Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 254 September, 1983

# CL1 Manual

Alan Bawden

ABSTRACT

CL1 is a prototype language for programming a Connection Machine. It supports a model of the Connection Machine as a collection of tiny conventional machines (processing elements), each with its own independent program counter.

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be papers to which reference may be made in the literature.

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1983

# 1. Introduction

CL1 is a prototype language for programming a Connection Machine [3, 2]. It supports a model of the Connection Machine as a collection of tiny conventional machines (processing elements), each with its own independent program counter. Processing elements communicate with each other using I/O-like primitives.

There is implementation and simulation of CL1 in MacLisp [4]. This manual documents that implementation. This manual is primarily aimed at people already familiar with the general form of the Connection Machine architecture who are interested in experimenting with programming it in a higher-level language.

This document only aspires to contain the basic information necessary to start using CL1 . It is not an introduction to the Connection Machine (see [2] for that). It is not an examination of the pros and cons of programming in CL1, nor does it contain any proposals for replacements for CL1 (papers are in preparation on those topics). It contains only occasional justifications for design decisions made in the course of implementing CL1.

The acronym "CL1" stands for "Cell Language 1": "cell" because programs are written from the point of view of a single processing element or "cell", and "1" to remind us that CL1 is only a first attempt at such a language. CL1 will probably never actually run on a Connection Machine. What we are learning by experimenting with it will shape successors better suited to that task.

CL1's model of the Connection Machine's processing elements as running with independent program counters is both a strength and a weakness. On the one hand, it frees the programmer from the duty of managing the possible states of the processing elements himself. On the other hand, the current implementation denies the programmer any control over when processing elements in particular states are run and advanced to new states. The CL1 programmer is unable to exploit the SIMD[1] features of the current Connection Machine design; instead, he must implement explicit protocols to govern execution order. Many known Connection Machine algorithms use these features (see [1]), and are thus inexpressible in CL1.

---

1. SIMD is an acronym for "Single Instruction, Multiple Data". It describes a class of shared instruction stream multiprocessor architectures of which the prototype Connection Machine is a member.

In the long run, attempts to exploit the features of a SIMD Connection Machine may prove to have been wasted effort. Initial designs for the Connection Machine called for a MIMD[1] architecture. If we ever return to that kind of architecture (in some future machine -- certainly there is no chance that the current hardware design will change that drastically), then those inexpressible algorithms will also become almost unusable.

CL1's design is based on the observation that even when programmers write Connection Machine programs directly in microcode, they still mostly think about the actions of the individual processing elements as if they were little conventional machines. Based on this observation, applicable parts of conventional compiler technology have been utilized to implement a language for programming the Connection Machine.

A typical Connection Machine microcode program alternates between selecting a set of processing elements to operate upon and broadcasting a stream of instructions to be executed by all selected processing elements. Register-machine-like actions are performed by those parts of the program that concern themselves only with the behavior of processing elements after the selection process has taken place: data is moved from place to place within a processing element and arithmetic and logical operations are performed upon it. The fact that these manipulations are being performed simultaneously by (potentially) thousands of little processors is irrelevant in the writing of such code.

Many microcoded Connection Machine programs also treat message transmission as an I/O operation. When the currently selected processing elements wish to transmit a message, the message sending/receiving microcode is broadcast and executed by *all* processing elements. This microcode might be thought of as part of the "operating system"; it simply transfers some data from a transmitting processing element to an input buffer in a receiving processing element. The use of a buffer isolates the local chain of events in the transmitting processing element from that in the receiving processing element.

The evaluation of many expressions in a conventional programming language can be compiled into "straight-line" code that only makes use of the fixed storage contained in the register file of a conventional machine. Since a single Connection Machine processing element contains about the same amount of storage as a conventional register file, such code can be executed on a single Connection Machine processing element. Thus we might try programming individual Connection Machine processing elements in such a language. Furthermore, if we are viewing message transmission and reception as I/O-like operations, we can support Connection Machine message passing by simply adding primitive input and

1. "Multiple Instruction, Multiple Data"

output operators to our language.[1] These are the basic characteristics of CL1.

It should be emphasized that CL1 is *not* the ultimate Connection Machine programming tool. We are using it for experimentation and as a convenient basis for extension languages that support higher-level models of the Connection Machine. It still requires the programmer to think about the actions of individual processing elements and about the transmission of each message and all the synchronization problems that that entails. Better ways of thinking about, and expressing, Connection Machine programs will certainly be found. In many ways, CL1 was just an obvious thing to try, but it has been very educational for us to actually work out all of the details.

# 2. General Information

The MacLisp implementation of CL1 can be accessed by loading the file PS:<ALAN.CM>STATES.FASL on the OZ machine. This file also contains the runtime support for the MacLisp CL1 simulation. Bug reports should be sent to BUG-CL1. (Keep in mind when reporting bugs that CL1 is an experiment, not a supported production program.)

CL1 is a compiled language; CL1 code is translated into MacLisp and then executed. There is currently no interpreter. The CL1 compiler runs whenever a DEFSTATE macro (see page 11) is being expanded. The CL1 compiler can be loaded into the MacLisp compiler so that CL1 code can be compiled first into MacLisp, and then into machine code.

The form

```
(EVAL-WHEN (EVAL COMPILE LOAD)
  (OR (GET 'DEFSTATE 'MACRO)
      (LOAD '((ALAN/.CM) STATES FASL))))
```

should be placed at the head of any file containing CL1 programs, to assure that both compiler and runtime support are present whenever the file is being processed in any way. The file may then be loaded into MacLisp, or compiled by the MacLisp compiler and the resulting FASL file loaded into MacLisp. As with ordinary MacLisp programs, CL1 programs should *not* be compiled by the MacLisp compiler until they are completely

---

1. An alternative might be to try viewing the rest of the machine as memory rather than as other processors. Unfortunately, it doesn't appear as a particularly well-behaved memory, since it can only be reliably written, not read.

debugged. The type checking performed by the uncompiled MacLisp output by the CL1 compiler is part of CL1's type checking mechanisms.

CL1 uses the MacLisp `HUNK` datatype to represent Connection Machine processing elements and other runtime datastructures. The default printed representation of hunks is somewhat confusing, especially when hunks are being used as structures. A better printed representation can be had by using the Dprint package. After loading CL1, the form `(DPRINT T)` will enable Dprint in your MacLisp. The Dprint package is documented in `PS:<LIBLSP>DPRINT.ALAN` (it is a general facility for getting Lisp Machine-like "named structures" [8]). Dprint is not automatically enabled because it cannot be used simultaneously with MacLisp's "smart strings", class system, or `DEFVST`-defined structures. If you don't know what those things are, then you aren't using them and Dprint is perfectly safe to enable.

The functions `INSPECT` and `DESCRIBE` (also part of the Dprint package and modeled after the Lisp Machine features of the same name) will attempt to display CL1's datastructures intelligently. To a large part they play the role of a debugger in the CL1 simulated runtime environment.

Someday CL1 may run on a Lisp Machine.

# 3. The CL1 Language

The CL1 language is a dialect of Lisp. It is fully tail-recursive like Scheme [6]. It is lexically scoped like Scheme and Common Lisp [7]. It stores functions as the values of variables as in Scheme. The names it gives its special forms are drawn mostly from Common Lisp. It inherits many of its primitive functions directly from MacLisp [4].

CL1 programs should be thought of as being executed by *single* Connection Machine processing elements. This limits the kind of programs that can be executed. Since each processing element has a fixed number of bits of state, programs that require unbounded state will not be executable.[1] CL1 disallows two features which in more powerful Lisps allow programs with unbounded state to be written: recursion other than tail-recursion,

---

1. In fact, the CL1 compiler may sometimes allocate more than a single processing element in order to have sufficient storage for state variables. But this is always a compile-time decision, and still results in a fixed-size virtual processing element.

and runtime closures. (These are really two views of the same feature.) Truncated in this manner, CL1 makes it impossible to write programs that require more than a fixed amount of state (determinable at compile-time). CL1 is basically a language for writing finite state machines.

To compensate for these restrictions the CL1 compiler is extremely good at optimizing CL1 programs into tail-recursive code. Additionally, inline-codeable functions are implemented correctly in CL1 (see page 10).

The values manipulated by a CL1 program in the MacLisp simulation are simply Lisp objects in the MacLisp world. The programmer is on his honor not to use these objects in ways that are clearly impossible on an actual Connection Machine. Integers, pointers (a datatype provided by the simulator), and symbols are clearly all implementable on a Connection Machine. Someday a typing scheme may be added to CL1 to facilitate the manipulation of aggregate datastructures, but for the moment, forming short fixed-length lists of objects will serve.

In the CL1 model of the Connection Machine, every allocated processing element is in some "state". Each state has associated with it a number of "state variables". The values of these variables at any processing element in that state are stored in bits allocated from that processing element's local storage. Also associated with each state is a straight-line (no branches) piece of code that any processing element in that state will run. That code performs computations with the processing element's state variables, computes a new state for it, and provides values for that new state's state variables.

It is the CL1 compiler's job to hide this state-machine model of program execution from the user wherever possible. However, sometimes it is necessary for the user to explicitly talk about a state with its associated variables and code -- for example when the user is initializing a newly allocated processing element. This is usually done using DEFSTATE, a *MacLisp* toplevel special form.

Here is an example of the use of DEFSTATE:

```
(DEFSTATE FACTORIAL-STATE (NUMBER)
  (LABELS ((FACT
             (LAMBDA (ACCUMULATOR COUNTER)
               (IF (< COUNT 2)
                   ACCUMULATOR
                   (FACT (* COUNTER ACCUMULATOR)
                         (1- COUNTER))))))
    (GO NEXT-STATE (FACT 1 NUMBER) NUMBER)))
```

This defines a state named FACTORIAL-STATE. It has a single state variable named NUMBER.

If a processing element is placed in state FACTORIAL-STATE and its first and only state variable is initialized to contain some integer, then that processing element will proceed to compute the factorial of that number. When it is finished it will set the state of the processing element to the state NEXT-STATE, and initialize its first state variable to contain the newly-computed factorial and its second state variable to contain the original number.

Several things to note about this example:

Despite its singular name, a typical DEFSTATE implies the existence of more than one state. In the example above three additional states are used to implement the iteration.[1] Calling it "DEFSTATE" rather than "DEFSTATES" is justified because the single starting state of the state-machine is the state being named.

The body of a DEFSTATE consists of some CL1 forms. In the example above the body was a single LABELS form that performed an iteration and then transferred to another state. Notice that the LABELS form did not ever return a value. If the final form in the body of a DEFSTATE returns, then that processing element simply drops back into the pool of available processing elements. (Actually, in the current CL1 simulation this is not strictly true; see the MacLisp variable *CELLS* documented on page 10.)

In the example above, LABELS, GO, LAMBDA and IF all introduce special forms in the CL1 language. All other symbols (FACT, COUNTER, *, NEXT-STATE, ...) are simply variables. The programmer's ability to take advantage of this fact by, for example, sending the value of the variable FACT as a message to another processing element, is limited by the restriction about the creation of runtime closures. Only closures with empty environments can be instantiated at runtime; in the example, the value of FACT *could* have be used at runtime.

The DEFSTATE form also defines FACTORIAL-STATE as a CL1 global variable and initializes its value when loaded into the simulator.

It cannot be overemphasized that a DEFSTATE form is a *MacLisp* special form. It is easy to become confused about when one is writing MacLisp forms, and when one is writing CL1 forms. Careful attention to the locations of these boundaries between CL1 and MacLisp will save the CL1 programmer much pain.

Besides giving a named state as the first "argument" to the GO special form, it is also useful

---

1. That is, the current CL1 compiler uses three. A clever human would only require one additional state!

to be able to name states when allocating new processing elements. For example, to allocate a new processing element and initialize it to the state FACTORIAL-STATE one would use a CL1 form like (MAKE-CELL FACTORIAL-STATE 259). (The MAKE-CELL function is documented on page 14. Its most interesting property is that it does not return the newly created processing element as a value as you might expect.)

CL1 does not support the SETQ special form. This is not due to any religious conviction about the evils of SETQ, but because the lack of SETQ leads to a particularly elegant and easily implemented compiler. It is almost always possible to avoid using SETQ, so the language isn't any less powerful. Additionally, this omission guarantees that the only "side effects" are those resulting from interprocessor communications. It's also an interesting exercise to learn how to program without SETQ; everyone should have the experience.

CL1 will treat the name of any MacLisp function it finds as the name of a primitive CL1 function; that is, it will compile code to simply call the MacLisp function from CL1. In the example, the functions <, *, and 1- are all inherited from MacLisp through this mechanism. This applies to MacLisp functions defined by the user as well as system functions, so the user can define new CL1 primitive functions simply by using MacLisp DEFUN. See also the (MacLisp) functions PROCLAIM-PRIMITIVE and PROCLAIM-NON-PRIMITIVE documented on page 15. Note that this feature increases the potential for confusing CL1 with MacLisp.

The CL1 compiler must make some assumptions about variables it hasn't previously seen defined or declared. It must choose between the possibilities that the variable will later be defined as a primitive (by DEFUN) or as a state (by DEFSTATE). In the former case, CL1 will generate a call to the MacLisp primitive of the same name; in the latter case, CL1 generates code to "go" to that state. As already mentioned, if the variable is defined or declared as a MacLisp function, CL1 declares it to be a primitive. In all other cases CL1 decides it must be the name of a user-defined state. This rule allows you to refer to states not yet defined without having to declare them.

Suppose you misspell the MacLisp function PLUSP as "PLUS?". CL1 looks at the symbol PLUS?, discovers it isn't the name of a MacLisp function, and therefore decides it must be the name of a state. You probably wrote (PLUS? N) rather than (GO PLUS? ...), so it is not quite clear what CL1 will try to do. Function calling is just a goto with a continuation [5], so in CL1 (PLUS? N) is exactly the same as (CATCH C (GO PLUS? C N)).[1] Unless you were

---

1. The fact that (F X) = (CATCH C (GO F C X)) is complemented by the fact that (LAMBDA (X) ...) = (KAPPA (C X) (THROW C ...)). (The KAPPA special form is documented on page 24.)

calling PLUS? tail-recursively, CL1 will probably have to construct a runtime continuation to compile the resulting code. Since this is illegal, an error message will result, reminding the user that closures and recursion are not supported.[1]

# 4. Sending Messages

Given only a few hundred bits, there is not much that a single processing element can compute. To do anything interesting, a Connection Machine program must use the machine's ability to transmit messages between processing elements. Since CL1 treats the individual processing elements of a Connection Machine as tiny conventional machines, it seemed natural to treat the sending and receiving of messages by those processing elements as stream I/O operations.[2]

The TRANSMIT function is the basic message-sending function. It takes two arguments: a pointer to another processing element and a message to be transmitted to that other processing element.

That much is simple, but doesn't indicate where a pointer comes from in the first place. (As previously mentioned, the MAKE-CELL function is *not* the way to create one.) A complication is that in the Connection Machine, a pointer points not just to a processing element, but to a specific location within that processing element. Since *any* location in a processing element might be pointed to, some mechanism must allocate locations within processing elements whenever a pointer is created. This is the job of the CL1 special form WITH-CHANNEL:

```
(WITH-CHANNEL (CHANNEL POINTER)
  (MAKE-CELL NEW-CELL-STATE POINTER)
  (INPUT CHANNEL))
```

The WITH-CHANNEL special form allocates a location within the executing processing element and creates a pointer pointing to that location. In this example the variable POINTER will be bound to that new pointer. The location will be deallocated when the body of the WITH-CHANNEL form is finished executing. Ideally, this deallocation should not take

---

1. This message is not the best way to tell the user that he has misspelled a variable name.
2. Well, actually it didn't seem all that natural; perhaps it would be more honest to say that I couldn't think of anything better at the time, so I implemented a relatively familiar mechanism.

place until after all other processing elements have released their copies of the pointer, but in general this seems hard (if not impossible) to guarantee.

The variable CHANNEL will be bound to a "channel object". In general, a channel object is only good for passing to the family of functions whose names begin with the word "INPUT". (Attempting to do anything else with a channel object, like transmit it to another processing element, will probably result in a friendly reminder from the CL1 compiler about the impossibility of runtime closures.) A channel object is the "other end" of the corresponding pointer created by the same WITH-CHANNEL form. Any message transmitted to this processing element using that pointer will be readable by doing input from the channel.

In the example code fragment a channel-pointer pair is created, then the pointer is made to be part of the initial state of a newly allocated processing element in state NEW-CELL-STATE. The creating processing element immediately tries to read a message from the associated channel. This is the method one would use to get a pointer to a new processing element. The state NEW-CELL-STATE would have to cooperate by having a definition that starts like

```
(DEFSTATE NEW-CELL-STATE (CREATOR)
  (WITH-CHANNEL (CHANNEL POINTER)
    (TRANSMIT CREATOR POINTER)
    ...))
```

This immediately allocates a channel and pointer and sends the pointer back to its creator. Now the creator can transmit messages through that pointer and the new processing element can read them from its channel.

Naturally, the issue of synchronization arises next. Since CL1 guarantees nothing about when a particular state will be scheduled to run, some processing elements will try to input messages from channels nobody has transmitted to yet, and others will try transmitting messages through pointers that already have unread messages buffered in them. What will happen? The answer depends on exactly which I/O functions you call.

The INPUT function causes a processing element that calls it to simply wait until input is available. Other functions whose names begin with the word "INPUT" perform related functions. (See the complete list starting on page 23.)

The TRANSMIT function signals an error if the receiving processing element already has an unread message in its buffer. While it will probably be cheap to implement on an actual Connection Machine, this function turns out to be inappropriate most of the time. More commonly useful is the function TRANSMIT-WAIT, which causes its caller to wait until the buffer it is trying to write into is empty.

# 5. MacLisp Interface

When you load CL1 into your MacLisp, it defines many MacLisp functions, variables, and special forms. This section documents those intended for users to use. It is in alphabetical order.

Again, remember the potential confusion between MacLisp code and CL1 code. *Everything* in this chapter is for use from MacLisp. Section 6 documents functions and variables for use from CL1.

***CELLS*** *Variable*

The value of *CELLS* is a list of every processing element ever allocated. Mostly it's for debugging convenience. If you are consing a great many processing elements, you might want to set it to NIL occasionally so that the garbage collector can reuse that storage. (Setting this variable to NIL does not make processing elements "go away"; for that, see the FLUSH-CELLS MacLisp function documented on page 13.)

**DEFINLINE** *Special Form*

Use DEFINLINE to define an inline-codeable function. Example:

```
(DEFINLINE FIRST (X)
  (CAR X))
```

This defines FIRST to have the usual meaning. Additionally, any occurrence of FIRST will compile *exactly* like an occurrence of CAR. Many common small functions that are traditionally written as macros in other Lisp dialects, are best written using DEFINLINE. Note that inline-codeable functions, like macros, must be defined before they can be used.

Inline codeable functions behave correctly with respect to lexical scoping rules: variables in the body of of a DEFINLINE will be looked up in the global environment. See also the CL1 special form LET-INLINE (page 25).

Recursive inline function definitions like

```
(DEFINLINE FACT (N)        ;This will not work.
  (IF (< N 2)
      1
      (* N (FACT (1- N)))))
```

will generate stack overflow errors inside the CL1 compiler.

CL1 doesn't have regular (non-inline) functions since processing elements are too small to have stacks.[1] Inline functions have been implemented in CL1 to help compensate for this lack.

DEFINLINE is one of the boundaries between the world of MacLisp forms and the world of CL1 forms: DEFINLINE itself is a MacLisp special form, but its body consists of CL1 forms.

**DEFSTATE** *Special Form*

(DEFSTATE *name* (*var1 var2 ... varN*)
  . *body*)

The DEFSTATE macro is the user's access to the CL1 compiler. The CL1 compiler runs when the DEFSTATE macro is being *expanded*; later, at load time, the results of this expansion will define the various states necessary for the simulator to implement the CL1 code in *body*. *name* will be defined as a CL1 global variable whose value is the starting state. That state will have *N* state variables, referred to as *var1* through *varN* within *body*.

Section 3 explains DEFSTATE in more detail. DEFSTATE is the most common boundary between MacLisp and CL1.

**DEFXMACRO** *Special Form*

Naturally CL1 has macros too:

```
(DEFXMACRO CONSQ (X Y)
  `(CONS ',X Y))
```

The body of a DEFXMACRO is a piece of MacLisp code that constructs and returns a CL1 form. The pattern part of a DEFXMACRO form can contain the keywords &OPTIONAL, &REST,

---

1. You can of course adopt a convention for allocating new processing elements to function as stack frames.

and &AUX, just as in the standard Lisp DEFMACRO.

DEFXMACRO has the greatest potential for confusion of any point on the boundary between MacLisp and CL1.

**DESCRIBE** *object* *Function*

Part of the Dprint package, DESCRIBE will try to describe any Lisp object in an informative way. In particular, when applied to a pointer to a processing element, or to a processing element itself, it will tell you the state of that processing element, what state variables that state has, and what their values are in the processing element.

**DPRINT** *flag* *Function*

Part of the Dprint package. (DPRINT T) will rationalize the way all of CL1's datastructures print in your MacLisp. (DPRINT NIL) will turn this off. See the complete documentation in PS:<LIBLSP>DPRINT.ALAN on OZ.

If Dprint is enabled, structures will print as follows:

States will print as

```
#<STATE FOO>
```

where "FOO" is the name of the state. Internal states (unnamed by the user, usually generated by the CL1 compiler while compiling the body of a DEFSTATE) are given generated names ending in "+" followed by a unique number. An internal state might look like

```
#<STATE FOO+17>
```

Processing elements will print as

```
#<CELL 662330 @FOO+17>
```

The octal number is simply for distinguishing among processing elements; the CL1 function CALL-ME can be used by a processing element to give itself a better name. Such a named processing element might print as

```
#<CELL ISHMAEL INPUT-HANG@BAR+3>
```

Following the atsign is the name of the processing element's current state. If a processing

element is waiting for some external event to happen before it can run, information to that effect is displayed before the atsign.

Pointers to processing elements print as

```
#<POINTER 7 FOO+17>
```

The number identifies the location of the input buffer of the processing element pointed at. "FOO+17" is the state occupied by that processing element. If a pointer points at a processing element that have given itself a name, then that name will be used instead of the processing element's state, for example:

```
#<POINTER 13 ISHMAEL>
```

**FLUSH-CELLS** *Function*

Reset the CL1 runtime so that it is no longer aware of any previously allocated processing elements. This is like pushing the boot button on the Connection Machine; it does *not* flush any previously defined states, primitives, special forms, or other declarative knowledge.

**GET-GLOBAL-VALUE** *symbol* *Function*

Used for accessing CL1 global variables from MacLisp. (GET-GLOBAL-VALUE 'X) will return the value of the CL1 variable X. See also the the MacLisp function SET-GLOBAL-VALUE (page 18).

**INSPECT** *object* *Function*

Part of the Dprint package, INSPECT displays objects in essentially the same format as DESCRIBE, with a command loop enabling the user to walk through his data structures. Type "?" for self documentation.

**MAKE-CELL** *Special Form*

This *MacLisp* special form allows the user to construct Connection Machine processing elements from within MacLisp code. (Something like this is necessary because you have to allocate your first processing element before a single line of CL1 code can be executed!) An instance of the MacLisp MAKE-CELL special form looks very much like a typical call to the CL1 function of the same name:

```
(MAKE-CELL FACTORIAL-STATE 259)
```

Of course FACTORIAL-STATE is a CL1 variable, not a MacLisp variable, so it wouldn't do to simply MacLisp evaluate it. This is why MAKE-CELL in MacLisp is a *special form.* The first "operand" to MacLisp MAKE-CELL must be a symbol that has previously been given a global CL1 value by DEFSTATE. The rest of its operands are MacLisp forms whose values will be used as the initial values of the state variables.

Like the CL1 function MAKE-CELL, the MacLisp MAKE-CELL does *not* return a useful value. There is no way to make a pointer except from CL1 code. Having a processing element running in the following loop has been found to be useful:

```
(DEFSTATE EAR ()
  (WITH-CHANNEL (CHANNEL POINTER)
    (SET 'EAR POINTER)  ;This is the MacLisp SET function!
    (ITERATE LOOP ()
      (SET 'VALUE (INPUT CHANNEL))
      (LOOP))))
```

Allocating a processing element in this EAR state, and letting it run, will cause it to create a pointer and make it the value of the *MacLisp* global variable EAR. Later, messages transmitted through that pointer will dutifully be made the value of the MacLisp global variable VALUE. This trick has proven to be a useful debugging aid, although it is completely unimplementable on a Connection Machine.

**PROCLAIM-CONSTANT** *symbol value* *Function*

The family of MacLisp functions whose name begin with "PROCLAIM" are used to manipulate the CL1 compiler's knowledge about various global variables. A typical use of PROCLAIM-CONSTANT would be to place the form

```
(EVAL-WHEN (EVAL COMPILE LOAD)
  (PROCLAIM-CONSTANT 'MESSAGE-SIZE 32.))
```

at top level in a file of CL1 source code. This proclaims to the CL1 compiler that the value of the variable MESSAGE-SIZE is permanently set to be 32. The compiler is free to build

knowledge of this fact into the generated code. If you change the value later, you will have to recompile any code which used the variable.

**PROCLAIM-GLOBAL** *symbol* *Function*

This function proclaims that its argument is an ordinary global variable. The compiler normally assumes this about variables it comes across while compiling CL1 code, so normally this function is unnecessary. However, it may be necessary to call this function to undo the effects of a PROCLAIM-CONSTANT, PROCLAIM-INLINE, or DEFINLINE.

**PROCLAIM-INLINE** *symbol form* *Function*

```
(EVAL-WHEN (EVAL COMPILE LOAD)
  (PROCLAIM-INLINE 'FIRST '(LAMBDA (X) (CAR X))))
```

is entirely equivalent to

```
(DEFINLINE FIRST (X) (CAR X))
```

although the latter is clearly preferable.

PROCLAIM-INLINE is documented because in actuality, any form can be used as the definition of an inline, not just a LAMBDA-expression. This results in a feature similar to the "atomic macros" of some Lisp dialects.

**PROCLAIM-PRIMITIVE** *symbol nargs* *Function*
**PROCLAIM-NON-PRIMITIVE** *symbol nargs* *Function*

CL1's heuristics for determining when a variable represents a primitive function inherited from MacLisp, and when it represents a vanilla CL1 variable are not fool-proof. Occasionally it may be necessary to correct them. This pair of functions can be used to do that.

PROCLAIM-PRIMITIVE proclaims to the CL1 compiler that *symbol* is the name of a MacLisp function that is to be called from CL1 code. PROCLAIM-NON-PRIMITIVE tells the CL1 compiler that *symbol* should *not* be treated in this way. Both these functions call PROCLAIM-GLOBAL on *symbol* first.

*nargs* should be a number or T or NIL. It is used to adjust CL1's idea of the number of arguments expected by the function the variable represents. NIL calls for no adjustment. A

number proclaims that the function demands exactly that many arguments. T proclaims that no number of arguments checking should be performed for this function (perhaps it is a "lexpr"). CL1 only uses number-of-arguments information to produce warning messages to catch programmer errors.

**REPORT** *Variable*

This variable controls the verbosity of the CL1 compiler. Its default value is NIL. Setting it to T will cause the CL1 compiler to emit a function-by-function account of its actions. This might be used during the loading of a large file of CL1 code to keep the user informed about CL1's progress.

**RUN** *Function*

This function runs the simulator. It returns when there are no more runnable processing elements.

The MacLisp function SINGLE-STEP (page 18) turns on single-stepping mode. In this mode, before any processing element executes the code associated with its current state, that processing element is printed, and the user is placed in a single-character command loop. The commands in this loop are:

Space — Causes the current processing element to execute the code associated with its current state. The next processing element in the current state is then printed and the user is returned to the command loop.

Rubout — Causes all remaining processing elements in the current state to run. A new state will then be selected, the first processing element in that state will be printed, and the command loop is returned to.

V — Displays the values of all state variables associated with the current state in the current processing element.

B — Toggles the breakpoint associated with the current state. If the breakpoint on a state is set, then processing elements in that state will be single-stepped through even if single-stepping mode is off. See also the function SET-BREAKPOINT (page 18).

S Toggles single-stepping mode itself. Single-stepping will continue until all processing elements in the current state have been run. If you don't want to watch those processing elements run, use the Rubout command next.

T Toggles state-tracing mode. State-tracing mode simply prints a message whenever a new state is selected for running, telling you what that state is and how many processing elements are in that state waiting to run. See also the function STATE-TRACE (page 19).

D Applies the DESCRIBE function to the current processing element.

I Applies the INSPECT function to the current processing element.

G Grinds and displays the MacLisp code that implements the current state. In reading such code, it is useful to know that a map, describing what state variables are stored where in a hunk that represents a processing element in this state, is included as a comment at the front of the function. Additionally, this function is *always* in the form of a DO loop that processes a linked list of processing elements. An example can be found on page 34.

If the DEFSTATE that defined the current state has been compiled by MacLisp, then the G command will not be able to show you anything.

Z Puts the current state "to sleep" for a while. The current state, along with all processing elements in that state, is moved to the end of the queue of runnable states. A new state is selected as the current state. This command gives the user a bit of control over when a particular state is run.

Q Causes the RUN function to return immediately, even if there are more runnable states. No information is lost; if the RUN function is called again, processing will continue where it left off. See the FLUSH-CELLS function (page 13).

The interrupt character Control-A can also be typed at any time turn on single-stepping mode.

**SET-BREAKPOINT** *state flag* *Function*

If *flag* is non-NIL, a breakpoint is set on *state*; otherwise all breakpoints are removed from *state*. *flag* is optional, and defaults to T. See the documentation of the RUN function (page 16) (in particular the "B" command), for a description of breakpoints. *state* can be any symbol that is the name of a state, for example:

```
(SET-BREAKPOINT 'FACTORIAL-STATE)
```

or

```
(SET-BREAKPOINT 'FACTORIAL-STATE+2)
```

*state* can also be an actual state.

**SET-GLOBAL-VALUE** *symbol value* *Function*

Modifies the value of a CL1 global variable from MacLisp code. For example (SET-GLOBAL-VALUE 'X T) sets the value of the CL1 global variable X to be T. See also the MacLisp function GET-GLOBAL-VALUE (page 13).

**SINGLE-STEP** *flag* *Function*

If *flag* is non-NIL, then single stepping is turned on; otherwise it is turned off. *flag* is optional, and defaults to T. See the documentation of the RUN function (page 16) for a description of single stepping.

**SOURCE-BACKTRACE** *Function*

At compile time, some errors will cause the compilation to stop and enter a break loop. (As usual, the user can also force his way into a break loop by typing a Control-B.) In such a break loop, this function will display the source code that the compiler was currently compiling. The innermost form the compiler was compiling is printed first, followed by the form that immediately contained it, and so forth, all the way up to the toplevel CL1 form the compiler was working on. If a form was macroexpanded, both the original form and the translated form will be printed.

**STATE-TRACE** *flag* *Function*

If *flag* is non-NIL, then state tracing is turned on; otherwise it is turned off. *flag* is optional, and defaults to T. See the documentation of the RUN function (page 16) (in particular the "T" command), for a description of state tracing.

**STATES-GC** *Function*

The CL1 compiler stores a fair amount of information on the property lists of symbols. Typically, immediately after a DEFSTATE compiles, the entire datastructure used to compile the body will be accessible from various symbols' property lists. As more compilations take place these datastructures will tend to be displaced by datastructures associated with newer compilations, but there will always be a certain amount of storage wasted by datastructures describing previous compilations. The STATES-GC function simply walks through the MacLisp world disconnecting such structures from symbols so that they will be collected as garbage.

It shouldn't ever be necessary to call this unless you are really hard up for just a few more words of storage.

See also the MacLisp variable *CELLS* (page 10).

**TRANSMIT** *pointer message* *Function*

This is the MacLisp version of the CL1 function of the same name. The message is delivered to the input buffer pointed to by the pointer. If there is already a message waiting to be read in the buffer, an error is signaled. Note that it is impossible to make the CL1 function TRANSMIT-WAIT available from MacLisp (who would wait?).

**XMEXP** *Function*

If you have your own macros, you need your own read-macroexpand-print loop to debug them. XMEXP is to DEFXMACRO as the Lisp Machine's MEXP is to DEFMACRO.

# 6. CL1 Summary

This section is an alphabetical listing of all CL1 functions, special forms, and constants. Remember that in addition to the functions listed here, any MacLisp function will automatically be treated as a primitive function by the CL1 compiler.

**%CASEQ** *Function*
**%FORGE-POINTER** *Function*
**%INPUT-CLEAR** *Function*
**%INPUT-LISTEN** *Function*
**%INPUT-WAIT** *Function*
**%LOCATION** *Special Form*
**%NULL** *Function*
**%TEST** *Function*

Any symbol whose name begins with a "%" is a sub-primitive known about in some special way by some part of the CL1 system. No user programs should ever need to call one of these. They are included in this list for completeness, and to alert users to the convention so they will not accidentally redefine them. The implementors reserve the right to add arbitrarily to this list.

***EMPTY-BUFFER-MARKER*** *Variable*

See the documentation for INPUT-LOOK on page 23.

**AND** *Special Form*

The traditional Lisp AND special form.

**BLOCK** *Special Form*

This is *not* the traditional Scheme BLOCK special form (which is called PROGN in CL1). This is the Common Lisp BLOCK special form, and is actually just a synonym for CATCH (page 21).

**CALL** *Special Form*

CALL is like MacLisp's FUNCALL function. In a language without function cells this special form is hardly ever needed. (If you had a variable named "IF", you might need it.)

**CALL-ME** *name* *Function*

If a processing element calls this function, whatever object is passed to it becomes part of the printed representation of the processing element. See the documentation of the MacLisp DPRINT function (page 12), where it is assumed that a processing element has done (CALL-ME 'ISHMAEL).

**CASEQ** *Special Form*

The familiar CASEQ special form. (Also called SELECTQ in Lisp Machine Lisp.) Both the symbols OTHERWISE and T are accepted as indicating the "else" clause. CASEQ is significantly faster than the equivalent COND form. CL1 also accepts SELECTQ as a synonym for CASEQ.

**CATCH** *Special Form*

(CATCH *tag* . *body*)

During the evaluation of forms in the body, the variable *tag* will be bound to a continuation suitable for use with the THROW special form. Unlike Scheme's CATCH, the value of *tag* can *not* just be applied as a function.

**COND** *Special Form*

Traditional.

**DO** *Special Form*

Traditional iteration construct. "Old-style DO" is *not* supported, nor is the MacLisp perversion where an empty list instead of a terminating clause makes DO behave as a "PROG with initial values".

See also TAGBODY (page 29) and RETURN (page 27).

**DOLIST** *Special Form*
**DOTIMES** *Special Form*

As in Lisp Machine Lisp. The Common Lisp extensions, where a third element in the variable-specification list is assigned a meaning, are not supported. The inclusion of DOLIST is pretty random, since nothing resembling a variable length list is likely to fit in a single Connection Machine processing element.

**GLOBAL** *Special Form*

(GLOBAL *form*)

will cause the form to be evaluated in the global (toplevel) environment rather than in the immediately surrounding lexical environment. Macros should use this special form in the code they produce to insulate themselves from the caller's environment where appropriate.

For example, if DEFINLINE did not exist, we might implement the FIRST function as a macro:

```
(DEFXMACRO FIRST (X)
  '((GLOBAL CAR) ,X))
```

Then, even if the user of the FIRST macro has the variable CAR bound in his environment, FIRST will continue to function as expected.

**GO** *Special Form*

A natural extension of the traditional special form:

(GO *state form1 form2 ... formN*)

*state* should evaluate to a state. The executing processing element is placed in that state, and its state variables are initialized to the values obtained by evaluating the forms.

GO can also be used in the familiar way with the PROG, DO, and TAGBODY special forms.

**IF** *Special Form*

(IF *test consequent*)

or

(IF *test consequent alternate*)

The traditional special form. The Lisp Machine extension of allowing an implicit PROGN of *alternate* forms is *not* supported.

**INPUT** *channel* *Function*

The executing processing element attempts to read a message from the channel. If the channel's input buffer is empty, the processing element waits until a message arrives. The message is read out of the buffer and returned, and the buffer is cleared to make room for another message.

**INPUT-CLEAR** *channel* *Function*

The executing processing element's input buffer for the channel is cleared of any buffered input.

**INPUT-LISTEN** *channel* *Function*

Returns T if there is any buffered input in the channel's input buffer, NIL if the buffer is empty.

**INPUT-LOOK** *channel* *Function*

Returns the contents of the channel's input buffer. If the buffer is empty, the value of the global variable *EMPTY-BUFFER-MARKER* will be returned, rather than waiting.

**INPUT-PEEK** *channel* *Function*

Just like INPUT except the message is not removed from the input buffer.

**INPUT-WAIT** *channel* *Function*

The executing processing element waits until the channel's input buffer contains a message. No useful value is returned.

**ITERATE** *Special Form*

As in Scheme, another way to write a LABELS.

```
(ITERATE FACT ((N N) (A 1))
  (IF (< N 2)
      A
      (FACT (1- N) (* A N))))
```

is just another way of writing:

```
(LABELS ((FACT (LAMBDA (N A)
                 (IF (< N 2)
                     A
                     (FACT (1- N) (* A N))))))
  (FACT N 1))
```

**KAPPA** *Special Form*

KAPPA is to states as LAMBDA is to functions; that is, a KAPPA-expression defines an anonymous state in the same way a LAMBDA-expression defines an anonymous function. A KAPPA-expression looks like

(KAPPA (*var1 var2* ... *varN*) . *body*)

where the variables are state variables and the body is some CL1 code, just like the CDDR of a DEFSTATE form. A KAPPA-expression evaluates to a state that behaves as it describes.

A typical use of a KAPPA-expression would be to obtain the first argument to pass to the MAKE-CELL function:

```
(MAKE-CELL (KAPPA (X)
             ...)
           (1+ Y))
```

This creates a new processing element, initializing its single state variable to one more than the value of the state variable Y in the executing processing element. The new processing element immediately begins executing the code in the body of the KAPPA-expression.

**LABELS** *Special Form*

As in Scheme or Common Lisp, LABELS allows the user to define several mutually recursive local functions. The CL1 LABELS allows KAPPA-expressions as well as LAMBDA-expressions; so this also allows the definition of mutually recursive local states.

**LAMBDA** *Special Form*
**LET** *Special Form*
**LET*** *Special Form*

All traditional.

**LET-INLINE** *Special Form*

Useful for making local inline codeable functions:

```
(LET-INLINE ((FIRST (LAMBDA (X) (CAR X))))
  ...)
```

Lexical scoping rules are carefully followed by the CL1 compiler when substituting such an inline codeable function. There is no semantic difference between LET-INLINE and LET; using LET-INLINE merely grants the CL1 compiler permission to be maximally liberal about substitutions involving the variables it binds. See also DEFINLINE (page 10).

**MAKE-CELL** *state args...* *Function*

Allocates and initializes a new processing element. MAKE-CELL's first argument must be a state, the rest of its arguments are initial values for that state's state variables. A new processing element is allocated, placed in that state, and its state variables initialized to those values. No value is returned.

**MULTIPLE-VALUE-BIND** *Special Form*

```
(MULTIPLE-VALUE-BIND (var1 var2 ... varN)
                     multiple-valued-form
  . body)
```

As in Common Lisp, and Lisp Machine Lisp. However, unlike Common Lisp and Lisp Machine Lisp, if not enough values are returned, the extra variables will *not* necessarily be bound to `NIL`; in fact, no guarantees are made about their values.

**NAMED-KAPPA** *Special Form*
**NAMED-LAMBDA** *Special Form*

```
(NAMED-LAMBDA name variables . body)
```

and

```
(NAMED-KAPPA name variables . body)
```

Not exactly analogous to Lisp Machine Lisp's `NAMED-LAMBDA`. During the evaluation of the body of a `NAMED-LAMBDA` or `NAMED-KAPPA` *name* is *bound* to the function or state itself. Users probably have little need for these, but the CL1 compiler is a lot cleaner internally than it would be without them.

**NIL** *Constant*

```
(PROCLAIM-CONSTANT 'NIL 'NIL)
```

Of course! Also `NIL` is illegal as a variable anywhere.

**NOT** *Special Form*

Traditional logical negation special form.

**OR** *Special Form*

Traditional special form.

**PERFORM** *Special Form*

(PERFORM *form* . *defs*)

is the same as

(LABELS *defs form*)

except there are less parenthesis, and you can read it in the proper order. Years ago Ed Kleban invented this trivial macro which sometimes improves the readability of large LABELS forms.

**POINTER?** *object* *Function*

Returns T if *object* is a pointer.

**PROG** *Special Form*

Implemented in the obvious way in terms of LET and TAGBODY. See also RETURN.

**PROG1** *Special Form*
**PROG2** *Special Form*
**PROGN** *Special Form*

Traditional.

**QUOTE** *Special Form*

Traditional.

**RETURN** *Special Form*

(RETURN *val*)

Returns *val* from the most recent containing PROG or DO. This is implemented in a kludgy fashion by surrounding PROG and DO forms with (CATCH *ITERATION-BLOCK* ...), and having RETURN use the current lexical value of *ITERATION-BLOCK*. Nothing else in CL1 is implemented so tastelessly.

Also:

(RETURN *val1 val2 ... valN*)

As in Common Lisp and Lisp Machine Lisp, this form of RETURN can be used to return multiple values.

**RETURN-FROM** *Special Form*

(RETURN-FROM *tag-form val1 val2 ... valN*)

Like the Common Lisp special form for returning from a BLOCK. *tag-form* should evaluate to a continuation obtained from a CATCH (or BLOCK) form. The values will be returned from the corresponding CATCH.

**SELECTQ** *Special Form*

Synonym for CASEQ.

**SETF** *Special Form*
**DECF** *Special Form*
**INCF** *Special Form*
**POP** *Special Form*
**PUSH** *Special Form*

As in Lisp Machine Lisp. However, since CL1 lacks the usual SETQ special form, and since few useful accessing functions are invertible, these are useless. SETF and its associates exist pending a more coherent theory of side-effects in a Connection Machine.[1]

---

1. Given a compiler that can optimize away unnecessary LAMBDAS, I couldn't resist writing a SETF that functions correctly (evaluates its arguments in the proper order, and returns the expected value).

**T** *Constant*

```
(PROCLAIM-CONSTANT 'T 'T)
```

Of course! It is also illegal to use T as a variable anywhere.

**TAGBODY** *Special Form*

(TAGBODY . *body*)

Like the body of the original Lisp PROG special form. *body* should consist of a sequence of non-atomic forms and atomic tags. The GO special form (in its single argument version) can be used to jump around in the usual fashion.

**THROW** *Special Form*

(THROW *tag-form* . *body*)

*tag-form* should evaluate to a continuation obtained from a CATCH (or BLOCK) form. The rest of the forms in the body are evaluated and the value(s) returned from the last one are returned from the corresponding CATCH.

**TRANSMIT** *pointer message* *Function*
**TRANSMIT-WAIT** *pointer message* *Function*

The message is delivered to the processing element pointed to by the pointer. If that processing element already has a message in its input buffer, TRANSMIT signals an error. Under the same circumstances TRANSMIT-WAIT causes the executing processing element to go into a wait state until that input buffer empties.

**UNLESS** *Special Form*

As recently introduced into Lisp Machine Lisp and Common Lisp,

(UNLESS *test* . *body*)

is equivalent to

(IF (NOT *test*) (PROGN . *body*))

**VALUES** *Special Form*

(VALUES *value1 value2 ... valueN*)

Traditional special form for returning multiple values. *N* values are returned as the value of this form.

**WHEN** *Special Form*

As recently introduced into Lisp Machine Lisp and Common Lisp,

(WHEN *test* . *body*)

is equivalent to

(IF *test* (PROGN . *body*))

**WITH-CHANNEL** *Special Form*

(WITH-CHANNEL (*channel pointer*)
  . *body*)

Allocates an input buffer, creating a pointer to that buffer for other processing elements to use to send messages to, and a channel that the executing processing element can read messages from. The variable *pointer* is bound to the new pointer, and the variable *channel* is bound to the new channel object within *body*. Explained in full in section 4.

**WITH-VALUES** *Special Form*

(WITH-VALUES (*vars form*) . *body*)

is the same as

(MULTIPLE-VALUE-BIND *vars form* . *body*)

In addition to the functions listed above, and those inherited automatically from MacLisp, the following traditional functions have been defined: <=, >=, FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH, SEVENTH, EIGHTH, LISTP, LOGAND, LOGANDC1, LOGANDC2, LOGEQV, LOGIOR, LOGNAND, LOGNOR, LOGNOT, LOGORC1, LOGORC2, LOGXOR, NULL, and REST. (Many of these function are implemented in MacLisp using macros, and so can not be automatically

inherited.)

# 7. Example

This section contains a complete example of using the MacLisp implementation of CL1. We will examine the execution of the following piece of CL1 code, which is assumed to be contained in a file named FIB.LSP in the user's connected directory on OZ.

```
(DEFSTATE FIB (CREATOR N)
  (CALL-ME '(FIB ,N))
  (IF (< N 2)
      (TRANSMIT-WAIT CREATOR N)
      (WITH-CHANNEL (CHANNEL POINTER)
        (MAKE-CELL FIB POINTER (- N 1))
        (MAKE-CELL FIB POINTER (- N 2))
        (LET ((ANSWER (+ (INPUT CHANNEL)
                         (INPUT CHANNEL))))
          (PRINT '((FIB ,N) => ,ANSWER))
          (IF (POINTER? CREATOR)
              (TRANSMIT-WAIT CREATOR ANSWER))))))
```

After MacLisp has digested this form, the value of the CL1 variable FIB will be the first state (named "FIB") of the finite state machine shown in figure 1. It's a rather poor way to use the Connection Machine to compute Fibonacci numbers. [1]

A processing element in state FIB first renames itself to something descriptive, so we can more easily distinguish between processing elements. Then the state variable N is tested. If it is less than 2, then that value is the answer, and it is immediately transmitted to the processing element's creator.

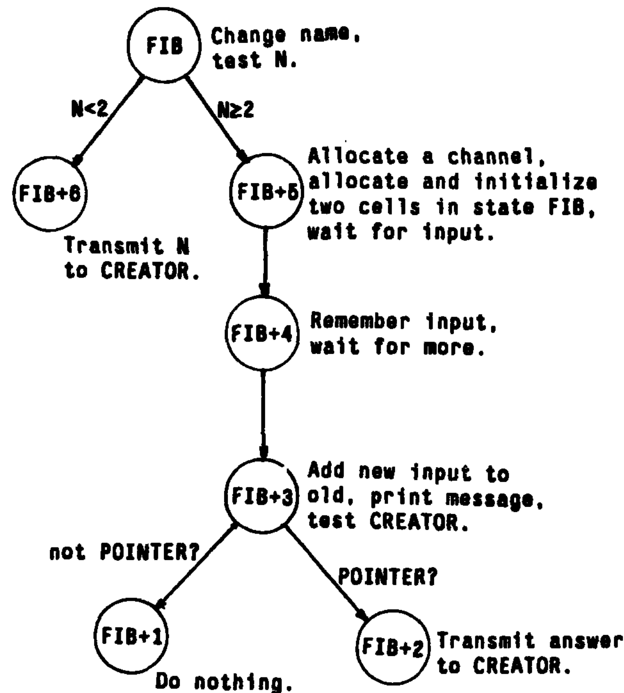
Otherwise, the processing element allocates two new processing elements. Both new processing elements are initialized to state FIB, and are given the same newly allocated pointer as the initial value of their CREATOR state variable. One processing element is given N-1 as the initial value of its N state variable, the other is given N-2.

The processing element then tries to input a value from the channel associated with the pointer it just passed to the two new processing elements. This causes the processing

---

1. Not because it's slow (it runs in linear time), but because it uses an exponential number of processing elements.

**Fig. 1. State machine for `FIB`:**




---

element to wait for that input to arrive before entering the next state. When one value arrives, the processing element immediately tries to read a second value from the channel. Once that value arrives, the processing element adds them up and prints a message informing us of the answer.

Notice how we depend on the commutativity of addition for this to work. There is no guarantee that the two answers will arrive in any particular order, all we know is that there will be exactly two. Notice also the use of `TRANSMIT-WAIT` rather than simple `TRANSMIT`, which protects us from losing either of these messages in transmission.

Finally the processing element tests the value of its `CREATOR` state variable. If it is not a pointer, then the processing element is done. If it is a pointer, then the answer is transmitted to the creator. This is done solely to make the example simpler. The first processing element allocated will have a `CREATOR` of `NIL`, rather than a pointer to another processing element, and will not bother to do anything with the answer it has computed.

We begin by starting a MacLisp, loading the CL1 compiler and runtime, setting few switches, and loading the file `FIB.LSP` (user input is indicated by using lower case, a "▮" indicates the position of the cursor):

```
@lisp

LISP 2140
ALLOC? n


*
(load '((alan/.cm) states))     ;Load CL1.
153163
(state-trace)                   ;Print each state before it is run.
T
(single-step)                   ;Pause before execution at each cell.
T
(dprint t)                      ;Rationalize the way things print.
T
(load 'fib)                     ;Load the file FIB.LSP.
T
▮
```

The CL1 compiler ran during the loading of the file (when the DEFSTATE macro in FIB.LSP was being expanded by MacLisp), so any compile-time errors would have been signaled then.

We allocate a single processing element and initialize its state variables using the MacLisp MAKE-CELL special form, then we invoke the RUN function to start the simulated Connection Machine running:

```
(make-cell fib nil 4)
#<CELL 666560 @FIB>
(run)
WORKING ON STATE #<STATE FIB>  (1 CELL)
#<CELL 666560 @FIB> ? ▮
```

Since state-tracing mode is on, CL1 first announces which state will be worked on next. In this case, there is only one processing element in the machine, and that processing element is in state FIB, so CL1 has no choice but to select FIB as the first state to be run. Since single-stepping mode is on, CL1 then picks one of the processing elements in the current state (there is only one), prints it, and enters a single character command loop. There are many useful commands available (typing "?" will list them). The V command will display the values of state variables in the current processing element:

```
#<CELL 666560 @FIB> ? v
 CREATOR = NIL
 N = 4
#<CELL 666560 @FIB> ? ▮
```

The G command will display the MacLisp code, produced by the CL1 compiler, that implements the current state:

```
#<CELL 666560 @FIB> ? g

(LAMBDA (CELL)
  (COMMENT VMAP= (CREATOR 7) (N 10))
  (DO ((NEXT)
       (S-111 (GET 'FIB+5 'STATE))
       (S-110 (GET 'FIB+6 'STATE)))
      ((NULL CELL))
   (SETQ NEXT (CELL-NEXT-IN-QUEUE CELL))
   (RPLACX 11 CELL (LIST 'FIB (CXR 10 CELL)))
   (SETF (CELL-NAME CELL) (CXR 11 CELL))
   (SET-STATE CELL (IF (< (CXR 10 CELL) '2) S-110 S-111))
   (SETQ CELL NEXT)))

#<CELL 666560 @FIB> ? ▮
```

This code is typical of that produced by CL1. It is a function that will be applied to the first processing element in a linked list of processing elements in this state. The function iterates over this list using CELL-NEXT-IN-QUEUE to retrieve the next processing element in the list. At the top, a comment tells the locations of the state variables CREATOR and N. A location is simply an index into the MacLisp hunk used to represent the processing element; the MacLisp functions CXR and RPLACX retrieve and store state variables using these indices. Several values are computed once when the DO-loop is started for use by each processing element; in this case the states FIB+5 and FIB+6 are fetched. For each processing element, the body of the loop modifies the processing element's name, and then sets its state based on the value of the state variable N. Temporary use is made of location 11 of the processing element during the computation.

CL1 is still waiting to be told what to do about this processing element. The Space command tells it to go ahead and run the current state in the current processing element:

```
#<CELL 666560 @FIB> ? <Space>
WORKING ON STATE #<STATE FIB+5>  (1 CELL)
#<CELL (FIB 4) @FIB+5> ? ▮
```

Not surprisingly, state FIB+5 has been selected to run next. Notice that our single processing element has successfully changed its name from 666560 to (FIB 4). When allowed to run, this processing element will allocate two new processing elements to compute (FIB 3) and (FIB 2):

```
#<CELL (FIB 4) @FIB+5> ? <Space>
WORKING ON STATE #<STATE FIB>  (2 CELLS)
#<CELL 666460 @FIB> ? v
 CREATOR = #<POINTER 11 (FIB 4)>
 N = 2
#<CELL 666460 @FIB> ? <Space>       ;Let (FIB 2) run.
#<CELL 666470 @FIB> ? v
 CREATOR = #<POINTER 11 (FIB 4)>
 N = 3
#<CELL 666470 @FIB> ? <Space>       ;Let (FIB 3) run.
WORKING ON STATE #<STATE FIB+5>  (2 CELLS)
#<CELL (FIB 3) @FIB+5> ? ▮
```

As more and more processing elements are allocated, it becomes tedious to use the Space command to step through the execution of each processing element in a particular state. The Rubout command runs all of the processing elements in the current state, and then advances to a new state:

```
#<CELL (FIB 3) @FIB+5> ? <Rubout>
WORKING ON STATE #<STATE FIB>  (4 CELLS)
#<CELL 666340 @FIB> ? ▮
```

Now four processing elements all in state FIB waiting to run. Let's return to MacLisp (using the Q command) and poke around a bit:

```
#<CELL 666340 @FIB> ? q
QUIT
*cells*
(#<CELL 666340 @FIB>
 #<CELL 666350 @FIB>
 #<CELL 666370 @FIB>
 #<CELL 666420 @FIB>
 #<CELL (FIB 2) INPUT-HANG@FIB+4>
 #<CELL (FIB 3) INPUT-HANG@FIB+4>
 #<CELL (FIB 4) INPUT-HANG@FIB+4>)
▮
```

Note that our original processing element and the two processing elements it allocated are all waiting for messages before they can begin state FIB+4. We now set a breakpoint at FIB+4, and at the following state, FIB+3, and also turn off single stepping.

```
(set-breakpoint 'fib+4)
#<STATE FIB+4>
(set-breakpoint 'fib+3)
#<STATE FIB+3>
(single-step nil)
NIL
(run)
WORKING ON STATE #<STATE FIB>  (4 CELLS)
WORKING ON STATE #<STATE FIB+5>  (1 CELL)
WORKING ON STATE #<STATE FIB+6>  (3 CELLS)
WORKING ON STATE #<STATE FIB>  (2 CELLS)
WORKING ON STATE #<STATE FIB+4>  (2 CELLS)
#<CELL (FIB 3) @FIB+4> ? ▮
```

Two processing elements have reached state FIB+4. We examine the state variables of the first:

```
#<CELL (FIB 3) @FIB+4> ? v
 CREATOR = #<POINTER 11 (FIB 4)>
 N = 3
 IO-BUFFER-5 = 1
#<CELL (FIB 3) @FIB+4> ? ▮
```

Sure enough, it has a message sitting in its input buffer. We let that processing element run, and examine the other one:

```
#<CELL (FIB 3) @FIB+4> ? <Space>
#<CELL (FIB 2) @FIB+4> ? v
 CREATOR = #<POINTER 11 (FIB 4)>
 N = 2
 IO-BUFFER-5 = 0
#<CELL (FIB 2) @FIB+4> ? <Space>
WORKING ON STATE #<STATE %NULL>  (3 CELLS)
```

Three processing elements have finished their tasks, and are entering the system supplied state named "%NULL".

```
WORKING ON STATE #<STATE FIB+6>  (2 CELLS)
WORKING ON STATE #<STATE FIB+3>  (1 CELL)
#<CELL (FIB 2) @FIB+3> ? v
 CREATOR = #<POINTER 11 (FIB 4)>
 N = 2
 IO-BUFFER-5 = 1
 A-55 = 0
#<CELL (FIB 2) @FIB+3> ? ▮
```

A processing element has finally received its second message, which is sitting in its input buffer; the old message is now in a state variable named "A-55". The code associated with state FIB+3 will add these two numbers, print a message, and then decide whether to transmit the answer to its creator. (Since we are no longer interested in the breakpoint on state FIB+4, we use the B command to clear it the next time it is triggered.)

```
#<CELL (FIB 2) @FIB+3> ? <Space>
((FIB 2) => 1)
WORKING ON STATE #<STATE FIB+4>  (1 CELL)
#<CELL (FIB 2) @FIB+4> ? b (STATE BREAKPOINT OFF) ? <Rubout>
WORKING ON STATE #<STATE %NULL>  (2 CELLS)
WORKING ON STATE #<STATE FIB+2>  (1 CELL)
WORKING ON STATE #<STATE FIB+3>  (1 CELL)
#<CELL (FIB 2) @FIB+3> ? <Space>
((FIB 2) => 1)
WORKING ON STATE #<STATE FIB+4>  (1 CELL)
WORKING ON STATE #<STATE %NULL>  (1 CELL)
WORKING ON STATE #<STATE FIB+2>  (1 CELL)
WORKING ON STATE #<STATE FIB+3>  (1 CELL)
#<CELL (FIB 3) @FIB+3> ? <Space>
((FIB 3) => 2)
WORKING ON STATE #<STATE %NULL>  (1 CELL)
WORKING ON STATE #<STATE FIB+2>  (1 CELL)
WORKING ON STATE #<STATE FIB+3>  (1 CELL)
#<CELL (FIB 4) @FIB+3> ? v
 CREATOR = NIL
 N = 4
 IO-BUFFER-5 = 2
 A-55 = 1
#<CELL (FIB 4) @FIB+3> ? ▮
```

Finally the original processing element has received both messages. After it runs, a few stragglers drop into the %NULL state, and the simulation halts and returns us to MacLisp:

```
#<CELL (FIB 4) @FIB+3> ? <Space>
((FIB 4) => 3)
WORKING ON STATE #<STATE %NULL>  (1 CELL)
WORKING ON STATE #<STATE FIB+1>  (1 CELL)
WORKING ON STATE #<STATE %NULL>  (1 CELL)
DONE
▮
```

# 8. Acknowledgments

Of great value at one time or another during the development of CL1 were Tom Knight, Gerald Sussman, Danny Hillis, Phil Agre, David Chapman, David Christman, Carl Feynman, Cliff Lasser and David Moon. Phil Agre deserves special thanks for being CL1's first user. Penny Berman helped translate this document into english. The author is entirely responsible for any faults that remain.

1. Christman, David P., "Programming the Connection Machine", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT (in preparation)

2. Feynman, Carl, "The Connection Machine", MIT AI Memo ??? (major revision of [3], in preparation).

3. Hillis, W. Daniel, "The Connection Machine (Computer Architecture for the New Wave)", MIT AI Memo 646 (Sept. 1981). See also [2].

4. Pitman, Kent M., "The Revised MacLisp Manual", MIT LCS TR-295 (May 1983).

5. Steele, Guy Lewis Jr., "Debunking the 'Expensive Procedure Call' Myth", Proc. ACM National Conference (Oct. 1977), 153-162. Revised as MIT AI Memo 443 (Oct. 1977).

6. Steele, Guy Lewis Jr., and Sussman, Gerald Jay, "The Revised Report on SCHEME: A Dialect of Lisp", MIT AI Memo 452 (Jan. 1978).

7. Steele, Guy L. Jr., et al., "Common Lisp Reference Manual", CMU (in preparation).

8. Weinreb, Daniel, and Moon, David, "Lisp Machine Manual", Symbolics Inc. (July 1981).