THE BERKELEY PROCESS-FLOW LANGUAGE: REFERENCE DOCUMENT

by

Christopher B. Williams and Lawrence A. Rowe

Memorandum No. UCB/ERL M87/73

15 October 1987

THE BERKELEY PROCESS-FLOW LANGUAGE: REFERENCE DOCUMENT

by

Christopher B. Williams and Lawrence A. Rowe

Memorandum No. UCB/ERL M87/73

15 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Berkeley Process-Flow Language: Reference Document

Christopher B. Williams
(williams@merlin.berkeley.edu)

Lawrence A. Rowe
(larry@postgres.berkeley.edu)

Computer Science Division – Dept. EECS
University of California, Berkeley
Berkeley, CA 74720

Version 1
October 15, 1987

Abstract

The Berkeley Process-Flow Language (BPFL) for integrated circuit manufacturing is described. The language specifies the manufacturing steps that describe how a semiconductor is fabricated. The language will be used to automate the manufacturing process and as an input and output language for other computer integrated manufacturing tools (e.g., process and scheduling simulators).

This research was supported by grants from the Semiconductor Research Corp., Fairchild Semiconductor, Harris Semiconductor, Philips/Signetics, and Siemens Corp., with a matching grant from the State of California's MICRO program. The first author was also supported by a fellowship from the Semiconductor Research Corporation.

THE BERKELEY PROCESS-FLOW LANGUAGE: REFERENCE DOCUMENT

by

Christopher B. Williams and Lawrence A. Rowe

Memorandum No. UCB/ERL M87/73

15 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

1. Introduction

This document describes the Berkeley Process-Flow Language (BPFL). It is a reference document for those who wish to specify processes with BPFL or to write process interpreters.

Specifications written in BPFL describe the fabrication of semiconductor devices. Fabrication requires many processing steps to be performed in a fixed sequence. The word *process* is commonly used to describe two different aspects of fabrication. The first usage refers to the entire sequence of operations that result in a working device (e.g., a "2 micron CMOS process"). The second usage refers to an operation within the sequence (i.e., a processing step), usually associated with one piece of equipment (e.g., an oxidation process, an implantation process, or a photolithography process).

A BPFL *process* is composed of a *sequence* of smaller, more specific processes. Thus, a *process* is a hierarchy, with the most general processes at the top. BPFL categorizes the processes in a hierarchy into three *process levels*. Each process level has a set of *process steps* that define the finest level of detail expressible at that level. At the highest level, called the *flow* level, the process steps specify the desired effect to be achieved (e.g., "grow 1000A of oxide"). At the second level, called the *generic-equipment* level, the process steps describe how to implement a flow step using a general class of equipment (e.g., a tube furnace or an ion-beam implanter). At the third and lowest level, called the *equipment-specific* level, the process steps describe in complete detail how to implement the generic-equipment steps on a specific piece of equipment.

A complete process specification, like "2 micron CMOS," is called a *process-flow*, while *process* can refer to any level of detail. In a BPFL specification of a process-flow,

processes are represented by *functions*. Functions can call other functions in sequence, just as processes can use other processes. Functions have local and global variables that hold data values, similar to variables in conventional programming languages.

Each function in a process-flow is specified as being at one process level (i.e., flow, generic-equipment, or equipment-specific). Statements in these functions may call functions at the same or lower level, but they may not call functions at a higher level. For example, a flow level function may call any function since it is at the highest level. An equipment-specific function may call another equipment-specific function, but cannot call a flow or generic level function.

The fabrication unit from a process design viewpoint is a *wafer*. In practice, however, work is performed on a set of wafers, called a *lot*. During a processing step, the wafers in a lot may be treated serially or in parallel, depending on the specific equipment. A lot and its controlling process specification is called a *run*. In a typical fabrication facility, several runs, each at a different stage of completion, will be under the control of one process specification. BPFL process specifications are written in terms of one run. Within a run, the lot may be divided into sub-lots, each of which can be processed separately from the rest.

The intent of BPFL is to represent all information about a fabrication process that will be needed by any application programs during the design, manufacture, and testing of semiconductors. Because different applications need different kinds of information, a process-flow actually specifies several different views of a process. For example, a production control application needs equipment-specific information, while a process simulator can use only generic-equipment information and can ignore the test wafer process-

ing. Most application programs will be *interpreters* that extract information from a BPFL process-flow description according to the needs of the application.

A process-flow specification has several semantic interpretations because the specification is interpreted by many applications. To allow one description to be used by many interpreters, basic semantic forms must be preserved by all interpretations. The basic structure of BPFL is taken from Common Lisp[Steele]. However, BPFL is intended to be a description language as opposed to a general purpose programming language. BPFL is distinguished from Common Lisp by several restrictions and two modifications of semantics. It is expected that BPFL will be used in an environment that relies on a database. Thus, the semantics of some constructs are defined in terms of a database rather than a main memory Lisp system. The programming functionality supported by BPFL was limited so that it would be easier to write a variety of interpreters.

The remainder of this report describes BPFL. It is organized as follows. The next section describes the syntax of BPFL. The third section describes the language semantics and functionality that must be supported by application interpreters. The fourth section provides some examples of descriptions written in BPFL.

2. BPFL Syntax

BPFL syntax is defined by its printed representation. BPFL could also be represented in a database or a Lisp system, but these forms, although easier to manipulate, are not as portable. Thus, all examples will be textual and set in `this typeface`. The first sub-section defines the data types. The remaining sections define special uses of the *list* data type for function calls and function definition.

2.1. Data Types

The BPFL data types are described by their printed representation and a general discussion. In what follows, a *newline* (i.e., a line separation character) may appear wherever a space is valid, unless otherwise specified. The behavior of the data types will be specified later by the functions that operate on them.

A *value* exists either as a constant or as a dynamic entity created by an interpreter. Data types describe the behavior of data values (i.e., values have a data type). A set of primitive or basic data types are defined (e.g., integer, floating point, and string). In addition, constructors are provided that can be used to define more complex, structured data types. The BPFL data types are described by first defining the basic data types and then defining the type constructors.

2.1.1. Basic Data Types

`integer` *[Data type]*

An integer is a signed number in the range that can be represented by a 32-bit two's complement binary number (i.e., -2147483648 to 2147483647) . Integers are printed in decimal with a preceding sign ("+" is optional).

Examples:

```
0   1   +256 -100000   2147483647
```

`ratio` *[Data type]*

The ratio data type allows exact representation of numbers such as *one-third.* The print representation is two integers separated by a slash ("/"). The first integer is the numerator of a rational number, and the second is the denominator. The numerator and

denominator should have a greatest common factor of one. The denominator must be strictly positive (i.e., non-zero) and has no preceding sign.

Examples:
```
1/2  1/3  -7/8  15/7 12831129/192282938
```

`float` [*Data type*]

BPFL has single precision and double precision floating point numbers analogous to those found in most computer systems. These have a precision of about 7 and 15 decimal places, respectively. The mantissa is required, and may be followed by an exponent, separated by a letter called an *exponent marker* (i.e., "scientific notation"). The mantissa and exponent are printed in decimal with a preceding sign ("+" is optional). The print representation must contain at least a decimal point or an exponent. A decimal point, if present, must be followed by at least one digit. If no exponent marker is given, or if the exponent marker is "`e`" or "`f`", then the number is single precision. If the exponent marker is "`d`", then the number is double precision. The number of digits given does not affect the precision of the value.

Examples:
```
1.0  .1  0.1  1e0  1.0e7  3.14159265358979317d0
```

`string` [*Data type*]

A string is a sequence of printable characters delimited by double quotes ("`"`"). A printable character is any non-control character in the ASCII character set. If the backslash character ("`\`") appears in a string, the following character is included in the string, even if that character is a backslash or a double quote. A string may contain zero characters, or up to 10,000 characters. A newline (textual line separator) may appear in a

string. It is printed as a newline and the rest of the characters in the string follow immediately at the beginning of the next line. The case of characters is significant. BPFL is not intended to be a text manipulation language. Thus the text handling capabilities are intentionally limited.

Examples:

```
"string"
""
"Hello world."
"\"Hi!\""
"This string goes from here
to here"
```

symbol [*Data type*]

A symbol is a sequence of printable characters, without any special delimiters. The characters may be alphanumeric, or any character in the string "+-*/@$%^&_<>.~". Note that the question mark ("?") and the exclamation mark ("!") are not allowed since they are reserved for future expansion of the language. The case of alphabetic characters is *not* significant and so a symbol may be printed with either upper or lower case letters. Symbols names may have up to 255 characters. A symbol may not consist entirely of dots. A symbol should not resemble a number. In the most general case, a number is a series of signs, dots, slashes, singleton letters, and at least one digit. Obviously, not all such combinations would be considered numbers as defined by BPFL. Anything that cannot be interpreted as a number is a symbol. However, to allow for future representations of numbers, such symbols should be avoided. In any case, a sequence of characters is a symbol and not a number if it ends with a sign ("+" or "-") or starts with a slash ("/"). Symbols can be used to name variables.

Examples:

```
hello   oxidation   *report-format*   1+   %%%   &aux   x
```

keyword [*Data type*]

A keyword has the same syntax as a symbol, except that the first character of its print representation is a colon (":"). A symbol and a keyword that print the same (modulo the initial colon) are said to *correspond*. Thus `:energy` is the corresponding keyword for the symbol `energy`, and *vice versa*. Except where noted otherwise, a keyword can be used in place of a symbol.

2.1.2. Data Type Constructors

These types have structures and print representations that include other data types.

object [*Data type*]

An object has no print representation and may not be written out by a user. An object may be used as a value generated by a function and may be passed as an argument to a function. An object belongs to a *class*, which defines a number of *slot* names and *method* names. If a class defines a particular slot, then each *instance* of that class (i.e., each object belonging to that class) can store a value under the name of that slot. Certain functions, called *access functions*, will return values stored in the slots of an object.

Methods are functions. Several related classes will use the same method name, but each will bind a different function to that name. Thus, when the method name is invoked on an object, the function used will be determined by the class of the object, and will thus be appropriate. For example, suppose all classes associated with furnaces defined a method `current-temperature` that determined the current temperature for a tube. For each class of furnace, the details of determining the current temperature are different.

However, given an object from a furnace class, invoking the `current-temperature` method will always work. BPFL presumes that objects will be stored in a database. Classes are not defined within BPFL process specifications. Classes used by a process specification must already exist in the object support environment.

`list` *[Data type]*

A list is printed as a left parenthesis ("`(`"), followed by zero or more values of valid BPFL types, followed by a right parentheses ("`)`"). Elements of the list are separated from each other by spaces or newlines. The empty list ("`()`") is synonymous with the symbol `nil`. Parenthesis need not be delimited by spaces. An apostrophe may be used to abbreviate a list whose first element is the symbol `quote`.

Examples:

```
(a b c)   (1 2 3)   ()   nil   (a (b c))   ((a))
(quote (1 2 3))
'(1 2 3)
```

`array` *[Data type]*
`vector`
`bit-vector`

Arrays, vectors, and bit-vectors contain data values that are identified by indicies. Elements of vectors and arrays may be values of any valid BPFL type. An element of a bit-vector is either `1` or `0`. Unless otherwise specified, a bit-vector can be used in place of a general vector.

A general vector prints as a hash mark ("`#`") followed by an optional non-negative integer indicating the length of the vector, followed by a list whose elements are the vector elements. The first element has the lowest index (i.e., 0). If no vector length was specified, then the vector has as many elements as were given. If a length was given,

then the number of given elements must not exceed that length. If fewer elements were given, then the remaining ones are set to the last value in the list (or `nil` if none was given).

An array prints as a hash mark, followed by a positive integer indicating the array rank (number of dimensions), followed by the letter "`a`" or "`A`", followed by a list whose elements define the first dimension of the array. If the array is one dimensional, this is equivalent to filling a vector. If the array has more than one dimension, then each element of the list is in turn a list that could fill an array of one less dimension. The number of elements in a list must exactly match the size of the appropriate dimension. Thus the first list used in a particular dimension determines the size of the dimension, and all other lists must match that size. BPFL does not support arrays with zero dimensions. Vectors are arrays with a rank of one.

A bit-vector prints as a hash mark, followed by an optional non-negative integer indicating the length of the vector, followed by an asterisk ("`*`"), followed by a series of "`0`" and "`1`" digits. At least one digit must be given if a non-zero length was specified. The series of digits is used to fill the bit-vector in the same way that a list is used to fill a vector.

Examples:

```
#(1 2 3)
```

These two vectors are equivalent:

```
#3(x y)
#(x y y)
```

A 2 by 2 array:

```
#2a((a b) (c d))
```

A 3 row by 2 column array:

```
#2A((a b) (c d) (e f))
```

```
#*101101
```

unit [*Data type*]

A unit value consists of a magnitude (an integer, ratio, or floating point number) and a unit designator (a symbol). A basic unit is separated from its exponent with a circumflex (“^”). The exponent can be negative. Thus, the unit “square meter” would be `m^2`. Basic units multiplied together should be separated by a dash (“-”). For example, “acre-feet” (a unit of volume) is `acre-ft`. The unit designator should contain one slash (“/”) to separate the numerator and denominator units. For example, “meters per second” is `m/s`. If there are no numerator units (e.g., “per second”), the unit destinator begins with a slash (i.e., `/sec`). If there are no denominator units, the slash is not present. Values without units have the unit designator “-”. Unit values with the same unit designator may be compared based on their magnitude. Since units may have negative exponents, unit designators must be formed consistently throughout a site. Thus, the unit “per square meter” should always be either `/m^2` or `m^-2`. These are considered different unit designators, unless the optional unit conversion facility is supported. A unit value is printed as “`#u` (*number unit-designator*) ”.

Examples:

```
#u(1000 A)
#u(20 um)
#u(10 mega_ohm)
#u(3.07E9 m/s)
#u(9.8 m/sec^2)
```

interval [*Data type*]

An interval value consists of two numbers or two unit values with the same unit designator. The first item should be less than or equal to the second. An interval is

printed as "`#i` (*item-1 item-2*)".

Examples:

```
#i(0 1/2)
#i(-1.0 1.0)
#i(#u(3 in) #u(5 in))
```

2.2. Function Calls

A list can be interpreted as a function call. The first element of the list is a non-keyword symbol, called the *header*. The header symbol will be discussed in more detail below in the section on semantics. The remaining elements in the list are the actual arguments to the function. There are two kinds of arguments: *positional* and *keyword*. For positional arguments, the elements in the list immediately following the header symbol are used. Each element corresponds to one argument. The order of the elements is significant when assigning values to the formal arguments of the function being called. For keyword arguments, the elements of the list are used in pairs. The first member of a pair must be a keyword. The second member can be any value. The keyword names the argument and the second part gives the value of the argument. The order in which arguments are given is not significant because the formal arguments are assigned by name. It is an error for a keyword argument to have a name and no value.

If both positional and keyword arguments are to be passed to a function, then the positional arguments must appear first in the list, following the header symbol. Positional arguments end with the first appearance of a keyword. If it is necessary to pass a keyword as a positional argument, then it may be passed with the `quote` function, either as "`(quote` *:keyword*)," or in the abbreviated form "`'` *:keyword*."

Examples:
A function call with two positional arguments:

```
(+ 1 1.3)
```

A function call with two keyword arguments:

```
(etch :thickness #u(10 um) :target 'si-oxide)
```

A function call with a positional argument and a keyword argument:

```
(sort input-list :descending t)
```

A function call with one positional argument:

```
(foobar ':bletch)
```

2.3. Function Definitions

A list can be interpreted as a function definition. The first element of the list is one of the symbols `defflow`, `defgeneric`, or `defequipment`, which indicates the *process level* of the function (i.e., flow, generic-equipment, or equipment-specific). The second element is a non-keyword symbol that names the function. While examples show names as simple identifiers, in general they would be unique within a site and would contain version control information. The third element is an *argument-declaration* list that defines the formal arguments to the function and its local variables. The remaining elements of the list (if any) make up the body of the function. The general syntax of a function definition is:

(`def`*level name* (*arg-list*) *body* ...)

The argument-declaration list is divided into two parts. The first part defines the formal arguments and includes all elements up to the end of the list or to the symbol `&aux`. The second part of the list defines local variables and includes all elements after the `&aux` symbol.

The elements of the argument-declaration list, other than the `&aux` symbol, are called *variable-specifiers*. Variable-specifiers for arguments and declarations have

similar syntax. Each variable-specifier must be either a non-keyword symbol or a list with one, two, or three elements. If it is a symbol, it names a variable, be it a formal argument or a local variable. If it is a list, the first element must be a non-keyword symbol, which names a variable. A variable-specifier list may have a second element, called an *initialization form* that can be used to initialize the variable. The third element of the list, which should be a non-keyword symbol, names another variable, called the *was-supplied* variable. Further interpretation of the argument-declaration list will be described in the section on semantics.

The body of the function has two parts: declarations and code. The declarations part consists of the zero or more elements at the beginning of the body that are either strings or lists whose first element is the symbol `declare`. The strings provide documentation. The remaining elements in a `declare` list are *declarations* and can be used to note special conditions. The only declaration supported by all interpreters is a list whose first element is the symbol `special` and whose remaining elements are symbols that are to be used as global variables. Global variables are variables that exist as long as the run being processed exists.

The elements of the function body following the declarations are generally function calls (i.e., lists). The function body is a boundary defining the scope of local variables and control flow. Local variable names do not affect the use of the same names in other function bodies. Control of the execution sequence within the function body is restricted to that function. Control may not be transferred to another function except by a function call or by exiting the current function body.

Object methods are functions bound to an object class. A list can be interpreted as a method definition. The first element of the list is one of symbols `defflow-method`, `defgeneric-method`, or `defequipment-method`, which indicates the process level of the method. The second element of the list is a non-keyword symbol that names the method. The third element of the list is an argument-declaration list and the remaining elements make up the body of the function. The first element of the argument-declaration list must be a variable-specifier list with two elements. The first element is the symbol `self` and the second element is a symbol that names the class under which this function will be bound to the method name. The remainder of the argument-declaration list and the function body are the same as for user-defined functions. Figure 1 gives some examples of function and method definitions.

```
(defflow CMOS-well-formation (&aux (thickness #u(1000 um)))

   "Dope and drive-in the well"
   (declare (special *mask-set*))
   (mask :mask 'WELL
         :thickness thickness)
   . . . )
```

This function is called `cmos-well-formation` and takes no formal arguments. It has one local variable `thickness` that is initialized to a unit value. The function uses one global variable and one function call in the function body is shown.

```
(defequipment-method run-sequence ((self tylan) sequence &aux recipe)
   "Given a sequence, find the appropriate recipe"
   . . .)
```

This function is bound to the `tylan` class under the method name `run-sequence`. It takes two formal arguments, one of which is the required `self` argument. The function has one local variable.

Figure 1 – Functions and method definition examples.

---

3. Basic BPFL Semantics

This section describes the semantics of BPFL that must be preserved by all interpreters of a BPFL description. Interpreters can provide additional functionality as long as these basic semantics are not violated.

Each step of a process specifies an operation to be carried out according to the process specification. The interpreter analog of a step is *evaluation*. Evaluation extracts a *result value* from another BPFL value, according to the contents of the value being evaluated. For example, the result of evaluating a variable is the value of the variable. It is illegal to evaluate objects, vectors, and arrays. The next three sub-sections describe the behavior of constants, variables, and function calls when they are evaluated. The remaining sub-sections describe functions that should be supported by all interpreters.

3.1. Constants

Constants evaluate to the value they denote (i.e., they *self-evaluate*). When a number (e.g., integer, ratio, or floating point) is evaluated, the result value is just the number (except that integral valued ratios may be converted to integers during evaluation). Strings, unit values, intervals, and keywords also self-evaluate. The symbols `t` and `nil`, representing the boolean values true and false, are the only self-evaluating symbols.

3.2. Variables

A non-keyword symbol can have a value. This value is returned when the symbol is evaluated. If the symbol has not had a value assigned to it, an error results. The special function `setf` (described below) is used to set the value of a symbol. The `quote` spe-

cial function may be used to return a symbol as the result of evaluation rather than the symbol's assigned value.

There are two classes of variables: *local* and *global*. Local variables include the formal arguments of a function. When a symbol is declared a local variable, local storage is allocated to hold the value of the variable. The same symbol can be used in different functions to name a local variable. These variables (i.e., storage locations) are distinct. Thus, a symbol has an assigned value for each function that it is used in. A value must be assigned in each function before the variable can be used.

A symbol that is declared `special` is a global variable. This variable can be accessed in any function. It is an error to declare a variable `special` when a local variable with the same name is being used. Global variables must also be initialized before use.

3.3. Function Calls

Evaluating a list causes a function call. There are three types of functions that can be called: *built-in* functions (i.e., those implemented directly by an interpreter, such as `format`, or `keywordp`), *user-defined* (i.e., a function defined by `defflow`, `defgeneric`, or `defequipment`), and *object methods* (i.e., a function bound by a class to a method name with `defflow-method`, `defgeneric-method`, or `defequipment-method`). The actual arguments to a function are determined in the same way for all types of functions, except for *special* built-in functions. The result of evaluating a list is the value returned by the function called.

The evaluation a function call is composed of the following steps:

1. Checking for special functions (i.e., those that do not evaluate their arguments

in the usual way),

2. Evaluation of actual arguments,

3. Determination of the function to be called,

4. Initialization of formal arguments and local variables, and

5. Execution of the function.

Given the function call list, the header symbol is examined. If it is a special built-in function, arguments are not processed in the usual manner. Special functions are discussed below. Otherwise, each argument is evaluated and the result becomes an argument to the function. In the case of keyword arguments, the value is evaluated and the name is not. The division between positional and keyword arguments is determined before evaluation begins (i.e., it is syntactically determined). Thus, there must be an even number of elements in the function call list following the positional arguments.

The header symbol of the list may or may not indicate which function will be called, depending on the interpreter. If the header symbol of the call list is recognized by the interpreter the function is *built-in* to that interpreter. If the function is not built-in and the keyword argument "`:implemented-by`" is given, the argument value specifies the user-defined function that will be called as follows. If the value is a symbol, it names the function. If the value is an object, the keyword argument `:method` specifies a method name. If the `:method` argument is not given, then the header symbol names the method. The function bound by the object's class under that method name is then called. If the "`:implemented-by`" argument is not given, the header symbol names the user-defined function that will be called. It is an error to evaluate a list when there is no built-in or user-defined function that can be called. Some examples of function determination may be found in Figure 2.

```
(if tube-available tube-name)
```
The header symbol `if` is a special function and the arguments are treated specially.

```
(+ line-width 1)
```
The + function is built-in to all interpreters. The first actual argument is the value of the variable `line-width`.

```
(grow :thickness #u(1000 ang) :implemented-by 'CMOS-init-oxide)
```
If `grow` is not built-in to the particular interpreter, then the function `CMOS-init-oxide` is called.

```
(cmos-init-oxide :thickness #u(1000 ang))
```
This is equivalent to the previous example, except when the `grow` function is built-in to the interpreter.

```
(furnace :sequence '((temp #u(950 C)) (wait #u(1 hr)))
         :implemented-by tube-object :method 'run-sequence)
```
If the value of the variable `tube-object` is an object, then the `run-sequence` method for the object's class is the function to be called.

Figure 2 – Examples of function determination.

Once a function body has been found for execution, the actual arguments are assigned to the formal arguments of the function. The variable-specifiers in the first half of the formal argument list (i.e., the argument-declaration list) name a sequence of variables, each of which is assigned a value, in order, as follows. If there are any unused positional arguments, the first one is marked used and its value is assigned to the variable. Otherwise, if there is a keyword argument with a name corresponding to the variable symbol, the value of that argument is assigned to the variable. Otherwise, if the variable-specifier for the variable is a list and contains an initialization form, the initialization form is evaluated and the resulting value is assigned to the variable. If there is no initialization form, the variable is initialized to `nil`. If a *was-supplied* variable is specified, it is initialized to `t` if the corresponding variable was initialized from the actual argument list. Otherwise, it is initialized to `nil`. It is an error if more positional arguments are supplied than there are formal arguments. It is not an error to supply a keyword argument that names a variable that is not formally declared.

When initializing the variables for a method, there is one small difference. The `self` argument, which must always be the first formal argument, is initialized to the object that was used in the method invocation. All other arguments are initialized from the actual argument list as usual.

The argument passing mechanism in BPFL is considerably different from the mechanism in Common Lisp. In the usual case, user-defined functions will either take all positional arguments or all keyword arguments, but in any case, all arguments are optional. Users may use *was-supplied* variables or default values to detect when arguments have not been given. It is expected that keyword arguments and default values will suffice for a large number of user-defined functions.

After the formal arguments have been initialized, the local variables are initialized in order of specification. If an initialization form is given, the form is evaluated and the result is used to initialize the variable. Otherwise, the variable is initialized to `nil`. *Was-supplied* variables for local variables are vacuous and are ignored. Initialization forms may reference any local variable (or formal argument) that has already been initialized. `Special` declarations are processed before initializing any formal arguments, and thus global variables may be used by any initialization form.

Once the arguments and variables have been initialized, the function is evaluated. The declarations and documentation part of the function body is completely ignored and is not evaluated in any way. Of course, the `special` declarations must be examined, but they are not evaluated in the usual sense. The code part of the function body is treated as a call to `progn`, which is described below. The result of this implicit `progn` is returned as the value of the function call.

3.4. Special functions

Some built-in functions do not evaluate arguments in the usual way. These functions are listed below. This set is exclusive – no other functions will evaluate arguments abnormally. These functions must be built-in to all interperters. Future extensions of this set will be minimal, since the cost of implementing special functions in interpreters is very high. If one of these special functions does not evaluate an element of the argument list, the element itself is used as an argument instead of the result of evaluating it.

`quote` *arg* [*Special Function*]

The `quote` function takes one argument and does not evaluate it. The return value is the un-evaluated argument. For convenience, the function call "`(quote x)`" may be written "`'x`". When evaluated, this list would return the symbol `x`.

`setf` *where what* [*Special Function*]

`Setf` takes two arguments. The second is evaluated, the first is not. Setf changes the interpretation environment so that if the first argument had been evaluated, then the result would be the value of the second argument to `setf`. Thus, if the first argument is a symbol, its assigned value becomes the second argument to `setf`. If the first argument is a function call (i.e., a list) on a function known to `setf`, then things are suitably modified such that the function call would return the desired value. `Setf` returns the value of its second argument.

`and` *arg ...* [*Special Function*]

Arguments are evaluated one at a time. As soon as an argument evaluates to `nil`, evaluation stops and the function returns `nil`. Remaining arguments are not evaluated.

If all arguments are evaluated, then the function returns the value of the last argument evaluated.

or *arg ...* [*Special Function*]

Arguments are evaluated one at a time. As soon as an argument does *not* evaluate to `nil`, evaluation stops and the function returns the value of that argument. Remaining arguments are not evaluated. If all arguments evaluate to `nil`, then the function returns `nil`.

if *condition if-true* [*Special Function*]
if *condition if-true if-false*

`If` evaluates its first argument. If the result is not `nil`, (i.e., false) then it evaluates its second argument, otherwise it evaluates the third, if any. The result value is the value of the last argument evaluated.

progn *arg ...* [*Special Function*]

The arguments are examined in order. If an argument is a string, it is ignored and assumed to be documentation. If an argument is a list, it is evaluated as a function call. If an argument is an integer or a symbol, it is not evaluated and is used as a positional tag. (See the `go` function below.) There may be any number of arguments. `Progn` returns the value of the last function call evaluated, or else `nil` if no functions ever got evaluated. Arguments may be evaluated more than once or not at all if the `go` function is used. If evaluating an argument to `progn` causes another `progn` in the same function body to be evaluated, then the first `progn` "surrounds" the second. The code section of a function body is treated as if it was a list of arguments to `progn`. This is called

an *implicit* `progn`. A number of special functions make use of an implicit `progn`.

go *tag* [*Special Function*]

This function causes the point of evaluation to be transferred within a `progn`. The *tag*, which is not evaluated, must appear in a function body or a `progn` surrounding the call to `go`. The `progn` will then begin evaluating the arguments that follow the tag in its list of arguments. If the tag appears in more than one `progn`, control is transferred to the "closest" `progn` (i.e., most recently began evaluating) to the `go`. If control is transferred to the end of the argument list for a `progn`, then the `progn` returns `nil`.

return *value* [*Special Function*]

The function currently being executed is caused to return the given *value*, or `nil` if no value is given. Even though `return` evaluates its argument, it is considered a special function because of its affect on function evaluation.

while *condition args ...* [*Special Function*]

The *condition* is evaluated. If it does not evaluate to `nil` (i.e., false), then the remaining arguments are treated as as call to `progn`. When the implicit `progn` returns, the condition is evaluated again and the process repeats. `While` always returns `nil`.

cond (*condition consequent ...*) ... [*Special Function*]

The arguments to `cond` are never directly evaluated. Each argument should be a list and they are considered in order. The first element of an argument is evaluated. If the result is `nil`, (i.e., false), `cond` skips to the next argument and begins again. (If there are no more arguments, `cond` returns `nil`). If the condition returns true, then the

remainder of the argument is evaluated as an implicit `progn`. The result of the implicit `progn` is returned by `cond` and no more arguments are considered.

`case` *subject* (*tag consequent ...*) ... [*Special Function*]

The *subject* is evaluated and should return a number, symbol, or keyword. This value is used to select one of the remaining arguments, which should be lists and are not evaluated. The first element of one of the remaining arguments should be a single item or a list of items. This element is not evaluated. If the subject item is equal to one of the tag items, then the remainder of the list is evaluated as an implicit `progn` and the result of the `progn` is the result value of the `case` call. No more arguments are considered. A given tag should appear only once. The tag `t`, if it appears, must be in the last argument and its consequent is always evaluated if no other tags are selected. The subject item and the tags are compared with the `eql` function.

3.5. Value Copies

When a BPFL construct is evaluated, it returns a value. Unless the value is a symbol or an object, it is a copy of the value that generated the result. This copying occurs both for the evaluation of arguments and for the evaluation that results in the return value from a function. For example, consider the call `(setf a b)`. Assume the value assigned to `b` is not a symbol or an object. The symbol `b` is evaluated and a copy of its value is assigned to `a`. Thus, if the value of `b` were an array, future changes to the elements of the array stored in `a` would not be visible in the array stored in `b`. The function call to `setf` also returns the value which was assigned. This will be another copy, distinct from the values assigned to `a` and `b`. A side effect of evaluation copying values is that circular, or cyclic, data structures can not be created. For example, if the value of

the variable `b` is an array, then

```
(setf (elt b 0) b)
```

would set the first element of the array to be a copy of itself from before the assignment was made. The copying semantics are a significant difference between BPFL and Common Lisp. BPFL uses copying semantics to allow for database storage of interpreter state and to make data garbage collection easier on interpreters that don't have a built-in garbage collector.

3.6. Documentation

Strings may appear in the main function body or in the implicit `progn` in a special function. These strings serve no purpose than to store information about the process specification for human readers. Within function calls to non-special functions, the keyword argument `:doc` may be used. By convention, no functions will use a formal argument named `doc`, and so this argument will always be ignored.

A more stylized form of documentation may be introduced with the `:tag` keyword argument to non-special functions. The argument should be a symbol or a string that mnemonically identifies the step. This is different from the `:doc` argument because the tag does not necessarily describe the operation. Tags may be used to help interpreters locate a step of interest. If tags are used in a consistent manner, some functions may use `tag` as a formal argument in order to record the calling step.

3.7. Lot Control

A run associates a process specification and a set of wafers to be processed. Each wafer is assigned an index, which is used for identification of wafers by the process

specification. Wafer indicies may be grouped into named sets called *lots* or *sub-lots*, depending on context. A set of wafers may be specified by a list whose elements are integers specifying wafer indicies, integer intervals, or symbols naming sub-lots. Using an integer interval is equivalent to specifying all integers on that interval. Using a symbol is equivalent to specifying all indicies contained in the named sub-lot. Individual interpreters may choose whether or not it is an error to use an unregistered sub-lot name in a wafer-set specification. If it is not an error, then the sub-lot is assumed to contain no wafers. If a wafer-set specification list consists of just one symbol, then that symbol may be used in place of the list. Thus, a wafer-set specification is either a symbol or a list.

When a run starts, it contains no wafers. Wafers are allocated to a run with the `allocate-lot` function. As each wafer is allocated, it is assigned an index for the life of the run. The index could also correspond to a position in a cassette or to an inscription on the wafer. Lot names may be registered at allocation time or with `set-lot` function. Two special lot names are always available. The first is `all-wafers`, which contains all wafers that are currently allocated. The second is `current`, which is the set of wafers that are currently being operated on. Each function being evaluated in a run has a separate definition of the `current` lot. The initial contents of the `current` lot for a function is determined when the function is first called. In the general case, the contents of the `current` lot for the calling function is copied to the `current` lot for the function being called. If the function being called is not special and the keyword argument `:lot` is given, the `current` lot for the function being called is set to the value of that argument. The `:lot` argument should be a wafer-set specifier. If it contains symbols or sub-lot names, these are expanded to their corresponding set of wafer indicies. The `current` lot is initialized for a function before formal arguments are

initialized. Once a function has begun evaluation, the `current` lot may be changed with the `set-current-lot` function. If the `current` lot is assigned to a wafer-set specifier of `nil`, the steps carried out will not affect any wafers. This feature may be used to specify equipment preparation. In the usual case, the `:lot` argument will only be used at the higher process levels of the specification. For example, photolithography steps are not performed on test wafers. Thus, the sub-lot containing production wafers would be specified for calls on a function that performed photolithography operations.

If a wafer breaks (i.e., the interpretation environment is a real production facility), the index of the wafer is removed from all sub-lots that contained it. If all of the wafers in a lot are broken, the sub-lot name is not invalidated.

`set-lot` *name index-list* [*Function*]

The first argument should be a symbol and is registered as a sub-lot name. If the name is already in use, the old registration is destroyed. The index-list is a wafer-set specification and the sub-lot name is registered as containing the specified wafers. The wafer-set specifier is expanded before the old sub-lot registration is destroyed. Thus, a sub-lot name may be redefined in terms of its old contents. The `all-wafers` and `current` lots may not be redefined with `set-lot`. If the index-list is `nil`, the sub-lot name is invalidated. Changing the definition of a sub-lot name will not affect the contents of other lot names. Wafer-set specification lists that contain sub-lot names are expanded to a set of indicies at the time of their use. Thus no connection is made

between a wafer set and the sub-lots that were used to define it.

set-current-lot *contents* [*Function*]

The current lot (i.e., the one that process steps will operate on in the current function) is changed to the set of wafers specified by the argument. The argument is a wafer-set specification.

allocate-lot :size *lot-size* [*Function*]
:type *doping*
:resistivity *value*
:crystal *orientation*
:quality *list*
:preparation *list*
:names *list*

Wafers are added to the run lot and assigned sub-lot names. This function should be called once at the beginning of a process-flow, although equipment-specific functions can request test wafers for local monitoring. The :size is the number of wafers to be allocated. :type should be the symbol p or n, or a list whose first element is one of these symbols. The remainder of the list would indicate doping type and concentration. The :resistivity is the electrical resistivity and should be given as a unit value or an interval of acceptable unit values. :crystal is one of the symbols <100> (the default) or <111>. The :quality argument is a list whose elements would indicate oxygen content or the manufacturer. The :preparation argument is a list whose elements dictate the steps that have already been performed on the wafers (e.g., an oxidation step). No formalisms for specifying this information exist yet, so strings for human readers are sufficient. The :names argument is a list whose elements are lists, each of which defines a sub-lot. The first element of a sub-lot definition list is a symbol that names the sub-lot. The remainder of the definition list consists of either integers or

integer intervals that indicate the index numbers of the wafers within the group being allocated. The first wafer is numbered 0 for each call to `allocate-lot`. Note that these are not the indicies that will be assigned by `allocate-lot` and used throughout the remainder of the process specification. The lots defined with the `:names` argument can only refer to the wafers currently being allocated. The `all-wafers` and `current` lot names may not be defined with the `:names` argument. The function returns a list containing the permanent indicies assigned to each wafer. This list will not use integer intervals. No wafer will be assigned an index that has been assigned previously.

`lot-indexes` *name* [*Function*]
`:interval` *arg*
`:names` *arg*

The name argument is a symbol naming a registered sub-lot. A list of the indicies of the wafers in the lot is returned. Unless the `:interval` argument is given and not `nil`, none of the list elements will be integer intervals. If the given lot name has not been registered or if all of the wafers in the lot have broken, the function returns `nil`. The call "`(lot-indexes 'all-wafers)`" will return `nil` before the first call to `allocate-lot`.

`wafer-identifiers` *arg* [*Function*]

The argument is a wafer-set specification. The return value is a corresponding list of wafer identifiers (e.g., serial numbers). The list elements will be either symbols or strings. This function is only useful when the interpreter is controlling physical produc-

tion or a detailed simulation.

lot-names [*Function*]

This function returns a list of all currently valid sub-lot names. The names `all-wafers` and `current` are not included. The names of sub-lots that have attritioned to empty sets, but have not been explicitly invalidated, are included.

deallocate-lot *arg* [*Function*]

The argument is a wafer-set specification. The indicated wafers are removed from the processing run as if they had been broken. This is how temporary test wafers may be forgotten. In addition, any lot names used in the wafer-set specification are invalidated.

3.8. General Functions

The following are built-in functions that should be supported by all interpreters. They are taken from Common Lisp in most cases. Any extensions to this set should try to follow Lisp when possible. General functions can be used by functions at any process level.

3.8.1. Type Predicates

integerp *arg* [*Function*]
ratiop *arg*
floatp *arg*
numberp *arg*
stringp *arg*
symbolp *arg*
keywordp *arg*
objectp *arg*
listp *arg*
arrayp *arg*
vectorp *arg*

bit-vectorp *arg*
unitp *arg*
intervalp *arg*

If the given argument is of the appropriate type, return `t`; otherwise, return `nil`. `Numberp` returns `t` if its argument is either an integer, a ratio, or a floating point number. `Arrayp` returns `t` if its argument is either an array, vector, or bit-vector. `Vectorp` returns `t` if its argument is either a general vector or a bit-vector. Both `symbolp` and `listp` will return `t` if their argument is `nil`.

3.8.2. Data Access Functions

The values stored in the slots of an object may be retrieved with access functions defined for the class of the object. Since BPFL does not define any classes, it does not prescribe any access functions other than `object-name`. Each interpreter will have built-in access functions for the classes that it supports.

object-name *object* [*Function*]

The argument is an object of any class. If the object has a name, then it is returned as a symbol or a string. A symbol is preferred when possible. If the object has no meaningful name, the return value is unique to that object, but otherwise arbitrary.

numerator *ratio* [*Function*]

The argument must be a ratio data value and the numerator (an integer) is returned.

denominator *ratio* [*Function*]

The argument must be a ratio data value and the denominator (a positive integer) is

returned.

symbol-value *symbol* [*Function*]

The actual argument is a symbol. Return the assigned value for the symbol, just as if the symbol had been evaluated. Note that the call "`(symbol-value a)`" does not return the value assigned to `a`, but the value assigned to the symbol that is assigned to `a`. This function may be used with `setf`.

unit-value *unit* [*Function*]

Return the magnitude (numeric) part of the argument, which must be a unit value. This function may be used with `setf`.

unit-designator *unit* [*Function*]

Return the unit-designator (a symbol) part of the argument, which must be a unit value. This function may be used with `setf`.

interval-lower *interval* [*Function*]

This function returns the first part of its interval argument, which is assumed to be the lower half. This function may be used with `setf`.

interval-upper *interval* [*Function*]

This function returns the second part of its interval argument, which is assumed to be the upper half. This function may be used with `setf`.

elt *construct n* [*Function*]

The *construct* is either a vector or a list; *n* is a non-negative integer. The function returns the *n*th element of the vector or list. The first element has index `0`. The index must not attempt to access an element beyond the end of the list or vector. This function may be used with `setf`.

`first` *list* [*Function*]
`second` *list*
`third` *list*
`fourth` *list*
`fifth` *list*
`sixth` *list*
`seventh` *list*
`eighth` *list*
`ninth` *list*
`tenth` *list*

Return the *n*th element of the list. If the list doesn't have enough elements, return `nil`. Note that these functions use ordinal indexes, as opposed to the cardinal indexes of `elt`. These functions are provided to emphasize how a list is being accessed. These functions may be used with `setf`.

`rest` *list* [*Function*]

The argument should be a list. The return value is a list that contains all the elements of the argument list except the first. If the list does not have more than one element, then return `nil`. This function may be used with `setf`.

`length` *arg* [*Function*]

The argument must be a list or a vector. The function returns the number of ele-

ments in the list or vector.

aref *array subscript ...* [*Function*]

The *array* is either an array or a vector. For a vector, there is just one *subscript* argument; for an array, there should be as many subscript arguments as there are array dimensions. Arrays are accessed in row-major order. The result of "`(aref '#2a((a b) (c d)) 0 1)`" is `b`. This function may be used with `setf`.

array-dimensions *array* [*Function*]

The argument is an array. The return value is a list whose length is the rank of an array and whose elements are the size of each dimension. Since a vector is just a one-dimensional array, `array-dimensions` would return a list whose sole element would be the length of the vector.

string *arg* [*Function*]

The argument must be either a string or a symbol. The function always returns a string. If the argument is a string, that string is returned. If the argument is a symbol, then a string containing the name of the symbol is returned. Whether the string for a symbol uses upper case or lower case characters depends on the interpreter. The function `string-equal` may be used to compare strings independent of case. Thus, the call `(string-equal "yes" (string 'yes))` will always return `t`. A keyword symbol is given, then the resulting string does not have the initial colon ("`:`").

subseq *arg lower upper* [*Function*]

The first argument must be a list, vector, or a string. (The first element has index `0`, as with `elt`). The second argument specifies an index into the vector or list. The

second argument is optional and specifies another index greater than or equal to the first. The result is a value of the same type as the argument (list, string, general vector, or bit-vector). The result contains all elements of the first argument with indicies greater than or equal to the first argument and strictly less than the third argument. If the third argument is not given, then all elements up to the end of the list, vector, or string are present in the result. If the second and third arguments are equal, the result is an empty list, vector, or string. Neither of the index arguments should be greater than the length of the first argument. This function may be used with `setf`, but the length of the item may not be changed.

3.8.3. Expressions

When math operations are performed on two numbers of different types (e.g., an integer and a ratio), the type of the result will be determined as follows. If a floating point number is involved, the result will be floating point. If only ratios and integers are involved, the result will be a ratio, unless the result value is integral, in which case the result will be an integer. When comparing numbers, if a floating point value is present, everything will be coerced to floating point.

When operating on two unit values, the unit designators should be the same. BPFL does not require that interpreters support automatic unit conversion. However, if unit conversion is supported, some semantics must be observed. When a function requires that two unit values have the same unit designator, the interpreter may attempt to scale one of the values so that it has the same unit designator as the other. This is similar to numeric coercion in that the user's original value is not modified.

With respect to an interval, a *scalar* number is either a number (i.e., integer, ratio, or floating point) or a unit value.

not *arg* [*Function*]
null *arg*

Return `t` if the argument is `nil`; return `nil` otherwise.

and *arg ...* [*Special Function*]
or *arg ...*

These functions were described above in the section on special functions. They perform boolean operations and only evaluate enough arguments to determine the result.

+ *arg1 arg2 ...* [*Function*]

The arguments are added. If the result overflows the range expressible by the appropriate data type, then the result is undefined. Unit values may be added if they both have the same unit designator. An interval may be added to a scalar. The result is an interval where both the lower and upper halves have had the scalar added in.

- *arg1* [*Function*]
- *arg1 arg2*

If one argument is given, the number or the magnitude of the unit value is negated. If the argument is an interval, then both halves are negated and their positions are swapped to maintain ordering. If two arguments are given, the second argument is subtracted from the first. Unit values may be subtracted if they both have the same unit designator. If one argument is an interval and the other is a scalar, then the result is the

same as negating the second argument and performing an addition.

** arg1 arg2 ...* [*Function*]

The arguments are multiplied. If the result overflows the range expressible by the appropriate data type, then the result is undefined. A unit value may be multiplied by a scalar number. Two unit values may be multiplied if the interpreter supports unit conversion. An interval may be multiplied by a scalar. The lower and upper halves of the interval are each multiplied and the result is an interval. If the scalar is negative, then both arguments are negated before multiplying.

/ arg1 arg2 [*Function*]
/ arg

The first argument is divided by the second. A unit value may be divided by a scalar number. Two unit values may be divided or a scalar may be divided by a unit value if the interpreter supports unit conversion. If the arguments are integers and they do not evenly divide, then the result is a ratio with a numerator of *arg1* and a denominator of *arg2*. If the arguments are an interval and a scalar number, then the result is an interval where the lower and upper halves of the argument interval participated individually in the division. If the scalar was negative, both arguments are negated before dividing. If only one argument is given, then the argument is reciprocated.

= arg ... [*Function*]
/= arg ...
< arg ...
> arg ...
<= arg ...
>= arg ...

The arguments must either all be numbers or all be unit values. If the arguments are unit values, the unit designators must all be the same. At most one argument may be an interval over a type compatible with the rest of the arguments. The functions test the following conditions:

=	– all arguments are numerically equal.
/=	– each argument is different from all the rest.
<	– arguments are monotonically increasing.
>	– arguments are monotonically decreasing.
<=	– arguments are monotonically nondecreasing.
>=	– arguments are monotonically nonincreasing.

The functions return `t` if the condition is satisfied and return `nil` otherwise. Type coercions are made to compare numbers. If only one argument is given, then the condition is always true. Intervals may be compared to scalar numbers. A scalar is equal to an interval value if it is on the closed interval defined by the upper and lower halves of the interval value. A scalar is strictly less than an interval if it is less than the lower half of the interval value. A scalar if strictly greater than an interval value if it is greater than the upper half of the interval value.

eq *arg1 arg2* [*Function*]
eql *arg1 arg2*
equal *arg1 arg2*
equalp *arg1 arg2*

Two arguments are `eq` if and only if they are the same symbol or object. Two arguments are `eql` if they are `eq` or if they are numbers of the same type and value. The = function is recommended for numeric comparisons. Two arguments are `equal` if they are `eql`. Two strings are `equal` if they match character for character. Two unit values or two intervals are `equal` if their components are `equal`. Two lists or bit-vectors are `equal` if their corresponding elements are `equal`. Arrays and general

vectors are never `equal`. Two arguments are `equalp` if they are `equal` or if they have isomorphic structures whose components are `equalp`. Thus, two arrays with the same number and sizes of dimensions are `equalp` if all of their corresponding elements are `equalp`.

string= *string ...* [*Function*]
string-equal *string ...*

These two functions compare their arguments, which should all be strings. `String=` returns `t` if its arguments match character for character, as with the function `equal`. `String-equal` returns `t` if its arguments match character for character or if they only differ by the case of the alphabetic characters. Both functions return `nil` otherwise. Thus `(string= "Hello" "hello")` returns `nil` and `(string-equal "Hello" "hello")` returns `t`.

random *integer* [*Function*]

The argument is a positive integer. The return value of the function is an integer greater than or equal to `0` and less than the argument. The number returned is chosen from a uniform distribution over the possibilities.

3.8.4. Constructors

These functions create data values from their components. Ratios are created with the "`/`" function.

list *arg ...* [*Function*]

The arguments are made into a list. The list will have as many elements as there were arguments given. If no arguments are given, then an empty list (i.e., `nil`) is

returned.

listify *arg* [*Function*]

If the argument is a list, then that list is returned. Otherwise, a list containing that one value is returned. This is useful when it is desirable to have a list.

make-interval *lower upper* [*Function*]

This function returns an interval whose bounds are the given arguments. The arguments must have the same type.

make-unit *magnitude unit-designator* [*Function*]

This function returns a unit value whose components are the given arguments.

make-array *dimensions initial-contents* `:bit` *arg* [*Function*]

The first argument is a list whose length is the rank of the array. Each element of the list is a positive integer that defines the size of the dimension corresponding to the element's position in the list. The dimensions are given in the same order that `aref` subscripts access arrays. If the array has just one dimension (i.e., it is a vector), then the dimensions argument can be an integer instead of a list. An array cannot have zero dimensions, although a vector may have zero length.

The second argument is optional and defaults to `nil`. If the second argument is `nil`, the array or vector elements are initialized to `nil`. Otherwise, the second argument should be a list of the same form as the initializer list of constant arrays and vectors (i.e., those written with a hash mark ("#")).

If only one dimension is given and the `:bit` keyword argument is given and is not `nil`, then a bit-vector is returned instead of a general vector. The bit-vector elements are initialized to `0` where general vector elements would be initialized to `nil`.

`vector` *arg ...* [*Function*]
`bit-vector` *arg ...*

These functions are similar to the `list` function. The arguments are gathered into the elements of a vector. The vector has the same length as the number of arguments given.

`concatenate` *arg ...* [*Function*]

The arguments are concatenated. The arguments should either be vectors, lists, or strings. Bit-vectors and strings may only be used with similarly typed arguments. General vectors and lists may be concatenated with each other and the result has the same type as the first argument. Note that this function operates differently than the Common Lisp function.

3.8.5. Miscellanea

`abort` *message* [*Function*]

This function never returns. The interpreter stops or suspends evaluation of the process-flow. This function is used when a process specification detects a situation that is undesirable or prevents further processing. The optional argument is a string that will

help explain what went wrong.

format *control-string arg ...* [*Function*]

Return a string similar to *control-string*, but with the other arguments inserted according to directives. This function is modeled after the Common Lisp function, which has a large number of directives and formatting options. An interpreter should approximate the Common Lisp functionality as much as possible, although this is not necessary in most cases. It is not an error to attempt to print an object, although the contents of the object need not be printed.

Values, other than objects, printed with the `~S` directive should be printed according to BPFL syntax. For example, strings would be printed with surrounding double quotes. Values printed with the `~A` directive should be more readable by people. Strings should be printed without the surrounding double quotes. Intervals should be printed as "*lower-value* `to` *upper-value*". Unit values should just be printed as "*magnitude unit-designator*".

interpreter *name* [*Function*]

This function returns `t` if the given name, which is a symbol or a string, is known to the interpreter executing the function. The function returns `nil` otherwise. This function can be used with the `if` special function to execute functions known only to a particular interpreter. In this way, a process specification can provide special purpose functions and still be used by a number of different interpreters. Examples of symbols to use with this function are `fabrication`, `scheduling`, `simulation` (when simulating a fabrication facility), or the name of a process simulator such as `suprem` or `simpl`.

3.9. Environment functions

These functions interact with the environment of the BPFL interpreter. Most of these functions are designed for process specifications that will be used to control production.

`wait-for` *condition* [*Function*]

The *condition* is a data value that will be recognized by an external monitoring system. This function is primarily used when then process specification is controlling equipment. An example condition might be `(done tube-7)`.

`user-dialog` `:title` *string* [*Function*]
`:summary` *string*
`:detail` *string*
`:reference` *string*
`:query` *string*
`:form` *object*

This function allows interaction with an operator in a fabrication facility or the user of an interpreter. The function can be used to display text or to display a form. In either case, the `:title` is required and should be a string indicating the nature of the conversation. The `:summary` and `:detail` arguments are strings or lists of strings (with newlines possibly embedded) that will be displayed as the user requests. Two strings in a list have an implicit newline between them. The `:query` argument is a string or a list of strings. Each string is presented to the user and the user enters a single BPFL value (which could include a list). The `:form` argument names a form to be displayed. A form can contain both information and data fields for the user to fill. This argument will generally not be used if the other arguments are used. Forms are assumed to already exist in a database. The return value of `user-dialog` is a list containing one element

for each query made. The return value if a form was used depends on the forms support system used by the interpreter.

As a side effect of presenting the requested information to the user, the function may also append such information as the run indentifier, the name of the function making the call, the wafers in the `current` lot, or any other information relevant to the interpretation environment.

`allocate` *what* [*Function*]

If it is physically processing wafers, this function allocates equipment and resources to a run. In any case, it indicates what equipment is required by the process specification. The argument is either a symbol or a list of symbols. The symbols name equipment. If a list is given, then any of the pieces of equipment will do. The return value is an equipment object representing the equipment that was allocated.

If more than one resource is needed, the argument can be a list whose first element is the symbol `and`. The remaining elements should be symbols or lists of symbols as described above. The return value will be a list of equipment objects, one taken from each request.

`deallocate` *what* [*Function*]

The given object, or list of objects, as obtained from the `allocate` function, is removed from the run's set of resources.

`log` *destination args ...* [*Function*]

This function puts information contained in the arguments into a history record. If the first argument is a symbol, then it names an object class. The remaining arguments

are used to instantiate a member of that class and the resulting object is stored in a database. If the first argument is a string, then it names a disk file. The remaining arguments are used in an interpreter dependent way to append information to the file. An interpreter should automatically include the time, run indentifier, calling function name, and current lot composition in log records when appropriate.

3.10. Flow Level Functions

The following functions are intended for use as *basic steps* at the flow process level. Some interpreters will consider these to be built-in functions. Others will need an `:implemented-by` argument specifying a generic or equipment level function. All of these functions state goals to be achieved on the wafers being processed. All of these functions take keyword arguments, many of which are optional.

Some of these functions specify a material to be deposited or removed. In the simple cases, this should be a symbol, such as `silicon`, `silicon-nitride`, `silicon-oxide` (as opposed to `silicon-monoxide`), `poly-silicon`, `photo-resist`, `aluminum`, `boron`, or `phosphorus`. Variations on a material, such as doping or impurities, and specializations, such as specifying atomic weight or density, should be described with a list. The first element of the list is the primary material and the remaining elements of the list are lists that add information. Each additional list starts with a symbol specifying the attribute and the remaining elements give the value or degree of the attribute. Whether a material is a specialization of another material or whether it deserves its own symbol is a matter of convention. Different molecules, and often different phases (`nitrogen` *vs* `liquid-nitrogen` and `water` *vs* `steam`), use different symbols. Whether elements such as `nitrogen`,

oxygen, or chlorine are atomic or diatomic must be determined by context or the presense of an ionization specialization attribute.

Examples:

```
(boron (atomic-weight 11))
(arsenic (ionization +1))
(photo-resist (polarity positive) (brand kodak-820))
(poly-silicon (dope boron #u(1e14 /cm^2)))
```

anneal :modify *property* [*Function*]
:contact-resistance *value*
:etch-rate *value*
:resistivity *value*

Change the properties of a target material. The :modify argument should be one of contacts, dielectric, or semiconductor. Depending on this argument, additional information may be given with the other arguments.

deposit :material *what* [*Function*]
:thickness *value*
:density *value*
:planarity *value*
:resistivity *value*
:crystal-specification *value*
:surface-state-density *value*

Deposit some material on the wafer. The :material argument should be a symbol, such as nitride. The :thickness argument should be a unit value specifying the desired thickness of the layer being deposited. The remaining arguments are optional and place restrictions on the properties of the layer being deposited.

drive-in :junction-depth *value* [*Function*]
:oxide-thickness *value*

Redistribute dopants. The target :junction-depth is given and a desirable

resulting `:oxide-thickness` may be specified as well.

etch `:material` *what* [*Function*]
`:thickness` *value*
`:undercut` *value*
`:selectivity` *value*

The given `:material` will be etched. The `:thickness` argument indicates how much will be *removed.* This argument may be a list where the first element is the symbol `all` and the second element is the expected thickness of the layer. The remaining arguments specify how much undercutting is acceptable and how selective the etching process must be to avoid other materials in adjacent or lower layers.

implant `:dopant` *what* [*Function*]
`:dose` *amount*
`:depth` *value*
`:orientation` *value*

Perform an ion implantation. The given `:dose` of `:dopant` ions are implanted over a region of the given `:depth`. If an ion beam is to be used, the the `:orientation` argument may be used to specify an incidence angle different from the default 7 degrees.

mask `:thickness` *value* [*Function*]
`:mask` *name*
`:polarity` *symbol*

Perform a photolithography deposition. Photoresist of `:thickness` depth is deposited. The `:polarity` will be either `positive` or `negative`. The polarity is between the original artwork and the final resist pattern. The `:mask` names the layer to mask. The mask set from which the mask is chosen should be bound to the global vari-

able *mask-set*.

measure :attribute *what* [*Function*]

The requested attribute of the lot is sampled and the results are stored away.

grow :compound *what* [*Function*]
:thickness *value*
:surface-state-density *value*

The requested :compound is grown to the desired :thickness. The compound is generally silicon-oxide. The :surface-state-density may be limited, if desired.

reflow :angle *value* [*Function*]

Smooth feature edges such that the maximum feature angle is :angle.

other [*Function*]

This function specifies a change in wafer state or topology that cannot be described with the previous functions. This should not be used for operations that have no major effect on a wafer. For operations such as cleaning, the generic or equipment level function should be called directly.

3.11. Generic Level Functions

The following functions are intended for use as basic steps at the generic-equipment process level. These functions may be built-in to some interpreters, but will require an :implemented-by argument in others to specify an equipment level function. Implementation may be either through an equipment object method, if an object has been allocated, or else through an equipment-specific function that will allocate equipment and

perform the desired operations.

Many of these functions take a `:sequence` argument. The `:sequence` argument specifies a recipe for the generic class of equipment. Recipes are fixed sequences of operations and are represented as a list. Each element of the list represents an operation appropriate to the class of equipment. The format of an operation is similar to a BPFL function call, but the operations are not evaluated by a BPFL interpreter. Control-flow operations will probably be rare, but should be modeled after BPFL special functions when present. An additional control operation is `repeat`, which takes a number for its first argument and a list of operations for its second argument. The list of operations is performed the specified number of times. The first operation of a recipe may be the `tag` pseudo operation. Its argument is a symbol that names the sequence of operations. It is quite possible that the `:sequence` argument will be ignored by implementing functions and only used for documentation.

`implanter` `:dopant` *what* [*Function*]
`:dose` *value*
`:energy` *value*
`:orientation` *value*

This describes the input to a generic ion implanter. The orientation defaults to 7 degrees. The dopant uses the materials specification format.

`furnace` `:sequence` *recipe* [*Function*]

This describes the average tube furnace. Some of the operations are `temp`, `gas`, `wait`, `pressure`, and `boat`. The arguments to `temp` are the target temperature and optionally a time value over which the ramping takes place. The arguments to `gas` are the gas to be used and the proportion it contributes to the total pressure. For operations at

atmospheric pressures, the proportion argument may simply be `on`, to indicate an ample supply (as with steam or nitrogen). The gas argument may be the symbol `all`, in which case all gasses of interest are affected. The `pressure` operation specifies the pressure to be maintained. If the pressure is the symbol `ambient`, then no pressure control is maintained. The `boat` operation takes the argument `in` or `out`. The `wait` operation simply extends the current conditions for the given amount of time. The default case is that the boat will push in, the temperature will ramp up in nitrogen and then the given recipe will start. After completion, the tube is cooled in nitrogen and the boat is pulled. The default ramp up does not occur if an initial `boat` operation is given. The default ramp down is not given if a final `boat` operation is given.

oven `:time` *value* [*Function*]
`:temperature` *value*

This is a medium temperature heating oven. No sequence argument is given because the number of major steps is almost always one.

plasma `:sequence` *recipe* [*Function*]

The operations for a plasma system are similar to those for a furnace. Additionally, there is an `energy` operation for striking a plasma. There is also an optional `end-point` operation for initializing an end-point detector.

spinner `:sequence` *recipe* [*Function*]

This function describes the operation of a spinner, such as is commonly used in photolithography steps. The recipe operations are `spin`, which takes a unit value specifying the frequency of revolutions, `temp`, which defaults to room ambient, and `dispense`, which takes a materials specification as its first argument and a flow rate as

its optional second argument.

wet-sink :sequence *recipe* [*Function*]

The operations for a wet process sink are `bath`, which takes a materials specification and an optional time argument, `wait`, which takes a time value argument, and `repeat-until`, which takes an argument that is used as a termination condition (e.g., `de-wet`), plus a sequence of operations to be performed repeatedly.

4. Example BPFL Specification

The following functions are an example of how the Berkeley CMOS process could be specified with BPFL. The first function, `cmos-nwell`, would be the first one called by an interpreter and its arguments would be given by the user. The function initializes some global variables and allocates wafers to be processed. It then calls a series of functions and specifies the wafers that they will operate on with the `:lot` argument.

```
(defflow CMOS-Nwell (analog-option (lot-size 10) mask-set)
  "3 um, N-well, single poly-Si, single metal"
  (declare (special *def-resist-thickness*)
           (special *mask-set*))

  ";; Set the global variable for use by low level routines"
  (setf *mask-set* mask-set)

  (allocate-lot :doc "Allocate a device lot and 3 test wafers"
                :size (+ lot-size 3)
                :type 'p
                :resistivity #i(#u(18 ohm-cm) #u(22 ohm-cm))
                :crystal '<100>
                :names (list
                        (list 'MAIN (make-interval 0 (- lot-size 1)))
                        (list 'WELL lot-size)
                        (list 'NCH (+ lot-size 1))
                        (list 'PSG (+ lot-size 2)))
                )

  "This parameter is determined by the mask-making functions"
  (setf *def-resist-thickness* #u(1.2 um))

  (CMOS-Well-Formation :lot '(MAIN WELL))
```

```
  (CMOS-Isolation-Formation :lot '(MAIN WELL))
  (CMOS-Threshold-Adjustment :lot '(MAIN WELL))
  (if analog-option
      (CMOS-Analog-Options :lot '(MAIN WELL)))
  (CMOS-Gate-Poly-Formation :lot '(MAIN WELL NCH))
  (CMOS-S/D-Formation :lot '(MAIN WELL NCH))
  (CMOS-Final-Steps :lot '(MAIN))
  "... At this point, CMOS-NWell returns and the run is terminated"
  )

(defflow CMOS-Well-Formation ()
  "This function groups together a set of flow-level steps"
  "that begin building the CMOS device."
  (declare (special *def-resist-thickness*))

  (grow :doc "Initial Oxidation"
        :tag 'init-oxide
        :compound 'silicon-oxide
        :thickness #u(1000 A)
        :implemented-by 'CMOS-init-oxide)

  (measure :doc "Measure oxide"
           :attribute 'oxide-thickness
           :lot 'WELL
           :tag 'init-oxide
           :implemented-by 'measure-oxide)

  (mask :doc "WELL mask -- note lot restriction"
        :mask 'WELL
        :polarity 'negative
        :thickness def-resist-thickness
        :lot 'MAIN
        :implemented-by 'CMOS-mask)

  (implantation :tag 'well-implant
                :dopant 'phosphorus
                :dose #u(4e12 /cm**2)
                :depth #u(.09 um)
                :implemented-by 'CMOS-implantation)

  (cmos-well-drive-in-clean)

  (etch :doc "Oxide Etch"
        :material 'silicon-oxide
        :thickness '(all #u(1000 a))
        :implemented-by 'CMOS-well-drive-in-etch)

  (etch :doc "Remove resist"
        :material 'photo-resist
        :thickness (list 'all *def-resist-thickness*)
        :lot 'MAIN
        :implemented-by 'CMOS-photo-resist-removal)

  (drive-in :junction-depth #u(3 um)
            :oxide-thickness #u(3000 a)
            :implemented-by 'CMOS-well-drive-in)
```

```
  (measure :attribute 'oxide-thickness
           :lot 'WELL
           :tag 'well-drive-in
           :implemented-by 'measure-oxide)

  (measure :attribute 'junction-depth
           :lot 'WELL
           :implemented-by 'measure-well-depth)
  )

(defflow CMOS-Isolation-Formation ()
  (declare (special *def-resist-thickness*))

  (etch :doc "Remove oxide from well drive-in"
        :material 'silicon-oxide
        :thickness '(all #u(3000 a))
        :implemented-by 'oxide-strip-bhf5/1)

  (grow :tag 'pad-oxidation
        :material 'silicon-oxide
        :thickness #u(200 a)
        :implemented-by 'dry-oxidation)

  (measure :attribute 'oxide-thickness
           :lot 'WELL
           :tag 'pad-oxide
           :implemented-by 'measure-oxide)

  (etch :doc "NOTE: only stripping control wafer"
        :material 'silicon-oxide
        :thickness '(all #u(200 a))
        :implemented-by 'oxide-strip-bhf5/1
        :lot 'WELL)

  (deposit :tag 'nitride-deposit
           :material 'silicon-nitride
           :thickness #u(1000 a)
           :implemented-by 'CMOS-deposit-nitride)

  (measure :attribute 'nitride-thickness
           :implemented-by 'measure-nitride
           :lot 'WELL)

  (mask :mask 'ACTV
        :thickness *def-resist-thickness*
        :polarity 'positive
        :implemented-by 'CMOS-mask
        :lot 'MAIN)

  (etch :material 'silicon-nitride
        :thickness '(all #u(1000 a))
        :selectivity '(not (or silicon-oixde photo-resist))
        :implemented-by 'plasma-etch-nitride
        :lot 'MAIN)

  (mask :doc "NOTE: double mask"
```

```
        :mask 'FDII
        :thickness *def-resist-thickness*
        :polarity 'positive
        :implemented-by 'CMOS-mask
        :lot 'MAIN)

  (implantation :tag 'field-implant
                :dopant 'boron
                :dose #u(1e13 /cm^2)
                :depth #u(0.4 um)
                :implemented-by 'CMOS-implantation
                :lot 'MAIN)

  (etch :material 'photo-resist
        :thickness (list 'all (* 2 *def-resist-thickness*))
        :implemented-by 'CMOS-photo-resist-removal
        :lot 'MAIN)

  (grow :tag 'locos-oxide
        :material 'silicon-oxide
        :thickness #u(5500 a)
        :implmented-by 'CMOS-locos-oxidation
        :lot 'MAIN)

  (measure :doc "Pick a random wafer from the device production lot"
           :attribute 'oxide-thickness
           :lot (list (nth (random (length (lot-indexes 'MAIN)))
                           (lot-indexes 'MAIN)))
           :tag 'locos-oxide
           )

  (etch :material 'silicon-nitride
        :thickness '(all #u(1000 a))
        :implemented-by 'CMOS-nitride-etch
        :doc "WELL control included again")

  )

(defflow CMOS-Threshold-Adjustment (&aux main-indexes main-size
                                         break1 break2)

  (grow :tag sacrificial-oxide
        :compound 'silicon-oxide
        :thickness #u(200 a)
        :implemented-by 'CMOS-sacrificial-oxide)

  "Divide the main production lot into thirds and"
  "give each sub-lot a slightly different dose"

  (setf main-indexes (lot-indexes 'main))
  (setf main-size (length main-indexes))
  (setf break1 (/ main-size 3))
  (setf break2 (+ break1 break1))

  (implantation :tag 'threshold-implant
                :dopant 'boron
```

```
                :dose #u(0.9e12 /cm^2)
                :depth #u(0.1 um)
                :lot (subseq main-indexes 0 break1))
  (implantation :tag 'threshold-implant
                :dopant 'boron
                :dose #u(1.0e12 /cm^2)
                :depth #u(0.1 um)
                :lot (subseq main-indexes break1 break2))
  (implantation :tag 'threshold-implant
                :dopant 'boron
                :dose #u(1.1e12 /cm^2)
                :depth #u(0.1 um)
                :lot (subseq main-indexes break2 main-size))
  )
```

The following functions are used to implement the flow level steps. Since there is one function defined per step, these functions may be thought of as specializations on the flow level specifications. All of these functions assume that they implment only the intended step and thus ignore their actual arguments. The functions could verify the actual arguments, but this is not done in this example.

```
(defgeneric CMOS-init-oxide (&aux tube)
  "Implement the flow level goal of 1000A of oxide on bare silicon"

  (setf tube (allocate :what '(tylan1 tylan2 tylan3 tylan4)))

  (furnace :doc "Clean the tube with TCA before use"
           :implemented-by tube
           :method 'run-recipe
           :lot nil
           :recipe "STCA")

  (std-clean-wafers)

  (furnace :implemented-by tube
           :method 'run-sequence
           :sequence '((tag cmos-init-oxide)
                       (temp #u(1000 C))
                       (gas oxygen on) (gas steam on) (wait #u(11 min))
                       (gas all 0)
                       (gas N2 on) (wait #u(20 min)))
           )

  (deallocate tube)
  )

(defgeneric cmos-well-drive-in-clean ()
  "Prepare for the cmos well-drive-in."
  "This furnace step is separated because flow level steps"
```

```
  "interveve between the clean and the drive-in"
  "Ideally, we could put this in the cmos-drive-in function and"
  "a scheduler would notice that the :lot was nil and migrate"
  "the function accordingly"
  (declare (special *drive-in-tube*))

  (setf *drive-in-tube* (allocate :what '(tylan1 tylan2 tylan3 tylan4)))

  (furnace :implmented-by *drive-in-tube*
           :method 'run-recipe
           :lot nil
           :recipe "STCA")
  )

(defgeneric cmos-well-drive-in ()
  (declare (special *drive-in-tube*))

  (std-clean-wafers)

  (furnace :implmented-by *drive-in-tube*
           :method 'run-sequence
           :sequence '((tag cmos-well-drive-in)
                       (temp #u(1150 c))
                       (gas oxygen on) (wait #u(4 hr))
                       (gas oxygen off) (wait #u(4 hr)))
           )
  (deallocate *drive-in-tube*)
  )

(defgeneric cmos-pad-oxidation (&aux tube)

  (setf tube (allocate :what '(tylan1 tylan2 tylan3 tylan4)))

  (std-clean-wafers)

  (furnace :implemented-by tube
           :method 'run-sequence
           :sequence '((tag cmos-pad-oxidation)
                       (temp #u(950 c))
                       (gas oxygen on) (wait #u(28 min))
                       (gas oxygen off) (wait #u(20 min)))
           )
  (deallocate tube)
  )

(defgeneric cmos-deposit-nitride (&aux tube)

  (setf tube (allocate :what 'tylan9))
  (furnace :implemented-by tube
           :method 'run-sequence
           :sequence '((tag cmos-deposit-nitride)
                       (temp #u(800 c))
                       (gas nitrogen off)
                       (pressure #u(35 mtorr))
                       (gas dichloride on)
                       (wait #u(22 min)))
```

```
        )
  (deallocate tube))

(defgeneric cmos-locos-oxidation (&aux tube)

  (setf tube (allocate :what '(tylan1 tylan2 tylan3 tylan4)))

  (furnace :implemented-by tube
           :method 'run-sequence
           :sequence '((temp #u(950 c))
                       (gas oxygen on) (gas steam on)
                       (wait #u(10/3 hr))
                       (gas oxygen off) (gas steam off)
                       (wait #u(20 min)))
           )
  (deallocate tube))
```

The following two functions are bound to methods in the `tylan` class. The `run-sequence` method is used with the generic equipment level `furnace` step. It expects recipes to start with the `tag` step so that it can look up the equipment specific recipe in a database. In this case, the database is contained in a `case` special function. The mapping of generic sequences to equipment recipes could also be done with a true database, or in the future, by compiling the generic sequence into an equipment-specific recipe. It would also be possible to use individual BPFL functions for each equipment-specific recipe and not use the object method.

```
(defequipment-method run-sequence ((self tylan) sequence
                                   &aux tag specifics)

  (if (eq 'tag (first (first sequence)))
      (setf (tag (second (first sequence))))
      (abort "Generic furnace recipe is untagged"))

  (setf specifics
        (case tag
              (cmos-init-oxide
               '("SWETOXB" "Time=11min, anneal=20min"))
              (cmos-well-drive-in
               '("SDRYOXA" "Time=4hrs, anneal=4hrs"))
              (cmos-pad-oxide
               '("SDRYOXA" "Time=28min, anneal=20min"))
              (cmos-deposit-nitride
               '("SNITC" ""))
              (cmos-locos-oxide
               '("SWETOXB" "Time=3hrs 20min, anneal=20min"))
```

```
                (t
                 (abort "Unknown recipe for given tag"))))

    (run-recipe :implemented-by self
                :recipe (first specifics)
                :parameters (second specifics))
    )

  (defequipment-method run-recipe ((self tylan) recipe parameters)

    (user-dialog
     :title "Load and start tylan tube"
     :summary (list (format "Load recipe ~a into tube ~a; parameters are"
                            recipe (object-name self))
                    parameters
                    "Load wafers into boat and start recipe")
     )

    (wait-for :what (concatenate "Tylan tube "
                                 (string (object-name self))))

    (user-dialog
     :title "Unload tylan tube"
     :detail
     (format "Alarm will sound when run is complete ~@
              Press ALARM ACK on ROP or type 'ACK tube#' to stop alarm. ~@
              Press the OUT button on the ROP. ~@
              Wait for wafers to cool ~@
              Press IN button on ROP.")
     )
    )
```

The following functions demonstrate possible uses of the `log` function. In both cases, we assume that a logging format has already been set up under the given object class names. Each call to log will create an instance of the class with the given values. For the `measure-oxide` function, the tag argument is used to mnemonically indentify which step within the process we are measuring. This function is equipment-specific because it assumes that the Nanospec measurment system will be used. The `nano-spec` function will return a value for the measured oxide thickness.

```
(defequipment measure-oxide (tag &aux
                                 (wafer-set (wafer-indexes 'current))
                                 result)

  "For each wafer in the current lot, measure and log the"
  "oxide thickness"
```

```
  (if (not (interpreter 'fabrication))
      (return))

  (while wafer-set
         (setf result (nano-spec :target 'silicon-oxide
                                 :lot (list (first wafer-set))))
         (log 'nano-spec-measurement
              :step tag
              :layer "oxide"
              :thickness result
              :lot (list (first wafer-set)))

         "Step through the list of wafers for the current lot"
         (setf wafer-set (rest wafer-set))
         )
  )

(defequipment measure-well-depth (&aux results)

  (if (not (interpreter 'fabrication))
      (return))

  (if (null (query-user-yn "Should the well depth (xj) be measured?"))
      (return))

  (setf results
        (user-dialog
         :title "Measure well depth"
         :summary '("Send the wafers to the valley for measurement.")
         :query '("Enter the average well depth:")
         ))
  (log 'cmos-well-depth
       :depth results)
  )
```

This function demonstrates how rework can become a fully controlled process.

```
(defgeneric cmos-mask ()

  (cmos-hmds-treatment)

  apply-resist
  (spinner :sequence '((tag resist-soft-bake)
                       (spin #u(4600 /min))
                       (dispense (photo-resist (brand kodak-820)))
                       (wait #u(25 sec)) (rpm 0)
                       (temp #u(120 c)) (wait #u(45 sec))
                       )
           :implemented-by 'cmos-resist/soft-bake)

  (cmos-expose-resist)

  (spinner :sequence '((tag develop-resist)
                       (spin 1.0)
                       (dispense developer) (wait #u(60 sec))
```

```
                        (spin 0.05)
                        (dispense water) (wait #u(20 sec))
                        (dispense off)
                        (spin 0.50) (wait #u(20 sec)))
            :implemented-by 'cmos-resist-develop)

  (if (null (cmos-inspect-resist))
      (progn
       (cmos-photo-resist-removal)
       (cmos-dehydration-bake)
       (go apply-resist)))

  (plasma :sequence '((tag resist-descum)
                      (pressure #u(35 mtorr))
                      (gas oxygen on)
                      (power #u(50 watt))
                      (wait #u(1 min)))
          :implmented-by 'cmos-technics-c)

  (oven :temperature #u(120 c) :time #u(20 min))
  )
```

This function demonstrates how simple variations on built-in functions can be made with BPFL functions. This function could also have been built-in to interpreters.

```
(defequipment query-user-yn (query-string &aux user-response)

  "This function returns t or nil according to the user's response."
  "Defined at the equipment level so that anyone can call it"

  (if (not (stringp query-string))
      (return nil))

  (setf query-string
        (concatenate query-string " (yes/no) "))

  retry-loop
  (setf user-response
        (user-dialog :title "Yes-no query"
                     :query query-string))

  "user-dialog will return a list; get the answer from the list"
  (setf user-response (first user-response))

  (if (or (eq user-response 'yes)
          (eq user-response 'y))
      (return t))

  (if (or (eq user-response 'no)
          (eq user-response 'n))
      (return nil))
```

```
"Retry until we get a reasonable answer"
(go retry-loop)
)
```