Flow Control in the Optimization Program Inverse

(FOR VERSION 3.11)

Igor Grešovnik

Ljubljana, 2005

Contents:

3. FLOW CONTROL

When the optimisation shell *INVERSE* is run, the command file interpretation is triggered immediately after the initialisation. Everything what happens in the shell from that point on is a consequence of the file interpreter's functions (commands) called in the command file.

Beside the functions, which perform shell's built-in algorithms and utilities, there are also flow control functions which make possible programming the command file in a similar way as programming with a high level programming language. This feature of the shell is additionally supported by concepts of user-defined variables and the expression evaluator. By the flow control, user can write portions of code which are executed several times dependent on specified conditions. The conditions can depend on anything what determines the current state in the shell, which includes the results of algorithms and operations which have already been performed.

The branching and looping functions execute specific portions (blocks) of code if certain conditions are true. The blocks of code and conditions must be given in the argument blocks of these functions. Such functions can be nested to an arbitrary depth.

New interpreter's functions can be defined by the file interpreter commands. Their definition block of code is executed simply by calling the newly defined function.

3.1 File Interpreter

3.1.1 File Interpreter Syntax

The syntax of the file interpreter is as simple as possible. The interpreted file consists of function calls (commands), which follow each other. Function calls consist of the function name and function argument block in curly brackets. Basically, the syntax of the shell's interpreter is the following:

funcname1 {argblock1} funcname2 {argblock2} funcname3{argblock3} ...

When the interpreter encounters a function call, it executes the corresponding shell internal function associated with that interpreter function. The interpreter does not pass the arguments from the argument block of the function call to this internal function. It only passes the information about where the argument block of the function can be found (i.e. the file and the position). The associated internal function of the shell extracts arguments from the argument block interpreting their meaning at the same time, and does what it is supposed to do.

Some functions associated with file interpreter functions are able to call the file interpreter to interpret specific blocks of code, usually parts of argument blocks of the corresponding file interpreter functions. Flow control is implemented by such functions.

The interpreter syntax allows that set of commands are enclosed in curly brackets. Such curly brackets, which do not follow a funccction call, are simply ignored.

3.1.2 Argument Passing

Most of the file interpreter's functions require arguments. On these arguments it depends what these functions do and how they do it.

Function arguments appear in curly brackets which follow the function name in the interpreted file. Every function call must have these curly brackets, but the brackets may be empty.

The space inside the curly brackets that follow the function name is called argument block of the function. Function arguments can be specified in this block. The file interpreter does not pass arguments to the shell's functions that are associated with the interpreter's functions. Instead, it passes the information about where the function's argument block is placed. The associated function itself must extract its arguments from the argument block. Therefore the complete freedom of interpreting arguments is left to the associated functions. The file interpreter does not impose any rules about how arguments should be interpreted.

Such freedom allows setting rules for argument formats which best suit the meaning and aim of arguments. On the other hand, this freedom allows for every function to require its own format of arguments. Therefore, some rules are imposed about the formats of arguments that are required by the file interpreter's functions. An overview of these rules is given below.

3.1.2.1 General Rules

Multiple function arguments are usually separated by spaces. Because some arguments (respectively strings that represent them) can themselves contain spaces, it must be unambiguous for each type of argument from where to where it extends. This is the basic requirements for formatting conventions. Let us show some examples how this requirement is took into account.

Strings must be in double quotes and may not explicitly contain the double quote characters. All double quotes within string arguments must be replaced by the appropriate sequences (*d*).

Mathematical expressions are usually contained in brackets. Because all brackets in expression must be closed, the brackets which contain an expression can not be mixed up with brackets which eventually appear within the expression.

Specifications of variable's elements consist of variable name and an optional index list in square brackets. Variable name extends either until the first space or until the first square bracket. If a square bracket is encountered after the variable name, it is clear that an index list is specified and the argument extends till the closed square bracket. Otherwise there is no index list and the argument extends till the last character of the name.

3.1.2.2 Numerical Arguments

Numerical arguments can be given in different ways. The most elementary way is to specify them as numbers, e.g. *124, -3.64567, 6.02e23*. Numbers can be written in a standard format used in most programming languages.

We can specify mathematical expressions or expression evaluator's variables in place of numbers. Numerical arguments can be specified by mathematical expressions in the form

${expr}

where *expr* is the expression which can be evaluated by the expression evaluator. The expression does not need to be in double quotes. Spaces are allowed between the $ sign and the bracket.

The specification of numerical arguments by the expression evaluator's variables has the form

$varname

where *varname* is the name of the calculator's variable.

Where numerical arguments are replaced by mathematical expressions or calculator's variables, these are first evaluated in the expression evaluator and the obtained values are used as arguments. The functions associated with the file interpreter's commands take care of that.

Some examples of numerical arguments specified by mathematical expressions or calculator's variables:

*${3*a+b^3} $a1 $ { getmatrix["Mat2",2,3] }*

3.1.2.3 Mathematical Expressions as Arguments

Some arguments are supposed to be mathematical expressions. Branching and looping conditions in flow control commands are an example of that. In this case, the expressions are not contained in curly brackets that follow the *$* sign. Numbers can be specified in place of these arguments, but are treated as mathematical expressions and are evaluated in the expression evaluator (although there is no need to do that).

When expressions are not the only arguments of a function, they must be somehow separated from other arguments. Usually they are contained in brackets (e.g. conditions in flow control commands). This is because expressions can contain spaces and commas, which are also used to separate arguments.

There is a simple intuitive rule about when numerical arguments and when expression arguments are used. Expression arguments are used if the appropriate arguments can be just numbers only in exceptional cases. This is for example in flow control conditions or in functions like **setmatrixcomponents**.

3.1.2.4 String Arguments

String arguments are seldomly used in the file interpreter's functions. An example of their use is in functions like **write** and in the **execute** (**system**) function.

Because string arguments can contain spaces and commas, they must be in double quotes. An exception is (currently) the **execute** (**system**) command, which takes only one argument and simply takes the whole argument block as string arguments.

String arguments can contain special characters which can not be written in interpreted ASCII files. This problem is overcome with two character sequences which represent these characters. The sequences are replaced by the appropriate special characters if they appear in string arguments. The first character of all such sequences is '\'. Sequences are the folowing:

Table 1: special character sequences.

sequence	**meaning**	**sequence**	**meaning**
\q	single quote (')	\1	(
\Q or \d	double quote (“)	\2	)
\\	backslash (\)	\3	[
\0	null character	\4	]
\n	newline character	\5	{
\r	carriage return	\6	}
\t	tab	\<	{
\s	space	\>	}
\#	hash (#)		

3.1.2.5 Specifications of User-defined Variables, Variable Elements and Sub-tables of Variable Elements

Some functions take arguments that are specifications of user-defined variables (e.g. **movematrixvar**), variable elements or sub-tables of variable elements (e.g. **copymatrix**). We must make difference between arguments that are specifications of variable elements of a specific type and arguments that are objects of a specific type (later are described in the following chapter).

Variables are specified simply by a variable name. For example, we refer to a matrix variable *m1* like this:

m1

Variables can hold whole tables of objects of a specific type. When we refer to an individual element of such variable, we must specify indices of that element. Indices must be listed in square brackets which follow the variable name and be separated by spaces or commas. There can be spaces between the name and square brackets. Indices in the index list can be replaced by mathematical expressions or variables of the expression evaluator according to standard rules. The following specifications of variable element are valid and refer to the same element if the value of the expression evaluator’s variable *i1* is 2:

v [2,3, 2]

v[${4-2} 3 2}]
*v[$i1 ${i1+1}, ${2*i1-2}]*

There must be as many indices in the index list as is the number of dimensions (i.e. rank) of the appropriate variable. If we refer to an element of a variable that has rank zero, the indices are not specified. We can simply specify variable name or eventually put empty square brackets after the name:

a1
a1[ ]

We refer to sub-tables of variable elements in a similar way than to individual elements. The only difference is that the number of indices specified in the index list does not necessarily equal the number of dimensions of the appropriate variable. The specified indices refer by turns to the first few indices of the variable's element table while the rest indices remain free. All elements with the remaining indices running from 1 to the appropriate dimension specify the element sub-table. If no indices are specified, the appropriate sub-table refers to the whole element table of a given variable. When as many indices are specified as the number of variable dimensions, the appropriate sub-table contains only one element. Examples:

v [4 2]
v[3]
m [ ]
m
m[2, 4]
*m [$a ${2*a}]*

3.1.2.5.1 Specification Through String Objects

Variable name is a part of variable, variable sub-table or element specification. It is treated as any other string argument, therefore the same rules as for string arguments apply for variable names in these specifications. Instead of stating variable name directly, we can reference an element of an existent string variable. Such reference consists of the hash sign (#) followed by specification of a string element in a usual form. Additional rule is that such specification must always include square brackets (index specification), even if they are empty since a zero-rank string variable is referenced.

Example:

Let us say that we have a rank 2 string variable *str* that holds a 2 by 3-dimensional table of string elements, and let its 2-1 element be initialised to "*vec1*". Let us then define a rank 1 vector variable that holds 4 vector elements, and name it as string *str[2 1]*. The fourth vector element of this variable can then be referred to as

#str [2 1] [4]

The part *#str [2 1]* now stands for string "*vec1*", the 2-1 element of string variable *str*, and specifies vector variable name, while the part *[4]* is index specification of the appropriate element of vector variable with such name. The above line is therefore identical to

vec1 [4]

as long as string object *str [2 1]* has value "*vec1*".

3.1.2.6 Objects of Various Types

When arguments are data objects of different types, there are basically two ways how to specify them in function's argument blocks. Either we specify values of the data object directly or refer to existing data elements of the user-defined variables of the appropriate type.

When we refer to an existent data element of a user-defined variable, this is done by the # sign followed by the element specification (see the previous chapter!). Example:

#m2[2 4]

 # m0

In this case a copy of the specified variable element is made and this copy is used by the function to which such specification was passed in the argument block. Spaces are allowed between the # sign and the element specification. The specification must be given as described in the previous chapter.

If we directly specify values of an object, the values must be specified according to the rules that apply for a specific data type. For the rules for various data types, refer to the descriptions of the functions for setting variable elements of that specific types.

For example, we can specify the value of a matrix object in the following way:

2 2 {{1 1:1.1}{1 2:1.2}{2 1:2.1}{2 2:2.2}}

Matrix components must always be enclosed in curly brackets. If no components are specified, there must be empty brackets instead.

The same rules as for matrices apply for data objects of other types.

There are some exceptions at specifying data objects. Some functions don't accept objects given by specifications of variable elements. Further, some functions do not require that the value specification is in curly brackets. Both apply for the functions which set elements of user defined variables, such as **setmatrix** and **setvector**.

3.1.3 Functions for Input and Output

3.1.3.1 **write** *{ arg1 arg2 ... }*

Prints the values of its arguments *arg1*, *arg2*, etc., to the standard output of the programme. Arguments can be strings, expression evaluator variables, mathematical expressions, or special character sequences. Arguments must be separated by blank characters (i.e. spaces, newlines, or tabulators).

Usual rules apply for string arguments. They must be in double quotes if they contain spaces. They can contain special character sequences, i.e. two character sequences that begin with the backslash character (\). These sequences are replaced by the appropriate special characters before the string is printed. String arguments can also be specified by referring to an existing string element. Such specification must begin by the hash character (#) followed by string variable name and optional element index specification. An example of string argument is

"This is a\nstring."

Expression evaluator's variables must be given by the *$* character followed by the variable name. Blank characters (spaces, newlines and tabs) are alllowed between the *$* character and the variable name. The current value of the expression evaluator's variable is printed. An example of an expression evaluator's variable as an argument of **write** is

$ v1

Mathematical expressions must be in curly bracked that follow the *$* character. Blank characters are allowed between the *$* character and the curly bracket. Blank characters are also allowed in the brackets since these characters are ignored in the mathematical expressions. The current value of the expression is printed.

$ { a+3+b }

Special character sequences consist of the backslash character and of the specification character that follows immediately the backslash. The appropriate special character is printed (see Table 1 for the meaning of sequences). An example of a special character sequence as an argument of **write** is

\t

Example:

Let us say that we have defined the expression evaluator's variables *a*=5 and *b*=2.5. Then the command

```
write { "The value of a is " $a "\nand the value of"
\n\t "2*a+b\n is " ${2*a+b} "." \n }
```

will generate output like this:

```
The value of a is 5
and the value of
        2*a+b
is 10.
```

The first argument is a string and is printed literally. The second argument is an expression evaluator's variable, therefore its current value is printed. The third argument is again a string. It is printed literally except that the sequence *\n* is replaced by the appropriate special character, i.e. the newline. Then we have two special character sequences, *\n* and *\t*. The appropriate special characters, i.e. the newline and the tabulator are printed because of them. The sixth argument is a string again and is printed literally except that the special character sequences are replaced by the appropriate characters. The seventh argument is a mathematical expression and its current value is printed, i.e. 10. Then we have a string with one character that is printed literally, and a special character sequence, which causes the newline character to be printed.

Let us have execute the following code:

```
setstring {username "John Walker"}
write { "My name is " #{username} ".\n"}
```

This would generate output

My name is John Walker.

3.1.3.2 fwrite *{ arg1 arg2 ... }*

Does the same as the **write** function, except that it prints to the programme's output file. This file is represented by the pre/defined file variable *outfile*.

3.1.3.3 dwrite *{ arg1 arg2 ... }*

Does the same as the **write** function, except that it prints both to the standard output and to the programme's output file. What is printed to the standard output is identical to what is printed to the programme-s output file.

3.1.3.4 read *{ varname }*

Reads a numerical value from the standard input and assigns it to the expression evaluator's variable named *varname*. If the expression evaluator's variable does not yet exist, it is created.

3.1.3.5 setoutputdigits *{ numdigits }*

Sets the number of digits used for output of decimal numbers to *numdigits*. This number is used e.g. at output of numbers with the **write**, **fwrite** and **dwrite** functions, but also with functions for printing vectors, matrices and other variables, e.g. **printvector** or **printvectorvar**.

Interfacing functions like **setparam** are affected by another function, namely **setintfcdigits**.

3.1.3.6 setoutputcharacters *{ numcharacters }*

Sets the minimal number of characters used for output of decimal numbers to *numcharacters*. This number is used e.g. at output of numbers with the **write**, **fwrite** and **dwrite** functions, but also with functions for printing vectors, matrices and other variables, e.g. **printvector** or **printvectorvar**. There is seldomly a need to use this function. One example is when we want to output numbers in a table format so that all numbers in a column occupy the same amount of space. In this case the number of characters for output must be set appropriately greater than the number of digits which

are written, so that the width of output numbers does not exceed *numcharacters* and all number outputs take the same amount of space. By default the minimal number of characters is less than the number of digits, so that each number that is written takes just as much space as necessary.

3.1.4 Expression Evaluator's Functions Concerning Output

3.1.4.1 getoutputdigits *[ ]*

Returns the number of digits currently used for output of decimal numbers. This function is seldomly used since we can usually set the number of output digits independent on its previous values.

3.1.4.2 getoutputcharacters *[ ]*

Returns the minimal number of characters currently used for output of decimal numbers. This function is seldomly used since we can usually set the number of output characters independent on its previous values.

3.2 Branches, Loops, etc.

Branching and looping commands execute portions of code if certain conditions are fulfilled. The expression evaluator evaluates conditions; therefore they must be given by strings which represent mathematical expressions that can be evaluated by the expression evaluator. Both conditions and portions of code must be given in the argument block of the appropriate commands.

Since the expression evaluator evaluates branching and looping conditions, they can include calls to expression's functions for accessing shell's variables. Through them these conditions can include virtually every information, which at a given moment exists in the shell, inoculating results of algorithms and operation which have been performed before the condition evaluation.

3.2.1 Branching, Looping and Other Flow Control Functions

3.2.1.1 **if** *{ (condition) [block1] < < else > [block2] > }*

The **if** command executes the portion of code *block1* if the *condition* evaluates to a non-zero value, otherwise it executes *block2* if it is specified.

condition is an expression which can be evaluated by the expression evaluator. It must be given in round brackets. *block1* and *block2* are portions of code which can be executed (interpreted) by the file interpreter. They must be given in square brackets. If *block2* is specified, the user can optionally insert the word *else* for clearness.

3.2.1.2 **while** *{ (condition) [block] }*

The while function repeats execution (interpretation) of the portion of code *block* as long as the *condition* evaluates to a non-zero value. If the condition is not fulfilled when the function is called, *block* is not executed at all.

3.2.1.3 **do** *{ [block] < while > (condition) }*

The **do** commands executes the portion of code *block* until the *condition* evaluates to zero. *block* is interpreted at least once because the condition is examined after its execution. When the *condition* evaluates to zero, *block* stops being executed and the **while** function exits.

Word *while* can be added optionally for clearness.

3.2.1.4 **interpret** *{ filename }*

Interprets the file named *filename*. The whole file is interpreted. After the file is interpreted, the interpretation continues after the end of the function's argument block.

3.2.1.5 **exit** *{ < numlev > }*

The **exit** command causes exiting *numlev* interpretation levels. Interpretation is then continued. If the number of levels to exit *numlev* is not specified, the interpretation is interrupted completely.

At the beginning of interpretation of the command file the level of interpretation is 1. The level increases by one every time a new block of commands is interpreted. This happens for example when a portion of code in a loop or branch is executed, when a definition block of a user-defined function of interpreter or calculator is executed, when the **analysis** function’s argument block is executed or when a new file is interpreted. At the end of interpretation of such block the interpretation level is again decreased.

3.2.1.6 block *{ blockcode }*

The **block** function simply interprets its argument block *blockcode*. The interpretation level is increased by one when the interpretation begins and is decreased to previous value when it stops. Sometimes it is useful to use the **block** command because anywhere within the basic level of *blockcode* the rest of it can be skipped simply by the command **exit**{1}. We must be careful if we use the **exit** command with the **if** command, for example. Then we must increase the number of levels to exit because the interpretation level of the body of the **if** command is one more than the level of the surrounding code. We must call **exit**{2} instead of **exit**{1} in such cases.

3.2.1.7 goto *{ labelname }*

The **goto** function causes a jump in the interpretation. The interpretation continues from the position of the label named *labelname*. The label position must be given in the code by the **label** command and is considered to be the first character after the closing bracket of the **label** command.

The label must exist in the same code block as the referring **goto** command, i.e. the **goto** command can be used only for jumps within the same interpretation level.

The **goto** function origins from the time when loops were not available in the shell and is considered a bit obsolete.

3.2.1.8 label *{ labelname }*

Specifies the position of the label named *labelname*. The command **goto** *{labelname}* causes continuation of the interpretation from the first character after the closing bracket of the **label** command-s argument block.

3.2.1.9 comment { comments }, * { comments }

Does nothing. This function is used for putting comments in the code. The comments are put to the argument block of the function.

3.2.1.10 trace *{ }*

Causes the interpreter to keep track of the calling sequence of functions that are currently being executed. This information is used in error reports, which make the location of errors easier by providing this additional information. When the spell checker or debugger is run, this capability is automatically used.

3.2.2 update *{ code }*

Makes replacements in *code* as it is indicated by replacement marks that appear in it and then interprets *code*. Replacement marks have the following syntax (note that there are two hash characters):

{ arg1 arg2 arg3 … }

What is pasted in the place of such replacement mark is exactly the same as would be written by function **write** with the same arguments *arg1 arg2 arg3*, etc.

The **update** command is usually used in definition blocks of user-defined interpreter functions to perform replacements of function arguments referenced by their sequential numbers (see description of command **function**!).

The function has also its alternative purpose. It can be used in cases when interpreter code is not completely known in advance, but is to some extent dependent on the current situation, e.g. on a vlaue of some variable. Since it is not known in advance what that code should look like, it must be generated at the time when this is known, i.e. between the execution of the optimization shell. Such situation can be handled in such a way that the code which should be executed is generated by shell functions and written into a file, which is then interpreted. This is not always necessary because sometimes only very small portions of code are not determined in advance.

Warning:

Since replacements are made before the code is executed, the user must be careful that *code* does not include loops in which variables on which replacement is dependent can change. The **update** command must be inside the loop in such cases. For the same reason it does not make sense if we have nested **update** (one **update** command within the argument block of another), except in a very special case when replacement marks within the inner **update** command are generated as a result of replacements in the outer **update** command.

Example:

Let us have two string variables *sname* and *sdef*, where the first one holds a name of a calculator function of two variables and the second one holds the definition of that function. We want to create an expression evaluator function with such name and definition. The following code would do that:

```
update {
  $ { ##{#sname}[x,y] : ##{#sdef} }
}
```

In the **write** function arguments starting by the hash (#) sign are replaced by contents of string elements whose specifications follow the hash sign, therefore if the value of string *sname* is “ff” and the value of the string *sdef* is “x*y”, the code within the **update** argument block will look like this after replacement (i.e. before execution):

```
  $ { ff[x,y] : x*y }
```

A more illustrative example is assignment of vector components to expression evaluator variables. Let us have a vector variable *vec* and want to assign its components to expression evaluator variables *v1*, *v2*, *v3*, …. (as many variables as there are components). The problem is that we don't know in advance how many components will *vec* have, therefore we can not write the code in the following way:

```
= { v1 : getvector["vec",1] }
= { v2 : getvector["vec",2] }
        …….
```

The following code would solve the problem:

```
= { j: 1}
={ dim : getvector[vec,0] }   *{ vector dimension }
while { (j<=dim)
[
  update
  {
        = { ##{ "v"  $j } : getvector[vec,i] }
  }
  = {j:j+1 }
]}
```

The replacement mark "*= { ##{ "v" $j }*" will be replaced at the call to the enclosing **update** by what function **write** would write when called with the same arguments as those in curly brackets following the double hash sign. For example, when *j* is 2, the replacement mark would be replaced by "**v2**", and the whole line of code would look like this:

```
= { v2 : getvector[vec,i] }
```

Note that <u>all replacements</u> are performed at the call of the **update** command. This means that we would get wrong results if the whole **while** loop in the above code would be included in the argument block of the **update** command, because replacement of the mark (only one) would be made when *j* would not have the right value. When the **update** command is within the **while** loop, a replacement is made at each pass of the loop and *j* has the expected value at that time.

The code in the above example is actually not optimal. It doesn't make sense that string "v" is included in the replacement mark because it is known in advance. The code could therefore look like this:

```
= { v1 : getvector["vec",1] }
= { v2 : getvector["vec",2] }
        …….
```

The following code would solve the problem:

```
= { j: 1}
={ dim : getvector[vec,0] }   *{ vector dimension }
while { (j<=dim)
[
  update
  {
        = { v##{ $j } : getvector[vec,i] }
  }
```

```
  = {j:j+1 }
]}
```

in this case we must be careful that there are no spaces between "v" and the replacement mark, because "v" and what is pasted in place of the replacement mark together form a variable name, which must not contain spaces.

3.3 Definition of New File Interpreter Functions

3.3.1 function *{ funcname (arglist) [defblock] }*

The **function** command defines a new file interpreter's function. The first argument *funcname* is the name of the newly defined function. A list of formal argument names *arglist* follows in round brackets. Finally, the function's definition block *defblock* is given in square bracket. This block is interpreted every time a newly defined function is called.

The formal argument names must be separated by spaces or commas. They must be strings consisting of letters and numbers starting with a letter (the same applies to *funcname*). These names are used in the function's definition block. Formal arguments are referenced in the *defblock* by the # sign immediately followed by argument's name. References to arguments are replaced by actual arguments when the function is called.

A function defined by the **function** command can be used as any other function. Its call consists of function name (*funcname*) followed by an argument block in curly brackets. When the file interpreter hits a call to such function, all references to function's arguments in the function's definition block *defblock* are replaces with actual arguments on the string basis, and then *defblock* is interpreted. In the function call arguments must be separated with spaces or commas. If the string which represent an argument contains spaces, it must be enclosed in curly brackets, otherwise it is not considered as one argument.

It is important to know that strings which represent actual arguments at a function call are pasted in the place of references to formal arguments in the *defblock* as strings. This means that there is no general way to check the appropriateness of arguments passed to the function. It is completely on the function's definition how to treat function's arguments, including possible checking of appropriateness. The good thing of such approach is that practically everything can be an argument of a user-defined function, including variables, mathematical expressions, strings, portions of interpretable code, etc.

Referencing arguments by their sequential numbers:

There exists another mechanisms of argument passing for user-defined function, which looks a bit exotic at first sight and can be avoided in most cases, but on the other hand introduces a lot of additional freedom and functionality into the shell's programming.

Besides referencing arguments in the *defblock* by their names specified in the *arglist* preceded by the # sign, the user can also reference function's arguments by their sequential numbers at the function call. The sequential numbers of actual arguments passed to a function must appear in curly brackets which follow the # sign, and can be given by mathematical expressions which can be evaluated in the expression evaluator. Portions of code, which include such references to function arguments, must be included in the argument block of the **update** command. The **update** command evaluates the expressions, which represent the arguments' sequential numbers within its argument block, replaces the references to arguments by actual arguments, and interprets its argument block. Argument references are simply replaced with strings which represent the appropriate arguments at the function call, the same as is the case with arguments specified in the *arglist*.

The expression evaluator's function **numfuncargs** can be used while referencing function's arguments in the *defblock* by their sequential numbers. When called within a definition block of a user-defined function, the **numfuncargs** function returns the number of arguments, which are actually passed, to the function at its call.

An important feature of referencing arguments by sequential numbers is that the replacements of arguments referenced by sequential numbers is not performed at function call as is the case by arguments referenced by names in the *arglist*, but at the call to the appropriate **update** command which includes the corresponding argument references in its argument block. If we have loops in the *defblock*, it can therefore make a difference if we include a whole loop or just a body of the loop in the **update** command's argument block, since the values of expressions which determine the argument's sequential number can vary at consequent passes of the loop's body. The effect of replacing argument references by actual arguments at the call to the **update** function is clearly shown in one of the examples below.

Nested calls to the **update** function do not make sense since argument replacement is performed at the outer-most **update** command. All calls to the **update** functions, which appear in the argument block of another update command, are therefore redundant.

It is important to remember that every reference to function's argument by its sequential number ***must*** be included in an argument block of the **update** command, otherwise it is not replaced by the corresponding argument of the appropriate user defined function's call.

The two mechanisms of argument referencing can be mixed. Argument names listed in the *arglist* always refer to the first *N* arguments in the appropriate order, where *N* is the number of arguments specified in the *arglist*. If an argument is listed in the *arglist*, this does not prevent referencing it by its sequential number rather by its formal name.

Examples:

The first example shows how to define a function *increment* which increments an expression evaluator's variable by 1:

```
function { increment ( var ) [ = {#var:#var+1} ] }
```

The function takes one argument, which is the name of the calculator's, variable. When it is called, it increments the calculator's variable with such name. Code where the functions is used can look like this:

```
= {x:1}
increment {x}
write { "x = " $ x "\n." }
```

This portion of code defines a new calculator's variable named *x* and assigns it the value 1, increments this variable by 1 using the user-defined function *increment*, and prints its value to the standard output.

The next example shows how the user can define the *for* loop which is not defined in the shell:

```
function { for ( begin condition end body )
[
        #begin
        while { ( #condition )
        [
                #body
                #end
        ] }
] }
```

This function takes four arguments: the *begin* block which is interpreted at the beginning and is typically an initialisation of the loop counter, the condition for the **while** loop, the *end* block which is interpreted at the end of the **while** block and typically includes incrementation of the counter, and the *body* block which represents the body of the *for* loop and is interpreted at the beginning of the **while** loop.

For example, here is a portion of code, which uses the newly defined *for* loop and writes numbers from 1 to 100 to the standard output:

```
for { ={i:1}  i<=100  ={i:i+1} { write { $i } write { "\n" } } }
```

When the *for* function is called in the above code, its arguments in the argument block replace formal arguments in the definition block of the function, which gives the following code:

```
        ={i:1}
        while { ( i<=100 )
        [
                write { $i } write { "\n" }
                ={i:i+1}
        ] }
```

This code is then executed which writes numbers from 1 to 100 to the standard output. The last argument was a block of two function calls and contained spaces, therefore it was enclosed in curly brackets.

The next example illustrates referencing the user-defined function's arguments by sequential numbers, which is mixed with standard approach, i.e. referencing by names listed in the *arglist*. The example shows how to define a new expression evaluator's function *evaluatefunctions* which evaluates an arbitrary number of expression evaluator's functions of one variable at a specific value of the independent variable and outputs the results to the standard output and programme's output file:

```
function {evaluatefunctions (x)
[
        ={icount:2}
        while { (icount<=numfuncargs[])
        [
                update
                {
                        write { "#{icount}[" ${#x} "] = "  ${ #{icount}[#x] }  \n }
                        fwrite { "#{icount}[" ${#x} "] = "  ${ #{icount}[#x] }  \n }
                }
                ={icount:icount+1}
        ] }
        write {\n}
        fwrite {\n}
] }
```

The code

```
evalfunctions {0.0 sin cos exp}
```

will generate the following output:

```
sin[0] = 0
cos[0] = 1
exp[0] = 1
```

When the function is called, all references to arguments by name are immediately replaced by the strings representing function's actual arguments in the function's definition block. In the above case all strings "*#x*" are replaced by the string "*0.0*", which gives the following code of the function's definition block:

```
        ={icount:2}
        while { (icount<=numfuncargs[])
        [
                update
                {
                        write { "#{icount}[" ${0.0} "] = "  ${ #{icount}[0.0] }  \n }
                        fwrite { "#{icount}[" ${0.0} "] = "  ${ #{icount}[0.0] }  \n }
                }
                ={icount:icount+1}
        ] }
        write {\n}
        fwrite {\n}
```

The replacement of the argument references by sequential numbers is accomplished each time the **update** command which contains the appropriate references is executed. This

happens in each execution of the **while** loop's body. This is executed for *icount*=2, *icount*=3 and *icount*=4, since the number of arguments passed at the function call is 4 and this is returned by the **numfuncargs** function of the expression evaluator. At the first execution of the **while** loop's body the value of *icount* is 2, therefore strings *#{icount}* in the argument block of the **update** command are replaced by the string "*sin*" and the code which is actually executed looks like this:

```
write { "sin[" ${0.0} "] = "  ${ sin[0.0] }  \n }
fwrite { "sin[" ${0.0} "] = "  ${ sin[0.0] }  \n }
={icount:icount+1}
```

In the next pass of the **while** loop the value of *icount* is 3, therefore strings *#{icount}* are replaced by the string "*cos*" which represents the third argument passed at the function call. The appropriate code that is executed at the second pass of the while loop therefore looks like this:

```
write { "cos[" ${0.0} "] = "  ${ cos[0.0] }  \n }
fwrite { "cos[" ${0.0} "] = "  ${ cos[0.0] }  \n }
={icount:icount+1}
```

Warning:

At the definition of new functions one must be careful with the use of variables. The interpreter does not automatically assume local variables, therefore all variable names are global unless specially defined otherwise by using the function pair *deflocvar* and *undeflocvar* (see the manual on user defined variables). If not using this, then auxiliary variables in the function definitions (counters, variables for storing intermediate results, etc.) must have different names as variables used for other purposes outside the function definition. Interference can usually be avoided by using long, strange or meaningless names for auxiliary variables used in the function's definition block. However, the use of *deflocvar* and *undeflocvar* is recommended in functions.

3.3.1.1 update *{ code }*

The file interpreter function **update** is usually used within a definition block of a user-defined interpreter function. This function first replaces all references to user-defined function arguments by sequential numbers with the strings representing the appropriate actual arguments, and then interprets its argument block (see the reference for the **function** command).

Warning:

Do not do nested calls to **update**! Nested calls don't make sense because all replacements are made in the outer-most **update**.

3.3.1.2 numfuncargs *[ ]*

The expression evaluator's function **numfuncargs** returns the number of arguments passed to a user-defined function in the definition block of which the **numfuncargs** function is evaluated. See the reference for the **function** command for more details!

3.4 Execution of Other Programmes, Printing Notes, Pausing Execution

3.4.1 **system** *{ command },* **execute**

Executes the system's command *command*. This can be any programme or command that can be executed by the operating system on which the optimisation shell runs. *command* is a string that can include the command-line arguments. It does not need to be in double quotes. Special character sequences are not allowed in *command*.

Programme waits until the execution of *command* is finished and then the interpretation of the command file goes on.

3.4.2 **systempar** *{ command },* **executepar**

Executes in parallel the system's command *command*. This can be any programme or command that can be executed by the operating system on which the optimisation shell runs. *command* is a string that can include the command-line arguments. It does not need to be in double quotes. Special character sequences are not allowed in *command*.

The command file interpretation goes on while *command* is being executed.

The function is currently available only on UNIX systems.

3.4.3 **sleep** *{ numsec }*

Delays execution of the programme for *numsec* seconds. The execution can be delayed only for integer number of seconds.

3.4.4 waituser *{ }*

Stops execution of the programme until the user presses the <*Return*> key. The user is promped to press <*Return*>.

3.4.5 waitusernote *{ arg1 arg2 … }*

Prints the values of *arg1*, *arg2*, etc., to the standard output and waits until the user presses the <*Return*> key. The syntax of arguments *arg1*, *arg2*, etc. is the same as at function **write**. The user is not promped to press <*Return*>, therefore the message printed should normally include such suggestion.

3.4.6 usernote *{ arg1 arg2 … }*

Does the same as the **waitusernote** function, except that the user is promped to press the <*Return*> key after the arguments *arg1*, *arg2*, etc. are printed.

3.5 Printing Error Reports

There are some special functions for printing error reports. The user can use standard output functions for this purpose, but these special functions automatically add some information that is normally not accessible to the user, for example, in which file and in which part of it the error has occurred.

3.5.1 reporterror *{ errorstring }*, error

Prints an error report to the standard output and programme output file. *errorstring* is a string that describes the error. It is not supposed to end by a newline character.

The error report contains basic information about where the error occurred (file, line number and function name).

This function is typically used at the definition of new functions via file interpreter.

3.5.2 **reporterrorcode** *{ errorcode errorstring }*, **errorcode**

The same as **reporterror**, except that user must provide a number that identifies the error *errorcode*, which is also printed in the error report.

3.5.3 **reporterrornote** *{ notearg1 notearg2 … }*, **errornote**

This function is similar to **reporterror**, except that the string that describes the error is specified by arguments *notearg1 notearg2* etc., which have the same syntax as arguments of the function **write**.

3.5.4 **reporterrorcodenote** *{ errorcode notearg1 notearg2 … }*, **errorcodenote**

This function is similar to **reporterrorcode**, except that the string that describes the error is specified by arguments *notearg1 notearg2* etc., which have the same syntax as arguments of the function **write**.