Monsoon Id World User's Guide

Version 002

[DRAFT]

Computation Structures Group Memo 334

January 6, 1992

R. Paul Johnson

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

Revision History:

This document supersedes:

Id World: Reference Manual, Rishiyur S. Nikhil, P. R. Fenstermacher, J. E. Hicks and R. P. Johnson Computation Structures Group, MIT Laboratory for Computer Science, November 1989.

Abstract

In this paper we describe Id World, an integrated development environment for Id, a general purpose high level parallel programming language on ETS, a parallel MIMD architecture. Id World supports preparation, execution, and statistics collection for Id programs run on hardware, Monsoon, or software, Mint, implementations of the ETS architecture.

Contents

Chapter 1

Introduction

Id World is a powerful, integrated programming environment produced by the Computation Structures Group at MIT's Laboratory for Computer Science for the experimenter who wishes to prepare and execute parallel programs and study their behavior, as quickly as possible.

When one codes an algorithm in a parallel language, there are many questions about its parallel behavior that are of interest. For example, it is quite popular nowadays to annotate a sequential language such as FORTRAN or C with "parallel" constructs, and then to compile it for a parallel architecture. In these languages, the *determinacy* of the program is left to the programmer, *i.e.*, the programmer must insert appropriate synchronizations to ensure that a given set of inputs always produces the same set of outputs. This makes debugging extremely difficult— for a given input, the program may produce different outputs for different machine configurations and/or scheduling policies, and this behavior may not immediately be obvious. Such timing-dependent errors may not even be reproducible in a debugger. So, a fundamental question might be: "What *is* the correct output for a given set of inputs, *i.e.*, is the parallel program executing correctly?"

Other questions that one might ask with respect to a particular algorithm are:

- What is the *Parallelism Profile*, *i.e.*, what are the operations that *could* be done in parallel assuming enough processors? Has encoding the algorithm into the parallel programming language extracted all the parallelism originally available?
- How many processors could be kept busy?
- What is the effect of increasing the communication latency between processing elements?
- How many instructions are executed, and what is the dynamic instruction mix?
- How many memory reads and writes are executed?
- How many synchronization events are needed?

Id World provides a powerful and convenient means to prototype parallel programs rapidly and to study these questions. Programs are written in the parallel programming language Id and executed on Monsoon, an implementation of the *explicit token store* architecture.

Id is a general-purpose high-level parallel programming language which may be regarded as a functional language extended with I-structures as a parallel data-structuring facility. Id supports higher-order functions and general recursion, loops, nested block structure and non-strict functions and data-structures. Unlike other parallel languages,

- The parallelism is *implicit* in the operational semantics of Id. The programmer does not explicitly break up a task into parallel components, and so does not worry about synchronization.
- Id programs are *determinate*, *i.e.*, the result of any Id program depends only on its inputs, and never on the machine configuration or the runtime scheduling policy. This is a *major* simplification in debugging.
- The only limits on parallelism are from data-dependencies and finite machine resources.

Id programs are compiled into *dataflow graphs*. The nodes in a dataflow graph are elementary machine instructions, such as ALU arithmetic operations, memory loads and stores, conditionals, *etc.* and the arcs specify only the *essential* data dependencies between the instructions. Thus an idealized parallelism profile derived in Id World reflects the "maximum" parallelism available in the given algorithm.

Dynamic dataflow architectures completely abandon the von Neumann program counter based instruction scheduling in favor of purely data-driven instruction-scheduling. Multiple threads of execution (due to data dependencies) can be interleaved at the instruction level, and synchronization is built into the hardware. The Explicit Token Store (ETS) is a parallel MIMD architecture that directly executes dataflow graphs. It relies upon compiler determined slots in an activation frame for operand storage. Thus, ETS eliminates associative matching and implicit storage allocation present in tagged-token dataflow models.

Even if one were interested only in more "conventional" approaches, such as parallel FORTRAN on a von Neumann-based multiprocessor, Id World can be of great use in quickly prototyping a parallel algorithm, checking what the correct output should be for a given set of inputs, and providing points of reference for the parallelism profile of the algorithm, instruction mix, *etc.*

1.1 Prerequisite Hardware and Software

Id World runs on Sun and Motorola Unix workstations that have Common Lisp.

On Sun 3/4 Workstations: Id World is built on Sun Common Lisp (Lucid Common Lisp) 3.0 / 4.0 or Allegro Common Lisp 3.0, needs at least 8MB of memory, about 10MB disk space, and about 30 MB of swap space.

On Motorola Delta Workstations: Id World is built on Allegro Common Lisp 3.0, needs at least 8MB of memory, 10MB disk space, and 30MB of swap space.

1.2 Prerequisite Expertise

Activities in Id World are patterned directly after corresponding activities in the Lisp program-development environment. Editor, and compiler commands are direct analogs of their Lisp counterparts. If you are already familiar with the mechanics of developing Lisp programs, you should be able to use Id World with very little training.

In general, you must know the following minimal things about your workstation:

- Machine preliminaries, such as booting, logging in *etc.*
- File system basics: the directory structure and file names. It is also useful to know how to print, copy, rename and delete files.
- Basic editor commands for text files.
- How to switch back and forth between windows.

Most of the commands you will normally use are described in this manual. However, you may occasionally have to consult some Lisp machine manuals and/or obtain the assistance of a Lisp wizard.

1.3 Related Documents

The following three documents specify the syntax and semantics of the parallel programming language Id:

Id 90.1 Reference Manual, Rishiyur S. Nikhil, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA 02139, July, 1991.

Id Reference Manual, Part I: Operational Semantics, Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA 02139, April 24, 1987.

Id Reference Manual, Part II: Operational Semantics, Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA 02139, April 24, 1987.

Idiosyncrasies of the current implementation are given in Appendix B of the Id 90.1 reference manual. The operational semantics for Id are expressed in terms of rewrite rules for a "kernel" language, together with translations from the full language into the kernel language.

The following document is useful for readers who are interested in a more detailed description of the ETS architecture.

Implementation of a General Purpose Dataflow Multiprocessor, Gregory M. Papadopoulos, Technical Report LCS TR-432, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

For readers interested in further information on the Monsoon software architecture.

Monsoon Software Software Interface Specifications, Kenneth R. Traub, Michael J. Beckerle, James E.Hicks, Gregory M. Papadopoulos, Andrew Shaw and Jonathan Young, Motorola Cambridge Research Center and MIT Laboratory for Computer Science, MCRC-TR-1 and CSG Memo 296, January 1990.

The following document is useful for readers who need an introduction to parallelism and dataflow. It provides a high-level overview of the dataflow approach to high-speed parallel computing.

Executing a Program on the MIT Tagged-Token Dataflow Architecture, Arvind and Rishiyur S. Nikhil, In Proc. PARLE (Parallel Architectures and Languages Europe), Eindhoven, The Netherlands, June 1987. (Revised: CSG Memo 271), MIT Laboratory for Computer Science, Cambridge MA 02139, June 1988.

For readers who are already familiar with functional languages, the following document points out limitations of functional data structures, and shows how I-structures address some of these limitations.

I-structures: Data Structures for Parallel Computing. Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, in Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, Springer-Verlag LNCS 279. Pages 336-369. Also: CSG Memo 269, MIT Laboratory for Computer Science, Cambridge MA 02139, February 1987.

The following document is useful for readers who are interested in the details of compiling programs to dataflow graphs:

A Compiler for the MIT Tagged-Token Dataflow Architecture. Kenneth R. Traub, TR-370 (Master's Thesis), MIT Laboratory for Computer Science, Cambridge, MA 02139.

1.4 Command Conventions

Id World commands are EMACS editor commands or Lisp interface functions. Most Id World commands are the same on all Id World implementations, differences are documented by the annotation "(`implementation:` *command*)". The GNUEMACS and ZMACS commands are also usually the same ; whenever they differ we use the GNUEMACS key binding and follow it with the annotation "(*ZMACS: Zmacs-Command*)". GNUEMACS key command bindings may be changed in your `.emacs` file.

1.4.1 Typing In Commands

For keys that have long names, we show the entire name in caps, *e.g.*, `RETURN`, `LINE FEED`, `TAB` *etc.* In GNUEMACS, you get help by typing `c-H`, and in the Id World Interface by typing `(iw:help)`.

We use the following abbreviations for the various "shift" keys of the Lisp machine:

`c`	the `CONTROL` key
`m`	the `META` 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 and typing `A`. (GNUEMACS: Additionally, `c-m-A` can be typed by first typing `ESC` and then `c-A`).

A command such as `m-X id compile file` is typed 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 "`M-x` ". Type in `id compile file` ending with the `RETURN` key. Note that in GNUEMACS, the words in a command are separated by '-' instead of a space; however, if you type spaces GNUEMACS's command completion will insert the appropriate dashes. Furthermore, you only have to type enough characters of a command word to unambiguously name a command, and both EMACS implementations will fill in the rest of the characters for you.

1.4.2 Numeric Arguments for Commands

In various commands, you may want to or have to supply a numeric argument. To do this, precede the command by typing `c-U` followed by the number. or type the number *while holding down any combination of the* `CONTROL` *or* `META` *keys.*

1.5 Overview of Id World

Id World consists of three major components:

1. Id Mode, a new major mode for editing Id code in EMACS,
2. Id Compiler, which compiles Id code into Monsoon Object Code (MOC),
3. Id World Interface, a suite of procedures for running, loading compiled Id programs, and manipulating statistics.

1.5.1 Execution Vehicles

Compiled Id programs can be executed in a number of ways including:

1. Execution on a single-processor Monsoon hardware configuration
2. Execution on a multi-processor Monsoon hardware configuration
3. Execution on MINT[1], a software emulation of the Monsoon architecture

The design of the software architecture for Monsoon is such that all Monsoon execution vehicles present a common machine interface[2], allowing the loader, Id debugger, execution manager and statistics viewer to be shared between Monsoon and MINT.

[1]*M*onsoon *INT*interpreter.

[2]*Monsoon Software Software Interface Specifications* contains a complete specification (see Section 1.3).

1.5.2 Id World Interface

All Id World commands may be invoked by a lisp function; we call this interface the Id World Interface (IW). Code block names, statistic names, mode names, statistics save places, statistic collection names and parameter names can be strings or symbols; they are compared to the appropriate "known names" without respect to packages. When the user sits down in front of the Id World (IW) listener, he gets a regular vanilla Lisp listener[3], except that (1) the current package is IW:, and (2) the readtable is the IW readtable (*i.e.*, with

`$(), $[], $<>, $', $T, $F, $E and NIL`

as currently implemented).

1.5.3 Lispified Syntax for Id forms

To type Id objects into the IW listener (*e.g.*, as arguments when invoking an Id function), you must use a special syntax so that the Lisp reader knows that it is an Id (and not Lisp) object.

For this kind of Id object	*You should type*
integer	Common Lisp integer
float	Common Lisp float
char	Common Lisp char
string	Common Lisp string
symbol `'nil` (symbol)	Common Lisp symbol `'nil`
`nil` (object)	Common Lisp `nil`
`true`	`$t`
`false`	`$f`
I-structure	`$`*lb*`[`*e ... e*`]`
n dimensional array	`$` $lb_0, ...lb_{n-1}$`[[` *e ... e*`]` ... `[` *e ... e* `]]`
Tuple	`$<`*e ... e*`>`
List	Common Lisp list
Function `f` as a value	`$'f`
Partial application of `f`	`$'(f` *e ... e*`)`

In the I-structure, tuple and partial-application syntax, the *e*'s are, recursively, the expressions for the elements of the structures and arguments for the application.

In the I-structure syntax, *lb* stands for the lower index bound. The upper index bound is automatically computed from the number of elements. The *lb* is optional; if no lower bound is specified, the lower bound will be 0.

When an I-structure *A* is returned as a value, its components can be selected using the form:

```
(iref A j1 j2 ...  )
```

[3]A "listener" is lisp-speak for a shell/command-interpreter.

where the j's are indices. The form:

```
    (setf (iref A j1 j2 ...  )  v)
```

may be used to set the value of a component of an I-structure.

When I-structure values are printed on the output, you may see some components printed as **$E**. These are components of the I-structure that are still "empty"— they were never assigned a value.

When a tuple T is returned as a value, its components can be selected using the form:

```
    (tref T j)
```

where j is a 0-based numeric index.

1.6 Overview of Editor Activities

In the editor window, you can enter and edit the text of your Id programs using standard text manipulation commands, as well as commands that know about Id syntax to assist in indentation, balancing brackets, *etc.* These commands are described in Section 3.2

On Unix machines, the Id Compiler and Id World Interface are run in separate processes and communicate via a special lisp interaction buffer.

The editor also has commands to compile some or all of the Id definitions in a buffer and to compile Id files. If the compiler reports errors or warnings, there are commands to place you directly at the points in the program text where the errors were discovered, so that they may be corrected. These commands are described in Section 3.3.

When Id procedures are compiled from editor buffers, the MOC code is loaded automatically into Monsoon. This means that they are ready for execution by **ID-RUN**.

When Id procedures are compiled from files, the object code is stored in a file. These may be loaded into Monsoon to prepare them for execution.

Chapter 2

A Walk-through Example

In this section we walk you through a complete example to give you a flavor of typical usage of Id World. The program examples are the Inner Product and Vector Sum programs from *Executing a Program on the MIT TTDA* (see Section 1.3).

2.1 Initialization

When you first start the EMACS editor you will only be able to edit Id code. To use the incremental compilation and automatic loading you need to start *MINT*, the *Id World* lisp image and initialize the runtime system. To initialize the runtime system, you select your execution vehicle, *boot* and *initialize* the run-time system.

You can startup the Monsoon emulator *MINT* by typing:

```
    M-x run mint
```

You can startup or goto a running *Id World* lisp image by typing:

```
    M-x id world
```

Enter the IW Listener by typing:

```
    (IW:LISTENER)
```

At the IW listener prompt, `IW>>` :

connect to and select MINT as the execution vehicle by typing:

```
        (MINT)
```

setup the runtime system by typing:

```
        (BOOT-RTS)

        (INITIALIZE-RTS)
```

2.2 Code Preparation: Compile-time errors; Loading

Suppose that the new source-code file for the example program is:

```
/home/jj/rpaul/example.id
```

In the editor, type

```
c-X c-F /home/jj/rpaul/example.id
```

This initiates a new buffer for the file, and sets it automatically to be in Id Mode.

Type in the text of the first example:

```
%%% Inner-product of two vectors with same bounds.


Def ip A B = {   l,u = 1D_bounds A ;
   typeof A = Vector I;
 typeof B = Vector I;
                s = 0
              In
                {For j <_ l To u Do          % Misspelled "<-"
                   Next s = s + A[j] * B[j]
                 Finally s }} ;
```

Remember that the **LINE FEED** and **TAB** keys are very useful in automatically indenting Id programs. There is a syntax error in the program: a misspelled keyword.

Assume that the cursor is just past the final semicolon. Attempt to compile the procedure by typing:

`c-C c` or `M-sh-C` (ZMACS: `c-sh-C`)

On discovering the syntax error, the compiler opens the editor's typeout window at the top of the buffer, reporting:

```
ERROR at Line 5, Column 22 (Character # 115) from module FILE-PARSER
{For j <_ 1 to u Do
         -
Syntax error: "<-" expected.
```

Correct the error and attempt to compile it again:

`c-C c` or `M-sh-C` (ZMACS: `c-sh-C`)

This time the compilation is successful, and the procedure is also loaded.

Now type in, compile and load two more Id procedures: the "vector- sum" and a test function:

```
%%% Vector sum of two vectors

Def vsum A B = {  l,u = bounds A ;
```

```
                typeof A = vector I;
                       typeof B = vector I;
                     In
                       {Vector (1,u) of
                        | [j] = A[j] + B[j] || j <- 1 To u }};

     Def test A B = ip (vsum A A) (vsum B B) ;
```

2.3 Running the Examples

Switch to the lisp interaction buffer by typing `M-x id world`.

Run the procedure test supplying two vectors `A` and `B`.

```
    (id-run :vsum $5[10 10 10] $5[10 11 12])
```

The expression "`$5[10 10 10]`" evaluates to an Id vector with 3 components with indices 5 through 7 and values 10, 10 and 10 respectively.

Monsoon will now run. After giving the message:

```
RUN-UNTIL-IDLE returned :HALT-INSTRUCTION
Answer is present
Signal is present
No errors were reported on PE 0.
```

the result is printed out:

```
    $5[20 21 22]
```

i.e., a vector of length 3 with indices 5 through 7 and values 20, 21 and 22, respectively.

"Capture" the last result by typing:

```
    (setq v *)
```

Here you make use of the fact that in the IW listener, "*" is a variable that is always bound to the last thing returned.

You can now type:

```
    (id-run :ip $5[10 10 10] v)
```

and you will get the result:

```
    630
```

2.4 Debugging

To demonstrate errors encountered while running on Monsoon, we will invoke `vsum`, deliberately supplying incorrect arguments.

```
    (id-run :vsum $5[10 10 10] $4[2 4 6])
```

Here, the arguments to `vsum` have mismatched index bounds. Id World prints:

```
RUN-UNTIL-IDLE returned :HALT-INSTRUCTION
Answer is absent
Signal is absent
One error was reported on PE 0.
Error from VSUM+110 in frame 11264 on PE 0: "SVC7": "Bounds Error in VSUM (#<PLACE 373[15,c34]>)
$E
$E
```

showing the location where the error occurred— the body of function **vsum**. The $E notation means the result is still "empty".

Chapter 3

Preparing Id Code

To prepare Id code, enter the EMACS editor by typing the shell command **emacs** (SYMBOLICS: `SELECT E`), (TI: `SYSTEM E`). Here you use standard EMACS text-editing commands as well as special Id Mode commands to enter and format your program. You use Id Mode commands also to compile all or part of your program. If there are compilation errors, there are more Id Mode commands to facilitate their correction.

3.1 Id Mode in ZMACS and GNUEMACS

Id Mode is a new major mode for GNUEMACS, a public-domain implementation of EMACS available for UNIX machines, and for ZMACS, the editor on Symbolics and TI Lisp Machines. We will henceforth refer to EMACS instead of ZMACS on Lisp Machines and GNUEMACS on Unix machines. When EMACS is editing a buffer in Id Mode, it has several commands which understand the structure of Id procedures, and interfaces with the Id Compiler and IW in order to simplify program development.

The preferred way to set a buffer to Id Mode is to initiate it with the *Find File* command (invoked by `c-X c-F`), and to specify a file name of the form *foo*`.id`. EMACS uses the "`id`" file type to set the buffer automatically to Id Mode.

However, if a buffer is not already in Id Mode, you can always set it using the command

```
    m-X id mode
```

When you are in Id Mode, the editor mode line (the top line in the small pane at the bottom of the editor window) will look similar to this:

```
    ---Emacs:  example.id (ID)---All-----------
```

Here, the `ID` symbol specifies the major mode for this editor buffer.

Editor Typeout Window, and Getting Help

At various points, EMACS and the Id compiler print informational messages by opening a *typeout* window over the editor buffer, growing it downward from the top. After reading the messages, you can usually get rid of the typeout window by typing `SPACE` or `c-C`. (ZMACS: `ABORT`, `c-L`, `REFRESH`, or `CLEAR SCREEN`).

To get help on any subject, say "compile", type:

> `c-h A` (ZMACS: `HELP A`)

and then type

> `compile`

in the minibuffer.

In the typeout window, EMACS will list all commands that contain the substring "compile". Each command is accompanied by a one-line explanation and the keyboard command that invokes it.

To get a brief summary of all Id Mode commands, type:

> `c-h D` (ZMACS: `HELP D`)

and then type

> `id mode`

in the minibuffer (ZMACS: `m-X Describe Command` and then Id Mode.

In the typeout window, EMACS will list all Id Mode commands with a short description.

3.2 Typing In and Editing Id Code

The following commands concern initiating and saving files.

`c-X c-F` *Find File*

This is how you begin editing a file. Prompts you for a file name, creates a new buffer for that file, reads the file into the buffer, and leaves the cursor at the top of the buffer. If the file name is "*foo*`.id`", the buffer is automatically put in Id Mode. Because a new buffer is created for each file, EMACS allows you to edit many files simultaneously.

`c-X c-B` *List Buffers*

Causes the names of all your buffers (which are normally similar to the names of the files they are associated with) to be listed in the EMACS typeout window. By mousing any of the names, you can switch to that buffer.

`c-X c-S` *Save File*

Writes the buffer out to (a new version of) the associated file. This is how you save editing changes.

In addition to the usual text manipulation commands, there are some special commands in Id Mode which understand Id syntax. Most of these commands are counterparts to Lisp Mode commands. In the following commands, the phrase *current procedure* refers to the Id procedure on which the cursor is currently placed.

c-m-A *Beginning of Id Procedure*
c-m-[

Moves to the beginning of the current Id procedure.

---

c-m-e *End of Id Procedure*
c-m-]

Moves to the end of the current Id procedure.

---

m-p *Backward Id Procedure*

Moves to the beginning of the current Id procedure, or to the beginning of the previous procedure if already at the beginning of a procedure. With a positive argument, moves to the *n*'th previous beginning; with a negative argument, moves to the *n*'th succeeding beginning.

---

m-n *Forward Id Procedure*

Like **c-m-A**, but moves to the end of Id procedures.

---

c-m-H *Mark Id Procedure*

Makes the current region be the current Id procedure. With an argument, marks that many procedures forward or backward.

LINE FEED (SYMBOLICS: LINE) *Indent New Line*

Like typing **RETURN** to begin a new line, except that it also automatically indents it according to the previous line.

Tab *Indent for Id*

Indents the current line according to the rules for indenting Id. The **Tab** key may be typed with the cursor *anywhere* on the line. Indeed, sometimes you must type **TAB** only *after* typing something on the line because the correct indentation depends on the contents of the line.

If the cursor was within the line's indentation, it is moved to the end of the indentation.

---

c-m- *Indent Region*

Like typing **TAB** on each line of the region, from top to bottom.

---

c-m-Q *Indent Id Procedure*

Like typing **c-m-H** followed by **c-m-**.

The following commands are useful for inserting and removing comments in Id code. We recommend that you adhere to guidelines on page 348 of the Common Lisp manual (Guy L.

Steele, Jr., Digital Press, 1984) for commenting code, except that you must use "%" instead of ";" as the comment character.

`m-;` *Indent For Comment*

If the current line has no comment, starts a comment by inserting a "%" at the right end. If the line already has a comment, realigns it to the correct column. Positions cursor at start of comment text.

`ZMACS: c-m-;` *Kill Comment*

If the current line has a comment, deletes it.

`ZMACS: m-P` *Up Comment Line*

`ZMACS: m-N` *Down Comment Line*

Moves cursor to beginning of comment in previous (respectively, next) line. If line has no comment, starts the comment by inserting "%".

`c-X c-;` *Comment Out Region*

Comments out all lines in marked region by inserting "%" in first column(s).

`c-X c-:  (ZMACS: c-X m-;)` *Uncomment Out Region*

Uncomments out all lines in marked region by removing "%" from first column(s).

`m-X uncomment region` *Uncomment Region*

Removes comments from all lines in marked region.

There are two EMACS variables of potential interest in Id Mode (they can be modified by the EMACS command `m-X set variable`):

`Id Indentation Increment` *[Id Mode Variable]*

The amount of indentation to add in certain circumstances, in columns. Default is 2.

`Comment Column` *[Id Mode Variable]*

The column to which comments are indented by `c-;` in characters (ZMACS: pixels). Default is 40 (ZMACS: 384 (64 columns)).

In addition, Lisp-hacking commands like `c-m-B`, `c-m-F`, `c-m-K`, `c-m-Rubout`, `c-m-N`, `c-m-U`, `c-m-P`, `c-TAB`, *etc.* do approximately the right thing, though they haven't been quite tuned for Id.

3.3 Compiling Id Code

The Id compiler takes Id source code as input and produces MOC code as output. The source code may be taken directly out of an editor buffer or from a file. In the former case, the object code is also loaded automatically into Monsoon; in the latter case, the object code is saved in a file that you must load explicitly.

Id source code is a series of *defs*. Because of the treatment of nested procedures, each *def* can expand into one or more *procedures*. Because of code-block partitioning, each *procedure* in turn can expand into one or more *code blocks*. MOC code is simply a series of *code block* descriptions.[1] The compiler generates various informational messages in the editor typeout window. If the compiler signals a WARNING or an ERROR, then whatever it is working on (a *constant* or *type* or *def* or *procedure* or *code block*, depending on how far along compilation has proceeded) may not be what the programmer intended, but continues to be processed by the compiler. If the compiler signals a FATAL error, then it abandons processing of the current *constant* or *type* or *def* or *procedure* or *code block* (again depending on how far along compilation has proceeded). In that case, the object code produced will be lacking some code blocks. If the compiler signals an UNRECOVERABLE error, the entire compilation process is abandoned, and no object code is produced.

When compiling from a file using the Id Mode command `m-X id compile file` or the Lisp procedure `ID:ID-COMPILE-FILE`, the compiler leaves both the object code in a file of the same name but with type "moc", which stands for "Monsoon Object code" and the symbol table in a file of the same name but with type "
tt etb", which stands for "External Symbol Table in the binary encoding" (other encodings are available, but not of interest to most users). The `moc` and `etb` files can be loaded into Monsoon at any time by using the Lisp procedure `ID-LOAD`.

Here are the Id Mode commands for compiling Id code:

`c-G (ZMACS: c-ABORT)` *Abort*

Stops the editor from displaying compiler output, see section 4.2 on how to abort the lisp process. On ZMACS the compiler is stopped while it is running.

`M-x Id Compile Definition` *Compile Id Definition*
`m-sh-C (GNUEMACS: c-C c)(ZMACS: c-sh-C)`

Compiles and loads into Monsoon the contents of the current definition (LISPM: or region).

`GNUEMACS: M-x id compile region` *Compile Id Region*
`GNUEMACS: c-C r`

Compiles and loads into Monsoon the contents of the current region.

`m-X id compile buffer` *Compile Buffer*
`GNUEMACS: c-C b`

Compiles and loads into Monsoon the entire buffer.

---

[1] *Monsoon Software Software Interface Specifications* contains a complete specification (see Section 1.3).

`m-X id compile file` *Compile File*

`GNUEMACS: c-C f`

Prompts for the pathname of an Id file and compiles it, writing `moc` and `etb` files. An alternative to using the Lisp function `ID:ID-COMPILE-FILE`. Note that this command has nothing to do with the current buffer.

The following commands are useful in correcting compilation errors. They allow you to scan simultaneously the error list produced by the compiler and the source code.

`ZMACS: m-X Edit File Messages` *Edit File Messages*

`ZMACS: m-X Edit File Warnings`

Prompts you for a file name of an Id file, the default being the file already in the current buffer. Then puts you in two-window mode, with the compiler messages for that file in the upper window, and the file itself in the lower window. Initially, the upper window's cursor (inactive) is on the first message, and the lower window's cursor (active) is on the corresponding place in the Id code. Each time you type `c-.`, the upper cursor is advanced to the next message, and the lower cursor moves to the appropriate place in the Id code. When no messages are left, typing `c-.` puts you back in one-window mode (the file's buffer).

Some messages issued by the compiler do not say exactly where in the Id code the problem lies. When you `c-.` to one of these messages, EMACS doesn't try to move the lower window's cursor at all, but instead prints a message saying that it doesn't know exactly where the error is.

When editing compiler messages, you are free to make changes to your Id code as you visit each message. If you do, however, then the messages you haven't yet visited may no longer point to the correct spot in the Id code. It may be wise to visit all the places with messages first, and then go back to them to make changes.

All of EMACS's editing commands work just fine while you are visiting compiler messages, including the command `c-X 1`, which returns you to one-window mode. If you do go to one-window mode before you have visited all compiler messages, then the next time you type `c-.` you will return to two-window mode, as you expect. Moving the cursor in the top window does the "right" thing: it affects which message you visit the next time you hit `c-.`.

`ZMACS: m-X Edit Compiler Messages` *Edit Compiler Messages*

`ZMACS: m-X Edit Compiler Warnings`

This is like `m-X Edit File Messages`, but allows you to edit messages for several files at once. You are not prompted for a file; instead, EMACS goes through each file for which there are messages recorded, and asks you if you want to see them. EMACS then goes into two-window mode, as before, except that the top window may have messages for several files. `c-.` works as before, and even makes sure the correct file is displayed in the lower window.

If you compiled Id from a buffer or buffers not associated with any file, then `m-X Edit Compiler Messages` will get these messages too. When you visit them, however, EMACS doesn't even know what file they came from, so the lower window will not necessarily contain relevant code.

`m-X id compiler messages` *Compiler Messages*

`ZMACS: m-X Compiler Warnings`

This is just like `m-X Edit Compiler Messages`, but displays all the in one-window mode. In ZMACS you can enter into two-window mode by typing `c-.`. You can then proceed as with `m-X Edit Compiler Messages`.

3.3.1 Incremental Compilation

(See also Section 4.5)

The Id Compiler is an incremental compiler, that is, individual definitions may be compiled in any order without affecting the results at run-time. The Id Compiler is also an optimizing compiler, so the more it knows about each definition, the more it can optimize code. Thus, the order of compilation does affect run-time performance even though it does not alter the correctness of the code generated.

There is one exception[2] to this — an algebraic type must be compiled before any of its disjuncts may be used in pattern matching (either in a pattern in a **Case** clause, in the pattern on the left hand side of a block binding or loop binding, or in the formals of a procedure definition). There are two ways this may fail. Consider the following example:

```
%%% Find the depth of a tree.
Def depth (empty v)  = 0
  | depth (node l r) = 1 + max (depth l) (depth r) ;

%%% Definition of Tree algebraic type:
Type tree *0 = empty *0 | node (tree *0) (tree *0) ;
```

If these two definitions are compiled in order, compilation will fail because `empty` and `node` are not known to be constructors of an algebraic type at the time of the compilation of `depth`, so the compiler does not know how to generate a test to see if the object passed to `depth` is an `empty` or a `node`. The compiler will yield the error message

```
FATAL ERROR at Line 18, Column 15 from module GENERATE-PROGRAM-GRAPH
   EMPTY is not the name of a known algebraic type constructor.
```

The type-checker is fairly sensitive to order of compilation; it makes assumptions about all procedures called and all types used within a definition. To ensure that the type checker infers correct the type for a definition, make sure that all the symbols referenced in the definition have been defined. The type checker will also make assumptions about any undefined symbols it sees, and compilation will still occur, but the type of the definition may not be as constrained as it should be.

The efficiency of the code generated for a procedure will still depend somewhat on the order in which definitions are compiled. If the compiler sees a full-arity[3] application of a procedure, then it will generate a *Fastcall-Apply* instead of the more expensive general *Apply*. If the called procedure is a `Defsubst` and there is a full-arity application of the procedure, then the compiler will wire in a copy of the body of the called procedure instead of an application of the procedure. Although these three forms of application differ in implementation, they have the same semantics — the run-time behavior will differ only in number and type of instructions executed; the results will be the same.

[2]Isn't there always?

[3]An application of an arity r procedure to r arguments.

Chapter 4

Running Id Code

4.1 Invoking Id Functions

Id functions on started on Monsoon by using the ID-RUN lisp function.

For example, suppose you have compiled and loaded a function **total** that has one formal parameter **A** (an array) and returns the sum of all the numbers in the array. When invoked from ID-RUN, you will be required to supply a single argument, **A**. For example, you may now type:

```
    (id-run :total $1[11 12 13])
```

i.e., Execute total with an argument representing an Id array of 3 elements with indices 1 through 3 and values 11, 12 and 13, respectively. The function is run on Monsoon, the result and run status is typed out in the IW listener:

```
    36
```

Typing Lisp "front-end" forms

When you are in the IW listener, you are conversing with a Lisp front-end that drives an Monsoon back-end machine. Thus representations of Id forms are entered in Lisp, these forms are converted to Id forms and fed to the back-end machine, and when the back-end machine produces Id results, the front-end machine types them out in a Lisp representation.

The IW listener is customized so that there are convenient syntactic forms for Id objects in Lisp. We call these forms the "Lispified" Id syntax, which is specified in Section 1.5.3. To enter the IW listener type:

```
    (iw:listener)
```

You can invoke any of your Id functions that have been previously loaded with ID-RUN, with arguments in their Lispified syntax. It is important to keep in mind this model of a front-end machine driving a back-end machine, as it can have a significant effect on the parallelism statistics you gather and on the contexts that you see in the debugger.

For example, suppose you have defined, compiled and loaded the following Id functions:

```
Def f x = ... ;

Def g x = ... ;

Def h x = f (g x) ;
```

There is a big difference between the following two invocations in the IW listener:

```
(iw:id-run :f (iw:id-run :g 23))

(iw:id-run :h 23)
```

Using Lisp's evaluation rules for the front-end function, the first form invokes Monsoon on `g` with argument 23. When this returns a result r, Lisp then invokes Monsoon again on `f` with argument r. Because of these separate invocations, if `f` puts you in the debugger, there will be no sign of `g`. Further, the statistics you see after this run will only be for the call (`f` r).

The latter form invokes Monsoon only once on `h` with argument 23, and consequently both `f` and `g` run in the same Monsoon invocation. Using Id's non-strict evaluation rules, `f` will be invoked before `g` has returned a result. If `f` puts you in the debugger, you will see contexts for both `f` and `g`. The parallelism statistics gathered will be for both `f` and `g`.

4.2 Aborting Lisp

While Lisp is running (*i.e.*, not waiting for terminal input), you can abort back to the "`IW>>:`" prompt of the IW listener by typing the abort command `c-ABORT` on the Lisp Machines. On Unix Lisps, you may interrupt lisp `c-C c-C` and abort or continue running function from the lisp debugger. (LUCID: :a or :c) (ALLEGRO: :pop or :continue)

4.3 Aborting an Id Run

While Id is running on Monsoon, Id World is waiting for a return result and status. To interrupt a program running on Monsoon, first interrupt lisp (see section 4.2), then force an interrupt to Monsoon by typing `(IW:FORCE-HALT)`.

4.4 Loading Standard Libraries

Many of the function names commonly used in Id programs are not part of the language *per se* but are *standard identifiers*. Examples include `not`, `Bounds`, *etc.* These functions exist in various standard libraries.

Id World normally pre-loads the standard libraries:

```
types-library.moc
basic-library.moc
transcendental-library.moc
```

Other libraries include:

```
accumulator-library.moc
array-library.moc
complex-library.moc
list-library.moc
set-library.moc
string-library.moc
```

to load array, list, set and other useful abstractions (a partial description is in the Appendices of the *Id 90.1 Reference Manual*).

4.5 Separate Compilation

(See also Section 3.3.1)

Id World fully supports separate compilation and dynamic linking of Id procedures. This means, for example, that if you make a change to one function in your Id Program, you need only recompile that function (via the editor command `M-sh-C` (ZMACS: c-sh-C), for example) and not the whole program.

When you invoke an Id procedure, the consistency checker verifies that all the procedures that may be referenced have definitions loaded. If there are undefined symbols, you may get a message that looks similar to this:

```
IW>> (id-run :foo 23)
Check...
WARNING:  Running procedure FOO might cause references to the following undefined symbols:

    BAR     GLURPH      MAKE_2D_ARRAY

Do you want to go ahead anyway? (Y or N)
```

This may occur because you forgot to define some procedures in your source code, or because you misspelled the name of a procedure, or because some standard library has not yet been loaded (as in `MAKE_2D_ARRAY` above).

You can still run the program without correcting the situation (by responding `Y`). Any dynamic reference to an undefined symbol is then simply treated as a run-time error.

You can correct the situation as follows. First respond `N` to abort the run and come back to the "`IW>> `" prompt. Switch to the editor and correct any errors there, remembering to recompile any changed definitions. Load any necessary standard libraries, as described in the next section. Finally, re-enter the expression in the IW listener (this is most easily done by using the `m-P` comand-line editor commands (TI: `c-C` and `m-C`) (Symbolics: `c-m-Y` and `m-Y`) to yank back the things you typed most recently.)

Occasionally, making a change to one function can create the need to recompile other functions, even though you have made no change to the source code for those other functions. For example, if you have an Id procedure `f` which is declared to be an inlinable procedure (via the Id keyword `defsubst`), and another procedure `g` which calls `f`, when you make a change to `f` you must recompile *both* `f` and `g`. Because the old definition of `f` was substituted into `g` at the time `g` was compiled, simply recompiling `f` does not affect the behavior of `g` at run time. Similar problems can arise when you change the number of formal parameters in the definition of a function, even when that function is not inlinable, because the compiler uses that information to optimize calls when compiling other procedures.

To assist you in keeping track of what needs to be recompiled, IW will perform a consistency check on your program each time you invoke an Id procedure from the IW listener, to make sure all necessary recompilations have been performed. If there is an inconsistency, you will get a message like this:

```
IW>> (id-run :top 23)
Check...
WARNING:  the following procedures may need to be recompiled:

    G

Do you want to go ahead anyway? (Y, N, or D)
```

You can get further information about the problem by answering "`D`" (for "Describe"):

```
IW>> (id-run :top 23)
Check...
WARNING:  the following procedures may need to be recompiled:

   G

Do you want to go ahead anyway? (Y, N, or D) D

  For procedure G:
       F's INLINABLE-DEFINITION has changed.

Do you want to go ahead anyway? (Y, N, or D)
```

If you choose to run the program anyway, your program may behave unpredictably. In general, you should not answer "yes" unless you know that no problems will arise (for example, you know that `g` will not be called given the particular set of arguments used for this run).

Because procedures are compiled separately, you must be sure to compile an inlinable procedure before compiling any procedures which refer to it. If you compile `g` before compiling inlinable procedure `f`, for example, the compiler cannot substitute the definition of `f` in place of a call that appears in `g`, since it is not known at the time `g` is compiled. When you use an editor command that appears to compile several procedures at once—`m-X id compile region`, `m-X id compile buffer`, or `m-X id compile file`—the procedures are in fact compiled separately, in the order of their appearance. This means that inlinable procedures should appear nearer the top of a buffer or file than procedures which refer to them.

Because the type checker is also sensitive to compilation order, you may get messages at invocation that a procedure may need to be recompiled because of incorrect assumptions about a symbol's `TYPE-ARITY`, `RECURSIVE-SET`, `DEFINED-TYPE`, or `NON-GENERIC-TYPES-ENCODING` properties. As before, you may proceed, ignoring the message, or you may recompile the definitions so that the type checker works with up-to-date information.

Chapter 5

IW: The Id World Interface

5.1 Id World Listener

The Id World Listener is identical to a normal Lisp Listener with the following differences:

1. The current package is IW:
2. The readtable is the Id World readtable.

You can start up an Id World listener by invoking one of two functions in the USER: package: `(id-world)` or `(iw)`.

You can leave the Id World listener by invoking one of two functions in the IW: package: `(exit)` or `(quit)`.

5.1.1 ID-WORLD-INTERFACE: package

All Id World Interface functions are in the IW: package, which has the following attributes:

Names ID-WORLD, IW

Uses LISP:, FOREIGN:, IOCF:, MACROARCH:, MMI:, LOADER:, EM:

Shadows COLOR-CODE-BLOCKS, FIND-CODE-BLOCK, GET-ANSWER, HARDWARE, INITIALIZE-RTS, MHD, MINT, SET-RTS-INITIALIZED, SETUP

Exports ANSWER-P, BOOT-RTS, CONTINUE-ID-RUN, END-ID-RUN, EXIT, EXSYM-GET, GET-ANSWER, GET-OPTION ID-COMPILE-FILE, ID-COMPILE-LOAD-FILE, ID-LOAD, ID-RUN, INITIALIZE-RTS, LISTENER, LOAD-LIBRARIES, MHD, MINT, SET-OPTION, SET-RTS-BOOTED, SET-RTS-INITIALIZED, SHOW-OPTION, SHOW-OPTIONS, WITH-IW-READTABLE

Imports and Exports ALL-CODE-BLOCKS, ALL-IDENTIFIERS, DM-POINTER-TO-VALUE, DM-WORD-TO-VALUE, IREF, LD-D, LD-DU, LD-INITIALIZE, LD-LOAD, LD-RLOAD, RUN-ONE-STEP, RUN-UNTIL-IDLE, SETUP, TREF, VALUE-TO-DM-WORD

5.1.2 Id World Readtable

Id World runs using an extended readtable, which is based on GITA's readtable for reading/printing Id objects. The syntax extensions provided in this readtable are as follows:

- NIL denotes Id's concept of the end of a list.
- $E denotes an empty slot in an I-structure.
- $T denotes the Id boolean "true".
- $F denotes the Id boolean "false".
- $'FN denotes a closure of the named function over zero arguments.
- $'(FN args...) denotes a closure of the named function over the specified arguments.
- $<v1 v2 ... > denotes an Id tuple.
- $(v1,v2,...) denotes an Id tuple.
- $low1,...lown[[v ... v] ... [v ... v]] denotes an Id array, or I-structure. The lower bounds are specified. The upper bounds are inferred from the lists of values.

5.1.3 Exported High-Level Functions

Here are the exported functions from the IW: package that you'll normally be using to compile, load, and execute Id programs.

Connecting to a Monsoon

`mint &optional` *contact-file-name* [*Function*]

Connects to a Mint server using the specified contact file name, or using `"~/default.mmi"` if no file name is specified. The queuing system will be set to the default Mint queueing system: :monsoon :grand-total.

`mhd &optional` *contact-file-name* [*Function*]

Connects to an MHD server using the specified contact file name, or using `"~/default.mmi"` if no file name is specified. The queuing system will be set to the default MHD queueing system: :monsoon :n-step-until-match, with the match token appropriately set to catch a successfully terminating Id program or Run Time System call. The timeout will be set to the maximum value.

Controlling the Id Run Time System

`boot-rts &optional` *(error-output t)* *[Function]*

Boots the Id Run Time System. Id World Parameter :rts-version specifies which directory contains the files that will be loaded, relative to the standard Id RTS root directory. Parameter :pmm-files is the list of files that will be loaded with the Restricted Loader (normally only the PMM.) Parameter `:rts-files` is the list of files that will be loaded with the normal loader.

`set-rts-booted` *[Function]*

This is needed only when you are switching back and forth among multiple Monsoons. It tells Id World that the Monsoon it is talking to has already had the Id Run Time System booted, and makes Id World read various parameters from the Monsoon.

`initialize-rts &key frame-size frame-area-k heap-area-k` *[Function]*

Initializes the Id Run Time system. `:frame-size` allows you specify the number of words in each frame. `:frame-area-k` and `:heap-area-k` are useful only when connected to Mint. Id World will use these values to artificially reduce the size of regions in the corresponding area in the PMM before initializing the RTS. This can reduce the initialization time under Mint to something bearable.

`set-rts-initialized` *[Function]*

Tells Id World to check whether the Id Run Time System is initialized and set various flags appropriately. Again, useful primarily if you are switching among multiple Monsoons.

Environment Commands

`set-option &rest` *parameter-value-pairs* *[Function]*

```
Ex: (set-option :rts-version "012-mcrc-02")
```

Sets one of Id World's parameters. A parameter is named by a keyword. A parameter is followed by a value. You can set more than one parameter at a time with this function. Currently specified parameters:

- :stats-spec `(set-option :stats-spec nil)` or `(set-option :stats-spec :none)` turns off statistics gathering. `(set-option :stats-spec :all)` will collect all statistics registers. `(set-option :stats-spec '`*(list of numbers)*`)` will collect a particular set of statistics registers.

- :stats-file `(set-option :stats-file` *file*`)` will open a CIOBL file for collecting statistics. `(set-option :stats-file NIL)` will close a statistics file.

- :stats-block-size `(set-option :stats-block-size 100)` specifies that statistics will be streamed out to the stats file every 100 timesteps.
- :answer-type Due to buggy type-propagation, the EM can't always tell just what is being returned by an Id procedure. You can use this option to tell it what to expect, regardless of type bits. `(set-option :answer-type` *keyword*`)`, where keyword is one of the following: `:void`, `:integer`, `:boolean`, `:character`, `:float`, `:closure`, or `:aggregate`.
- :aggregate-type For a similar reason, the EM can't always tell what the types are inside an aggregate (tuple or i-structure). Use `(set-option :aggregate-type keyword)`, where keyword can one of the following: `:void`, `:integer`, `:boolean`, `:character`, `:float`, or `:closure`.
- :rts-version `(set-option :rts-version "014")` Tells Id World to find the RTS files in subdirectory "014" within the normal id-rts directory root.
- :aggregate-overhead Different heap managers don't necessarily follow the IOCF for specifying the active length of aggregates. This option allows the EM to work correctly with any of them. "first-fit" requires this to be 1, whereas "quick-fit" requires it to be 0.
- :number-of-pe/is When you connect to a Monsoon using `(mint)` or `(mhd)`, the server will tell Id world what the node configuration is. If you wish to use a run time system that supports a different configuration that is a subset of what the server supports, use this option. `(set-option :number-of-pe/is '(1 2))` says to use 1 PE and 2 IS boards.
- :pmm-files `(set-option :pmm-files '(`*list of files*`))` specifies the list of files to be loaded with the restricted loader before (Id-initialize) is called. Normally, this is just the PMM.
- :config-files `(set-option :config-files '(`*list of files*`))` specifies a list of files to be loaded with the normal loader immediately after the PMM is loaded and the loader is initialized. Normally, this will be a configuration file defining the various loader areas.
- :rts-files `(set-option :rts-files '(`*list of files*`))` specifies a list of files to be loaded with the normal loader immediately after the config files are loaded. Normally, this will be the run time system.
- :heap-mgr-files `(set-option :heap-mgr-files '(`*list of files*`))` specifies a list of files to be loaded with the normal loader immediately after the rts files are loaded. Normally, this will be the heap manager.
- :frame-size `(set-option :frame-size` *n*`)` sets the frame size in words to be used in subsequent `(initialize-rts)` calls.
- :frame-area-k `(set-option :frame-area-k` *n*`)` sets the size of the frame area in K words to be used in subsequent `(initialize-rts)` calls.

- :heap-area-k `(set-option :heap-area-k n)` Sets the size of the heap area in K words to be used in subsequent `(initialize-rts)` calls.
- :queueing-system `(set-option :queueing-system '(`*qsys qmode qmodeargtypes qmodeargs*`))` Sets the queueing system to be used for running your Id function.
- :timeout (set-option :timeout *n*) Sets a timeout value, specified in number of clock ticks, for running your Id function. This is accomplished by selecting a queueing mode based on the currently selected queueing system/queueing mode that allows such a timeout. For example, `:grand-total/:normal` will turn into `:grand-total/:n-step`. `:monsoon/:until-match` will turn into `:monsoon/:n-step-until-match`. `(set-option :timeout NIL)` will transform your queueing system in the other direction, removing any timeout.
- :rts-queueing-system `(set-option :rts-queueing-system '(`*qsys qmode qmodeargtypes qmodeargs*`))` sets the queueing system to be used by the EM for running RTS functions.
- :rts-timeout This is similar to option `:timeout`, but applies to the `:rts-queueing-system`.
- :code-block-colors `(set-option :code-block-colors '((0 :pi1 :pi2) (1 :foo) (2 :bar)))` This sets the colors to be applied to specified functions prior to setting up your Id function. All code blocks not listed will get color -1, which means inherit the color from your caller.

`get-option &rest` *parameters* [*Function*]

`Ex: (get-option :frame-length)`

This function will get the current setting of specified Id World options, or of all of them, if none are specified.

`show-option &rest` *parameters* [*Function*]

This function prints out the current settings of specified Id World parameters, or all of them if none are specified.

`show-options` [*Function*]

This function prints out the current setting of all Id World parameters.

`exsym-get` *procedure key* [*Function*]

Searchs the loaded Exsym tables for the specified procedure and returns the value of the parameter specified by the key.

Compiling and Loading

`id-compile-file` *input-file* `&key <compiler-specific-keys>` [*Function*]

```
Ex: (id-compile-file "~/id/wavefront")
```

Compiles a file of Id code producing a MOC file and an ETB file. The currently available keys are:

libraries libraries to compile with. The defaults are "types", "basic", and "trans".

`id-load` *file-or-files* `&key (error-mode :continue) (error-output t)` [*Function*]
`(ld-verbose t) (etb-verbose t)`

```
Ex: (id-load '("~/id/wavefront" "~/id/pi")
```

Calls the Monsoon loader to load one or more MOC files. After loading, the corresponding ETB (exsym table) files will be read into Id World. :error-mode can be either :continue or :abort. :error-output can be T or NIL. If NIL, the loader's error output will be suppressed. :ld-verbose controls whether the pathnames of files are printed before they are loaded. :etb-verbose controls whether the names of EXSYMs are printed as they are loaded.

`id-compile-load-file` *file-or-files* `&rest` *args* [*Function*]

Calls (id-compile-file) on each of the files with the specified args. Then calls id-load with the resulting MOC files and default arguments.

`load-libraries &rest` *names* [*Function*]

```
Ex: (load-libraries "trans")
```

Loads the specified libraries into the Monsoon. If your functions use any libraries other than "types" or "basic", you will need to load the library before executing your program. The consistency checker will tell you if necessary functions are not loaded.

Executing Id programs

`id-run` *code-block-name* `&rest` *arguments* [*Function*]

```
Ex: (id-run :simple 1 2 5 2 5)
```

Executes the code block named by CODE-BLOCK-NAME. The arguments are the arguments expected by the code block. If Id World is able to determine the arity of the code block, this will be compared to number of arguments given. The Consistency Checker is called, and the function is executed on the Monsoon. After the Monsoon stops running, the

Answer and Signal are checked for, and are extracted from Monsoon memory, if present. They are stored in variables *answer* and *signal* and are returned as the values of this function.

`force-halt` *[Function]*

If you get impatient during the execution of your Id procedure, you can type a Control-C to get the Lisp Error Handler. At that prompt, you can invoke the force-halt function to force the Monsoon the stop executing. When you continue from the Error Handler, id-run will return :halt-forced. You are then free to poke around and figure out what state your program is in.

`continue-id-run` *[Function]*

Use this to continue running your Id program, after getting a :halt-forced.

`end-id-run` *[Function]*

Use this to tell Id World that you will not be continuing the Id program. Id World will gather the final statistics and flush out the statistics file.

`answer-p` *[Function]*

Returns two boolean values. The first says whether the answer is present. The second says whether the signal is present.

`get-answer` *[Function]*

Returns two values: the answer and signal. These are also stored in variables *answer* and *signal*. Fetching of the answer is controlled by the setting of the **`:answer-type`** and **`:aggregate-type`** parameters.

Statistics Collection and Viewing

Id World provides low level support for statistics collection and a general interface to the statistics processor/viewer. The following options (described above under set-option) pertain to statistics collection:

- :stats-spec
- :stats-block-size
- :stats-file
- :queueing-system
- :code-block-colors

Additionally, the following macro can usefully surround a call to `(id-run)`:

`with-stats-file` (*file block-size*) {*form*}* [*Macro*]

Opens the specified stats file, sets the specified block size, and executes all the forms, returning the final value. The stats file will be closed when the macro is left.

Id World also loads in the Statistics Processor and Viewer. This program is able to generate X windows and Postscript files containing plots of "Statistics Profiles" collected by Id World. A Profile is a curve whose X axis is "Timesteps" and whose Y axis is an arithmetic expression on various Monsoon statistics registers.

There are two functions that interface to this program:

`sv-toplevel` *file* [*Function*]

Reads an *sv* input file and calls the stats viewer.

`sv-display` `&optional` *splot-file-name profile-definition-list graph-list h-graph-definition-list stats-definition-list table-definition-list (error-stream t)* [*Function*]

This is the low-level interface to the Stats Viewer.

A *profile-definition-list* is a list of profile definitions. A *profile definition* is a list defining one profile. A profile has a name, expressed as a keyword, and a *profile expression* that defines which Monsoon statistics registers will be combined in what manner. There are three variants of profile expression.

1. (:id *pname fname*) refers to a stats register identified by *pname* to be read from file *fname*. *pname* and *fname* can be strings, symbols, or keywords.
2. (:unary *y-operation operand1*) refers to a unary operation over an operand. Available *y-operations* include :negate and :integrate. *operand1* can be any profile expression.
3. (:binary *y-operation operand1 operand2*) refers to a binary operation over two operands. *y-operation* can be :plus, :minus, or :divide. *operand1* and *operand2* can be any profile expression.

Example:

```
(:profile-1
  (:unary :integrate (:binary :minus (:id "STAT-10" "stats.out")
     (:id "STAT-11" "stats.out"))))
```

is a profile definition defining a profile named :profile-1 with a profile expression that says to take the integral of the difference between statistics registers "STAT-10" and "STAT-11", both from file "stats.out".

In addition to defining the profiles that you want the stats processor to caluclate for you, you must define the graphs that you want the stats viewer to generate for you. The "graph-list" is the mechanism for this.

A *graph-list* is a list of *graphs.* A *graph* is a list of four elements: The name of the graph (a keyword), the title of the graph (a string), *graphic-info* and a list of *output profiles.*

graphic-info is a list of parameters defining the overall format of the graph:

(*graphic-medium xlabel ylabel xtick ytick xmin ymin xmax ymax fill graph-size*)

graphio-medium can be either :x (to plot the graph in an X window) or a list of two elements specifying the Postscript file to be created. The first element is a string containing the name of the file. The second is T if an existing file will be appended to, or NIL if an existing file should be overwritten. *xlabel* and *ylabel* are strings that will be printed by the X and Y axes of the graph. *xtick* and *ytick* are integers that specify how often a tick mark is generated on the X and Y axes. *xmin* and *ymin* are integers defining the origin. *xmax* and *ymax* are integers defining the length of the axes. *fill* is a boolean (T or NIL) specifying whether the area beneath the curves should be filled with gray scale. *graph-size* is a list of two floating point numbers: *xsize* and *ysize* giving the physical size of the graph in inches.

An *output profile* defines the appearence of a *profile* (as defined above) in the context of a *graph.* It consists of a list of parameters:

(*name xmin xmax xcons-ratio curve-info definition*)

name is the name of this output profile, expressed as a string, keyword, or symbol. *xmin* and *xmax* are integers specifying the range of timesteps to be included in the graph. *xcons-ration* specifies the *X consolidation ratio* to be applied to the data. An X consolidation ratio of 10, for example, means to display one value for every 10 input values. *curve-info* is a list of parameters describing this curve:

(*curve-type curve-symbol curve-gray font label cons-curve-repr*)

curve-type can be :solid or :dotted. *curve-symbol* can be :none, :triangle:, :diamond, :square, or :circle. *curve-gray* is a floating point number between 0.0 and 1.0. *font* is an integer. (What does it mean?) *label* is a string. *cons-curve-repr* chooses which Y-value to display when X-consolidation is performed: :mean, :min, :max, :max-and-min, or :all.

definition is the string, symbol, or keyword that names the *profile definition* associated with this curve.

h-graph-definition-list, stats-definition-list and *table-definition-list* are not implemented in the initial version of the Stats Processor.

Here is a grammar that documents how the various parameters are constructed. Upper case symbols are non-terminals. '(' and ')' refer to those specific characters – i.e. are used to surround a Lisp list. "keyword" means any Lisp string, symbol, or keyword. "string" means any Lisp string. "integer" means any integer. :XXX means the keyword XXX.

```
PROFILE-DEFINITION-LIST ::= '(' PROFILE-DEFINITION* ')'
PROFILE-DEFINITION ::= '(' NAME PROFILE-EXPRESSION ')'
NAME ::= keyword
PROFILE-EXPRESSION ::= PROFILE-EXPRESSION-ID | PROFILE-EXPRESSION-EXP
PROFILE-EXPRESSION-ID ::= '(' :id PNAME FNAME ')'
PNAME ::= keyword
```

```
FNAME ::= keyword

PROFILE-EXPRESSION-EXP ::= UNARY-EXPRESSION | BINARY-EXPRESSION
UNARY-EXPRESSION ::= '(' :unary Y-OPERATION OPERAND1 ')
BINARY-EXPRESSION ::= '(' :binary Y-OPERATION OPERAND1 OPERAND2 ')
Y-OPERATION ::= :plus | :minus | :divide | :negate | :integrate
OPERAND1 ::= PROFILE-EXPRESSION
OPERAND2 ::= PROFILE-EXPRESSION

GRAPH-LIST ::= '(' GRAPH* ')'
GRAPH ::= '(' NAME GRAPH-TITLE GRAPHIC-INFO OUTPUT-PROFILE-LIST ')'

GRAPH-TITLE ::= string
GRAPHIC-INFO ::= '(' GRAPHIC-MEDIUM XLABEL YLABEL XTICK YTICK XMIN YMIN
 XMAX YMAX FILL GRAPH-SIZE ')'
GRAPHIC-MEDIUM ::= :x | '(' PS-FILENAME APPENDP ')
PS-FILENAME ::= string
APPENDP ::= T | NIL
XLABEL ::= string
YLABEL ::= string
XTICK ::= integer
YTICK ::= integer
XMIN ::= integer
YMIN ::= integer
XMAX ::= integer
YMAX ::= integer
FILL ::= boolean
GRAPH-SIZE ::= '(' XSIZE YSIZE ')'
XSIZE ::= float
YSIZE ::= float

OUTPUT-PROFILE-LIST ::= '(' OUTPUT-PROFILE* ')'
OUTPUT-PROFILE ::= '(' NAME XMIN XMAX XCONS-RATIO CURVE-INFO DEFINITION ')
XMIN ::= integer
XMAX ::= integer
XCONS-RATIO ::= integer
CURVE-INFO ::= '(' CURVE-TYPE CURVE-SYMBOL CURVE-GRAY FONT LABEL CONS-CURVE-REPR ')'
CURVE-TYPE ::= :solid | :dotted | :fill
CURVE-SYMBOL ::= :none | :triangle | :diamond | :square | :circle
CURVE-GRAY ::= float
FONT ::= integer
LABEL ::= string
CONS-CURVE-REPR ::= :mean | :min | :max | :max-and-min | :all
DEFINITION ::= keyword
```

```
H-GRAPH-DEFINITION-LIST ::= '(' H-GRAPH-DEFINITION* ')'
H-GRAPH-DEFINITION ::= '(' ')'

STATS-DEFINITION-LIST ::= '(' STATS-DEFINITION* ')'
STATS-DEFINITION ::= STATS-PROFILE | STATS-HISTOGRAM | STATS-TABLE
STATS-PROFILE ::= '(' :profile NAME STATS-FILE-INFO OUTPUT-PROFILE-LIST ')'
STATS-HISTOGRAM ::= '(' :histogram NAME STATS-FILE-INFO ')'
STATS-TABLE ::= '(' :table NAME STATS-FILE-INFO ')'

STATS-FILE-INFO ::= PROPERTY-LIST BLOCK-SIZE

PROPERTY-LIST ::= '(' ')'
BLOCK-SIZE ::= integer

TABLE-DEFINITION-LIST ::= '(' TABLE-DEFINITION* ')'
TABLE-DEFINITION ::= '(' NAME FILE-NAME ')'
FILE-NAME ::= keyword
```

Imported Useful Functions

Id World's package does a (USE-PACKAGE) on the "FOREIGN", "IOCF", "MACROARCH", "MMI", "LOADER", and "EM" packages; you can use any exported function from those packages freely from within the Id World Listener.

Id World also imports and exports a collection of useful functions from its component systems. Some of these are useful in and of themselves, but some are primarily useful if you want to write your own `(id-run)`, for example.

`all-code-blocks` *[Function]*

From the Execution Manager. Searches the Code Block Descriptor Table and returns a list of all loaded code blocks.

`all-identifiers` *[Function]*

From the Execution Manager. Searches the Identifier table and returns a list of all loaded identifiers.

`dm-pointer-to-value` *pointer* `&optional` *type* *[Function]*

From the Execution Manager. Takes an mmi:pointer and reads Monsoon Data Memory at that address. Heuristically interprets the resulting DM-WORD as an Id object and returns the corresponding Lisp object. Optional argument "type" can be used to override the type bits on the DM-WORD.

`dm-word-to-value` *dm-word* `&optional` *pointer* *[Function]*

From the Execution Manager. Heuristically interprets the DM-WORD as an Id object and returns the corresponding Lisp object. Optional argument "pointer" is the pointer to DM where this word came from. Not necessary unless the DM-WORD has type bits "header".

`value-to-dm-word` *value* [*Function*]

From the Execution Manager. Takes an arbitray Lisp object and conses the equivalent Id object on the Monsoon, returning a DM-WORD. Non-aggregates don;t need consing.

`iref` *id-array* `&rest` *indices* [*Function*]

From the Execution Manager. Does an Id array reference on an Id array. (setf (iref ...) value) works.

`tref` *id-tuple index* [*Function*]

From the Execution Manager. Does an Id tuple reference on an Id tuple. (setf (tref ...) value) works.

`color-code-blocks` [*Function*]

Takes the current setting of the :code-block-colors option and colors the Code Block Desriptors on the Monsoon. Done automatically by `(setup)`, which is called by `(id-run)`.

`setup` *function* `&rest` *args* [*Function*]

Compares the arity of the function against the number of arguments, colors the code blocks, and calls (em:setup)

`run-until-idle` *event-handler stats-list stats-stream* [*Function*]

From the Execution Manager

`run-one-step` *event-handler stats-list* [*Function*]

From the Execution Manager.

`ld-rload` *path* `&key (error-mode :abort) (error-output t) (verbose t)` [*Function*]

From the Loader. Do a restricted load of a file.

`ld-initialize` [*Function*]

From the Loader. Initialize the Loader Internal area on the Monsoon.

`ld-load` *paths* `&key (error-mode :abort) (error-output t) (verbose t)` [*Function*]

From the Loader. Load a file or group of files.

`ld-d &optional` *list* [*Function*]

From the Loader. Get the list of dependents for the specified symbol.

`ld-du &optional` *list* [*Function*]

From the Loader. Get the list of undefined dependents for the specified symbol.

`with-iw-readtable &rest` *body* [*Macro*]

Execute the BODY with the Id World readtable in effect.

Acknowledgments

This document borrows heavily from the Id World Reference Manual by Rishiyur S. Nikhil, Revised: P. R. Fenstermacher, J. E. Hicks and R. P. Johnson

With minor editing, section 5 is taken from *Monsoon Id World Specification*, by Peter L. DeWolf, June 1991, version 010.

Mint, the loader, the Id World Interface, the execution manager, and statistics viewer were written by the members of the Motorola Cambridge Research Group whose members include: Tony Dahbura, Ken Traub, Mike Bekerely, Peter DeWolf, Maria Carlon, Venkat Natarajan, Norm Rubin, Andrea Carnevali, and Edward Zyszkowski.

The current Id compiler (Version 2) was written by Ken Traub; it was updated to Id 90.1 by Shail Aditya Gupta ; earlier versions were written by Vinod Kathail, Steve Heller, Traub and Jamey Hicks. Shail Aditya Gupta wrote the type checker for the current compiler.

The consistency checker was written by Alex Caro.

Id Mode was designed and implemented by Dinarte Morais and Ken Traub. R. Paul Johnson implemented Id Mode for GNUEMACS.

Appendix A

Bug Reports, Comments, Questions, Discussion

Bug reports, comments and questions may be sent to:

```
BUG-ID-WORLD@MC.LCS.MIT.EDU
```

For general discussion of Id World amongst people interested in Id World (not necessarily users), there is an electronic mailing list:

```
INFO-ID-WORLD@MC.LCS.MIT.EDU
```

To be placed on this mailing list, send your request (including your electronic mail address) to

```
INFO-ID-WORLD-REQUEST@MC.LCS.MIT.EDU
```

Appendix B

Example Screen Images

The following pages show many examples of screen images showing various stages in the example walk-through of Section 2.