LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Id World: User's Manual

Computation Structures Group Memo 266
June 1986

Dinarte R. Morais

Reproduced verbatim from Mr. Morais' bachelor's thesis of May 1986.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This appendix is a reference manual for ID WORLD, currently being used by the members of the Computation Structures Group in the Laboratory for Computer Science at M.I.T.

ID WORLD interfaces the ID Compiler Version 1, GITA, the GITA Debugger, and ID Mode, in order to simplify the development of dataflow programs written in ID. The ID Compiler Version 1 was written by Ken Traub and Steve Heller, and is based in large part on the ID Compiler Version 0 by Vinod Kathail. GITA, the Graph Interpreter for the Tagged-Token Architecture, was designed by Ken Traub and Richard Soley and was originally implemented by the author and Richard Soley. Improvements to GITA have been made by David Culler, Greg Papadopoulos, and Steve Heller. The GITA Debugger and ID Mode were designed and implemented by the author.

In order to begin using ID WORLD, you must make sure that the machine you are using has loaded all of the programs which make up ID WORLD. Currently these are:

- The Graph Interpreter for the Tagged-Token Architecture (GITA)
- The ID Compiler Version 1
- ID Mode

If you are using one of the Lisp Machines belonging to the CSG group at the M.I.T. Laboratory for Computer Science, whether it be a Symbolics 3600 series or T.I. Explorer, then ID WORLD is already loaded into your environment.

If you are on a non-CSG Lisp Machine, you can load ID WORLD by loading the file `OAK:>ID-WORLD>ID-WORLD.LISP`. This file contains all the commands necessary to load all the parts of ID WORLD.

ID Mode is a new major mode for ZMACS, the editor available on T.I. and Symbolics Lisp Machines. It defines several commands which understand about the structure of ID procedures, and interfaces the ID Compiler and GITA in order to simplify program development.

In the commands which follow, abbreviations are used for the control and shift keys. They have the following meanings: `c` means the *control* key, `m` means the *meta* key, `s` means the *super* key, `h` means the *hyper* key, and `sh` means the *shift* key. A command such as `c-sh-C` is typed by holding down the *control* and *shift* keys and typing the `C` key. The command `c-m-A` is typed by holding down the *control* and *meta* keys any typing `A`.

Commands such as `m-X Compile File` are typed in as follows. First hold down the *meta* key and type `X`. In the small window at the bottom of the editor window you will be prompted with `Extended Command:`. Type in `Compile File` and hit `<RETURN>`.

2.1 The File Attribute List

It is a good idea to put the following line (called the *attribute list*) at the top of every file containing ID procedures that you will executing in ID WORLD:

```
! -*- Mode: ID; Package: GITA -*- !
```

Whenever a file which has this line at the top is loaded into the editor, ZMACS will automatically set the current major mode to be ID Mode, and the current package to be the GITA package. If you load in a file containing ID procedures which does not have this line at the top, then you should add it and then type `m-X Reparse Attribute List`. This will tell the editor to look at the attribute list and set the major mode and package to ID and GITA, respectively.

Alternatively, the following two editor commands can be used to construct the attribute list at the top of a file, or correct it if it is wrong.

m-X ID Mode *ZMACS Command*

Sets the major mode for the current buffer to ID Mode, a mode for editing ID programs.

m-X Set Package *ZMACS Command*

Prompts for the name of a package to become the default for the current buffer. When editing ID programs you should answer the prompt with **GITA**.

After each of these commands you will be asked if you want the attribute list at the top of the buffer to be updated. You should answer **Yes** to the question. The attribute list will either be created if one doesn't exist, or the appropriate field will be updated with the correct information. These two commands should be invoked whenever a new file or a new buffer is created in order to correctly set up ID Mode.

You can tell if you are in ID mode because the editor mode line (the top most line in the small window at the bottom of the screen) will say something like:

ZMACS (ID) SIMPLE.ID >ARVIND> OAK:

The symbol in parenthesis after the word **ZMACS** always specifies which major mode is currently active. If you switch to another buffer containing LISP code, for example, the major mode becomes LISP. However, it will revert back to ID mode whenever you go back to editing a buffer containing ID.

You can also tell which package you are currently in by looking down around the center of the very bottom line on the screen. There you will find a symbol with a colon after it. This symbol tells you what the current package is. When you are editing ID programs and are in ID mode you should see **GITA:** down there. While a procedure is being compiled you will see it change to **IDV1:**. What this means is that the ID Compiler Version 1 is busy compiling the procedure. When it finishes, the package will automatically be reset to **GITA:**.

2.2 Commands in ID Mode

There are a few special commands in ID Mode which "understand" ID procedures. When the following commands talk of the *current procedure* what they mean is the ID procedure on which the cursor is currently placed. If the cursor is between two procedures, then the earlier one is said to be the current one.

c-m-A *ZMACS Command*

Moves the cursor to the beginning of the current procedure. That is, the cursor is moved so that it is on the 'p' of the keyword 'procedure'.

c-m-E *ZMACS Command*

Moves the cursor to the end of the current ID procedure. That is, the cursor is moved just past the last character in the current procedure.

c-m-H *ZMACS Command*

Sets the region to be the current procedure (marks the current procedure). That is, the point is set to the beginning of the current procedure, and the mark is set to the end. See the ZMACS manual for an explanation of points, marks, and regions.

The following commands are used to edit comments at the end of a line of code.

c-; *ZMACS Command*

If the current line contains a comment, the cursor is moved to the beginning of the text of the comment. Otherwise, a comment is started on the current line. When a comment is started, the cursor is moved to the comment column (a horizontal position where single-line comments are started by default), and beginning and ending comment characters are automatically added around the cursor.

c-m-; *ZMACS Command*

If the current line contains a comment, this keystroke removes it. Use this if you accidently type **c-;** by mistake.

The following commands are used for compiling procedures, regions, buffers, or entire files of ID code. Except for **m-X Compile File** the object code from the compiler is automatically loaded into GITA. If the ID Compiler detects a syntax error while compiling a procedure a message will be printed in the typeout window (a window which grows down over the text in the buffer), and the compilation of the procedure with the error is aborted.

c-sh-C *ZMACS Command*

If there is a region, then each procedure in the region is sent to the ID compiler. Otherwise the current procedure is sent to the ID Compiler. The object code for each procedure successfully compiled is automatically loaded into GITA.

m-X Compile Buffer *ZMACS Command*

Each procedure in the current buffer is sent to the ID Compiler. The object code for each procedure successfully compiled is automatically loaded into GITA.

m-X Compile File *ZMACS Command*

Prompts for a file to compile, and sends the entire file to the ID Compiler. Unlike the previous two commands, this command does not automatically load the object code into GITA. Instead, a .CMC file is generated which contains the object code output by the compiler. Use **m-X Load File** to load the .CMC file into GITA.

m-X Load File *ZMACS Command*

Prompts for a file to load. If the file has a .NMC or .CMC extension it is given to GITA for loading. See the function **LOAD-NMC** below.

2.3 Restrictions in ID Mode

There are two restrictions imposed by ID Mode on the way ID code is organized in a file. While the ID Compiler does not care about either of these restrictions, they must be adhered to whenever using ID Mode in order for it to work correctly.

The first restriction is that the keyword **procedure** at the beginning of each ID procedure must be flush with the left margin. That is, the **p** in **procedure** must always be in column 0. ID Mode uses this fact in order to find the beginning and end of ID procedures without having to perform lexical or syntactic analysis.

The second restriction has to do with the way in which procedures may be commented out. As far as the ID Compiler is concerned the following is a correct way of commenting out the procedure **foo**:

```
!
procedure foo(x)
  x * x
!
```

As a result of the first restriction, however, ID Mode sometimes gets confused as to

whether procedure **foo** is "inside" or "outside" of a comment. The only way to know for sure is to scan forward from the beginning of the current buffer until you get to the procedure in question. But this scanning can be very time-consuming, so instead ID Mode requires that the procedure be commented out as follows:

```
! procedure foo(x)
  x * x !
```

The idea is to make sure that the keyword **procedure** does *not* begin in column 0 if it is commented out. When uncommenting out the procedure you must remember to once again position the keyword **procedure** so that the **p** is in column 0.

Before you can execute an ID procedure you must first compile it using the ID Compiler, and then tell GITA to load the output of the ID Compiler. This can be done in several ways.

The easiest way is to load a file containing ID procedures into ZMACS and use ID Mode commands to compile one, two, or even all of the procedures in the buffer. As explained above, ID Mode will arrange for the given procedures to be sent to the ID Compiler, and will automatically load the output of the ID Compiler into GITA.

If you already have a file which contains compiled ID procedures (which you can get by using the command **m-X Compile File** in ID Mode), then you can use the ID Mode command **m-X Load File**, described above, or the function **load-nmc** to load this file into GITA.

(**load-nmc** *pathname* **&optional** *silent?*) *Function*

Tells GITA to load all of the compiled ID procedures in the file *pathname*. *pathname* should have a .CMC or .NMC extension. If no extension is provided, then a file with extension .CMC is looked for, followed by a file with extension .NMC. If *silent?* is non-nil, then the ID procedure names contained in the file will not be printed out as they are loaded.

When GITA is told to load a file containing compiled ID procedures, it creates a structure known as a POBJ (procedure object), which contains all of the necessary information to allow it to interpret the procedure. In order to make it easy to execute these procedures, GITA creates a LISP procedure corresponding to each ID procedure loaded. The LISP procedure has the same name and takes the same number of arguments as the corresponding ID procedure. When called, the LISP procedure tells GITA to interpret the ID procedure with the given arguments, and when GITA finishes executing, the LISP procedure returns the results of the ID procedure as multiple values.

For example, suppose we wish to execute the following ID procedure:

```
procedure factorial(n)
   (if n = 0 then 1
       else n * factorial(n - 1))
```

One way to do this is to enter ZMACS, tell it to set the mode to ID Mode and the Package to GITA (as described above), type in the above procedure, and use the ID Mode command **c-sh-C** to compile and load this procedure into GITA. After doing this, GITA will create an internal representation (a POBJ) of the ID procedure **factorial**, along with a LISP procedure like the following:

```
(defun factorial (n)
  (gita:run-code-block 'factorial n))
```

You can then call the LISP procedure **factorial** with an argument, which will cause GITA to interpret the ID procedure by pushing tokens around the dataflow graph stored in the POBJ, until the answers came out the bottom of the graph. The answers which come out of the graph will be the values returned from the LISP procedure.

Note that this mapping of ID procedures to LISP procedures allows you to mix both ID

and LISP. For example, if you type:

```
(+ (factorial 3) (factorial 4))
```

to a Lisp Listener, you will cause GITA to be run twice, once for each call to **factorial**, and then LISP will compute and return the sum of the results of the two ID procedures.

Furthermore, you can take the result of one ID procedure and use it as an argument to another ID procedure. For example, the following is perfectly legal:

```
(factorial (factorial 5))
```

It will simply cause GITA to be run twice, first to compute the factorial of 5, and then again to compute the factorial of 120.

You can even call ID procedures from LISP procedures (but not the other way around!). The bottom line is that you are free to treat ID procedures just like LISP procedures. As far as the user of ID WORLD is concerned, there is no difference.

If the only reason you wish to execute an ID procedure is to know the result, then the previous section has already explained how to do this. All you have to do is get to a Lisp Listener, call the LISP procedure corresponding to the ID procedure, and wait for a result to be returned.

When writing ID programs in ZMACS you will probably want to be doing this quite often to debug procedures as you write them. Whenever you write an ID procedure and wish to test it, you should use the ID Mode command **C-sh-C** to compile and load the current procedure into GITA. The next thing to do is find a place where you can invoke the ID procedure via its LISP procedure. One place to do this is in a Lisp Listener. You can select a Lisp Listener (by typing **<SELECT>-L** or **<SYSTEM>-L** on Symbolics or T.I. Lisp Machines, respectively), execute the procedure, and then return to the editor.

If you are already in the editor, however, a convenient place to quickly test an ID procedure is in the editor typeout window. The editor typeout window is a window which "grows" down over the text in the buffer, and behaves just like a Lisp Listener. On a Symbolics Lisp Machine you can get to the editor typeout window by typing the **<SUSPEND>** key. On a T.I. Lisp Machine you would use the **<BREAK>** key. Once the editor typeout window is exposed, you can type any LISP form, such as the LISP functions which execute the ID procedures you have just written and compiled. When you wish to go back to editing your programs, just type the **<ABORT>** key until the typeout window goes away.

After you have finished using ID Mode to write, compile, and test your programs, you should select the GITA Frame to execute your ID programs. You can select the GITA Frame by typing **<SELECT>-G** on a Symbolics Lisp Machine (**<SYSTEM>-G** on a T.I. Lisp Machine).

6.1 Organization of the GITA Frame

When the GITA Frame first comes up, it is divided into three sections. The top section is the *profile pane*, and is used to draw parallelism profiles (described below). About two-thirds of the way down the screen is a menu with items such as *Load* and *Execute*. Finally, the bottom portion of the screen is a Lisp Listener which you can use to execute ID procedures via their LISP functions.

There are actually two configurations which the GITA Frame can be in. The first, which is what you see when you first select the GITA Frame, is called the *profile configuration* because a large portion of the screen is reserved for the displaying of parallelism profiles.

The second configuration is called the *debugger configuration*. When the GITA Frame is in this configuration, the menu will be all the way at the top of the frame, and the rest of the frame will be the Lisp Listener. This configuration is used whenever the GITA Debugger is entered, since it more convenient to use the debugger in a large window.

6.2 The GITA Frame Menu Items

This section briefly describes each of the items in the GITA Frame menu.

Load *GITA Menu Item*

Prompts for a file to load, then calls the function **load-nmc** with that file.

Execute *GITA Menu Item*

Pops up a menu of all loaded ID procedures. Click on one to execute it. You will be prompted in the Lisp Listener for each argument required by the procedure. GITA will then execute the procedure, and print the results in the Lisp Listener.

GITA Debugger *GITA Menu Item*

Changes the configuration of the GITA Frame to the debugger configuration, and starts up the GITA Debugger. See below for a description of the GITA Debugger.

Show Profile *GITA Menu Item*

Changes the configuration of the GITA Frame to the profile configuration. Then pops up a menu of all the profiles which can be shown. Click on one to cause the profile be drawn in the profile pane. See below for a description of profiles.

6.3 Collecting and Viewing Statistics

So far the only reason for using GITA was to execute an ID procedure. But GITA can also be used to collect and view various statistics. This section describes how to tell GITA to collect these statistics.

6.4 Idealized vs. Emulated Statistics

When GITA collects statistics it does so either in *idealized mode* or in *emulation mode*. It is important to understand the difference between these two modes since the same profile can look very different depending on which mode was used to generate it.

The main difference between idealized mode and emulation mode is in how each defines the term *timestep*. In emulation mode a timestep is simply a certain fixed amount of time, such as 2 seconds. In idealized mode, however, a timestep varies in the amount of time it takes to complete. It is based on the following assumptions:

- All activities ready to fire are fired in the same timestep.
- All activities take the same amount of time to execute.
- There is zero communications overhead.
- Unbounded resources are available.

Emulation mode measures how GITA actually performed in executing some ID procedure. If the timestep were 1 second, for example, then the first sample in the ALU-operations profile (described below) gives the number of ALU operations which fired during the first second. The next sample gives the number of operations which fired during the second second, etc.

For idealized mode, however, the ALU-operations profile is interpreted as follows. The first sample indicates how many ALU operations fired when the procedure was first started. (GITA arranges for there to be one operation ready to fire at the beginning -- it drops the first token into the dataflow graph to get things started.) The second sample indicates how many ALU operations fired during the second timestep. That is, all those operations which became ready to fire as a result of operations firing in the first timestep are all fired in the second timestep. An informal definition of a timestep in idealized mode is. *Fire all and only those operations which are ready to fire at the beginning of each timestep.*

The following functions control which statistics mode GITA is in.

(emulation-mode) *Function*
Tells GITA to collect statistics in emulation mode.

(idealized-mode) *Function*
Tells GITA to collect statistics in idealized mode.

(no-stats) *Function*
Tells GITA not to collect any statistics.

6.5 Statistics in GITA

This sections describes the statistics which can be collected while an ID procedure is being executed by GITA. Note that the term *timestep* has a different meaning depending on which statistics mode GITA is in.

ALU Operations Profile *Statistic*
Collects a profile of the number of ALU operations which were fired during each timestep.

Wait-Match Profile *Statistic*

Collects a profile of the number of tokens which were in waiting-matching sections at the end of each timestep.

Invocations Profile *Statistic*

Collects a profile of the number of procedure invocations which occurred during each timestep.

Terminations Profile *Statistic*

Collects a profile of the number of procedures which terminated during each timestep.

I-Fetch Profile *Statistic*

Collects a profile of the number of I-Structure fetches were done during each timestep.

I-Store Profile *Statistic*

Collects a profile of the number of I-Structure stores were done during each timestep.

Deferred-Reads Profile *Statistic*

Collects a profile of the number of I-Structure fetches which were deferred during each timestep.

I-Structure Storage Profile *Statistic*

Collects a profile of the amount of I-Structure storage in use at the end of each timestep.

The following statistics are only collected when in emulation mode.

Queued Tokens Profile *Statistic*

Collects a profile of the number of tokens in the token queue at the end of each timestep.

Active Code-blocks Profile *Statistic*

Collects a profile of the number of code-blocks which were active at the end of each timestep.

The following statistic is only useful when in emulation mode and running on more than one physical processor.

Idle Profile *Statistic*

Collects a profile of the amount of time each PE was idle per timestep.

6.6 Viewing a Statistic

After setting the statistics mode and executing a procedure, you can view any of the statistics by clicking on *Show Profile* in the GITA menu and selecting a statistic to view. The profile will be shown in the profile pane.

(gita-debugger) *Function*

Invokes the gita debugger, making the current context be the root context.

7.3 GITA Debugger Command Loop

After invoking the GITA Debugger the root context will be displayed, including the name of the procedure and its arguments, and the cursor will be to the right of a right-arrow, the GITA Debugger's prompt. Whenever the cursor is just to the right of the right-arrow, you are at top-level in the debugger. Whenever you are at this level, there are several things you can do.

- You can hit the **<ABORT>** key to exit the debugger.
- You can type one of the GITA Debugger commands, described below.
- You can type in a form to evaluate, just as if you were typing to a Lisp Listener.

Whenever you type a form to evaluate, the command loop automatically sets the global variable * to the value returned from the evaluation of the form. Thus * can be thought of as holding on to the last thing returned. Similarly, the global variable ** holds on to the second to last thing returned, and *** holds on to the third to last thing returned. In the descriptions of debugger commands which follow, whenever a command says that it "returns" an object, the variable * can be used to refer to that object. For example, after typing the debugger command **c-A**, which returns the ISD for the arguments of the current context, you can then type **(setq foo-args *)** to make the variable **foo-args** hold on to the ISD.

Some of the commands which follow refer to a *numeric argument*. A numeric argument is a number which is typed just before a command is issued, usually specifying which of a set of *n* things should be done. To type a numeric argument, you hold done one of the control keys (either *control*, *meta*, *super*, or *hyper*) and type the digits of the number. You then type the command. For example, to type the **c-m-A** command with a numeric argument of 12, you could type **c-1 c-2 c-m-A**. The section on ID Mode explains how to type commands such as **c-m-A**.

The GITA debugger is in many ways like the normal LISP debugger. The main difference is that the LISP debugger allows you to look up and down a *stack* of frames, while the GITA debugger allows you to look around a *tree* of contexts.

7.1 GITA Debugger Definitions

In GITA, a *context* roughly corresponds to a *stack frame*. Whenever a procedure is invoked a *context* is created to hold its arguments and results, etc. Because a procedure can execute sub-procedures in parallel, however, GITA must maintain a tree rather than a stack of contexts.

In the description of the GITA Debugger which follows, the *current context* means the context which is currently being examined. As you move around the tree of contexts, the *current context* is changing to reflect your position in the tree. The *root context* never changes and is the one context which has no father. It corresponds to the top-level call you made when you first told GITA to execute an ID procedure. Finally, the *anchor context* is a context which usually corresponds to a context at which an error occurred, although is can be changed by using the `c-.` debugger command. This anchor context is used in order to facilitate moving up and down the tree of contexts. See the section on Movement commands, below.

7.2 Invoking the GITA Debugger

There are two ways to enter the GITA Debugger. The first has already been described -- when in the GITA Frame, you click on the menu item *GITA Debugger*. Alternatively, you can use the following function from wherever you were when you executed an ID procedure -- the editor typeout window, for example.

7.4 Debugger Commands for Error Handling

When GITA detects an error during execution of a procedure a message is printed saying in which part of the machine the error occurred. Execution continues, however, until there are no more activities ready to fire. At the end of execution GITA reports the total number of errors it encountered. The following function allows you view the errors.

(show-errors) *Function*

Prints a report for each error encountered during the last GITA run. The errors are printed in reverse chronological order and include the time the error occurred and an explanation of the error.

7.5 Dealing with Error objects

When the GITA debugger is active, the following commands let you deal with errors.

c-E *GITA Debugger Command*

Prints a list of all errors encountered in the last GITA run. This command simply executes the **show-errors** function.

c-m-E *GITA Debugger Command*

Without a numeric argument, this command returns the current error object. Otherwise, the error object given by the numeric argument is returned. This is one way to get your hands on the arguments to the ALU operation which failed.

c-m-G *GITA Debugger Command*

Causes the anchor context to become the one where a particular error occurred. You must use a numeric argument to specify which error you are interested in.

7.6 Backtrace Commands

A *backtrace* is a listing of the contexts in reverse order starting from the current context and ending at the root context. There are two backtrace commands which differ only in how much detail they provide about each context.

c-B *GITA Debugger Command*

Displays a brief backtrace, showing the name of the procedure for each context.

m-B *GITA Debugger Command*

Displays a verbose backtrace, showing the name of the procedure and its arguments for each context.

7.7 Examining the Current Context

There are many commands in the GITA debugger designed to return information from the current context.

c-m-A *GITA Debugger Command*

Returns the n^{th} argument in the current context. Use a numeric argument to specify *n*.

c-A *GITA Debugger Command*

Returns the ISD used to hold the arguments for the current context.

c-m-V *GITA Debugger Command*

Returns the n^{th} value being returned from the current context. Use a numeric argument to specify *n*.

c-V *GITA Debugger Command*

Returns the ISD used to store the return values of the current context.

c-m-L *GITA Debugger Command*

Returns the n^{th} local variable (token) from the current context. Use a numeric argument to specify *n*.

c-m-F *GITA Debugger Command*

Return the procedure object from the current context.

c-m-C *GITA Debugger Command*

Returns the current context object, an object of type **dbg-context**. Note that this is not the same as the actual context object--a **dbg-context** only *refers* to the actual context object through its PE and INDEX slots. The GITA debugger uses this object to store the replies from remote servers so that it won't have to ask again.

7.8 Movement Commands

Moving around the tree of invocations in the GITA debugger is not quite as easy as moving up and down the stack in the LISP debugger. A context will have at most one father (the caller), but may have several sons (each corresponding to a procedure invocation which has not yet terminated).

Moving "up" the tree of invocations is straightforward, the current context gets set to its father. Moving "down" the tree of invocations, however, requires that a branch be selected from among its sons. To make moving down the invocation tree easier, the GITA debugger

will sometimes pick the branch which is considers the "obvious" choice. Taking the "obvious" choice is done by using the **c-N** command without any numeric argument. If the GITA debugger cannot determine an "obvious" choice, then a numeric argument specifying which branch to go down must be given. The GITA debugger determines the "obvious" choice as follows:

- If the current context has no sons, then there is no way to go down, and thus no "obvious choice".
- If the current context has only one son, then the "obvious" (and only) choice is to go down to the son.
- If the current context has more than one son, then the "obvious" choice is the son which is closest to the anchor context. If none of the sons are in the path to the anchor context, then there is no "obvious" choice.

By defaulting the "obvious" choice, you should be able to stick with using **c-P** and **c-N** and never have to specify a branch unless you want to go off and inspect a context at some other part of the tree.

c-P *GITA Debugger Command*

Goes up one context towards the caller. With a numeric argument, goes up that many contexts towards the caller.

m-P *GITA Debugger Command*

Like **c-P** except that detailed information about the target context is displayed.

m-< *GITA Debugger Command*

Goes to the root context (the oldest in the invocation tree).

m-> *GITA Debugger Command*

Goes to the anchor context.

c-N *GITA Debugger Command*

With no numeric argument, goes down the "obvious" branch of the invocation tree. Otherwise goes down the n^{th} branch, where n is specified by the numeric argument.

m-N *GITA Debugger Command*

Like **c-N** except that detailed information about the target context is displayed.

c-. *GITA Debugger Command*

Sets the anchor context to the current context.

7.9 Searching

The following commands search for a context with a procedure whose name contains a given substring. The contexts are searched starting from the father of the current context toward the root context. This is useful for quickly jumping to some context shown in a backtrace. For example, if you type **c-B** and the backtrace is:

```
FOO [1] <- BAR [301] <- BAZ[8] <- QUUX [503]
```

Then searching for "BA" will make the current context be **BAZ [8]**, and searching for "F" will make the current context be **FOO [1]**.

c-S *GITA Debugger Command*

Prompts for a substring and searches up from the current context for one whose procedure name contains the substring. If it finds one, then that context becomes the current one.

m-S *GITA Debugger Command*

Like **c-S** except that detailed information about the target context is displayed.

7.10 Other Debugger Commands

An invocation tree shows the full tree of invocations still active at the time the last GITA run finished executing. The sons of a context are all indented to the same column underneath the father.

c-T *GITA Debugger Command*

Shows an invocation tree of all known contexts.

Use these next two commands to clear the screen, in one of two ways.

c-L *GITA Debugger Command*

Clears the screen and displays the current error message along with the procedure name and arguments of the current context.

m-L *GITA Debugger Command*

Clears the screen and displays detailed information about the current context, including the procedure name, its arguments, results, and locals (tokens).

These next few commands let you look inside of various objects.

c-m-D *GITA Debugger Command*

Describes the last thing printed out. This keystroke is equivalent to typing **(DESCRIBE *)**.

c-I *GITA Debugger Command*

Pretty-prints the last thing printed out. This command is especially useful for viewing ISDs. When an ISD is pretty-printed the contents of the I-Structure it references is printed out.

c-m-I *GITA Debugger Command*

Typing this command toggles the ISD pretty-print flag. When the flag is set all ISDs are automatically pretty-printed.

Finally, you can get online help by typing the **<HELP>** key.

<HELP> *GITA Debugger Command*

Prints a concise description of each of the debugger commands.

GITA can be made to run on multiple machines. One of the design goals of ID WORLD, however, was that the user should not be concerned with the number of machines which are in use. Accordingly, ID Mode, the GITA Debugger, and the statistics have all been designed to be used in exactly the same way no matter how many machines are participating in the execution of an ID procedure.

8.1 Setting up Multiple Machines

Note that this section is likely to change in the near future. The interface to multiple machines at the moment is neither very robust nor general, and is being redesigned. Therefore, this section will provide only a quick introduction to some of the functions which are currently used to run experiments on multiple machines.

`*default-processors*` *Variable*

This variable contains a list of the machines, including the local machine, which may be used by the local machine to execute ID procedures.

`(select-firstn-processors` *n*`)` *Function*

Sets the variable **`*default-processors*`** to the names of the first *n* processors in the MEF.

`(select-processors &rest` *machines*`)` *Function*

Sets the variable **`*default-processors*`** to the list of machines given as arguments to this function.

`(initialize-gita-servers)` *Function*

Starts up a GITA-Server processes on each of the machines in **`*default-processors*`**.

`(reset-gita-servers)` *Function*

Makes sure that the GITA-Server process is still running on each of the machines in **`*default-processors*`**. Will reestablish a connection to any of the machines if it has failed.

(initialize-network) *Function*

Causes each of the machines in ***default-processors*** to derive the connectivity of the Circuit Switch network. This must be done once before GITA can execute ID procedures on all the machines.

(show-netstate) *Function*

Prints out a textual representation of the connectivity of the Circuit Switch for each of the machines in ***default-processors***.

(draw-netstate) *Function*

Draws a pictorial representation of the connectivity of the Circuit Switch for each of the machines in ***default-processors***.

(pes *n*) *Function*

Selects the first *n* processors in ***default-processors*** to participate in the next execution of an ID procedure. Setting *n* to 1 will force the local machine to be the only one involved in the execution.

8.2 Using the GITA Server

The following functions may be used to force some or all of the machines in ***default-processors*** to perform some action.

(all *form*) *Macro*

Causes all processors, including the local machine, to evaluate *form*. The results are discarded. Does not wait for the servers to finish evaluating *form* before returning.

(all-eval *form*) *Macro*

Causes all processors, including the local machine, to evaluate *form*. The results from each machine are collected and returned.

(others *form*) *Macro*

Causes all processors except the local machine to evaluate *form*. The results are discarded. Does not wait for the servers to finish evaluating *form* before returning.

(others-eval *form*) *Macro*

Causes all processors except the local machine, to evaluate *form*. The results from each machine are collected and returned.

(execute-on *pe form*) *Macro*

Causes *form* to be evaluated on the pe^{th} machine in ***default-processors***. The results are discarded. Does not wait for the server to finish executing *form* before returning.

(eval-on *pe form***)** *Macro*

Causes *form* to be evaluated on the pe^{th} machine in ***default-processors***. The results are collected and returned.

(load-on *pe file***)** *Macro*

Causes *file* to be loaded on the pe^{th} machine in ***default-processors***.

(all-load *file***)** *Macro*

Causes all processors, including the local machine, to load *file*.