Syntax Checker and Debugger for Programme INVERSE

(FOR VERSION 3.11)

Igor Grešovnik

Ljubljana, 27 September, 2005

Contents:

8. SYNTAX CHECKER AND DEBUGGER FOR PROGRAMME *INVERSE*

8.1 Introduction - Motivation and History

Syntax checker and debugger arose to support quick and efficient command file writing. Both tools support an user of the system at discovering and eliminating errors which lead to the behaviour of the system other than expected.

Syntax checker mainly serves for identification of errors which are not a consequence of the user's insufficient understanding of programme's functioning or the problem being solved or logical deficiencies in solution procedure planning, but are rather a result of typing errors or forgetting the exact command names or syntax. Such errors usually don't cause termination of the programme, but often corrupt the results without user's awareness. Although they are almost always reported in the shell's output file, the user can easily overlook these reports. Besides, it is a waste of time if such errors are detected after the program has used several hours of the CPU time. It is therefore recommendable to run a syntax checker over every newly created command file before you run it with a purpose of solving a problem. A normally long command file is checked by the syntax checker in a matter of moments if there are no errors.

It is important to know that identification of errors is limited, especially function arguments are not checked for all functions. This is partially a consequence of the interpreter concept itself since the meaning of arguments is many times known only during the runtime. Checking arguments for all commands would also require a tremendous overhead in the code, which would be hard to justify by the gained benefit. However, the syntax checker can discover some frequent and problematic errors like parenthesis mismatches and misspelling of function names.

When you suspect that you don't get right results or something is going wrong in the solution procedure, but you can't detect any errors in your work, you can run the command file in the debugger. This tool enables you to examine the state of the shell before and after the execution of any function in the command file, so you can locate the points where the discrepancy between the expected and actual state (or behaviour) of the programme arises. The debugger gives you the power of controlling or changing the values of any variable of the shell at any point of execution, and enables you to execute additional commands on-line, during the interpretation of the existing command file. You can therefore check the behaviour of any functional part of the shell in all situation that can appears during the interpretation of the command file.

8.2 Syntax Checker

The syntax checker is invoked by adding the "***-c***" option in the command-line. If you want, for example, to check the command file named "0.cm" and the shell is installed with the name "invan" on your system, this command will do it:

```
invan  0.cm  -c
```

The syntax checker will scan the command file and for every syntax error it finds it will inform you by printing an error message and wait until you press the <Return> key. It will also print a portion of code around the place where the error is located and denote the line which contains the error, and print some additional information, e.g. the name of the file and the number of the line in which error is located.

If you correct the reported errors simultaneously, you must run the checker from the beginning each time you make a correction, otherwise the checker can become confused because of the shift of absolute positions in the file. You can also correct errors after the checker checks the whole command file. In this case it is recommendable that you check the command file again after you make all corrections. You can also make errors when you are making corrections, and new errors can become visible to the checker when the old ones are eliminated.

If you intend to correct all errors after the checking is complete, it can be helpful to make the checker to write error reports also to a file, not only to standard output. You can do it by specifying a file name immediately after the "-*c*" option in the command line (the name must be separated from option by a space). For example, the command

```
invan 0.cm -c report.dat
```

will run the syntax checker over the command file named "0.cm" and write all error reports in the file named "report.dat".

8.3 Debugger

The debugger is invoked by adding the "***-d***" option in the command-line. If you want, for example, to check the command file named "0.cm" and the shell is installed with the name "invan" on your system, this command will do it:

```
invan  0.cm  -d
```

The command files for the shell *INVERSE* are **interpreted**, not compiled and run. This fact imposes some particularities to the shell's debugger which make its functioning different from the functioning of debuggers from programming languages.

None the less, the complete functionality offered to the user is similar than with debuggers for programming languages, only the principles are slightly different. The user can execute the command file step by step, where he has the possibility of stepping over or going into the block of commands or to execute arbitrarily long portions of the command file without stopping. During the breaks in execution, the user gains control through a simple interpreter in which he can input his instructions and requests. Here the user can view portions of code, print the values of variables, executes additional commands understandable to the file interpreter, launch further execution (interpretation) of the command file and specify the position of the next break.

8.3.1 Viewing Portions of Code

Debugger automatically prints portions of code near the current position when the control is transferred to the user. The line of the code in which the current position is contained is marked by the "==>" sign. When the user views a part of code in which the centerline does not contain the current position, the centerline is marked by the ">>" sign. Centerline is a line which is said to be a center of the printed code, although the number of printed lines before and after the centerline may not be equal. This line is marked and its number in the file is printed when a code segment is viewed.

The user can set the number of code lines which are printed each time by the command ***nv*** or ***nview***. This command takes one or two arguments which must be integers. Negative or zero arguments set the number of printed lines before the current position to their absolute value. Positive arguments set the number of printed lines from the current position of the interpretation. If the command is input with no arguments, the user is requested to input both values.

When the control is transferred to the user, he can view any part of the interpreted file. The ***v*** or ***view*** command, when input without arguments, shows a portion of code around the current position (as have been told, the number of lines printed before and after the current position can be set by the ***nv*** command). When ***v*** is called with a numerical argument, another segment of the code is printed. Its centerline is shifted for the number of lines specified by the command's argument which must be an integer. It is shifted according to the centerline of the last viewed segment of code.

With the ***vr*** or ***viewrel*** the user prints a segment of code, in which the centerline is shifted for a certain number of lines with respect to the line of current position. The user must input the number of rows as an argument of the command.

With the ***va*** or ***vabs*** command the user can print a portion of code around the line specified by an argument of this command. The argument must be a positive integer. It means the successive number of the centerline in the interpreted file.

8.3.2 Controlling Execution

The user can execute single commands, commands with their sub-blocks of commands, a certain number of commands or a portion of code till a specific place in the command file at once.

At the beginning of debugging session, the debugger stops interpretation immediately before the first command in the command file and waits user's instructions. Further execution depends on the commands which are input by the user.

With ***s*** or ***step*** command the user executes the next function in the command file. If this function executes one or more sub-blocks of functions, the interpretation is stopped before the first function in the first sub-block and the control is transferred back to the user. If the user wants to execute at once the function's sub-blocks together with the function, he must use the ***S*** or ***stepover*** command. In this case the user gains the control when the next function with all its sub-blocks is executed.

The user can execute a certain number of functions at once. This is done by the ***n*** or ***nstep*** command. This command takes one argument which must be a positive integer and specifies how many functions must be executed before the control is transferred back to the user. If any of the executed functions executes one or more sub-blocks, its execution together with sub-blocks is not regarded as an execution of a single function, but the executions of the functions in its sub-blocks are also counted. After as many functions as specified by command's argument are executed, the control is transferred back to the user who can input new commands.

The ***N*** or ***nstepover*** command is similar to ***nstep***, except that execution of functions that execute one or more sub-blocks of code is regarded as execution of a single function. Besides, the control is transferred back to user if the execution level in which the ***nstep*** command was executed is exited.

The ***x*** or ***exit*** command executes the code without stopping until a lower lever of execution is reached. If it is called without arguments, the control is transferred to the user when the first level is exited. Otherwise, the control is transferred back to the user when as many levels as specified by command argument are exited.

With the ***c*** or ***continue*** command the user triggers the execution of the remaining code. The execution is interrupted only if an active break appears in the interpreted code.

The user can execute a portion of code from the current position till any desired position within the command file or its included files. To do that, he must first set a *break* in the interpreted file at the position when he would like to interrupt execution and retrieve the control. The break is set within the interpreted file with the *break* function. This interpreter's function has the following syntax:

break *{ id [block] }*

id is an integer identification number and is optional. This number enables user to activate or suspend individual breaks. It can be given as a number, as a variable defined in the expression evaluator (a name following the '$' sign) or as an expression which can be evaluated in the expression evaluator (in curled brackets following the '$' sign). *block* is a block of commands which is also optional. This block is executed right before the break interrupts the interpretation and transfers the control to the user (if it does).

Whenever the interpretation hits the *break* command, the identification number *id* is read (evaluated if it is given as a variable or expression) and the break status is checked. If the identification number is not given, it is regarded to be 0. More than one break can have the same identification number. If it is established that breaks with the given identification number are suspended, the interpretation is carried on. Otherwise the *block* block is executed, then the interpretation is interrupted and the control is transferred to the user.

When control is transferred to the user, he can activate or suspend (deactivate) breaks with specific identification numbers. Breaks are activated by the ***ab*** or ***actbreak*** and suspended by the ***sb*** or ***suspbreak*** command. Both command require an integer argument which specifies the identification number of breaks which are activated or suspended. The number can be replaced by the '*' character which means "all breaks". More than one number can be input as parameters of both commands.

The user can at any time check which breaks are active. The ***pb*** or ***printbreak*** command prints the information about active and suspended breaks. The ***tb*** or ***tellbreak*** command examines only breaks with a specific identification number and tells the user if these breaks are active. The command requires an integer argument which specifies which identification numbers to examine.

The user can stop a debugging process by typing in the ***q*** or ***quit*** command.

8.3.3 Examination of the State

The user can check or set the values of variables when he debugs the command file, and he can also check the behaviour of arbitrary interpreter's function at arbitrary point of execution.

The user can calculate the value of any expression which can be evaluated by the expression evaluator at a specific point of execution. This is done by the ***e*** or ***calc*** command. This command requires as an argument the expression which should be evaluated. With the same command the user can also set values of expression evaluator's variables or define new expression evaluator's variables and functions. An expression evaluator's variable is defined by an argument which consists of variable name, a colon and an expression which defines the variable, for example:

e *a:(b+f[c])/2*

If the user wants to assign a numerical value to a variable, he must just replace an expression by a number. The definition of a function looks similar except that a list of formal arguments follows the function name:

e *g[x,y]:sin[x]*sin[y]*

If the ***e*** or ***calc*** command is input without arguments, a calculator which reads expressions and evaluates them is run. For this calculator the same rules as for the ***e*** or ***calc*** commands apply. The only difference is that the user can input as many expressions as you want. To exit the calculator, he must input the string :\q".

The user can use the ***e*** or ***calc*** command not only for examination of the values of expression evaluator's variables, but also for examination of other user defined variables of the shell. This is possible because all user defined variables are accessible through the pre-defined functions of the expression evaluator. If the user wants to know, for example, the current dimension of the user defined vector "vec", he can find out the value by the command

e *getvector["vec",0]*

The ***e*** or ***calc*** command can also serve as an auxiliary pocket calculator.

The user can automatically observe values of a group of expressions whenever he gains the control (a so called “watch” function). An expression is added to the group of the observed expressions by the ***w*** or ***watch*** commands. The expressions which he wants to observe among the others is an argument of this command. If the command is input without arguments, values of all expressions in this group are printed. The users can list only expressions in the group (without their values) by the ***pw*** or ***prwatch*** command.

The user can remove expressions from the group of observed expressions by the ***dw*** or ***delwatch*** command. A serial number of the expression which will be removed from the group must be an argument of the command. The user can determine the expression’s serial number if he inputs the ***pw*** or ***prwatch*** command.

Values of all expressions in the group of observed expressions can be automatically printed each time the user gains control. This can be achieved by the ***aw*** or ***autwatch*** command. This command requires an integer argument. If the argument is 0, automatic printing of expression is switched off, otherwise it is switched on.

The most powerful commands for examination of the state of the shell is the ***r*** or ***run*** command. This command runs additional commands in the file interpreter. Commands which should be run can be an argument of the ***r*** or ***run*** command. If the commands with their argument blocks are too long, the user can input the ***r*** or ***run*** command without arguments. Then he can input several lines of commands which he wants to execute by the file interpreter. At the end he inserts an empty line and the commands which he has input are interpreted.

The user can use any function installed in the file interpreter by the ***r*** or ***run*** command. This way he has a total control over the state in which the shell is in a given moment. This command can replace examination of the shell’s user defined variables using the ***e*** or ***calc*** command because the file interpreter’s *write* command can be used for printing expression values and = or *$* commands can be used to define variables or functions of the expression evaluator or to assign values to its variables. For example, the command

```
r  write {“a+b = “ ${a+b} \n}
```

does the same as the command

```
e  a+b
```

In many cases printing values of the user defined variables using the ***r*** or ***run*** command is more comfortable than using the ***e*** or ***calc*** command because the user can use the file interpreter’s functions which are specially designed for printing variables of complex types. For example, the user can print a whole vector *vec* simply by

```
r  printvector{vec}
```

Of course, the user can also set all types of user defined variables by using the ***r*** or ***run*** command since he has access to all file interpreter’s functions and can do by typing in commands during the debugging whatever he could do by writing commands into the command file. This way the user can check how everything would look like if the values of specific variables would be different than they are, which can sometimes be very useful. The user can also examine how different file interpreter’s functions would behave if they were called at a specific point of interpretation or even how the whole system would behave if whole segments of code were different.

Individual lines of file interpreter code can also be input directly, without putting the **r** or **run** command in front. In this case, one line is always run at a time. In fact, any time input sent to the debugger contains a curly bracket (*{*), this line is run in the interpreter. Difference between this and the **r** or **run** command is that commands input directly can not be repeated by inserting an empty line.

Command ***rd*** or ***rundebug*** is similar to ***r*** or ***run***, except that the code which is input by the user is not only run, but also debugged.

Commands which are input by user with the ***r*** and ***rd*** are written to a file. The user can specify its name by the ***rf*** or ***runfile*** command, otherwise the default file "*0debug.cm*" is used.

During the debugging procedure, the user can not only examine the state within the shell, but also the state in the system in which the shell is run. By the ***!*** or ***system*** command, the user can run any command known to the system in which the shell is run. The system command which the user wants to run, together with its arguments, should be input as an argument of the ***!*** or ***system*** command. If the command is input without arguments, a simple shell is invoked in which the user can input system commands one by one until he exits the shell. He exits this shell by inserting "*x*" or "*q*".

8.3.4 Some General Instructions

Debugging of a shell's command file consists of two main types of events: interpretation of code segments by the file interpreter and execution of user's commands. Between interpretation of the code segments the control is passed to the user so that he can input his instructions for the debugger. At these points a simple command interpreter is run which waits for user's commands and executes them. Some of these commands cause exiting the debugger's command interpreter and interpretation of another segment of code in the command file, the length of which depends on user's instructions.

The user must only remember the main concepts of the debugging because he can obtain a list of command by the ***?*** or ***h*** command. The arguments required by individual commands are briefly described along the command names.

Most of the debugger's (line interpreter) commands require some arguments. If they are input without them, the user is usually prompted to insert these arguments separately. In this case, at the commands which set the values of something, the user is also informed about the current state of what should be set by the specific command. The user therefore doesn't need to remember for each commands what kind of arguments it requires.

Some commands (***e*** or ***calc***, ***!*** or ***system*** and ***r*** or ***run***) invoke a simple interpreter. In this case the user is informed what he is supposed to do.

A useful feature is remembering the last command. If the user inserts an empty line in the debugger's line interpreter, the last command is repeated.

8.4 A List of Debugger’s Commands

Debugger’s commands are listed in this chapter. Their descriptions follow in a logical order according to their function. All the command names are listed in an appropriate title, separated by commas. Argument lists of commands follow their names. Short descriptions of what the commands do follow the titles.

8.4.1 ?, h

Prints a short help.

8.4.2 q, quit

Finishes the debugging process. The interpretation stops at the point where this command is invoked.

8.4.3 s, step

Executes the next file interpreter’s command. If this command contains sub-blocks of commands, the interpretation is interrupted before the first command of the first sub-block which will be executed.

8.4.4 S, stepover

Executes the next file interpreter’s command. Interpretation is not interrupted in a sub-block of the command unless it contains the **break** function. If this is the case, the ***c*** or ***continue*** command will launch interpretation until the end of the function containing this sub-block.

8.4.5 n, nstep *num*

Executes the next *num* commands in the command file. Functions which contain one or more executable blocks are not treated as single functions, therefore the interpretation can also break within a block of such function.

8.4.6 N, nstepover *num*

Does the same as ***nstep***, except that execution of functions that execute one or more sub-blocks of code is regarded as execution of a single function. Besides, the control is transferred back to user if the execution level in which the ***nstep*** command is executed is exited.

8.4.7 X, exit *num*

The ***x*** or ***exit*** command executes the code without stopping until a specified lower lever of execution is reached. If it is called without arguments, the control is transferred to the user when the first level is exited. Otherwise, the control is transferred back to the user when as many levels as specified by command argument are exited.

8.4.8 c, continue

Executes the code till the next active **break** command in the command file or till the end of interpretation.

8.4.9 ab, actbreak *id*

Activates all breaks with the identification number *id*. If *id* is "*" instead of a number, it activates all breaks.

8.4.10 sb, suspbreak *id*

Suspends all breaks with the identification number *id*. If *id* is "*" instead of a number, it suspends all breaks.

8.4.11 pb, prbreak

Prints information about active breaks.

8.4.12 tb, tellbreak *id*

Prints if breaks with identification number *id* are active or not.

8.4.13 v, view *shift*

Views s segment of code around the current position of interpretation. If the command is input by an argument (*shift*), it first shifts the current position in the code which is viewed for *shift* lines and prints few lines of code around this position.

8.4.14 vr, viewrel *shift*

Print a segment of code in the currently interpreted file around the line shifted for *shift* lines from the current position.

8.4.15 va, viewabs *linenum*

Prints a segment of code in the interpreted file around the line *linenum*.

8.4.16 nv, nview *num1 num2*

Sets the number of lines before (*num1*) and after (*num2*) the centerline which are printed when the code is viewed.

8.4.17 e, calc *expr*

Evaluates the expression *exp* in the expression evaluator. If the argument is missing, it runs an interpreter which reads and evaluates expressions one by one.

8.4.18 w, watch *expr*

Adds expression e*xpr* to the watch table. If the command is input without arguments, values of all expressions in the watch table are printed. This is also done every time the control is passed to the user if the option for automatic watching is set (see the **aw** or **autwatch** command).

8.4.19 dw, delwatch *num*

Removes the expression with serial number *num* from watch table. Serial numbers of expressions can be seen by invoking the ***pw*** or ***prwatch*** command. The user should be careful when checking serial numbers of expressions in the watch table because serial numbers can change when expressions are added to or deleted from the watch table.

8.4.20 aw, autwatch *switch*

If *switch* is zero, it turns automatic watching off, otherwise it turns it on. When automatic watching is turned on, values of all expressions in the watch table are printed whenever the control is passed to the user.

8.4.21 pw, prwatch

Prints all expressions from the watch table.

8.4.22 r, run *comblock*

Interprets commands *comblock* by the file interpreter. If *comblock* is not specified, it runs a simple shell where the user can input the commands one by one. These commands are then interpreted.

Individual lines of file interpreter code can also be input directly, without putting the **r** or **run** command in front. In this case, one line is always run at a time. In fact, any time input sent to the debugger contains a curly bracket (*{*), this line is run in the interpreter. Difference between this and the **r** or **run** command is that commands input directly can not be repeated by inserting an empty line.

8.4.23 rd, rundebug *commands*

The same as ***r*** or ***run***, except that the code input by the user is also debugged, not only run.

8.4.24 rf, runfile *filename*

Sets the name of the file into which the user's commands will be written, to *filename*. The default file name is "*0debug.cm*", so that user's commands for the file interpreter are written to this file, if not specified differently by the ***rf*** or ***runfile*** command.