LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

ID Compiler User's Manual

Computation Structures Group Memo 248

Version of February 7, 1986

Steve Heller

Ken Traub

This research was supported in part by Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract no. N00014-75-C-0661. Ken Traub is supported by a fellowship from the National Science Foundation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

1. Introduction

1.1. Overview

The Functional Languages and Architectures Group at MIT's Laboratory for Computer Science is designing a Dataflow Multiprocessor called the Tagged Token Dataflow Architecture (TTDA) which will run the dataflow language ID. ID was originally described in a paper by Arvind, Gostelow, and Plouffe[1]; this manual describes the current ID compiler.

The ID compiler has evolved over several years, beginning with Vinod Kathail's Version 0 in 1982. Written in MACLISP, it ran on MIT-XX (a DECSYSTEM-20) and was never fully debugged; it is no longer supported. After being transported to Symbolics 3600-series Lisp Machines, enhanced, and debugged, Version 0 became Version 1, the version described in this manual. Version 1 has been both available and supported since January, 1985, and has successfully compiled programs as large as 1200 lines. Version 2 will be designed to be far more flexible than its predecessors, facilitating compiler related research and experiments. This version is currently in the planning phase and should be available by the summer of 1986.

Figure 1-1 shows the relationship of the ID compiler to FLA's other software facilities. It's primary purpose is to produce executable code for the TTDA simulator and TTDA emulator, on which dataflow experiments are performed. The ID compiler also produces other forms of output which are useful when debugging ID programs, the simulator and emulator, and the compiler itself. Another facility, called IDSYS, exists for interpreting ID programs directly, without the intermediate step of producing TTDA code; it achieves this by translating ID into MACLISP. While IDSYS is available on MIT-XX, it is no longer being supported, and the dialect of ID it accepts differs somewhat from the version accepted by the ID compiler. In particular, the interpretation of structures differs in a subtle way.

---

[1] Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, University of California, Irvine, Department of Information and Computer Science, Technical Report TR-114, December, 1978. At that time, ID stood for Irvine Dataflow; perhaps now it stands for Institute Dataflow!

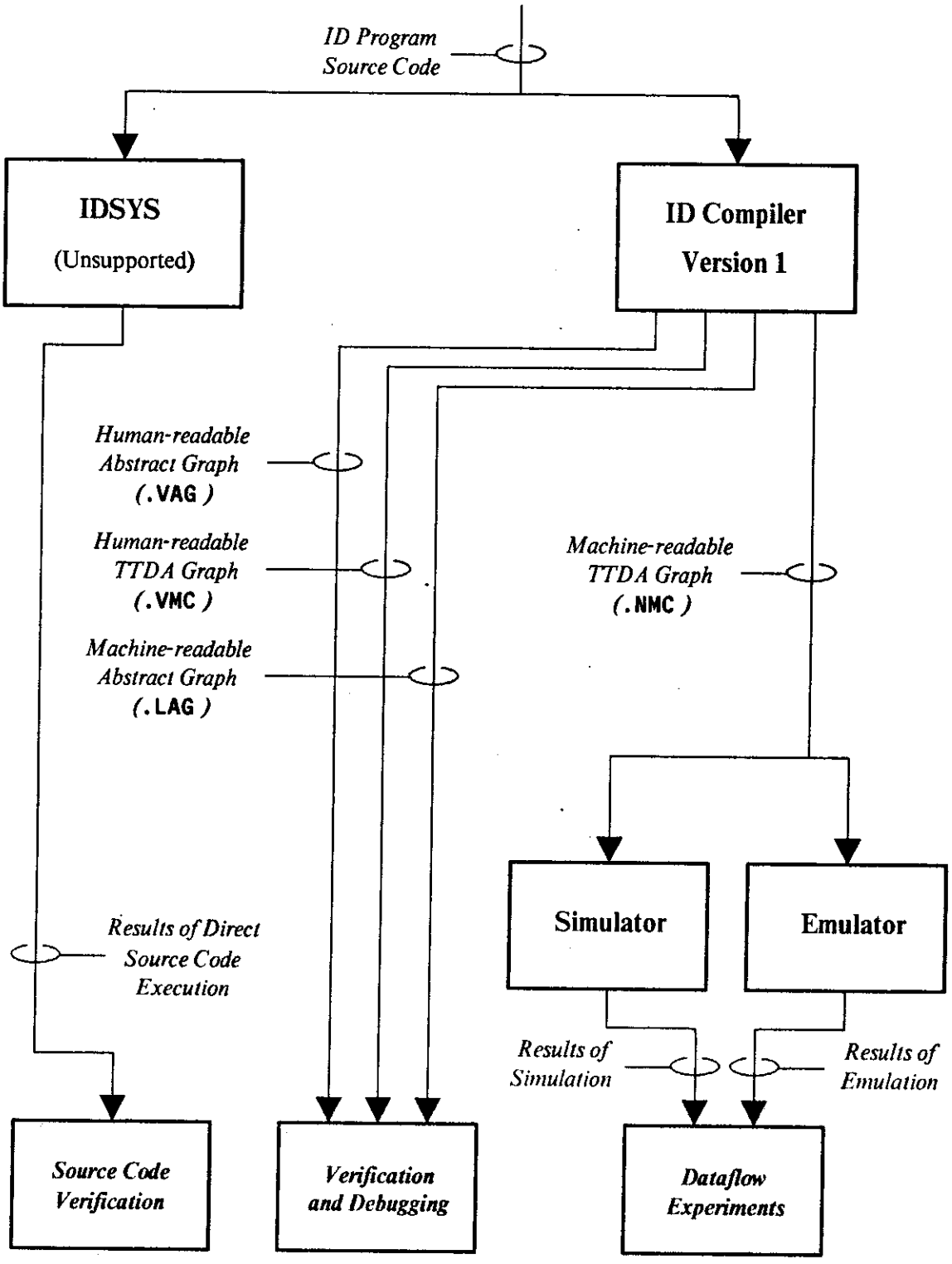

Figure 1-1: The ID Compiler and Related Facilities

1.2. The ID Compiler

The ID compiler is a large LISP program that takes an ID source file as input and produces an object file and a listing file as output. The object file contains the compiled version of the ID program; this may be either a U-interpreter style graph or machine code for the Tagged-Token Dataflow Architecture, in either a human-readable or a machine-readable form, as desired. The listing file contains a listing of the source program, together with line numbers, statistics, and error messages produced by the compiler.

The ID compiler currently runs on FLA's Lisp Machines. This means that you have to have access to one of FLA's Lisp Machines to run the compiler directly, although the file system of the Lisp Machine allows the source, object, and listing files to reside on any machine accessible to that Lisp Machine. This means, for example, that you can still keep your files on XX even though you invoke the compiler from JACARANDA. If you have access to XX but not to one of FLA's Lisp Machines, you can still use the compiler through the Remote ID Compiler (RIC) facility described in Chapter 3. The compiler that used to run on XX is no longer available.

The information in this manual corresponds to Version 1.13 of the ID Compiler.

2. Using the Compiler

The ID compiler can be used on any of FLA's Lisp Machines.[2] Unlike many Lisp Machine programs, there is no special window associated with the ID Compiler. Instead, you use a Lisp Listener to invoke the compiler (you can get a Lisp Listener by typing SELECT-L). The ID Compiler is built into the "world" of FLA's Lisp Machines, so you need not do anything special before running it.

To compile a file, you give the **`Id Compile`** command to the command processor. (For more information about the Symbolics Command Processor, see the appropriate Symbolics documentation.) Here is the description of the **`Id Compile`** command, following the format of Symbolics' documentation.

Id Compile Command

`Id Compile` *source-file object-file listing-file keywords*

Compiles an ID source file (collection of ID procedure definitions).

source-file The name of the ID source file. The default extension is ID; the default host, directory, and name are taken from the usual file default.

object-file The name for the file that will receive the object code. The default host, directory, and name are taken from the corresponding field of *source-file*. The default extension depends on the type of object code produced; see the **`:Object Type`** keyword below.

listing-file The name for the file that will receive the compiler listing. The compiler listing includes the original ID source annotated with line numbers, any error messages produced, an indication of the correspondence between unique code block names produced by the compiler and the user's procedure names (or line numbers in the case of loop code blocks), and statistics related to the size of the program.

---

[2] If you need to use it on some other Lisp Machine, see Steve Heller or Ken Traub.

keywords can be:

:Object Type **{Numeric-Mc Verbose-Mc Lisp-Graph Verbose-Graph}**
The type of object code to be produced; the default is **Numeric-Mc**. This option also determines the default extension for *object-name*, as described below:

Numeric-Mc	Machine readable form of code for the Tagged Token Dataflow Architecture. This is the form understood by the simulator and emulator. Default *object-name* extension: NMC.
Verbose-Mc	Human readable form of code for the Tagged Token Dataflow Architecture. Default *object-name* extension: VMC.
Lisp-Graph	Lisp-style representation of the abstract U-interpreter graph; theoretically machine readable. Default *object-name* extension: LAG.
Verbose-Graph	Human readable representation of the abstract U-interpreter graph. Default *object-name* extension: VAG.

Examples of each of these types of output are given in Appendix I.

:Constant Area {yes no}
Whether to use constant areas in loops. The default is yes.

:New Symbols {yes no}
Whether to generate unique names for code blocks different from names generated by earlier compiles. The default is no, so that if you compile the same program twice the same unique names will be generated. Adding this keyword to your **Id Compile** command is the same as **:New Symbols yes**.

As an example, the command

```
Id Compile XX:<ARVIND>SIMPLE
```

will try to compile the file <ARVIND>SIMPLE.ID on XX into a numeric machine code file <ARVIND>SIMPLE.NMC and a listing file <ARVIND>SIMPLE.LISTING, both on XX.

One restriction on the use of the ID Compiler is that only one compilation per machine can

take place at once. This means, for example, that you can't start a compilation from Lisp Listener 1 and then start another in Lisp Listener 2 while the first is still in progress.

3. The Remote ID Compiler

The Remote ID Compiler (RIC) facility permits the ID Compiler to be used even when an FLA Lisp Machine console is not available. By using the network, a host can send a file to a Lisp Machine for compilation, and receive the object and listing files, as well as the compiler's informatory messages. Currently, RIC is only available on MIT-XX and other TOPS-20 machines, but there are plans to implement versions for Unices (*e.g.*, MIT-NEWTOWNE-VARIETY) and VM/370s (*e.g.*, MIT-BIG-BLUE).

3.1. Using RIC on MIT-XX

Before invoking the compiler, it is necessary to type the following command to the EXEC:

```
Declare PCL PS:<ID.SS.RIC>RIC.PCL
```

It is a good idea to put this command in your **LOGIN.CMD** file so that you won't have to worry about it. This command defines a new EXEC command, **ID-COMPILE**, which is similar to the Lisp Machine command described in Chapter 2. Cosmetic differences are necessary because of the two systems' different command processors.

The most general format of the **ID-COMPILE** command is:

ID-COMPILE *Source-file Object-file Listing-file Options*

The *Source-file*, *Object-file*, and *Listing-file* parameters have exactly the same meanings and defaults as the corresponding parameters for the Lisp Machine's **Id Compile** command. Of these three parameters, only *Source-file* need be specified. Compiler options, while usually appearing at the end of the command, may actually appear anywhere after the word **ID-COMPILE**. Each option consists of a slash followed by a keyword, and they have no associated value as they do on the Lisp Machine; an option takes effect by virtue of its presence in the command line. The following options may be specified:

/NUMERIC-MC Has the same meaning as **:Object Type Numeric-Mc** on the Lisp Machine.

/VERBOSE-MC Has the same meaning as **:Object Type Verbose-Mc** on the Lisp Machine.

/LISP-GRAPH Has the same meaning as **:Object Type Lisp-Graph** on the Lisp Machine.

/VERBOSE-GRAPH Has the same meaning as **:Object Type Verbose-Graph** on the Lisp Machine.

/NO-CONSTANT Has the same meaning as **:Constant Area No** on the Lisp Machine. Not giving this option has the same effect as **:Constant Area Yes** on the Lisp Machine.

/NEW-SYMBOLS Has the same meaning as **:New Symbols Yes** on the Lisp Machine. Not giving this option has the same effect as **:New Symbols No** on the Lisp Machine.

The options **/NUMERIC-MC**, **VERBOSE-MC**, **/LISP-GRAPH**, and **/VERBOSE-GRAPH** are mutually exclusive; only one of these options may be given. If none of these options are given, **/NUMERIC-MC** is assumed. As on the Lisp Machine, these four options affect the default extension for *Object-file*. As an example, the command

```
ID-COMPILE <SABROBST>EXPER1 /VERBOSE-GRAPH
```

will try to compile the file <SABROBST>EXPER1.ID into a verbose abstract graph file <SABROBST>EXPER1.VAG and a listing file <SABROBST>EXPER1.LISTING.

RIC works by trying to find an FLA Lisp Machine that is willing to accept a compilation request, and then establishing network connections for the source, object, and listing files to that machine. With the exception of LIVE-OAK, the FLA Lisp Machines will not accept a compilation request if the console has not been idle for at least one hour. This is done so that people doing work on Lisp Machines will not be slowed down by the compiler. Since LIVE-OAK is not really a user machine, it will accept a compilation request even if the console has not been idle. To avoid disturbing people working on OAK, RIC will only try to use OAK if all the other machines refuse.

3.2. A Note to Lisp Machine Users

If your console is idle more than an hour, your machine might be used by RIC to compile ID programs. It is a very good idea to make sure to save your work (editor buffers, etc.) if your console will be idle for more than an hour, in case the compiler does something bad to your machine (uses up the last cons cell, for example).

There are certain cases when you might want to prevent remote compilations from taking place on your machine even if the console is idle for a long time. An example might be if you start a long number-crunching program on your machine. The following variable will then be of interest:

id:sv*minimum-console-idle-time *Variable*

The Lisp Machine will only accept remote compilation requests if the console idle time is greater than the value of **id:sv*minimum-console-idle-time**, measured in 60ths of a second. The initial value of this variable is 216000 (1 hour).

By setting **id:sv*minimum-console-idle-time** to some large value (**(setq id:sv*minimum-console-idle-time (* 60. 60. 60. 24. 365.))** will set it to one year, for example) you can prevent remote compilations from disturbing your machine. You should not do this unless you have good reason.

4. ID Syntax

This chapter attempts to define those features of the ID language that the ID compiler compiles properly. Other features of the language are implemented by IDSYS, and even by the syntax checker of this compiler, but they are not given here.

Uppercase and lowercase letters are interchangeable in ID.

4.1. Basic Syntactic Units

4.1.1. Whitespace and Comments

Whitespace (blanks, tabs, and/or line breaks) is required to separate adjacent constants, identifiers, and keywords.

Comment:

! *any text not containing an exclamation point* !

4.1.2. Constants

There are two kinds of constants in ID, as described below.

4.1.2.1. Numeric Constants

All numeric constants in ID are base 10. A numeric constant is any sequence of digits, possibly including a decimal point, and possibly followed by an exponent. An exponent is indicated by the letter **E** followed by an optional minus sign (**-**) and then a sequence of digits.

Examples:

```
90      3.14159      .677  6.023E-23   10e80
```

4.1.2.2. Boolean Constants

The boolean constants **true** and **false** may be used.

4.1.3. Identifiers

Identifiers in ID are used to stand for expressions, procedure names, and formal parameters. An identifier is a sequence of letters, digits, and underscores, of any length, that begins with a letter or underscore. The following keywords may not be used as identifiers:

abs	**cos**	**float**	**max**	**return**
all	**decrement_rc**	**from**	**min**	**returns**
allocate	**do**	**if**	**new**	**select**
allocrec	**each**	**imports**	**not**	**sin**
and	**else**	**in**	**or**	**sqrt**
append	**endif**	**increment_rc**	**pass_left**	**then**
atan	**end_let**	**init**	**proc**	**to**
but	**false**	**initial**	**procedure**	**true**
by	**fi**	**lambda**	**remainder**	**until**
compose	**first**	**log**	**repeat**	**while**
cons	**fix**	**let**	**rest**	

Examples:

```
x     bottom_r    _size a37
```

4.2. Expressions

The types of expressions are presented here in a way that explicitly shows the precedence and associativity of operators.

In ID, an expression may denote several values. The *arity* of an expression is the number of values denoted by that expression, and may be any positive integer.

4.2.1. Primary Expressions

Primary expressions group left to right. Thus, **x[i][j]** is equivalent to **(x[i])[j]**.

Primary-expression:
- *Constant*
- *Identifier*
- **<>**
- **(***Expression***)**
- *Primary-Expression***[***Expression***]**
- *Builtin-Function-Name***(***Expression***)**
- *Identifier***(***Expression***)**
- *Identifier***(***Expression***) returns** *Constant*
- *Block-Expression*
- *Conditional-Expression*
- *Loop-Expression*

Constants and identifiers are of arity 1.

The primary expression **<>** denotes a new I-structure of size 100, and is of arity 1. This is an obsolete construct; **allocate(100)** should be used instead.

The primary expression **s[i]** denotes the **i**th element of the I-structure **s**. A multiple-expression as subscript is syntactic sugar for a series of individual subscripts; thus **s[i,j,k]** is equivalent to **s[i][j][k]** which is equivalent to **((s[i])[j])[k]**. The primary-expression **s** must evaluate to an I-structure; the selectors must evaluate to integers. The arity of such a primary expression is 1.

The primary expression **p(x) returns c** denotes the result of invoking procedure **p**, which returns **c** values, with argument(s) **x**. The form without the **returns c** portion may only be used if the procedure returns one value or if the name of the procedure appears in the import list of the procedure in which the call occurs (see Section 4.4). The arity of this primary expression is the number of values returned by the procedure.

The following built-in procedures exist, all of which are of arity 1:

Name	*Description*
abs	Absolute value. (1 argument)
allocate	Allocate a new I-structure of a given size. **allocate(3)** returns an I-structure of size 3, whose indices run from 0 through 2. (1 argument)
allocrec	Allocrec is exactly the same as allocate; it exists solely for the benefit of those doing type-checking experiments. (1 argument)
append	**Append(s,i_1,i_2,...,i_n,v)** is equivalent to **s + [i_1,i_2,...,i_n]v**. (3 or more arguments)
atan	Inverse tangent of a quotient. (2 arguments)
cos	Cosine. (1 argument)
decrement_rc	**Decrement_rc(s,i)** returns **s**, which should denote an I-structure, and adds **i** to its reference count. Note that **i** should be negative to actually decrement the reference count. Unlike **increment_rc**, **decrement_rc** returns **s** immediately, without waiting for its reference count to be adjusted. (2 arguments)
fix	Floating point to integer conversion. (1 argument)
float	Integer to floating point conversion. (1 argument)
increment_rc	**Increment_rc(s,i)** returns **s**, which should denote an I-structure, after adding **i** to its reference count. Unlike **decrement_rc**, **increment_rc** returns **s** only after the adjustment to the reference count has been acknowledged.

(2 arguments)

`log`	Natural logarithm. (1 argument)
`max`	Greater of two arguments. (2 arguments)
`min`	Lesser of two arguments. (2 arguments)
`pass_left`	`Pass_left(x,y)` returns `x` after both `x` and `y` arrive. This is useful for introducing additional dependencies into graphs. (2 arguments)
`remainder`	Remainder from dividing two numbers. (2 arguments)
`select`	`Select(s,`i_1`,`i_2`,...,`i_n`)` is equivalent to `s[`i_1`,`i_2`,...,`i_n`]`. (2 or more arguments)
`sin`	Sine. (1 argument)
`sqrt`	Square root. (1 argument)

4.2.2. Arithmetic Expressions

All arithmetic expressions are of arity 1.

4.2.2.1. Unary Expressions

Unary-expression:
Primary-expression
+Primary-expression
-Primary-expression

The primary expression must be of arity 1. The arity of a unary expression is 1.

4.2.2.2. Exponent Expressions

Exponent-expression:
Unary-expression
Exponent-expression ↑ *Unary-expression*

4.2.2.3. Multiplicative Expressions

Multiplicative-expression:
Exponent-expression
Multiplicative-expression * *Exponent-expression*
Multiplicative-expression / *Exponent-expression*

4.2.2.4. Additive Expression

Additive-expression:
Multiplicative-expression
Additive-expression + *Multiplicative-expression*
Additive-expression - *Multiplicative-expression*

4.2.2.5. Relational Expressions

Relational-expression:
Additive-expression
Relational-expression < *Additive-expression*
Relational-expression <= *Additive-expression*
Relational-expression > *Additive-expression*
Relational-expression >= *Additive-expression*
Relational-expression = *Additive-expression*
Relational-expression ~= *Additive-expression*

4.2.2.6. Not Expressions

Not-expression:
Relational-expression
not *Relational-expression*

4.2.2.7. And Expressions

And-expression:
Not-expression
And-expression **and** *Not-expression*

4.2.2.8. Or Expressions

Or-expression:
And-expression
Or-expression **or** *And-expression*

4.2.3. I-structure Expressions

All I-structure expressions are of arity 1.

4.2.3.1. Append Expressions

Append-expression:
Primary-expression
Append-expression + [*Expression*] *Primary-expression*

An append-expression that is just a primary-expression should be of arity 1 and denote an

I-structure. An expression like `s+[i]v` denotes the I-structure denoted by `s` with the value denoted by `v` stored in element `i`. The primary-expression `v` must be of arity 1. If `i` is of arity *n*, the meaning of the append-expression is *n - 1* selects followed by an append, consistent with the interpretation of higher-arity subscripts (see Section 4.2.1).

4.2.4. Expressions

Expression:
Or-expression
Append-expression
Primary-expression
Expression , *Or-expression*
Expression , *Append-expression*
Expression , *Primary-expression*

The arity of a group of primary expressions separated by commas is the sum of the arities of the individual primary expressions.

4.3. Compound Expressions

4.3.1. Conditional Expressions

Conditional-expression:
`(if` *Expression* `then` *Expression* `else` *Expression* `)` (This is the preferred syntax.)
`if` *Expression* `then` *Expression* `else` *Expression* `endif`
`if` *Expression* `then` *Expression* `else` *Expression* `fi`

The first expression must be of arity 1. The arity of the entire expression is the arity of the second and third expressions, which must be consistent.

4.3.2. Block Expressions

Block-expression:
`(let` *Block-binding-list* `in` *Expression* `)` (This is the preferred syntax.)
`let` *Block-binding-list* `in` *Expression* `end_let`

Block-binding-list:
Block-id-list `<-` *Expression*
Block-binding-list `;` *Block-id-list* `<-` *Expression*

Block-id-list:
Identifier
Block-id-list , *Identifier*

The number of identifiers in each block-id-list must be equal to the arity of the corresponding expression. The bindings define the values corresponding to the identifiers in the block-id-lists. These bindings hold not only for the expression following the keyword **in**, but also for the expressions in the binding list itself. Thus it is syntactically possible to express a circular binding, but the compiler will flag this as an error. The arity of a block expression is the arity of the expression following the keyword **in**.

4.3.3. Loop Expressions

Loop-expression:
(*Loop-iterator Loop-binding-list* **return** *Expression*)
(**initial** *Block-binding-list Loop-iterator Loop-binding-list* **return** *Expression*)

Loop-iterator:
while *Expression* **do**
for *Identifier* **from** *Expression* **to** *Expression* **do**
for *Identifier* **from** *Expression* **to** *Expression* **by** *Expression* **do**

Loop-binding-list:
Loop-id-list **<-** *Expression*
Loop-binding-list **;** *Loop-id-list* **<-** *Expression*

Loop-id-list:
Loop-id
Loop-id-list **,** *Loop-id*

Loop-id:
Identifier
new *Identifier*
Primary-expression[*Expression*]
new *Primary-expression*[*Expression*]

Identifiers used on the right hand sides of the bindings following the keyword **initial** always refer to identifiers defined outside of the Loop-expression. Note that this is different scoping than for the bindings of a Let-expressions, in which identifiers on the right hand sides refer to bindings defined *within* the Let-expression, if those identifiers are defined there. Naturally, identifiers appearing in the expressions of the Loop-iterator, in the right hand sides of the bindings in the body of the Loop-expression, and in the Expression following the keyword *return* refer to bindings made within the Loop-expression, if those bindings exist.

When there is no **new** keyword preceding an identifier, the binding establishes an intermediate value for use in the current iteration. When an identifier is preceded by the **new** keyword, the identifier is bound for use in the next iteration.

All expressions in the loop-iterator must be of arity 1. The number of loop-ids in each loop-id-list must be equal to the arity of the corresponding expression. The arity of a loop expression is the arity of the expression following the keyword **return**.

4.4. Procedure Definitions

Procedure-definition:
proc *Identifier* (*Formal-list*) *Procedure-body*
procedure *Identifier* (*Formal-list*) *Procedure-body*
proc *Identifier* (*Formal-list*) **imports** *Import-list Procedure-body*
procedure *Identifier* (*Formal-list*) **imports** *Import-list Procedure-body*

Procedure-body:
Primary-expression
Or-expression
Append-expression

Formal-list:
Identifier
Formal-list , *Identifier*

Import-list:
Identifier **returns** *Constant*
Import-list , *Identifier* **returns** *Constant*

The arity of the procedure body is the number of values returned by the procedure. The import-list may be used to indicate the number of values returned by procedures called within the expression. An invocation of a procedure whose name appears in the import-list need not be suffixed by **returns** *Constant*. The constants in the import list must be positive integers.

4.5. ID Source Files

An ID source file consists of one or more procedure definitions. Each procedure is compiled separately; the procedures are linked together by whatever program reads the object file.

5. Compiled Code Considerations

The following is a collection of useful facts about the code produced by the ID Compiler.

- The code generated for procedures and procedure invocations enforces the requirement that procedures be *strict* in all their arguments: execution of the procedure will not begin until all arguments have been received. Similarly, the returned results will not be made available to the calling procedure until all results are ready.

- When compiling a loop, the code for the loop body and return expression will be placed in a separate code block from the code block of the surrounding code. This loop code block will be invoked in much the same way as a procedure is invoked, except that strictness is not enforced as it is for procedures. The expressions in the **initial** part of the loop will be compiled into the code block *invoking* the loop. The exception to this are **initial** bindings which contain either <> or **allocate(***Expression***)**, which are compiled into the loop code block itself.

- In the body of a loop, **new s <- s + [i]v**, **s[i] <- v**, and **new s[i] <- v** are semantically equivalent, but there is a difference in code generation. In the first case, the I-structure descriptor for **s** will be circulated along an arc, while in the second and third cases the descriptor will be stored in Constant Area. If the **:Constant Area no** option is given to the compiler while compiling the second or third cases, they will be exactly the same as the first.

6. Bugs

Nobody claims that the ID compiler is entirely bug free. If you think the compiler is producing incorrect object code, bring the source file, object file, and what you think the object file should be to either Steve Heller or Ken Traub. If the compiler enters the Lisp Machine error handler while it is compiling, hit the ABORT key and send the source file to either Steve or Ken. Note that whenever you abort out of the compiler, it deletes the object and listing files it was producing.

Appendix I
Sample Output

To get a feel for what kind of output the compiler can produce, this appendix gives a short ID program, the listing file generated therefrom, and the four kinds of object files that can be obtained through various compiler options. This example is intended to show the different types of compiler output, not to illustrate programming in ID. See Appendix II for examples of ID programs.

Actual object files have several blank lines before and after the text of the file. These blank lines are not indicated here.

I.1. ID Source File

```
procedure test (x,y)
  x+y
```

I.2. Listing File

The abbreviation AG stands for Abstract Graph, and the abbreviation MC stands for Machine Code.

```
    1  procedure test (x,y)
    2    x+y

-----                    TEST  (P0003):       3 AG Ops;     19 MC Ops;     147 Bytes
-----
-----                   Total                 3 AG Ops;     19 MC Ops;     147 Bytes

----------------------- COMPILER STATISTICS -----------------------

 Total Procedures:      1
Total Code Blocks:      1
     Total AG Ops:      3
     Total MC Ops:     19
      Total Bytes:    147

                        | Procedure | Code Block |  AG Op  |  MC Op
------------------------+-----------+------------+---------+---------
      Average Bytes per |     147.0 |      147.0 |   49.00 |    7.74
     Average MC Ops per |      19.0 |       19.0 |    6.33 |
     Average AG Ops per |       3.0 |        3.0 |
Average Code Blocks per |      1.00 |
```

I.3. Numeric Machine Code (.NMC)

```
BEGIN

CODE-BLOCK EXTERNAL PROCEDURE TEST P0003 147 2 1 0 0 0 0 0
PARAMETERS 0 0 4 64 1 0 0 0
MANAGERS   *SUPER* 1 135
INTEGERS   8 128 2 113 1 106 0 99 -1 79 2 64 1 52 -1 34 0

122 73 241 192 0 26 1 0 10 0
114 65 0 113 5 0 106 5 0 99 6
114 65 0 52 4
114 65 0 34 1 0 41 4
74 73 97 0 0 41 6
114 65 0 79 1 0 64 1 0 52 6
125 25 97 0 0 94 6
 0 0 0 0 0
74 73 97 0 0 71 0
114 65 0 21 5 0 142 4
74 73 97 0 0 86 0
114 65 0 21 7 0 142 6
114 65 0 120 0
125 25 97 0 0 94 4
76 121 97 0 0 99 4
76 121 97 0 0 106 6
67 65 0 128 0
 0 0 0
121 133 97 0 0 135 0
120 132 196 0 0 0 0
1 97 0 113 6
END
```

I.4. Verbose Machine Code (.VMC)

```
BEGIN
Code for procedure TEST
     Unique name is P0003
     Expected number of arguments:  2
     Number of returned values:  1
     Maximum subdomain size:  4
     Maximum bytes per PE using maximum subdomain size:  64
```

Addr	Opcode	Disp T1	Disp T2	Const C	Const Source	Dest Flag	Dest/ConstSpec NT/ Length	Dest/ConstSpec PNum/ Class	Dest/ConstSpec Addr/ Value	Dest Flag
0	EXPAND	1	0	2	0	1	1	15	192	
							0	0	26	1
							0	0	10	0
10	IDENTITY	1	0	0	0	1	1	0	113	1
							1	0	106	1
							1	1	99	0
21	IDENTITY	1	0	0	0	1	1	0	52	0
26	IDENTITY	1	0	0	0	1	0	0	34	1
							1	0	41	0
34	FORM-ADDRESS-I-FETCH	1	0	2	0	1	1	6	0	
							1	1	41	0
41	IDENTITY	1	0	0	0	1	0	0	79	1
							0	0	64	1
							1	1	52	0
52	DECREMENT-RC	0	1	2	0	1	1	6	-1	
							1	1	94	0
64	FORM-ADDRESS-I-FETCH	1	0	2	0	1	1	6	1	
							0	0	71	0
71	IDENTITY	1	0	0	0	1	1	0	21	1
							1	0	142	0
79	FORM-ADDRESS-I-FETCH	1	0	2	0	1	1	6	2	
							0	0	86	0
86	IDENTITY	1	0	0	0	1	1	1	21	1
							1	1	142	0
94	IDENTITY	1	0	0	0	1	0	0	120	0
99	DECREMENT-RC	0	1	2	0	1	1	6	-1	
							1	0	94	0
106	FORM-ADDRESS-I-STORE	1	3	2	0	1	1	6	0	
							1	0	99	0
113	FORM-ADDRESS-I-STORE	1	3	2	0	1	1	6	1	
							1	1	106	0
120	EXTRACT-TAG	1	0	0	0	1	0	0	128	0
128	COMPRESS	2	0	1	0	1	1	6	2	

```
                                                    0       0       135     0
136     USE                     2   0   1   0       0       4       12      *SUPER*
142     +C                      1   2   0   0       1       1       1       113     0
END
```

I.5. Lisp-Style Abstract Graph (.LAG)

Note that the .LAG file in this example has been formatted for easy readability. The actual output from the compiler has no line breaks or extra whitespace.

```
((CONS-CODE-BLOCK (NAME = TEST )
                  (U-NAME = P0003 )
                  (TYPE = PROCEDURE )
                  (EXT-OR-INT = EXTERNAL )
                  (PREVIOUS = *SUPER* )
                  (EXTERNAL-PROC = NIL )
                  (ARGS-&-RET = (2 . 1) )
                  (ESSENTIALS = (*UNDEFINED*) )
                  (NO-OF-ESSENTIALS = (*UNDEFINED*) )
                  (CODE = 64 )
                  (END = 65 )
                  (A-L-LIST = NIL )
                  (CONSTANT-LIST = NIL )
                  (ESSENTIAL-LIST = NIL )
                  (UNCONNECTED = (*UNDEFINED*) )
                  (HEADER-OPS = (*UNDEFINED*) )
                  (MACRO-NODES = NIL ))
 (CONS-OPERATOR (INSTR-NO = 64 )
                (OPCODE = BEGIN )
                (CONSTANTS = NIL )
                (OUT-LIST =
                          (LIST (*EMPTY*)
                                (CONS-OUTPUT (NAME = X )
                                             (DEST-LIST =
                                                        (LIST
                                                         (*EMPTY*)
                                                         (CONS-DEST
                                                          (DESTINATION
                                                           =
                                                           66)
                                                          (PORT
                                                           =
                                                           1)))))
                                (CONS-OUTPUT (NAME = Y )
                                             (DEST-LIST =
                                                        (LIST
                                                         (*EMPTY*)
                                                         (CONS-DEST
                                                          (DESTINATION
                                                           =
                                                           66)
                                                          (PORT
                                                           =
                                                           2)))))
                                (CONS-OUTPUT (NAME =
                                                   ID*TRIGGER)
                                             (DEST-LIST =
                                                        (LIST
                                                         (*EMPTY*))))))
                (ALTERNATE-LIST = (LIST (*EMPTY*) ) )
                (IN-LIST = (LIST (*EMPTY*) ) )
                (MARK1 = T )
                (MARK2 = (*UNDEFINED*) ))
 (CONS-OPERATOR (INSTR-NO = 65 )
                (OPCODE = END )
                (CONSTANTS = NIL )
                (OUT-LIST = (LIST (*EMPTY*) ) )
```

```
                (ALTERNATE-LIST = (LIST (*EMPTY*) ) )
                (IN-LIST =
                        (LIST (*EMPTY*)
                              (CONS-INPUT (NAME =
                                           (*UNDEFINED*))
                                          (ORIGINATOR = 66 )
                                          (PORT = 1 )
                                          (WEIGHT =
                                                 (*UNDEFINED*)))))
                (MARK1 = T )
                (MARK2 = (*UNDEFINED*) ))
(CONS-OPERATOR  (INSTR-NO = 66 )
                (OPCODE = + )
                (CONSTANTS = NIL )
                (OUT-LIST =
                         (LIST (*EMPTY*)
                               (CONS-OUTPUT (NAME =
                                             (*UNDEFINED*))
                                            (DEST-LIST =
                                                   (LIST
                                                    (*EMPTY*)
                                                    (CONS-DEST
                                                     (DESTINATION
                                                      =
                                                      65)
                                                     (PORT
                                                      =
                                                      1)))))))
                (ALTERNATE-LIST = (LIST (*EMPTY*) ) )
                (IN-LIST =
                        (LIST (*EMPTY*)
                              (CONS-INPUT (NAME = X )
                                          (ORIGINATOR = 64 )
                                          (PORT = 1 )
                                          (WEIGHT =
                                                 (*UNDEFINED*)))
                              (CONS-INPUT (NAME = Y )
                                          (ORIGINATOR = 64 )
                                          (PORT = 2 )
                                          (WEIGHT =
                                                 (*UNDEFINED*)))))
                (MARK1 = T )
                (MARK2 = (*UNDEFINED*) )))
```

I.6. Verbose Abstract Graph (.VAG)

```
CODE FOR PROCEDURE TEST
     UNIQUE NAME IS P0003
     NUMBER OF EXPECTED ARGUMENTS =   2
     NUMBER OF RETURNED VALUES =   1

                                -- INSTRUCTIONS --
NO.  OUTPUT     OPCODE          DESTINATIONS       CONSTANT/              SOURCE
     NAME                                          INPUT NAME
-------------------------------------------------------------------------------
  1              BEGIN
     X                         TO         2 @1
     Y                         TO         2 @2

  2              +
                                                              X @1  FROM     1
                                                              Y @2  FROM     1
                               TO         3 @1

  3              END
                                                                @1  FROM     2
-------------------------------------------------------------------------------
```

Appendix II
Sample ID Programs

This appendix contains several examples of ID programs that demonstrate various features of the ID language. The preferred syntax is used, and we recommend that you adopt our indentation style.

II.1. Arithmetic Expression

Expr is an arithmetic expression.

```
!!!
Expr computes the discriminant of a quadratic polynomial whose
coefficients are a, b, and c.
!!!
procedure expr (a, b, c)
b*b - 4*a*c
```

II.2. Block Expression

Let_expr is a block expression. Note the scoping structure; x and y are defined and used in the same block binding list. If the order of the bindings were changed, there would be no semantic difference. Also, semicolons are used as binding separators, not terminators, so there is none after the last binding.

```
!!!
Let_expr shows how variables can be used to name intermediate
results.
!!!
procedure let_expr (a, b, c, d)
(let x <- a*b;
     y <- c*d;
     s <- x+y;
     t <- x-y
 in s/t)
```

II.3. If Expression

If_expr returns two values based on the input. Foo calls if_expr and indicates the arity inline; Bar calls if_expr and indicates the arity in the imports list.

```
!!!
If_expr computes both the larger and the absolute value of the
difference of its inputs.
!!!
procedure if_expr (a, b)
(if a>b
    then a, a-b
    else b, b-a)

procedure foo (a, b)
(let x, y <- if_expr (a, b) returns 2
 in 2+x, 3+y)

procedure bar (a, b)
imports if_expr returns 2
(let x, y <- if_expr (a, b)
 in 2+x, 3+y)
```

II.4. For Loop

For_loop demonstrates a for loop. The variable sum_increment does not circulate, and hence has no new keyword. The variable sum does circulate and therefore uses the new keyword to specify the value for the next iteration.

```
!!!
For_loop computes the nth triangular number (the sum of the
integers from 1 to n).
!!!
procedure for_loop (n)
(initial sum <- 0
 for i from 1 to n do
     sum_increment <- sum + i;
     new sum <- sum_increment
 return sum)
```

II.5. While Loop

While_loop uses a while loop to compute square roots using Newton's method. Note the usage of new_sqrt. new_sqrt gives us a handle on the newly computed sqrt value which is normally referred to as sqrt in the next iteration. A similar technique can be used to save old values from a fixed number of previous iterations (*e.g.*, to code a higher order recurrence relation). It was necessary to bind sqrt as float(x) rather than x in case the user provided an integer as input. Then epsilon would always be a non-zero integer due to round-off error (unless the input were a perfect square), and the loop would never terminate.

```
!!!
While_loop computes the square root of its input using Newton's
method.
!!!
procedure while_loop (x)
(initial sqrt <- float(x);
         epsilon <- 1
 while epsilon > .001 do
     new_sqrt <- ((x/sqrt)+sqrt)/2;
     new epsilon <- abs (x - new_sqrt*new_sqrt);
     new sqrt <- new_sqrt
 return sqrt)
```

II.6. Simple Use of Arrays

Conz demonstrates the use of arrays.

```
!!!
Conz uses a structure to perform cons.  "Cons" is not used as it
is a keyword.
!!!
procedure conz (a, b)
allocate (2) + [0]a + [1]b
```

II.7. Array Calculation

Array_expr demonstrates an array based calculation. Note the zero based indexing and allocation. By the way, all the calls to f can proceed in parallel.

```
!!!
Array_expr fills an array of size n by calling the function f
for each position.  Then the array is consumed by adding together
the elements.  The producer and the consumer of the array should
be able to proceed in parallel.  Since f is the square procedure,
array_expr computes the sum of the first n squares starting
with 0.
!!!
procedure array_expr (n)
(let x <- (initial x <- allocate(n)
           for i from 0 to n-1 do
               x[i] <- f (i)
               ! fill the array by calling f for each position !
           return x);
     sum <- (initial sum <- 0
             for i from 0 to n-1 do
                 new sum <- sum + x[i]
                 ! consume each position in the sum !
             return sum)
in sum)

procedure f (i)
     i * i
```

II.8. Complex Use of Arrays

Wave demonstrates the use of an array of arrays to encode a two dimensional structure.

```
!!!
Wave allows matrix computation to proceed in a wave fashion.
First the top row is loaded up with integers corresponding
to their positions.  Then the rest of the rows are defined
by initializing the leftmost element to the index of the
row and letting A[i,j] <- 1+min(A[i-1,j], A[i,j-1]).
The wave proceeds from upper left to lower right in a
diagonal fashion.  The values in the matrix correspond
roughly to the order in which computation proceeds.  The
matrix produced for a call with high=5 looks like:

        0 1 2 3 4
        1 2 3 4 5
        2 3 4 5 6
        3 4 5 6 7
        4 5 6 7 8

Wave returns the element stored in the highest position, which
will turn out to be 2(n-1).
!!!
procedure  Wave (high)
(initial ocean <- allocate (high)
                  + [0](initial row <- allocate (high)
                        for i from 0 to high-1 do
                            row[i] <- i
                        return row)
 for i from 1 to high-1 do
     ocean [i] <- (initial  row <- allocate (high) + [0]i
                   for j from 1 to high-1 do
                       row[j] <- 1 + min (ocean[i-1,j], row[j-1])
                   return row)
 return ocean)[high-1,high-1]
```

II.9. Simple Recursive Program

Recursive is a recursive program that unfolds into a balanced binary recursion.

```
!!!
Recursive divides its input as evenly as possible until each call
receives an input of 1.  A balanced binary invocation tree with n
leaves and hence 2n-1 total invocations is generated.
!!!
procedure recursive (n)
(if n = 1
    then 1
    else (let fixn <- fix(n/2)
          in recursive (fixn) + recursive (n - fixn)
          )
    )
```

II.10. Higher Order Recurrence

Recur is a 4th order recurrence relation. The variables bit0, bit1, bit2, bit3, and bit4 form a software history of past computation.

```
!!!
Recur returns an array containing the first n pseudo random
numbers of a particular sequence.  Numbers are generated by
shifting a five bit binary register to the right, and shifting
in the exclusive nor of two of the bit positions.
!!!
procedure recur (n)
(initial sequence <- allocate (n);
         bit0 <- 0;
         bit1 <- 0;
         bit2 <- 0;
         bit3 <- 0;
         bit4 <- 0
 for i from 0 to n-1 do
     sequence[i] <- 16*bit4 + 8*bit3 + 4*bit2 + 2*bit1 + bit0;
     new bit0 <- bit1;
     new bit1 <- bit2;
     new bit2 <- bit3;
     new bit3 <- bit4;
     new bit4 <- (if bit2=bit0 then 1 else 0)
 return sequence)
```

Table of Contents

List of Figures