Idsys Manual

Computation Structures Group Memo 211
11 December 1981

Keshav Pingali

Vinod Kathail

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139

This work was supported by the Advanced Research Project Agency of the Department of Defense under Office of Naval Research contract no. N00014 -75-C-0661.

Table of Contents

1 Some Remarks about our Id Implementation

The Id compiler you will be using implements Id as described in section 2 of the paper "An Asynchronous Programming Language and Computing Machine", Arvind, Gostelow, K.P. and Plouffe, W.E., TR-114a, University of California, Irvine, Dec. 1978, with the following differences/addenda :

1. The Id reader automatically converts all upper case characters to lower case, unless it is reading a string. Therefore, you can use either case in an Id program.

2. The **compose** operator has not been implemented.

3. A file given to the compiler can only have procedure definitions in it. You are *not* allowed to compile Id expressions of any other kind.

4. Procedures in TR-114 Id cannot have free variables in them. This means that a TR-114 Id program must be an expression that is executed for value. Since we would like to define libraries of procedures that can be used many times, and we do not want to pass these library procedures explicitly into every procedure that uses them, we permit free-variables in a restricted sense - any variable that is used but not assigned to in an Id procedure is assumed to be a procedure in your environment, which will get defined before you execute this procedure. At compile time, a warning is printed to this effect.

5. The syntax for BLOCKs in our Id is

   BLOCK ::= (**let** *ASSIGNMENT-STATEMENT-LIST*
   in *EXPRESSION-LIST*)

   For example, the following is a legal Id block -

```
let x <- y+z;
    y <- 3
in x+y, x*y)
```

6. The following relational operators are defined in our Id:

OPERATOR	ACTION	PRIORITY -LEVEL	EXAMPLE
or	BOOLEAN OR	1	**true or false**

and	BOOLEAN AND	2	**true and true**
not	BOOLEAN NOT	3	**not false**
=	EQUAL	4	3 = 3
~=	NOT EQUAL	4	3 ~=4
<	LESS THAN	4	3 < 4
<=	LESS THAN OR EQUAL	4	3 <= 4
>	GREATER THAN	4	4 > 3
>=	GREATER THAN OR EQUAL	4	4 >= 3

7. IDSYS executes an assignment statement by completely evaluating the right-hand side of the statement, and then performing the assignment. Therefore, no data dependencies are allowed between variables being assigned to by the same statement. For example, the following code is illegal.

```
a,b  <-  2,  a+2;
```

Note that such dependencies are allowed in TR-114 Id.

Similarly, the compiler orders the statements in a block or a loop according to the data-dependencies of their right-hand sides. The following code is, therefore, illegal.

```
a  <-  f(b);
b,c  <-  3,g(a);   !Circular   data-dependency
                    between   the  two  statements!
```

8. For user convenience, we support a variety of structure formats. In an Id program, the data structure <1 : 1, 2 : <>> (i.e., a structure with the value 1 at selector 1 and <> at selector 2) can be written in any of the three forms shown below :

```
<> + [1]1 + [2]<>

<1: 1, 2: <>>

<1,<>>
```

Only the last format needs explanation - an expression list enclosed in angle-brackets is understood to define a structure with selectors 1,2,3 ..... (with as many selectors as there are values). The expression list can consist of arbitrary Id expressions. One note of caution : owing to a kink in Id syntax, a structure of the form <-1:1> will cause the compiler as well as the IDSYS command interpreter to protest. The reason is that <- is also the assignment symbol (!). If you must use structures in this format with negative selectors, then put a space between the '<' and the '-'. For instance, the above structure could be re-written as < -1:1>, and you will not get any complaints. (Sorry about this, but we are going to fix this soon.)

9. In addition, the current Id implementation permits I/O. This is described in the next section.

2 I/O in Id

The right way to implement I/O in Id is through streams. Although we have not implemented streams, we have provided I/O commands which we have implemented using I/O in LISP. The following is an informal introduction to I/O in Id.

Let us define an "I/O object" to be a number, string, boolean (i.e., TRUE or FALSE) or structure. For example, the following are I/O objects -

```
20000.45E4

"THIS IS A STRING"

TRUE

<1 : 23, 3 : 45>
```

The following is not an I/O object -

```
procedure silly(a)
        (a+1)
```

The basic "unit" of I/O in Id is an I/O object. This means that a single I/O operation will

result in the input or output of one I/O object. Notice that although a structure is itself composed of I/O objects, it is read in or printed with just *one* I/O operation. The Id view of a file on which I/O is being done is that it is a *sequence* of I/O objects. The following example should make this clear.

Consider a file that contains the following :

```
"The first selector in" <1 :2,2 :<>> "is" 1 "-" TRUE
```

For the purpose of this discussion, you can imagine a pointer that advances through this file each time an I/O object is read from it. Before we can do any I/O on a file, we must first create an association between this pointer (which we will call a file-pointer) and the file. Initially,this pointer is at the beginning of the file. This is shown below by the "↑" mark -

```
"The first selector in" <1 :2,2 :<>> "is" 1 "-" TRUE
↑
```

The Id procedure that reads objects in from a file is **idread** When an **idread** operation is executed on this file, there are two things that happen:

1. the string "The first selector in" will be returned to you

2. the file pointer is advanced, so that it is *ready* for the next **idread** (if any).

The file will now look like this:

```
"The first selector in" <1 :2,2 :<>> "is" 1 "-" TRUE
                        ↑
```

The next call to **idread** will result in

```
"The first selector in" <1 :2,2 :<>> "is" 1 "-" TRUE
                                    ↑
```

and the value <1:2, 2:<>> will be returned to you, and so on. One important point - when the last value in this file (i.e., TRUE) is read, it is passed to you as a *Boolean*, and *not* as a string (remember that all strings must be enclosed in double quotes).

The picture for output is very similar.

Therefore, to do I/O on a file, you must do the following :

1. associate a pointer with the file. This is called "opening" the file.

2. use this pointer to read or write on this file

3. when all your I/O on this file is done, you must remove this association between the pointer and the file. This is called "closing" the file.

At any given time, a file must be open either for input or for output - it is an error to attempt to open the file for, say, output while it is open for input. There is absolutely no reason why this *has* to be so - it was an implementation decision. You can, of course, open a file for output, write I/O objects into it, close it and *then* open it for input and read back the I/O objects. The only situation in which you are allowed to have many pointers to one file is this : you can open a file any number of times for *input* - this gives you many pointers to the same file. You can use each of these pointers separately to read from this file. Of course, you must close each file pointer separately as well.

The terminal is considered to be a special kind of file, and the commands for I/O from files can be used for doing I/O from the terminal. The terminal is the only exception to the rule that a file cannot be open for both input and output at the same time. When IDSYS is started, the terminal is opened for I/O automatically. You should *not* open the terminal for input or output or anything strange like that.

Let us now consider the Id command for opening files for input. The syntax for this command is

idopen–in(FILE–STRING, ... FILE–STRING)[1]

where a FILE–STRING is a string which the operating system understands as the name

[1]Owing to a font bug, all our underscores look a little like minus signs. Minus signs can occur in an Id program only to represent the minus sign, inside strings or as the second character of the assignment symbol <-. They cannot occur inside an identifier name. Hence, there should be no confusion.

of a file. For example, in XX,

- "program.output"
- "program.output.15"
- "<id>program.output.15"
- "ss:<id>program.output.15"

are all valid file-strings. The usual defaults for directory, version number etc. apply.

This command is treated by the Id system like it would treat any ordinary procedure call. The value returned by the **idopen−in** command is a file-pointer - the only thing you should do with it is to assign it to an identifier. You can, of course, pass it as an actual parameter to a procedure call, return it from a procedure call etc. but ultimately, it should get assigned to an identifier. Any I/O on this file must have the *file-id* as a parameter - not the file-string. For example, the following program fragment is legal:

```
in_file   <-  idopen−in("input.file");
b,c       <-  idread(2,  in_file)
```

The file-id for the terminal is the reserved word IDTTY. When a call to **idread** with idtty as an argument is encountered during the execution of your program, IDSYS will prompt you and ask you to type out the appropriate number of values on the terminal. When typing out these values, you must use spaces, tabs, or <LF>s[2] as separators and end the list with a <CR>.

Some other points to note are the following:

1. It is an error to attempt to read in any Id value that is not an I/O object. I/O objects in the file must be separated by "white space characters" i.e., blanks, <CR>, <LF>, TABs etc.

2. You can give any Id value as an argument to **idwrite**. However, the result is

[2]The Line Feed Character

meaningful only if it is an I/O object. For example, if it is a procedure, the string "**COMPILED PROCEDURE" will be printed out. Similarly, giving it a file-id will result in the printing of the file-string that corresponds to the file-id. A space is printed automatically after every I/O object that is written on a file. This enables you to read back I/O objects that you wrote on a file. When a structure is written, it is pretty-printed. The printing of every structure is always started on a new line. Any values printed after a structure are also printed starting on a new line. This is for readability.

3. A call to **idwrite**, **idprint** or **idclose** always returns exactly **one** value - true.

4. If you attempt to read in more things than there are on the file, you will get an "End of File" error.

The syntax for the various commands is given below :

- OPENING A FILE

  * **idopen—in**(FILE-STRING, ...,FILE-STRING) - This causes the Id system to open for input all the files specified in the argument list. Executing this command returns an expression-list that must be assigned to an identifier list of the same arity. For example,

```
x<-   idopen—in("program.file")

x,y<-   idopen—in("program.file","foo.bar")

x,y<-   5,  idopen—in("program.file")
```

  are all correct statements in Id. The last statement is supposed to illustrate that the expression list returned by an **idopen—in** can be treated like any old expression list in Id.

  * **idopen—out**(FILE-STRING, ...FILE-STRING) - This is similar to **idopen—in** except that the files are opened for output.

  * **idopen—append**(FILE-STRING, ...FILE-STRING) - This is similar to **idopen—out** except that if there is a pre-existing file with the same name, the output is appended to the end of this file, whereas **idopen—out** would create a new instantiation of the file, and start writing from the beginning of the file.

- CLOSING A FILE

* **idclose**(FILE-ID, ... FILE-ID) - This closes the files that correspond to these file-ids. Idclose always returns one value and that is *true*. If a file corresponding to a file-id is already closed, nothing happens. If one of the arguments is not a file-id, then a non-fatal error occurs, and an error message is printed out.

- *READING FROM A FILE*

* **idread**(ARG1, ARG2) -where arg1 must be an integer and arg2 must be a file-id. This is a command to read arg1 number of values from the file specified by arg2. The default for arg1 is 1, and for arg2 is idtty. For example, the following are legal Id expressions

  - **idread**(n, my – file) reads in n arguments from my – file

  - **idread**(n, idtty) reads in n arguments from the terminal

  - **idread**(n) same as the previous example

  - **idread**(my – file) reads in 1 argument from my – file

  - **idread**( ) reads in 1 argument from the terminal

- *WRITING TO A FILE*

* **idwrite**(ARG1,.....ARGn) - where ARG1,...ARGn-1 are expressions whose values are to be output and ARGn is file-id. There is *no default* for ARGn - if output to the tty is desired, ARGn must be the reserved word idtty. **idwrite** can also be called with only one argument. This argument must be a file-id, and doing this results in a <CR><LF> being output to the file. **idwrite** always returns *true*. Anything written by **idwrite** can always be read back in by an **idread**. For example,

  - **idwrite** (2+3, 3+4, my – file)

  - **idwrite** (w, idtty)

  - **idwrite**(idtty)

  are all correct expressions.

* **idprint** (ARG1,....ARGn) - This is exactly like **idwrite** except that double-quotes are not printed around strings. This exists for fancy I/O only - attempts to read back things printed with **idprint** may result in errors.

The following program is an example of I/O in Id. Consider the file we had earlier. The program

```
(let
    ptr1,ptr2   <- idopen-in("in.file","in.file");
    a,b,c,d,e   <- idread(5,ptr1);
    f           <- (if idread(1,ptr1)  then idread(1,ptr2)
                                       else b);
    dummy1      <- idclose(ptr1,ptr2)
in f)
```

will return the string "The first selector in".

This completes our discussion on how Id as implemented by our compiler differs from TR-114 Id.

3 IDSYS - A PROGRAMMING ENVIRONMENT FOR ID

We will now describe how to enter IDSYS and write, compile, debug and run Id procedures. To enter IDSYS, type IDSYS<CR> to the monitor. IDSYS should announce itself with a message and type its prompt character (=>) out on the screen. Typing a ? will print out a list of the commands that IDSYS recognizes along with information about these commands. Typing a question mark after a command name will print out information about that specific command. IDSYS is essentially a command interpreter which reads in a command line from the terminal, executes it and prints information on the terminal about the result of the execution. To exit temporarily from IDSYS, type ↑C. To restart IDSYS, type IDSYS<CR> to the monitor, and you will restart at the point that you typed a ↑C. Executing the QUIT command (typing QUIT<CR> to IDSYS), will kill IDSYS and take you back to the monitor.

3.1 Users View of the System

What the user sees of the system may be logically divided into three parts :

1. the procedure environment

2. the file environment

3. IDSYS commands to interface the user with these two environments

The procedure environment of IDSYS consists of all user-defined procedures that can be called from IDSYS. When you start IDSYS, your procedure environment is empty. However, IDSYS contains the following procedures which you can use without having to define them. These are not considered to be part of the procedure environment - for example, when you ask for your procedure environment to be printed out, these procedures will not be listed.

- SQRT - the square-root function

- MAX - the "maximum" function. For example, MAX(2,3.0, 1.0E-9) returns 3.0

- MIN - the "minimum" function.

- SIN - the "sine" function

- COS - the "cosine" function

- ATAN - the "inverse tangent" function

- NLOG - the "natural log" function

- FIX - the "smallest-integer-less-than-or-equal-to" function

- FLOAT - the "make floating-point" function

You can add to your procedure environment by compiling a file containing Id procedures and loading the object-code file produced. Every top-level Id procedure in this file will get added to your environment. However, if a procedure with the same name already exists in your environment, IDSYS will ask you if you want to load in the new procedure definition or retain the old procedure definition. Opting to load in the new procedure definition will

,ofcourse, remove the old definition from your procedure environment. If you choose to retain the old definition, then loading resumes from the next top-level procedure. You are allowed to redefine IDSYS procedures like SQRT, SIN etc.

Once a procedure has been added to the environment, it can be called from IDSYS. There are IDSYS commands to delete procedures from the environment, find out what procedures there are in the environment, how many arguments a certain procedure in the environment takes etc. The section on IDSYS commands should be self-explanatory.

The other environment you work with is your file environment. At any point, the file environment consists of all files that are currently open. When you start IDSYS, your file environment is empty. You can open files inside Id programs, as described above, or at the Id system level. The syntax in these two cases is different. The IDSYS commands **infiles**, **outfiles** and **appfiles** enable you to open files at the system level for input, output and appending respectively and add them to your file environment. Section 11 explains these commands in more detail. Any files opened by an Id program you are running are also put into this environment. You must use the **idread**, **idwrite** and **idprint** commands to read from or write in files. The command FILES lists all files that are in the file environment as well as whether they have been opened for input, output or appending. By using the **idclose** command to close files, you delete them from the file environment.

Some additional details:

- The command

  => *RESET(a)<cr>*

  will close the file *a* and then open it again, and assign it back to *a*. Think of this command as moving the file-pointer back to the beginning of the file.

- Suppose you call a procedure FOO in your environment from IDSYS. If you open some files in FOO (or in procedures that FOO calls), and forget to close them, then IDSYS automatically closes them just before it returns you to the top level. You should not, however, rely on this to close your files - it is bad programming practice. Similarly, all files that were opened at the top level (i.e.,

by **infiles**, **outfiles** etc.) are closed by IDSYS if you give it the QUIT command.

4 CREATING AND EDITING FILES

You can edit ID programs in EMACS. If you would like to use EMACS while in IDSYS, type EMACS<CR> to IDSYS. After editing, you must exit EMACS by typing either ↑C or ↑X↑Z. The preferred mode of exit is ↑X↑Z.

5 COMPILING ID FILES

5.1 RUNNING THE COMPILER

=>compile("foo.baz")<cr>

The COMPILE command compiles ID procedures in the specified file into object code that can be loaded and run by IDSYS. Normally, this command will create two new files named FOO.LISTING and FOO.LSP. The file FOO.LISTING will contain the code from the file FOO.BAZ with line-numbers inserted in front of every line as well as any messages generated by the compiler. The object code will be in the file FOO.LSP.

The compiler generates two types of messages - WARNINGs and ERRORs.

An ERROR message is generated when the compiler detects a syntactic or semantic violation of the rules of ID. In case of a syntactic violation, the compiler will print out the line it was parsing, the previous line and the place where the unexpected syntactic entity was encountered. This information is sent to both the terminal and the listing file. All files opened by this run of the compiler are then closed. A syntactic error therefore results in the termination of the compilation.

A semantic violation occurs in situations like the violation of the single assignment rule, circular definitions among variables etc. Error messages in these cases contain line numbers and some information as to why the compiler did not like your program. As before, error messages are sent to both the terminal and the listing file, but in this case, the

compilation is not terminated. However, no object code is produced for a procedure in which a semantic error occurred.

A WARNING message is generated when the compiler detects a condition that is not illegal but is peculiar enough (in its opinion) that the user be alerted. For example, a warning message is generated when an identifier is assigned to but not used in a program.

Sometimes, the names of the default listing and object files (*.LISTING and *.LSP) may not be convenient. This may happen if a file with that name is already open, or is being used for some other purpose. In that case, type COMPILE<CR>. IDSYS will ask you for the names of the input, listing and object files. IDSYS also suggests the usual defaults for the listing and object files - if that is satisfactory, type <CR>. Otherwise, type the file name you want. An example of this is shown below (user input is shown in italics).

```
=>compile<cr>
INPUT FILE NAME:super.test<cr>
FILE FOR PROGRAM LISTING (DEFAULT IS PS:<KESHAV>super.LISTING)<cr>
FILE FOR TARGET CODE (DEFAULT IS PS:<KESHAV>super.LSP) super.listing<cr>
UNABLE TO OPEN THE FILE "SUPER.LISTING" --- IT IS ALREADY OPEN.
TRY ANOTHER FILE NAME ->super.lsp<cr>
    MULTIPLY COMPILED
COMPILATION OVER
=>
```

In the example above, IDSYS tried to open the file super.listing for outputting the object code, but found it was already open. Hence, it generated a message and asked for another file name. If IDSYS is unable to open this file as well, all files opened by this run of the compiler are closed and control is returned to the top-level of IDSYS.

As each procedure is compiled, IDSYS types out the name of the procedure on the terminal. At the end of the compilation, the message COMPILATION OVER is printed on the terminal.

6 CALLING ID PROCEDURES

=> foo(<1 :2,2 :3>, 3, "CHECK", my−proc, **idread***(2,my−file)<cr>*

Id procedures in the procedure environment can be called from IDSYS. The actual parameters of a procedure call in IDSYS can be :

- numerical, string, boolean and structure constants
- names of procedures in the procedure environment
- file−ids of files in the file environment
- calls to **idread**

In the example above, FOO is a procedure in the procedure environment that takes in 6 arguments. The first argument is a structure, MY−PROC should be the name of a procedure in the procedure environment while the call to **idread** will read in 2 arguments from the file my−file.

If all goes well, and the procedure call is executed correctly, the values returned by the procedure will be printed out on the terminal. If any errors occur, appropriate error messages are printed out on the terminal.

7 DEBUGGING ID PROGRAMS

As of now, the only debugging facility in IDSYS is **trace**. If a procedure is being traced, then whenever it is called, IDSYS will print out the arguments that are passed to it (i.e., the actual parameters of the procedure call) as well as the values returned by the procedure to the caller. By specifying some procedure names in the FROM-LIST part of the **trace** command, it is possible to get the trace information printed out only when the procedure is called (directly or indirectly) by one of the procedures in the FROM-LIST. The following are legal **trace** commands in IDSYS:

trace(MY – PROC)
This traces all calls to MY – PROC.

trace(MY – PROC FROM FOO/BAZ, OUR – PROC FROM BUZZ/FUZZ)
This traces calls to MY – PROC from FOO and BAZ,
and calls to OUR – PROC from BUZZ and FUZZ.

8 AN EXAMPLE

In this example, we compile a file called "sieve.buggy" which contains two procedures called "driver" and "sieve". Driver is called with an integer n - it returns a structure containing all prime numbers less than or equal to n. The procedure "sieve" had a minor syntactic error in it which was caught by the compiler.

THE BUGGY PROGRAM

```
procedure driver(n)
    (initial list <- <>; A <- <>;
     for i from 2 to n do
          new list[i-1]  <- i;
     return sieve(list,n-1))

procedure sieve(list,   n     !LET'S  HAVE  A SILLY  ERROR!
    (initial p <- <>;  i <- +1;
     while n = 0 do
          p[i]  <- list[1];
         new i <- i+1;
         new list,  new n <-
                   (initial a<-  <>;  k<-  1;
                             prime  <- list[1]
                    for j from 2 to n do
                         new a, new k<-
                                (if mod(list[j],prime)=0
                                     then a+[k]list[j],k+1
                                     else a,k);
                    return a,k-1);
    return p)
```

INTERACTION WITH IDSYS

```
=>compile("<keshav>sieve.buggy")

--DRIVER COMPILED
***WARNING- IDENTIFIERS UNDEFINED IN THIS PROCEDURE ->
        SIEVE  USED IN LINE 5
+++ERROR--- UNEXPECTED TOKEN "(" ENCOUNTERED IN LINE 8.
   7    procedure sieve(list, n
   8            (initial p <- <>; i <- +1
                ↑
=>emacs
.
.
.
=>compile("sieve.buggy")
--DRIVER COMPILED
***WARNING- IDENTIFIERS UNDEFINED IN THIS PROCEDURE ->
        SIEVE  USED IN LINE 5

--SIEVE COMPILED
COMPILATION OVER
=>load("sieve.lsp")
----- Loading file sieve.lsp .....
      Procedure DRIVER  defined.
      Procedure SIEVE  defined.
----- Done.
=>driver(100)
<1 : 2,2 : 3,3 : 5,4 : 7,5 : 11,6 : 13,7 : 17,8 : 19,9 : 23,
10 : 29,11 : 31,12 : 37,13 : 41,14 : 43,15 : 47,16 : 53,
17 : 59,18 : 61,19 : 67,20 : 71,21 : 73,22 : 79,23 : 83,
24 : 89,25 : 97>
=>quit
----- All open files closed.
@
```

9 IDSYS COMMANDS

There are two types of IDSYS "commands" - interrupts and commands that are interpreted directly by the IDSYS command interpreter. The interrupt commands you can type in IDSYS are :

↑A - This interrupts whatever IDSYS is doing and returns you to the IDSYS command interpreter level. Apart from< your procedure and file environments, everything is reset. This command is useful for breaking out of infinite loops etc.

↑C - Ofcourse!

↑G - This is like ↑A except that, in addition, your procedure and file environments are thrown away.

↑T - This is really an XX command that prints out some information about the number of CPU seconds your job has got etc. Not very exciting.

The following commands are interpreted by the IDSYS command interpreter. Unless otherwise specified, IDSYS commands cannot be nested.

? Prints this information.

QUIT
Returns to the EXEC and resets the IDSYS fork. All open files are closed.

EMACS
To edit ID code. To return to IDSYS from EMACS, type ↑X↑Z.

COMMANDS FOR COMPILING AND LOADING PROGRAMS:

COMPILE(file−string)
Calls the Id compiler. File−string and enclosing parentheses are optional.

LOAD(file−string, ...)
Adds the procedure definitions contained in the files to the current environment. File−string is XX compatible file name. If a name conflict occurs (i.e., a procedure in your procedure

environment has the same name as that of one of the procedures in the file you are loading in), IDSYS asks you whether you want to retain your old definition or replace it with the new one.

COMPLOAD(file-string)
Effect of this command is equivalent to doing COMPILE followed by LOAD except that if a name conflict occurs, IDSYS does not prompt you, but throws away your old procedure definition and replaces it with the one being read in from the file.

COMMANDS RELATED TO PROCEDURE ENVIRONMENT:

PROCS(arg1, ...)
Prints information about procedure named arg1 if it is defined in the current environment. With no arguments it will print the information about all procedures in the environment.

REMOVE(arg1, ...)
Deletes the definitions of procedures arg1, ... from the environment.

APPEND(arg1, ...)
Adds the definitions of procedures arg1, ... to the environment. It will ask you the name of the file which contains these definitions.

COMMANDS FOR CALLING AND TRACING A PROCEDURE:

proc-name(arg1,arg2, ...)
Calls procedure named by proc-name. Result values are printed on the terminal.

TRACE(proc-name FROM from-list, ...)
Turns on the printing of trace information for procedure named by proc-name. From-list is a list of procedure names separated by / , for example FOO/ BAR/ ZAP.

UNTRACE(proc-name FROM from-list, ...)
Turns off the printing of trace information for procedure named by proc-name. Argument specification is similar to TRACE.

COMMANDS FOR OPENING AND CLOSING THE FILES:

INFILES(file-id <- file-string, ...)
Opens the files for input. File-string is XX compatible file name. File-name is an identifier. File named by file-string is opened

and assigned to file–id.

OUTFILES(file–id <- file–string, ...)
Opens the files for output. Arguments are similar to INFILES.

APPFILES(file–id <- file–string, ...)
Opens the files for appending. Arguments are similar to INFILES.

CLOSE(file–id, ...)
Closes the file associated with file–id. If no argument is supplied all open files are closed.

RESET(file–id, ...)
Closes the file associated with file–id; opens the same file again and assigns it to file–id.

COMMANDS FOR READING AND WRITING:

IDREAD(arg1,file–id)
Reads arg1 number of values from file specified by file–id. Default value for arg1 is 1, and default value for file–id is IDTTY.

IDWRITE(arg1,arg2, ... , file–id)
Writes values of the arguments on the file specified by file–id. If file–id is IDTTY arguments are printed on terminal.

IDPRINT(arg1,arg2, ... , file–id)
Similar to IDWRITE except that strings are not enclosed in quotes.