The following formula triggers a UDF named Trigger, which raises an alert panel when the hypothetical budget exceeds the predetermined limit:

```
=IfX ( Budget > Limit, Trigger ( ), "Budget is realistic." );

\\ UDF: Trigger

Alert("ALERT!", "You're over budget!");
```

In the above example, the IfX formula checks the named range called Budget. If the value in Budget ever exceeds the value in Limit, the UDF called Trigger raises an alert panel, and displays the appropriate warning.

Simulating functions currently not available in ParaSheet.

For example, suppose you want to compute the number of periods required for an initial investment at its present value to grow to a future value at a given interest rate. Using the UDF shown below, you can

You can have more than one macro window open and visible at a time, which makes moving and copying text between macros easy.

As you edit macros, you can choose functions, commands, and statements from the Functions panel, or you can type them. If you type macro commands and functions in your macros, don't worry about capitalization—ParaSheet accepts macro text regardless of capitalization (except for variables, labels, macro names, filenames, and path names, which are case sensitive).

Saving a Macro

Macros are automatically saved when you do one of the following:

- Save the macro's spreadsheet (local macros only)
- Check the macro's syntax
- Click *Run* in the macro's window
- Close the macro's window (local macros only)
- Close the spreadsheet from which a local macro was created (local macros only)
- Choose Save when a global macro window is the key window (global macros only)
- Quit ParaSheet

Macros are stored differently, depending on their type. A global macro is always stored with the ParaSheet application under its default name (unless you rename it). It is always available when you're working in ParaSheet.

A local macro is always stored with its parent spreadsheet under its default name (unless you rename it). For example, suppose you have a local macro called *CalculateExpenses* in a spreadsheet called MonthlyBudget. You can always find *CalculateExpenses* in the local macro list in the Macro panel when MonthlyBudget is the active spreadsheet.

Working with User-Defined Functions

A *user-defined function* (UDF) is a macro called by name from a cell or another macro. Like a function, it can be included in a formula (in the same way as Sum, Avg, and so on).

If you don't use the *Check Syntax* button after writing or editing a macro, ParaSheet automatically checks the syntax of a macro when you run it.

Adding Comments to a Macro

You can incorporate *comments* into your ParaSheet macros. Because comments don't execute when the macro is run, they can be used to describe the instructions in your macro, or to write general notes. Putting comments in your macros helps you to remember what certain commands do and will help explain your macros to other people who use them.

To add a comment to a macro, simply type the text of the comment, preceding it with two backslashes (\\ ), as shown below. When the macro is running, everything between the two backslashes and the end of the line is ignored. If your comment requires more than one line, make sure each comment line starts with two backslashes.

```
=Macro{
\\ Macro Name: AnnualBudget
Select ( B2:M2 );
\\ Select cells B2 through M2
Align ( "Center" );
\\ Center the contents of B2 through M2
Bold ( "Yes" );
\\ Make the contents of B2 through M2 bold
ChartNew ( "3DBar", B3:E7 );
\\ Create a 3D bar chart from data in
\\ cells B3 through E7
Save ( );
\\ Bring up the Save panel
}
```

When you are debugging a macro, you can precede an instruction with two backslashes to keep it from executing. This technique is useful when you want to experiment with different instructions without deleting the ones you've already written.

Editing a Macro

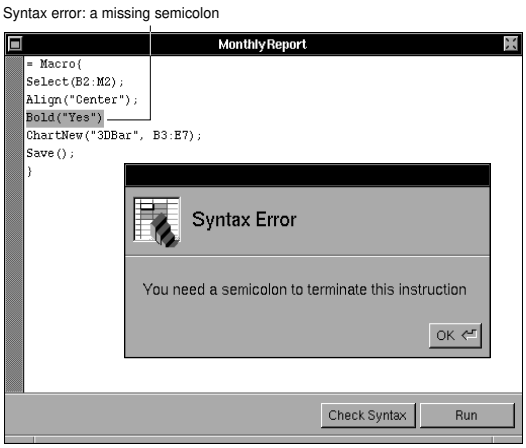
Once you're familiar with WILMA, editing macro instructions is just like editing any other text. First, open the macro. Then, click the insertion point anywhere in the macro and begin typing, using the arrow keys to move the cursor around, dragging to select text for cutting or copying, or using the *Select All* command in the Edit menu to select all the text.

- Variable names, labels, macro names, filenames, and path names are case sensitive and must be typed exactly as they were created. Other macro text is not case sensitive.

For more information on individual commands, functions, statements, and their arguments, refer to Chapter 2 in this manual.

Checking Syntax

While you're writing or editing a macro, you can use the *Check Syntax* button in the Macro window at any time. When you click this button, the compiler checks the macro for syntax errors, and highlights the first error it finds (as shown below), if it finds any. An alert panel also appears describing the error. You can fix the errors right away, or later. However, the macro will not run until the errors are fixed.



The following are some common syntax errors:

- Misspelling a statement
- Omitting a required argument
- Omitting commas between arguments when multiple arguments are required
- Omitting a semicolon at the end of a macro instruction
- Omitting the macro opening (=Macro{) or macro terminator (})
- Including square brackets to demarcate optional arguments

- The functions, commands, or statements in that category appear on the right side of the panel.
- Select a function, command, or statement from the right side of the panel.
- To insert the contents of the Syntax area directly into your macro, click either *Insert* or *Insert with Arguments*.
- If you click *Insert*, enter any necessary arguments.
- If you click *Insert with Arguments,* replace the name of each argument with an appropriate entry. (For more information on arguments and entries, see a complete description of the function, command, or statement in Chapter 2 of this manual.)

To close the Functions panel, click its close button .

Writing a Macro

There are certain conventions that must be followed when writing a macro. These conventions might vary, depending on which command, function, or statement you're using.

Following Syntax Rules

General syntax rules for WILMA are:

- Each macro instruction must end with a semicolon (;). You may include more than one instruction per line, as long as each instruction is terminated by a semicolon.
- The correct number and type of arguments for a function, command, or statement must be included. If you're in doubt about the arguments, check the command format in the Functions panel or in Chapter 2 in this manual.
- Commas must separate multiple arguments.
- Some commands have text arguments that must be surrounded by quotes. These are noted in the descriptions for each command. If you don't include quotation marks when they're needed, WILMA assumes you've entered a range name. If the argument takes only a single value, WILMA uses the value in the top left cell of the named range.
- When a decision or loop control statement includes multiple instructions to be executed as a result of a condition or as part of the loop, the block of instructions must be surrounded by braces ({ }).

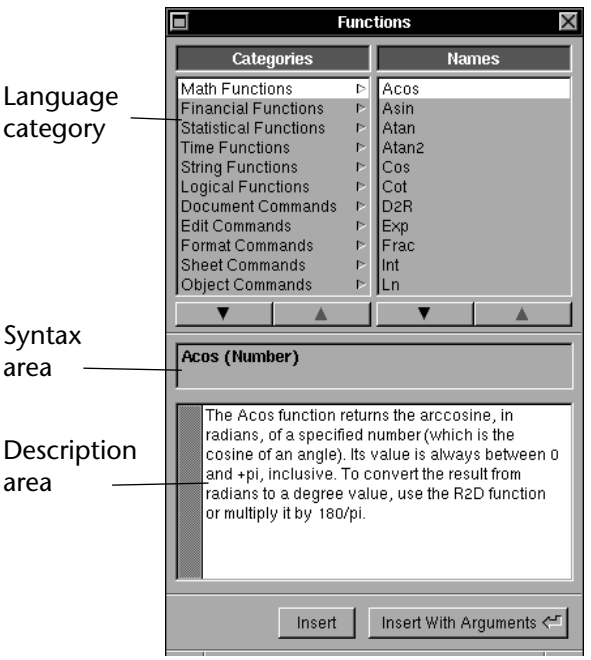
Shortcut

Instead of using the Macro panel, you can choose New Local or New Global from the Macro menu to open a new local or global macro window.

For more information on local and global macros, see "The Types of Macros" section on page 4. For more information on the Functions panel, see the next section, "Using the Functions panel."

Using the Functions panel

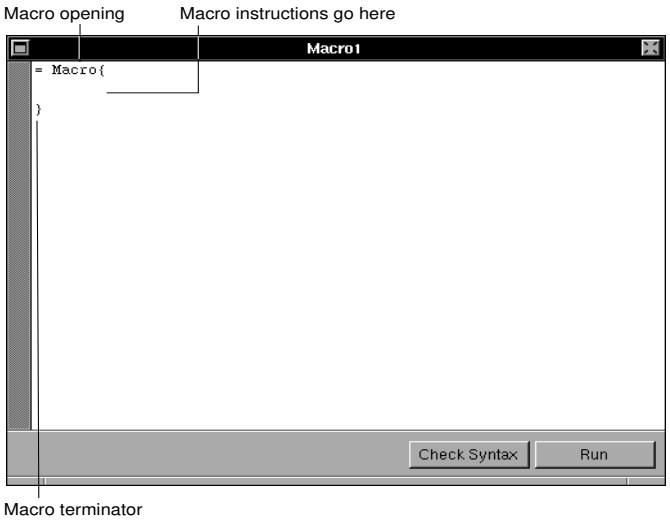
The Functions panel contains every element of the WILMA language, including the functions, commands, and statements discussed earlier in this chapter. It is useful as a reference to WILMA and is a convenient source of correct syntax and arguments.



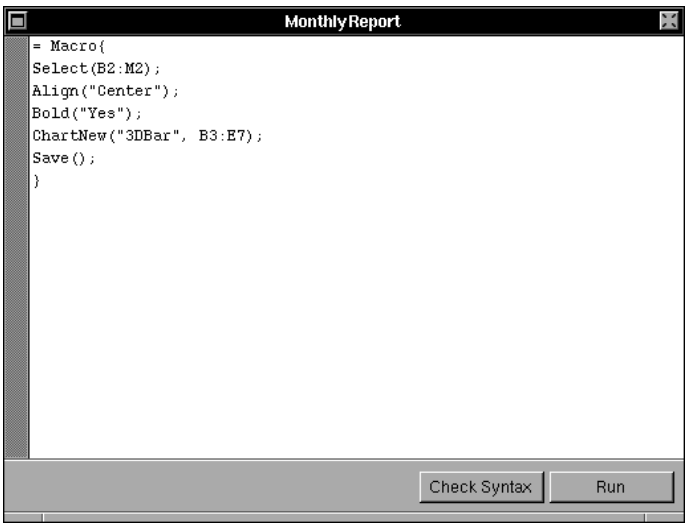
The contents of the Functions panel are categorized for easy reference. To get information on a function, command, or statement:

- Choose *Functions* from the Tools menu, or click the f(x) button on the icon bar.
- Select a category name from the scrolling list on the left side of the panel.

- A new macro window opens, as shown below.

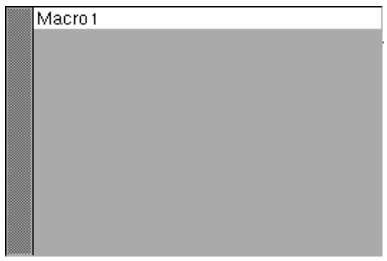


- Enter the instructions for your macro, terminating each instruction with a semicolon (;), as shown below. You can type the instructions, or choose them from the Functions panel and just fill in the necessary arguments.



- From the pop-up list, choose *Local* or *Global*, depending on what type of macro you are creating.

- Click *New*
- The new macro is assigned a default name in the macro list, as shown below.



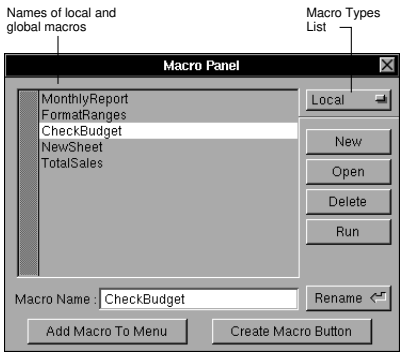
- If you want, enter a descriptive name in the *Macro Name* field, and press Return. Macro names can consist of alphanumeric characters and underscores.

- Click *Open*

Optional arguments are surrounded by square brackets ([ ]) when listed in Chapter 2 in this manual and in the Functions panel.

Writing and Editing Macros

Most of the tools for creating, writing, and managing macros are in the Macro panel.



Generally, the instructions in a macro are executed sequentially, in the order of their occurrence in the macro—except when an instruction transfers control to another part of the current macro or to a different macro. You can combine the different kinds of instructions in a macro in any way that accomplishes the task the macro is performing—as long as the macro conforms to the syntax rules of WILMA. (For more information on syntax rules, see the "Following Syntax Rules" section on page 13.)

Creating a New Macro

New local and global macros, or user-defined functions, are created in a macro window. The macro window is a standard NEXTSTEP window—it can be sized, scrolled, miniaturized, closed, and moved anywhere on the screen. As with any NEXTSTEP window, you can control the font and print the contents. Each macro window contains only one macro.

To create a macro:

- If you're creating a local macro, make sure the spreadsheet for which you're creating the macro is the active spreadsheet.
- Choose *Macro Panel* from the Macro menu.

About Statements

Statements are instructions that control how functions, commands, and even other statements operate in a macro. WILMA statements are organized as below:

Assignment	Decision Control	Loop Control	User Interaction
Global	Branch	For	Alert
Local	If…Else	While	Pause
Return			Prompt

If you have some programming experience, you may find that these components of the macro language are already familiar to you. By understanding how statements work and including them in your macros, you will be able to take the greatest possible advantage of WILMA to write custom applications in ParaSheet. For more information on individual statements, see the "Programming in WILMA" section on page 20 in this chapter, and Chapter 2 in this manual.

About Arguments

Arguments are information that certain functions and commands require in order to return useful information back to the spreadsheet or macro in which they are used. In WILMA, arguments are surrounded by parentheses and follow the function or command name. Multiple arguments are always separated by commas. For example, to create a 3D bar chart, use the WILMA command *ChartNew* with the *ChartType* and *Selection* arguments as follows:

```
ChartNew ( "3DBar", B2:D8 );
```

You can enter most arguments in one of two ways: an English-like word, or its numeric equivalent. These choices are listed for each function and command in Chapter 2 of this manual, and in the Functions panel.

Some functions and commands do not require arguments. For example, to open an existing file without providing a filename, you can just enter:

```
Open ( );
```

This displays the Open panel so you can select a filename.

Document
Activate
Close
Import
Maximize
Miniaturize
New
Open
OpenLinks
Redraw
RevertToSaved
Save
SaveAs
SaveTo
WindowMove
WindowSize
Zoom

Edit
Clear
ClearSpecial
Copy
Cut
Delete
Enter
Fill
IncFill
Insert
Move
Paste
PasteSpecial
Sort
SortKeyDel
SortKeyReset
SortKeySet

Format
Align
Bold
FmtCustomAdd
FmtCustomDel
FmtTypeSet
FontSet
Formats
Italic
PageLayout
Plain
PointSize

Sheet
GoTo
GridShow
HeadingsShow
NamedRngAdd
NamedRngDel
NamedRngEdit
PageBreakSet
PageBreaksShow
Recalc
RecalcAll
RecalcOptions
SheetSizeSet
SizeSet

Object
2DViewSet
3DViewAngleSet
3DViewDepthSet
AxisGridShow
AxisLabelsSet
AxisLabelsShow
AxisScale
AxisTicksCalcAuto
AxisTicksCalcMan
AxisTicksShow
BorderShade
BorderWidth
BringToFront
ChartAttribsSet
ChartNew
ChartTextDel
ChartTextNew
FillShade
InterpSet
ObjInfo
ObjMove
ObjSize
PieMarksSet
PieStartAngleSet

Selection
Cell
ChartSelect
HIndex
HLookup
NCols
NextCell
NRows
Offset
Scroll
Select
SelectAdd
SelectAll
TextSelect
VIndex
VLookup

Get
Get2DView
Get3DViewAngle
Get3DViewDepth
GetArrowType
GetAxisGridShow
GetAxisLabels
GetAxisLabelsShow
GetAxisTicksCalc
GetAxisTicksShow
GetAxisTicksValues
GetAxisXScale
GetAxisYScale
GetBorderShade
GetBorderWidth
GetChartAttribs
GetColNum
GetDocName
GetFillShade
GetFmtType
GetFont
GetInterp
GetNumObjs
GetNumPages
GetNumParts
GetObjName
GetObjNum
GetObjPosition
GetObjSize
GetPageNum
GetPieMarks
GetPieStartAngle
GetRange
GetRowNum
GetSelection
GetSize
GetUDFCol
GetUDFRow
GetWindowPosition
GetWindowSize

ParaSheet
Hide
Info
Prefs
Print
PrintNow

Commands combined with the other elements of the macro language are powerful building blocks for writing programs in WILMA. For more information on individual commands, see Chapter 2 in this manual.

function determines whether a value in a cell is text or a number. All functions return some kind of result back to the spreadsheet; if the function is part of a formula or macro, the result is typically used in a calculation. WILMA functions are organized as shown below.

Math	Financial	Statistical	Time	String	Logical
Acos	Annuity	Abs	Date	ASC	Choose
Asin	Compound	Avg	Day	Chr	Err
Atan	DB	Ceiling	DOW	Left	Error
Atan2	FV	Count	Hour	Length	False
Cos	IRR	Floor	Minute	Mid	IfErr
Cot	NPV	Max	Month	Num2C	IfX
D2R	Pmt	Min	Now	Right	IsNull
Exp	PV	Round	Second		IsNum
Frac	SL	StdDev	Time		IsText
Int	SOYD	Sum	Time2C		NA
Ln		Variance	Year		True
Log					
Mod					
Pi					
R2D					
Random					
Sign					
Sin					
Sqrt					
Tan					

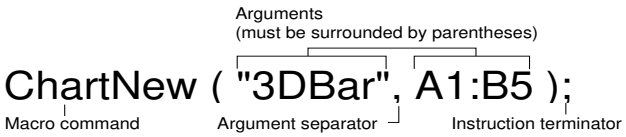
To use a function in a macro you can either type it, or choose it from the Functions panel and fill in the necessary arguments. For more information on individual functions, refer to Chapter 2 in this manual.

About Commands

Commands are instructions that are equivalent to the actions you take when working in a spreadsheet. Some commands directly imitate the ParaSheet menu commands. For example, the macro command *New* corresponds to the Document menu command called *New*. Both commands open a new spreadsheet window. Other commands get information from the spreadsheet so you can use it elsewhere, perhaps in a calculation. For example, the *GetFont* command returns the font in which specified characters are displayed. Still other commands select data or objects; for example, the *Select* command. WILMA commands are organized as shown below:

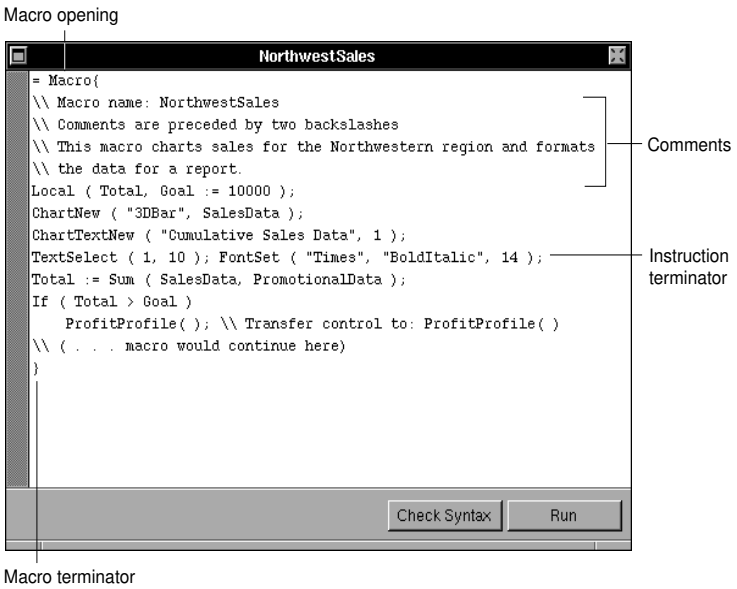
Inside ParaSheet Macros

A macro is made up of *instructions* that typically contain elements of WILMA such as *functions*, *commands*, *statements*, and their *arguments*.



Each macro instruction begins with a function, command, or statement, and must be terminated by a semicolon. (Macro instructions can also begin with variables and macro names, covered later in this chapter.) You can place each macro instruction on a separate line, or put several instructions on the same line.

The structure of a typical macro is shown below:



About Functions

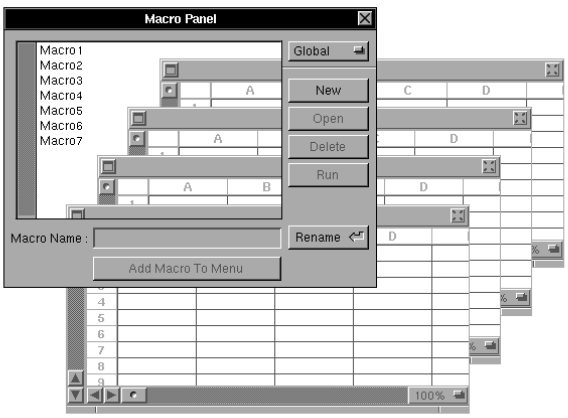
Functions are predefined instructions that perform specific actions on data. Typically these actions involve making calculations. For example, the Sum function adds the values in a range of cells. Other functions perform actions that aren't exactly calculations. For example, the IsText

The Macro menu gives you access to most of the commands for creating, using, and managing your macros. Macro menu commands are described in later sections of this chapter.

The Types of Macros

ParaSheet has three types of macros: *local*, *global*, and *user-defined functions*. They can all complete the same kinds of operations. Whether a particular macro should be global, local, or a user-defined function depends on how you'll use it. In general, make the macro *local* when it represents actions specific to one spreadsheet. For example, if one of your spreadsheets requires a unique format, you could create a local macro to set up that format. A local macro is only available when its parent spreadsheet is open.

Make the macro *global* if the procedure you're automating can be used in more than one spreadsheet. For example, if your printed reports always have the same format, put the formatting commands into a global macro. Or if many of your spreadsheets summarize results across four quarters (Quarter 1, Quarter 2, and so on), create a global macro that formats column labels. Global macros must be created in a macro window. They're always available when you're using ParaSheet, regardless of which spreadsheets are open.



User-defined functions are special macros that can be local or global, and can run from cells or a macro window. For more information, see the "Working with User-Defined Functions" section on page 16.

- WILMA can be used in spreadsheet cells *and* in macro windows.
- Virtually all macro capabilities are available from cells in the spreadsheet. Macros can be run directly from cells and included in formulas, making them part of normal spreadsheet recalculation.
- WILMA lets you use spaces, tabs, and returns freely to format macros in the macro window.
- With WILMA, you can provide a range reference instead of a number, text, or expression to get the value or values necessary to execute the command.

An Introduction to Writing Macros

Macros are written using WILMA, the ParaSheet programming language. You don't have to be a programmer to write macros in WILMA. However, the more you understand about some fundamental programming concepts, the more you'll be able to take advantage of macros.

This chapter describes how elements of the WILMA language can be used and combined to make a macro work; it does not teach programming techniques. Chapter 2 in this manual provides a complete reference to WILMA.

Basic Steps

The basic steps to create a macro are:

- **Plan** — Decide what you want your macro to do, and map out the basic steps that need to be completed. While changing macros in ParaSheet is easy, you can save time and effort with some preparation.
- **Name** — Give your macro a descriptive name; one that you'll remember long after you've created the macro.
- **Write** — Enter commands, functions, and statements by using the Functions panel or by typing them. As you write, you can have ParaSheet check the macro for misspellings, missing arguments, or other syntactical errors.
- **Run** — Run the macro and check the results to ensure that the structure of your macro is correct.
- **Edit** — If necessary, revise the macro or fix any errors.

Macros are stored with identifying names that you can choose like regular commands — or you can run them like programs. The figure below shows an example of a macro:

```
= Macro {
Enter ( "January", B2 );
Enter ( "February", C2 );
Enter ( "March", D2 );
Enter ( "April", E2 );
Enter ( "May", F2 );
…
Align ( "Center", B2:M2 );
}
```

All macros express instructions as commands, functions, and statements from WILMA, the ParaSheet programming language. The example above shows part of a macro that enters the months of the year across the top of a spreadsheet and centers them. English-like words represent the actions in many cases. For example, the commands:

```
Enter ( "January", B2 );
Align ( "Center" );
```

are equivalent to selecting cell B2 and typing January, pressing Return, and choosing *Center* from the Text submenu. Virtually any command you can choose from the menus can also be expressed in a macro language instruction.

What Makes ParaSheet Macros Special

Most spreadsheets make a distinction between *text*, *formulas* and *macros*. If you've used spreadsheets before, you know that text is typically used for labels. Formulas are combinations of cell references, numbers, named ranges, operators, and functions that perform calculations and produce results, and are written using the program's *spreadsheet* language. And, if formulas are not powerful enough, you might use macros—using a separate *macro* language.

In ParaSheet, the spreadsheet and macro languages are combined into a single programming language—WILMA (Windowing Interactive Language for Macro Applications).

ParaSheet is therefore different from most spreadsheets in the following ways:

- It has only one language, WILMA, which includes everything you need to write both formulas and macros.

Chapter 1

Creating and Using Macros

Macros are useful for quickly completing repetitive procedures. For example, at the end of every month, you need to turn your sales activity spreadsheet into a report for the division manager. Rather than repeat the steps to create these reports month after month, you can create a macro that contains the steps, and run the macro whenever you need a monthly report.

What Is a Macro?

A *macro* is a stored series of actions that automates tasks. A macro can be as simple or as complicated as you want to make it. For example, when you choose commands from the ParaSheet menus, you're completing certain spreadsheet actions one at a time. Now, imagine a macro that consists of several ParaSheet menu commands. When you run this macro, it completes the entire series of actions in one step.