In: "Combinators and Functional Programming Languages"
G.Cousineau, P-L.Curien and B.Robinet Editors,
Lecture Notes in Computer Science n.242, Springer-Verlag, 1986

# The Amber Machine

*Luca Cardelli*[1]

AT&T Bell Laboratories, Murray Hill, NJ 07974

## Abstract

The Amber machine is a stack machine designed as an intermediate language for compiling higher-order languages. The current version is specialized for the Amber language. The machine supports a set of basic and structured data types, functional closures, signals, bitmap graphics, persistent objects and meta-level execution. The latter is needed as the Amber compiler is entirely written in Amber (above the Amber machine level) and needs to switch level when executing a program it has just compiled.

A set of implementation strategies are admissible for this machine, including byte-code interpretation, threaded code interpretation and compilation to native code. The current implementation is based on a byte-code interpreter and a one-space compacting collector, and runs on a Macintosh.

## Introduction

The Amber machine is a stack machine designed as an intermediate language for compiling higher-order languages, in the tradition of SECD machines [Landin 64] and combinator machines [Turner 79] [Curien 86]. This is a revision of the Functional Abstract Machine described in [Cardelli 83, Cardelli 84], and is specialized for the Amber language [Cardelli 86].

The amount of specialization required for a particular language in the general class of higher-order algorithmic languages is marginal; it mostly involves the set of primitive data types and does not affect the basic organization of the machine.

The machine supports a set of basic data types, functional closures, signals, persistent objects and meta-level execution. The latter is needed as the Amber compiler is entirely written in Amber (above the Amber machine level) and needs to switch level when executing a program it has just compiled.

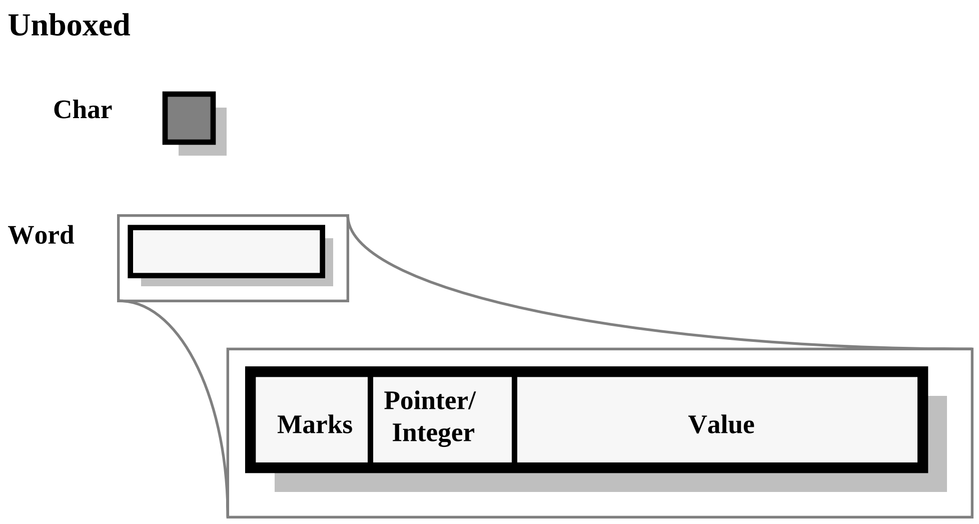
[1] Current addresss: DEC SRC, 130 Lytton Ave, Palo Alto, CA 94301.

A set of implementation strategies are admissible for this machine, including byte-code interpretation, threaded code interpretation and compilation to native code. The current implementation is based on a byte-code interpreter and a one-space compacting collector, while the similar Functional Abstract Machine for the ML language is compiled to VAX code and has a two-space compacting collector.

## The Format Level

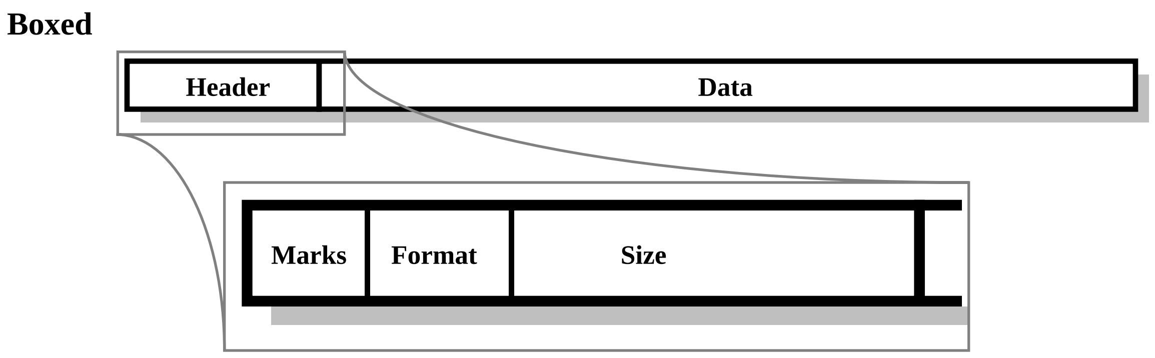
The format level is the lowest level of the Amber machine. This level is language independent and can be considered as a general-purpose heap manager. It supports garbage collection and data persistence, without any knowledge of the language or even the data structures supported at higher levels.

Data can be *boxed* or *unboxed*. Unboxed data exists on the stack or inside other data structures, and does not require memory allocation. We distinguish two kinds of unboxed: characters (normally one byte) and words (normally four bytes).



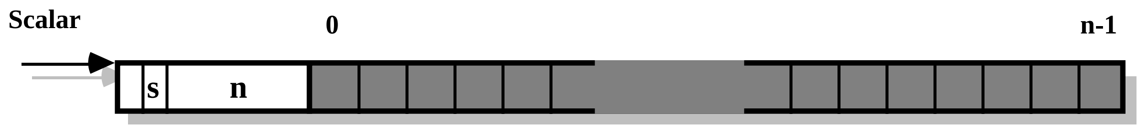
Unboxed words are further structured. They contain mark bits (used by the garbage collector and the persistence mechanism), a flag indicating whether this is an integer or a pointer (again, needed by the garbage collector), and an integer or pointer value.

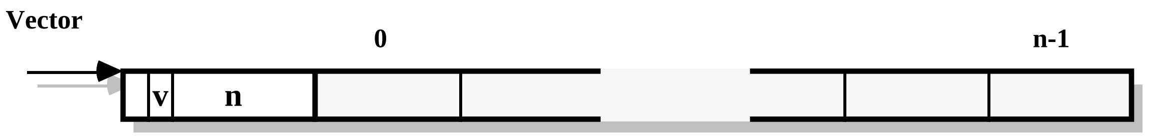
Boxed values are allocated in the heap. They have a header and a data section:



The header contains mark bits (the same as in unboxed words), a code indicating the format of the data section, and a size (number of items in the data section).

Corresponding to each kind of unboxed data, there is a format of boxed data. A *scalar* is a sequence of unboxed characters, and a *vector* is a sequence of unboxed words.





The following routines are supported for the manipulation of data formats:

**mkInt(i:int):Word**
convert an ordinary integer to an unboxed integer
**mkPtr(p:ptr):Word**
convert an address to an unboxed pointer
**isInt(w:Word):bool**
test wheter an unboxed word is an unboxed integer
**isPtr(w:Word):bool**
test whether an unboxed word is an unboxed pointer
**deInt(w:Word):int**
convert an unboxed integer to an ordinary integer
**dePtr(w:Word):ptr**
convert an unboxed pointer to an address

**allocScalar(n:int):Boxed**
allocate a boxed scalar of size n
**allocVector(n:int):Boxed**
allocate a boxed vector of size n
**isScalar(b:Boxed):bool**
test whether a boxed is a scalar
**isVector(b:Boxed):bool**
test whether a boxed is a vector
**scalarSize(b:Boxed):int**
get the size of a boxed scalar
**vectorSize(b:Boxed):int**
get the size of a boxed vector
**scalarNth(b:Boxed,n:int):Char**
get the n-th item of a boxed scalar
**vectorNth(b:Boxed,n:int):Word**
get the n-th item of a boxed vector
**setScalarNth(b:Boxed,n:int,c:Char)**
set the n-th item of a boxed scalar
**setVectorNth(b:Boxed,n:int,w:Word)**
set the n-th item of a boxed vector

A *hold stack* is provided to store boxed data temporarily or permanently. The garbage collector traces data starting from the hold stack, and recycles all the structures which are not accessible from the hold stack. The hold stack is a stack of indirections into the heap; such indirections are called *holds*. Often it is necessary to manipulate holds, instead of directly manipulating data, because the garbage collector may unexpectedly start and invalidate direct pointers to data.

**pushHold(b:Boxed):Hold**
push a boxed on the hold stack, obtaining a hold to it
**popHold():Boxed**
pop the hold stack, obtaining the boxed on top of it
**deHold(h:Hold):Boxed**
extract a boxed from a hold
**collect()**
garbage collection, normally called by allocScalar and allocVector.

Garbage collection can be implemented in a variety of ways. The current Amber implementation uses a one-space compacting collector. This technique requires two mark bits in boxed and unboxed data, and an extra relocation word for every boxed datum (which is prefixed to the header field and is invisible to all the above operations).

Finally, two routines provide a basic persistence mechanism. *extern* takes any arbitrary boxed or unboxed datum and writes it to a file, being careful to preserve all sharing and circularities. *intern* does the inverse: reads and recreates an object exactly as it was before it had been externed (modulo relocation). However, two executions of extern on the same object will duplicate it in persistent storage, and two invocations of intern on the same file will make two copies of the same object in memory.

**extern(filename:string,d:Data)**
save an datum to persistent storage
**intern(filename:string):Data**
fetch an datum from persistent storage

Extern and intern can be implemented purely iteratively, by pointer reversal techniques, so that there is no restriction on the size of persistent objects, other than memory size.

Here is the encoding used to extern objects. It is called **dex** (data exchange) format.

♦ Every dex file describes a data object which can have loops and shared subobjects.
♦ A dex file starts with a "[" and ends with a "]".
♦ Between brackets there are N data segments.
♦ Every data segment can refer to any other data segment, by "^n" (see below), where n is the order number of another segment (the first segment in the file is number zero).
♦ The last data segment is the object described by a dex file.

♦ Data in a dex file is encoded by four kinds of descriptors:
- Indirects. A "^" is followed by a non-negative varhex (see below) number, representing the number of a data segment.
- Numbers. An "x" is followed by the varhex representation of the number.
- Scalars. An "s" is followed by a non-negative varhex number (n). Then a ":" is followed by n chars (0-255).
- Vectors. A "v" is followed by a non-negative varhex number (n). Then a ":" is followed by n descriptors.

♦ No other characters appear in a dex file.

- ♦ A varhex is a variable-size hex representation of a number.
  - The hex digits 0,1..15 are represented by "@","A".."O"
  - A negative varhex n starts with "-", followed by the representation of -n.
  - Zero is represented as "@".
  - A positive varhex is represented by its hex encoding, with no leading zeros.
  - A varhex is terminated by any character different from "@".."O".
  - Ex: 3="C"; -1="-A"; 16="A@".

## The Data Structure Level

The data structures needed for a particular language are built on top of data formats. The data structures used in Amber are described below. This level supports operations for allocating, accessing and modifying these data structures, based on the operations described in the previous section.

### *Bool*

Booleans are represented as unboxed integer words, 0 for false and 1 for true.

### *Int*

Integers are represented as unboxed integer words.

### *String*

Strings of characters are represented as scalars.

### *Bitmap*

Bitmaps for graphics are represented as scalars.

### *Region*

Regions are compressed bitmaps, and are also represented as scalars.

### *Record*

Records are represented as vectors of even length. The even positions point to a string representing the field name, and the odd positions contain the field value.

### *Variant*

Variants are represented as vectors of length two. The first item points to a string representing the variant tag, and the second item is the content of the variant.

### *Array*

Arrays are represented as vectors.

***Ref***

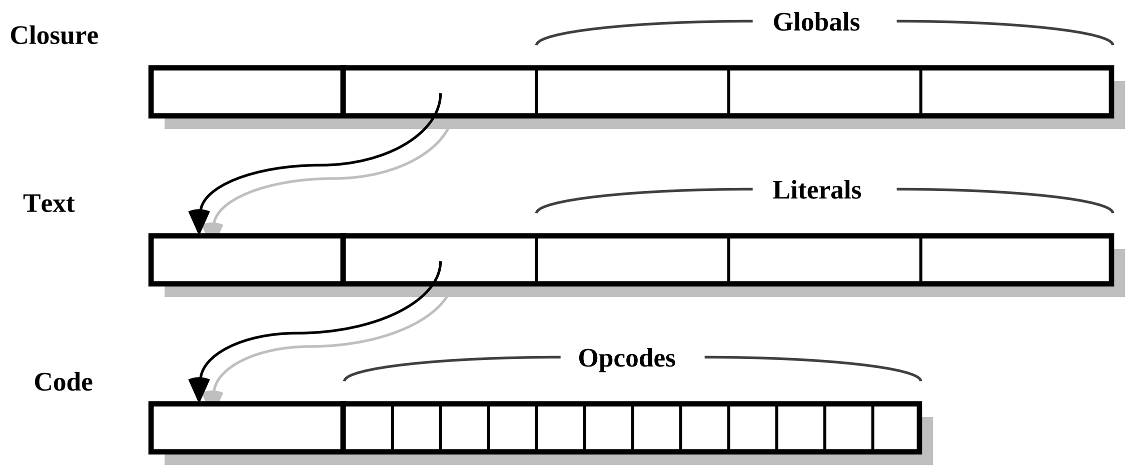
Assignable variables have an extra indirection, called a ref object, represented by a vector of length one.

***Function***

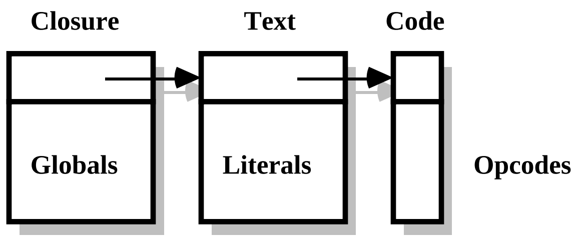
Functions are represented by a combination of a scalar and two vectors. The compiled code of the function is contained in a scalar called the *code*.

The code section cannot directly contain pointers to literals (like strings embedded in the program code). Literals are hence collected in an object called *text*, which is a vector whose first element is a code, and the remaining elements are literals for that code. Literals are normally either strings or other text objects arising from nested function definitions. Text objects are built at compile-time an do not change during execution.

The global variables of a function are collected in a vector called the *closure*, whose first element is a text and the remaining elements are values for the global variables. Closures are built at run-time, and many different closures can share the same text object.

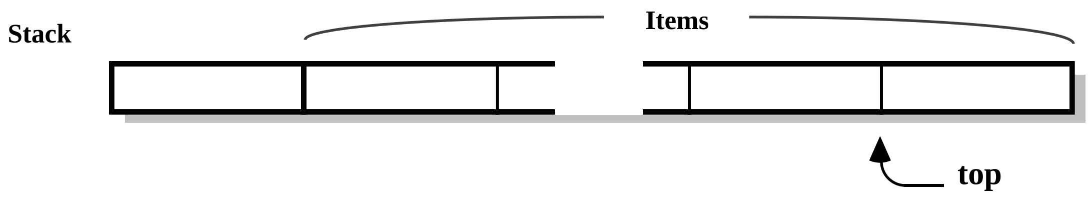


In the sequel we shall use closures extensively, and they will be drawn in the following, more compact, way:

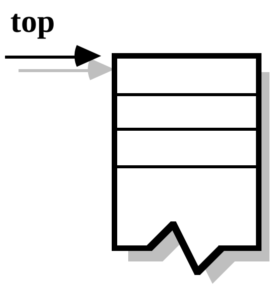


***Stack***

Some instructions having to do with meta-level execution  directly manipulate execution stacks. For this reason, stacks are legal Amber machine objects allocated in the same heap as data structures, and are represented as vectors. A stack pointer is just an index into such a vector.

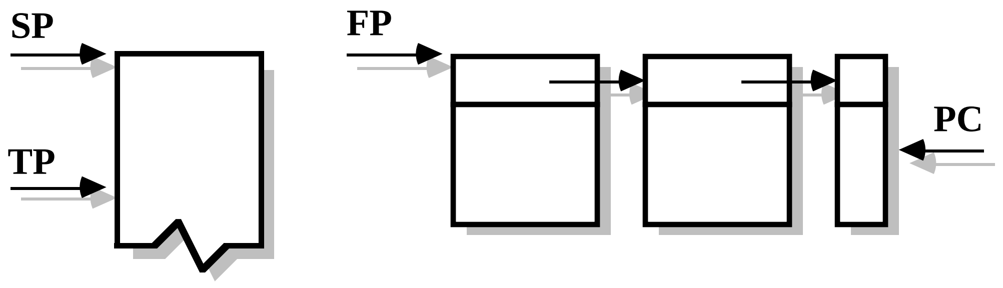


As stacks are used frequently in picures, they are shown in the following space-efficient way:



## The Operation Level

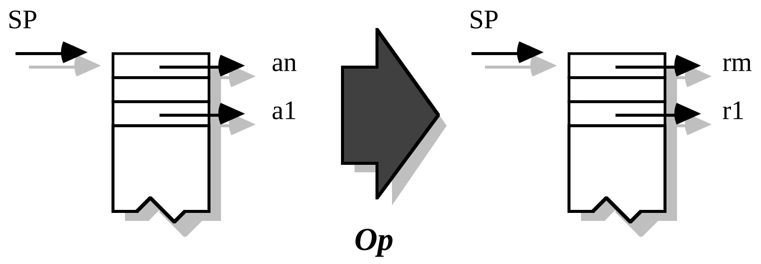
The Amber machine is a stack machine. Its state is determined by the stack pointer SP  indicating the top of the evaluation stack, by the trap pointer TP pointing to the most recent trap frame (for trapping signals), by the frame pointer FP pointing at the current closure, and by the program counter PC pointing at the next instruction to be executed. PC always points at the code segment belonging to the current closure, and TP always points at the same stack as SP.



The machine computes by performing state transitions. If a state transition involves only a subset of the state, we show only the parts which change, with the exception of the PC moving to the next instruction which is always assumed when not shown otherwise. Most transitions are determined by the instruction the PC is pointing to. In many such transition the PC changes to point to the next instruction; this is sometimes indicated by the notation PC+ (the value of PC at the instruction following the present one). The exact meaning of PC+ depends on the particular implementation strategy.

***Data operations***

Simple data operations affect only the stack (and possibly the heap): they take n arguments on the stack and produce m results, where both n and m can be greater or equal to zero. The *first* argument on the stack is the deepest, similarly for results. Hence, data operations peform state transitions of the following kind:



Here are the simple data operations. Some of them have parameters (e.g. OpBool(b)), which are taken from the instruction stream.

**OpSame:** takes two arguments and returns true if they are the same unboxed value (i.e. if they are the equal unboxed integers or the same unboxed pointer), false otherwise.

**OpPrint:** takes one argument (any value) and returns none. This is a low-level printing routine which shows the format structure of any value.

**OpBool(b):** takes no arguments and returns the boolean b.

**OpBoolNot:** takes one boolean argument  and returns its negation.

**OpBoolAnd:** takes two boolean arguments and returns their boolean and.

**OpBoolOr:** takes two boolean arguments and returns their boolean or.

**OpInt(n):** takes no arguments and returns the integer n.

**OpPlus:** takes two integer arguments and returns their sum. Signals "+" on overflow.

**OpDiff:** takes two integer arguments and returns their difference. Signals "-" on overflow.

**OpMult:** takes two integer arguments and returns their product. Signals "*" on overflow.

**OpDiv:** takes two integer arguments and returns their quotient. Signals "/" on divide by zero.

**OpMod:** takes two integer arguments and returns their module. Signals "%" on mod zero.

**OpLess:** takes two integer arguments and returns true if the first is less than the second, false otherwise.

**OpLessEqual:** takes two integer arguments and returns true if the first is less or equal to the second, false otherwise.

**OpMore:** takes two integer arguments and returns true if the first is greater than the second, false otherwise.

**OpMoreEqual:** takes two integer arguments and returns true if the first is greater or equal to the second, false otherwise.

**OpString:** takes two arguments (two integers) and returns a string whose length is the first argument, all initialized to the character whose ascii representation is given by the second argument. Signals "string" if the first argument is negative, or if the second argument is not in 0..255.

**OpLength:** takes one argument (a string) and returns its length.

**OpGetAscii:** takes two arguments (a string and an integer n) and returns the nth character of the string as an integer. Signals "getascii" if n is out of range.

**OpPutAscii:** takes three arguments (a string, an integer n and an integer v) and returns none. It replaces then n-th element of the string by the integer v (converted to a character). Signals "putascii" if n is out of range, or if v is not in 0..255.

**OpSub:** takes three arguments (a string s, an index i and a size n) and allocates and returns a string initialized to the substring of s starting at i and of length n.Signals "sub" if i or i+n are out of bounds.

**OpSetSub:** takes three arguments (a destination string, an index and a source string) and returns none. It side-effects the destination string by replacing the source string for the substring of the destination string starting at index and having the same length as the source string. Signals "setsub" if index or index + length of source are out of bounds.

**OpStringBlit:** takes five arguments (a source string s, a source index sx, an integer n, a destination string ds, and a destination index dx) and copies a substring of length n of s starting at sx, into d starting at dx. If s and d are the same string, the order of copy is chosen appropriately: from the end of the substring if sx is less then dx, and from the beginning of the substring otherwise. Does not signal on out of bounds.

**OpSearch:** takes four arguments (a string p, a string s, an index i and a boolean d) and searches the first occurrence of any of the characters in p inside of s, starting at position i of s. The search is forward if d is true, backwards otherwise. Returns the index of the occurrence, or signals "search" if not found.

**OpEqual:** takes two string arguments and returns true if they are equal string, false otherwise.

**OpBitmap:** takes four arguments (integers x,y,hor,ver) and allocates and returns a bitmap with bounding tile x,y,hor,ver.

**OpBitmapTile:** takes one argument (a bitmap) and returns the four integers (x,y,hor,ver) forming its bounding tile.

**OpPixel:** takes three arguments (a bitmap and two integers x,y) and returns one (a bool). The result is true if the pixel x,y in the bitmap is black, false otherwise, or if x,y is out of bounds.

**OpBitblit:** takes ten arguments (a source bitmap, four integer x,y,hor,ver, a destination bitmap, two integer dx,dy, an integer code and a region) and returns none. It transfers the tile x,y,hor,ver from the source to the destination at position dx,dy, using the code (0..3 for copy, or,xor,clear). The destination is clipped to the region.

**OpTexture:** takes eight arguments (a source bitmap, a destination bitmap, four integers x,y,hor,ver, an integer code and a region) and returns none. Replicates the source bitmap in the tile x,y,hor,ver of the destination bitmap, which is further clipped by the region, using the code (0..3 for copy, or, xor, clear). The replicated pattern is aligned to the origin of the destination bitmap.

**OpLine:** takes seven arguments (a bitmap, four integers x1,y1,x2,y2, an integer code and a region) and returns none. It draws a line from x1,y1 to x2,y2 using the code (0..3 for copy, or, xor, clear) and clipping to the region.

**OpScreen:** takes no arguments and returns the screen bitmap.

**OpCursor:** takes no arguments and returns two (integers x,y) which are the current position of the mouse cursor.

**OpSetCursor:** takes two arguments (integers x,y) and returns none. It moves the mouse cursor to the x,y position.

**OpButton:** takes no arguments and returns a boolean: true if the mouse button is pressed, false if it is not. Only one button is assumed.

**OpCursorIcon:** takes no arguments and returns a bitmap which is the current cursor icon. Changes to it change the cursor image.

**OpCursorTip:** takes two integers (nx,ny) and returns two integers (ox,oy). Changes the tip (in cursor coordinates) of the cursor, i.e. the pixel determinig the cursor position, to nx,ny. Returns the old value of the tip, ox,oy.

**OpNullRegion:** takes no arguments and returns the null region (a region with no points and an arbitrary coordinate system).

**OpTileRegion:** takes four arguments (integers x,y,hor,ver) and returns a rectangular region of size x,y,hor,ver.

**OpMakeRegion:** takes a bitmap and returns a region, which is a compressed version of the bitmap. The region has the same coordinate system as the bitmap. Blank regions are all equivalent to each other, and are called null regions.

**OpOffsetRegion:** takes three arguments (a region and two integers dx,dy) and returns a region. It moves the argument region by dx,dy with respect to its coordinate system and returns it as a new region: the argument region is not affected.

**OpInsetRegion:** takes three arguments (a region and two integers dx,dy) and returns a region. Shrinks (or expands for negative dx,dy) a region by moving all its points inward (or outward) by dx,dy.

**OpUnionRegion:** takes two argumens (regions) and returns one region. The result region is the set union of the points in the argument regions.

**OpSectRegion:** takes two argumens (regions) and returns one region. The result region is the set intersection of the points in the argument regions.

**OpDiffRegion:** takes two argumens (regions) and returns one region. The result region is the set difference of the points in the argument regions.

**OpXorRegion:** takes two argumens (regions) and returns one region. The result region is the set symmetric difference of the points in the argument regions.

**OpCarveRegion:** takes a region and returns four integers (x,y,hor,ver) forming a maximal tile entirely contained in the region, and signals "carveregion" if the region is empty. Does not affect the region.

**OpPointInRegion:** takes three arguments (two integers x,y, and a region) and returns a boolean. The result is true if the point x,y belongs to the region.

**OpTileSectRegion:** takes five arguments (four integers x,y,hor,ver and a region) and returns a boolen. The resul is true if the tils x,y,hor,ver intersects points in the region.

**OpEqualRegion:** takes two arguments (regions) and returns a boolean. The result is true if the two regions have the same coordinate system and set of points. Two empty regions are always equal, no matter what their coordinate system is.

**OpVariant(l):** takes one argument (any value) and returns a variant object whose tag is the l-th literal in the current closure, and whose contents is the argument.

**OpSetVariant(l):** takes two arguments (a variant and a value). If the variant tag is equal to the l-th literal in the current closure, the contents of the variant are set to the second argument, otherwise signals "setvariant".

**OpRecord(l(0),..,l(n-1)):** takes n arguments and returns a record whose (n-i-1)-th label is the l(i)-th literal in the current closure, and whose i-th field is the i-th argument.

**OpSelect(l,g):** takes one argument (a record) and returns the value of the field whose label is equal to the l-th literal in the current closure (a string). The parameter g is the initial *guess* (an index) of where that label can be found in the record. If the guess is wrong, the whole record is searched for that label, and the index of that field is stored back into the code stream as the guess for the next time around.

**OpSetRecord(l,g):** takes two arguments (a record and a value) and returns none. It assigns the value as the new contents of the record field whose label is equal to the l-th literal in the current closure (a string). The parameter g is used as in OpSelect.

**OpArray:** takes two arguments (an integer n and a value v) and allocates and returns an array of size n with all items initialized to v. Signals "array" if the first argument is negative.

**OpArraySize:** takes one argument (an array) and returns its size.

**OpIndex:** takes two arguments (an array and an index n) and returns the n-th item of the array. Signals "index" when n is out of range.

**OpUpdate:** takes three arguments (an array, an index i and a value v) and returns no results. It updates the i-th element of the array with the value v.

**OpRef:** takes one argument (a value v) and allocates and returns a ref data structure containing it.

**OpDeRef:** takes one argument (a ref data structure) and returns its contents.

**OpAssign:** takes two arguments (a ref and a value v) and returns no results. It assigns the value v as the new contents of the ref.

**OpExtern:** takes one two arguments argument (a string and any value) and returns none. It writes the value into the file whose name is the string, using the **extern** routine. Signals "extern" on disk errors.

**OpIntern:** takes one argument (a string) and returns a value. It reads the value from the file whose name is the string, using the **intern** routine. Signals "intern" on disk errors.

**OpInfile:** takes one argument (a string) and returns none. It redirects the input stream, which is initially bound to the screen. Old input streams are saved on an input stream stack. OpInfile redirects the input stream to the file whose name is given by the string argument (it signals "infile" on I/O errors). Or, if the string is empty, it redirects the input stream to the previous stream (it signals "infile" if the input stream stack is empty).

**OpOutfile:** takes one argument (a string) and returns none. It redirects the output stream, which is initially bound to the screen. Old output streams are saved on an output stream stack. OpOutfile redirects the output stream to the file whose name is given by the string argument (it signals "outfile" on I/O errors). Or, if the string is empty, it redirects the output stream to the previous stream (it signals "outfile" if the output stream stack is empty).

**OpInput:** takes no arguments and returns one (an integer). Reads the next character from the input stream. If the input stream is currently bound to a file and an end-of-file is found, the input stream is redirected to the previous stream on the input stream stack and an ascii blank character is returned. If the input stream is bound to the screen and there are no characters to be read, it waits. Signals "input" on I/O errors.

**OpCanInput:** takes no arguments and returns one (a boolean). Tests whether there are currently characters to be read from the input stream. It always returns true if the input stream is currently bound to a file. Signals " caninput" on I/O errors.

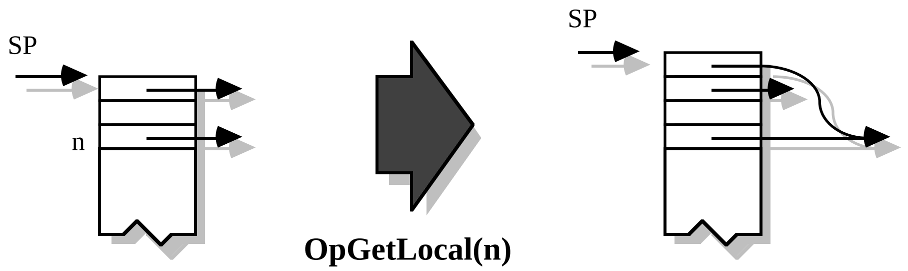
**OpOutput:** takes one argument (an integer) and returns none. Writes the integer on the output stream as the next character. Signals "output" if the argument is not in 0..255 or on I/O errors.

**OpOutString:** takes one argument (a string) and returns none. Writes the string on the output stream. Signals "output" on I/O errors.

***Stack operations***

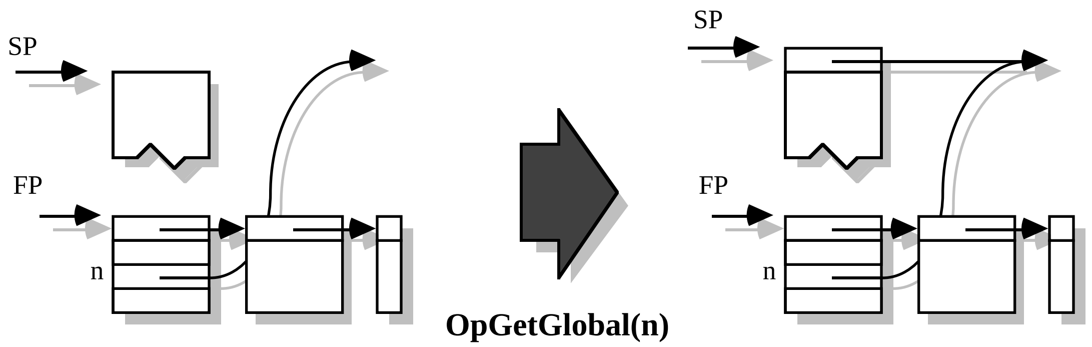
These are general purpose stack-shuffling operations.

**OpGetLocal(n)** copies the n-th element of the stack on top of it. The first element on the stack has displacement 0.



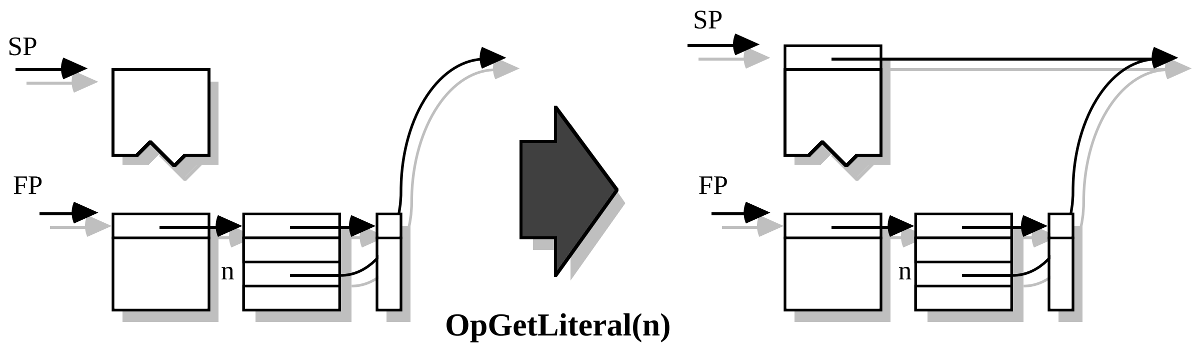
**OpGetLocal(n)**

**OpGetGlobal(n)** copies the n-th global variable on top of the stack.OpGetGlobal(0) fetches the first global variable in the closure; note that because of the structure of closures this is the element of index 1 in the closure vector.



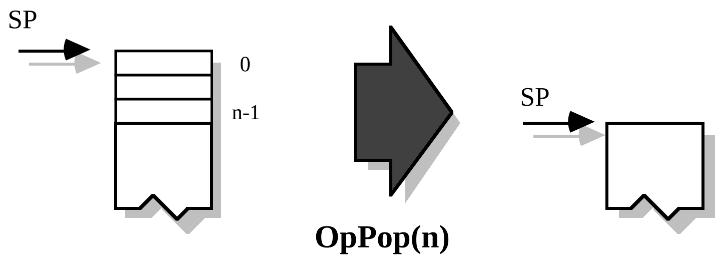
**OpGetGlobal(n)**

**OpGetLiteral(n)** copies the n-th literal on top of the stack. OpGetLiteral(0) fetches the first literal in the text; because of the structure of text objects this is the element of index 1 in the text vector.



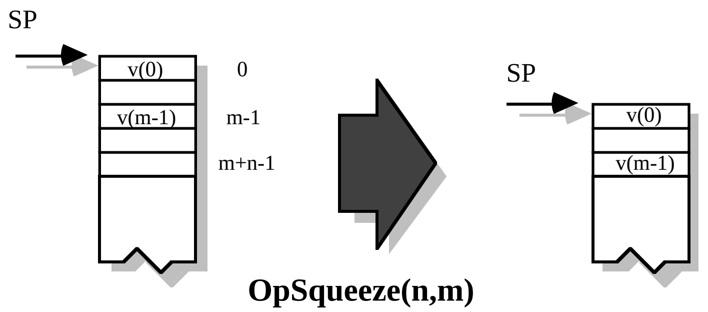
**OpGetLiteral(n)**

**OpPop(n)** removes the first n values on top of the stack.



**OpPop(n)**

**OpSqueeze(n,m)** removes from the stack n elements, starting at depth m. OpSqueeze(n,0) is equivalent to OpPop(n)

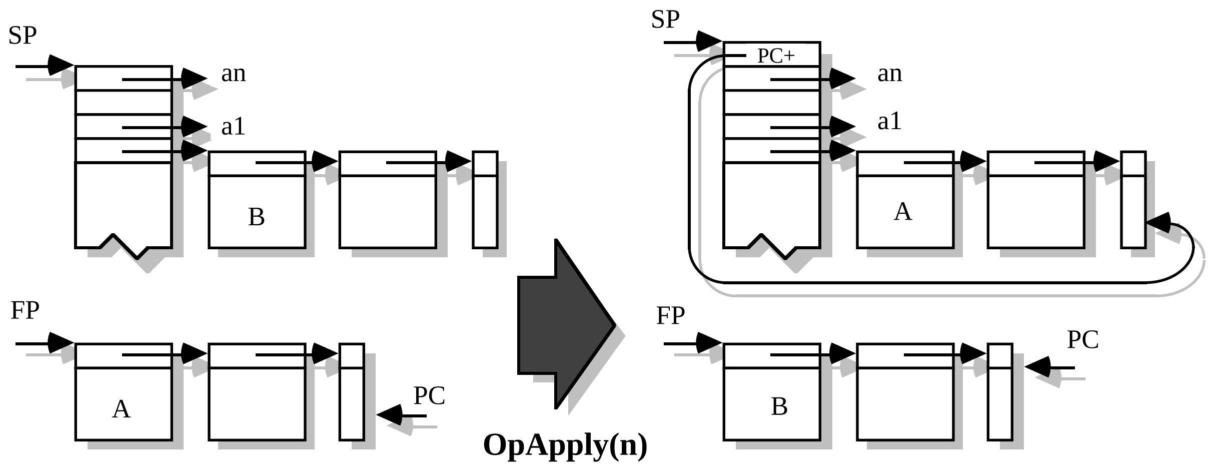


**OpSqueeze(n,m)**

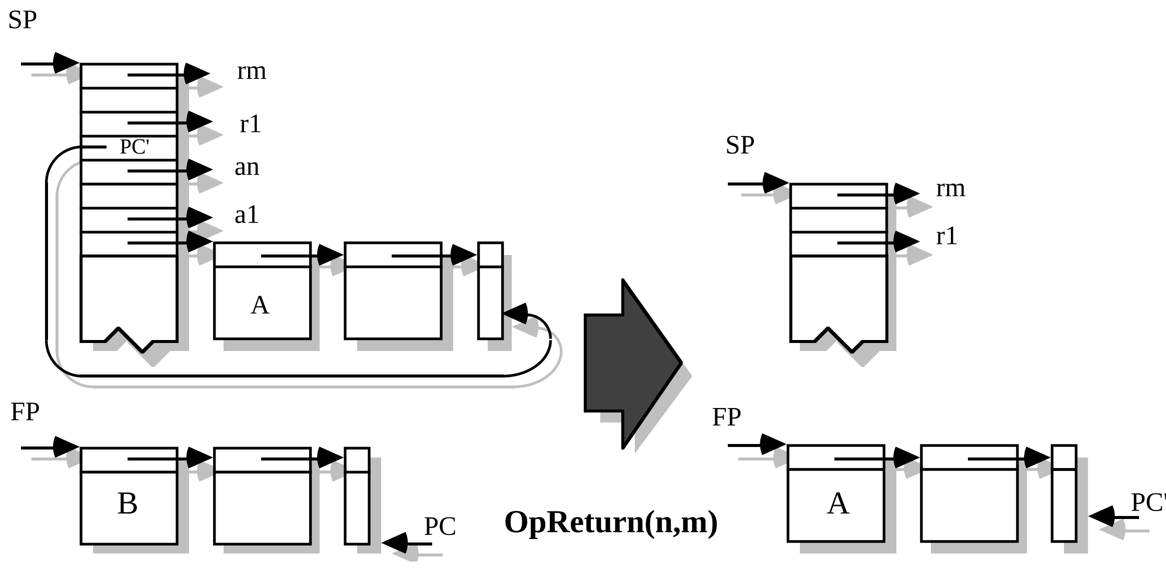
***Control operations***

Control operations affect the order of execution of instructions, i.e. they modify the PC in ways other than simply moving it to the next instruction.

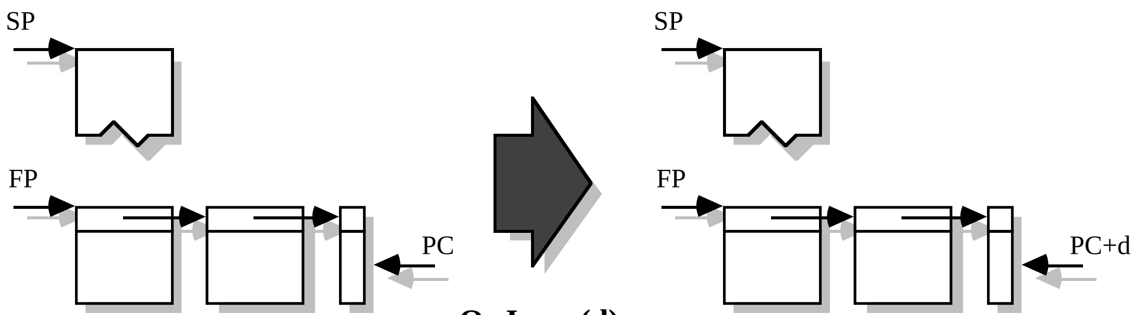
Function application is achieved by the **OpApply(n)** instruction, where n is the number of arguments for the application. The current (calling) closure is saved in place of the called closure, which is installed as the current one. The value of PC+ is saved on the stack and PC is set to the beginning of the called closure.



The **OpReturn(n,m)** operation takes care of function returns, where n is the number of arguments of the current call, and m is the number of results. The old (calling) closure is restored as the current one, and the saved value of the PC becomes current. The calling closure, its arguments and the saved PC are squeezed out of the stack.

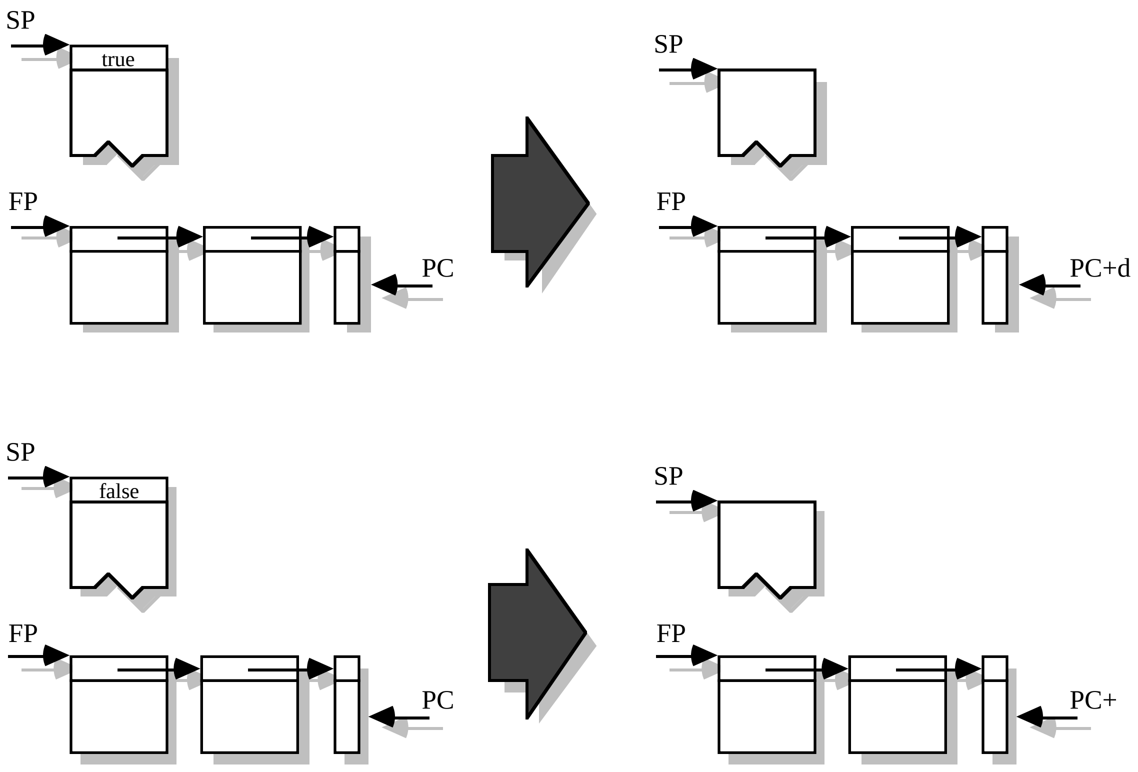


Unconditional jumps are obtained by the **OpJump(d)** instruction, where d is a relative displacement from the current PC. The exact meaning of d depends on implementations.



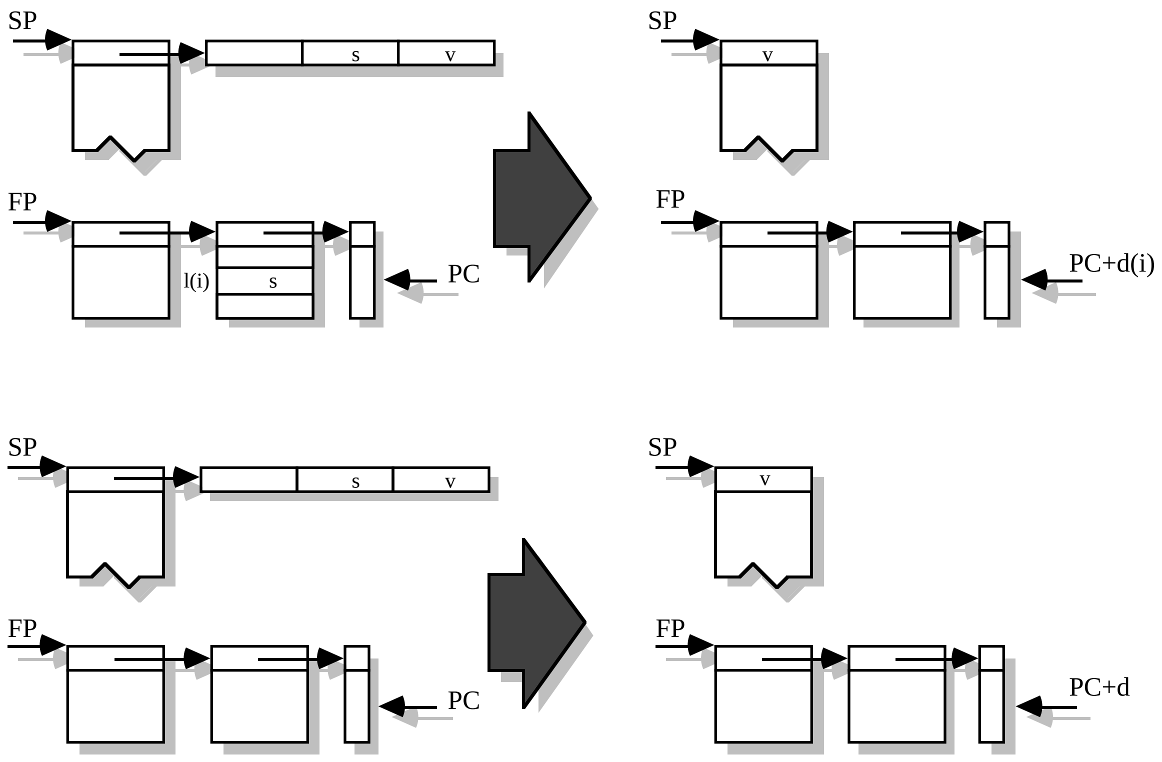
**OpJump(d)**

Conditional jumps are obtained by the **OpTrueJump(d)** and **OpFalseJump(d)** instruction, where d is a relative displacement from the current PC. Again, the exact meaning of d depends on implementations. OpTrueJump jumps if the top of the stack has the value true in it, otherwise proceeds to the next instruction. In both cases the top value is popped away. The following picture shows the two transitions for OpTrueJump; OpFalseJump is symmetric.



**OpTrueJump(d)**

**OpCase(l(1),d(1),...,l(n),d(n),d)** is a case operation for variant objects. The top of the stack must have a variant with tag s (a string) and contents v. If the string s is equal to the literal of index l(i) in the current closure, then the PC is incremented by the corresponding relative displacement d(i). Otherwise the PC is incremented by d. All the literals at index l(i) must be different from each other. In any case, the variant on top of the stack is replaced by its contents.

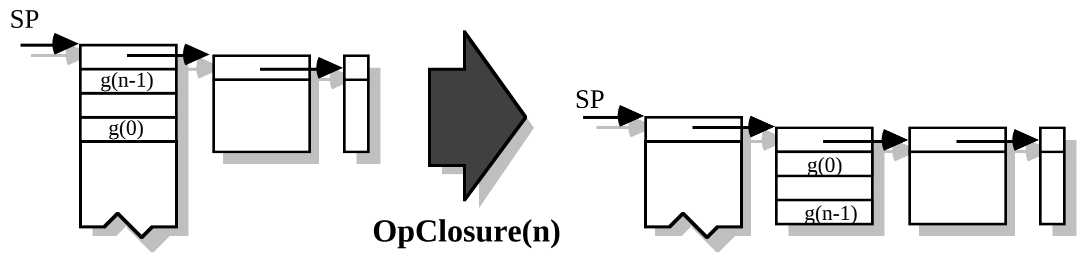


**OpCase(l(1),d(1),...,l(n),d(n),d)**

## *Closure operations*

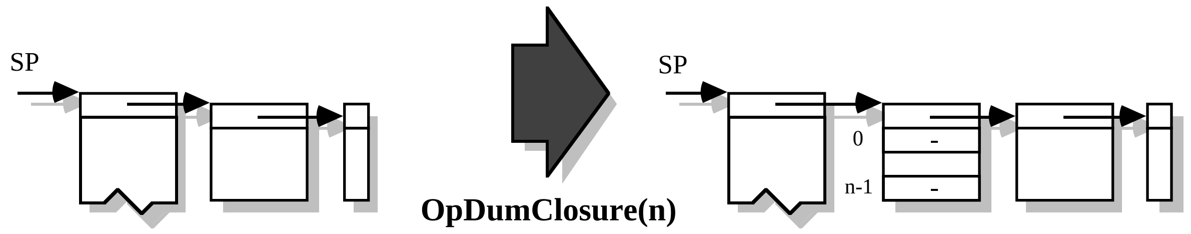
Closure operations really belong to the Data Operations section, but are listed separately because of their importance to the general workings of the machine.

**OpClosure(n)** builds a closure with n global variables, given a text object (obtained by an OpGetLiteral) and n values on the stack. The values on the stack are in inverse order to the globals in the closure.



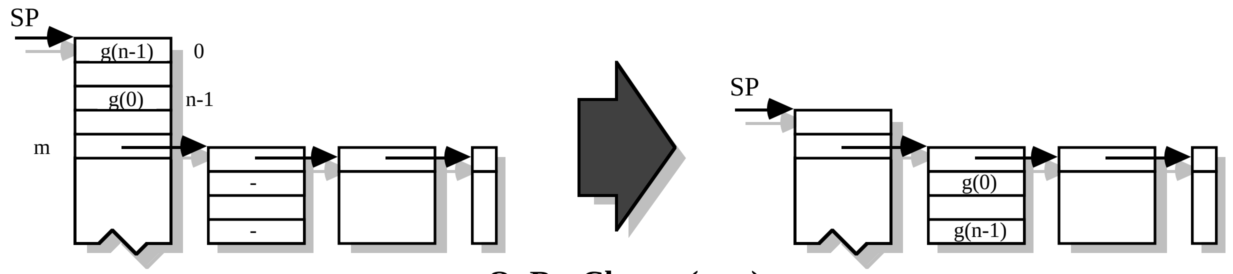
**OpClosure(n)**

Recursive and mutually recursive functions need recursively defined closures. These are built in two steps, first allocating empty closures and later filling them. **OpDumClosure(n)** builds dummy closures of n globals, given a text object on the stack.



**OpDumClosure(n)**

**OpRecClosure(n,m)** (with m≥n) fills an existing dummy closure at depth m on the stack with the first n values on top of the stack, and pops those values away. The values on the stack are in inverse order than the the globals in the closure.

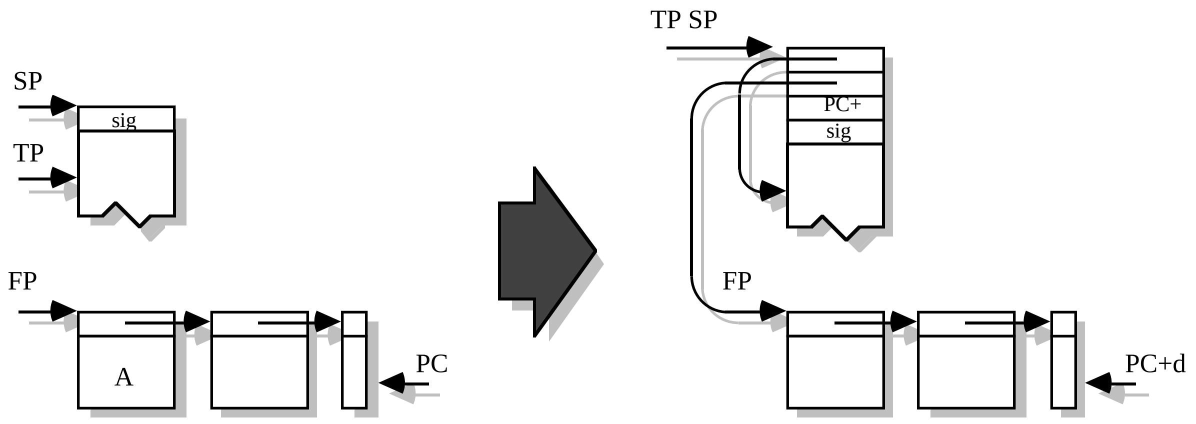


**OpRecClosure(n,m)**

***Signal operations***

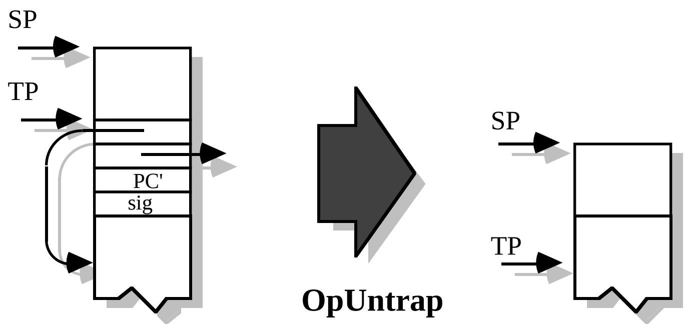
Many machine operations may *signal* exceptional conditions, for example OpDiv signals the string "/" on divide by zero. Signals are global jump-outs which unwind the stack. Traps can be set up to intercept signals and recover from them.

**OpTrap(d)** sets up a trap record to intercept a given class of signals. The string on top of the stack is the name of the signal intercepted by this trap. On top of that are saved PC+, the current closure and the current trap pointer TP. TP is set to the top of the stack. Finally PC is incremented by the relative displacement d (the code immediately following OpTrap(d) is used to recover from signals).



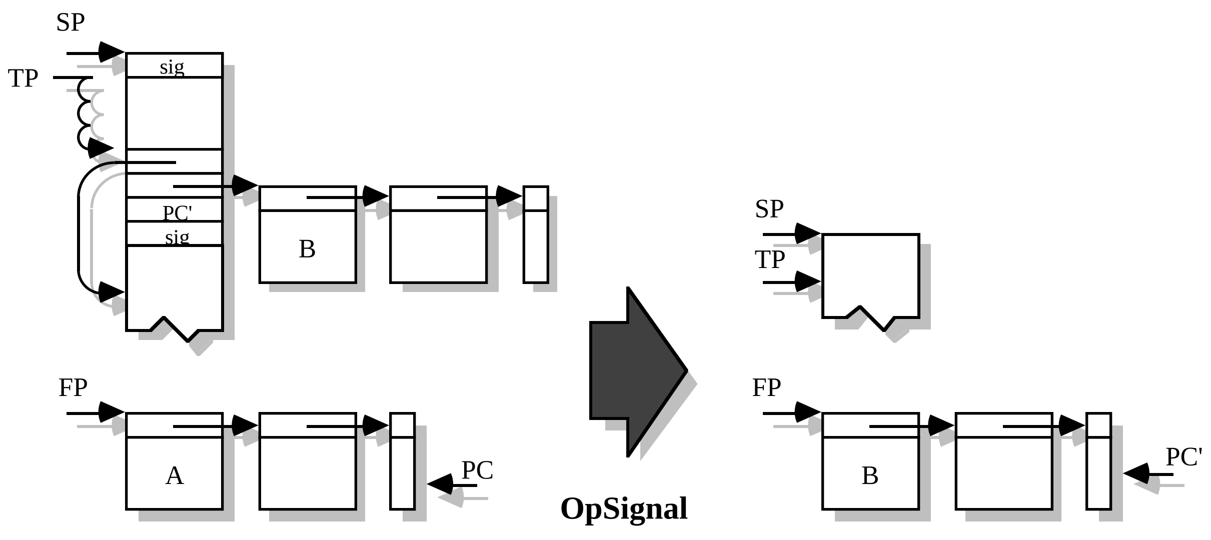
**OpTrap(d)**

**OpUntrap** eliminates a trap record, in case no signal was issued during an evaluation and the results of that evaluation are now sitting on top of the stack. TP is set to the previous TP, and the trap record is squeezed out of the stack.



**OpUntrap**

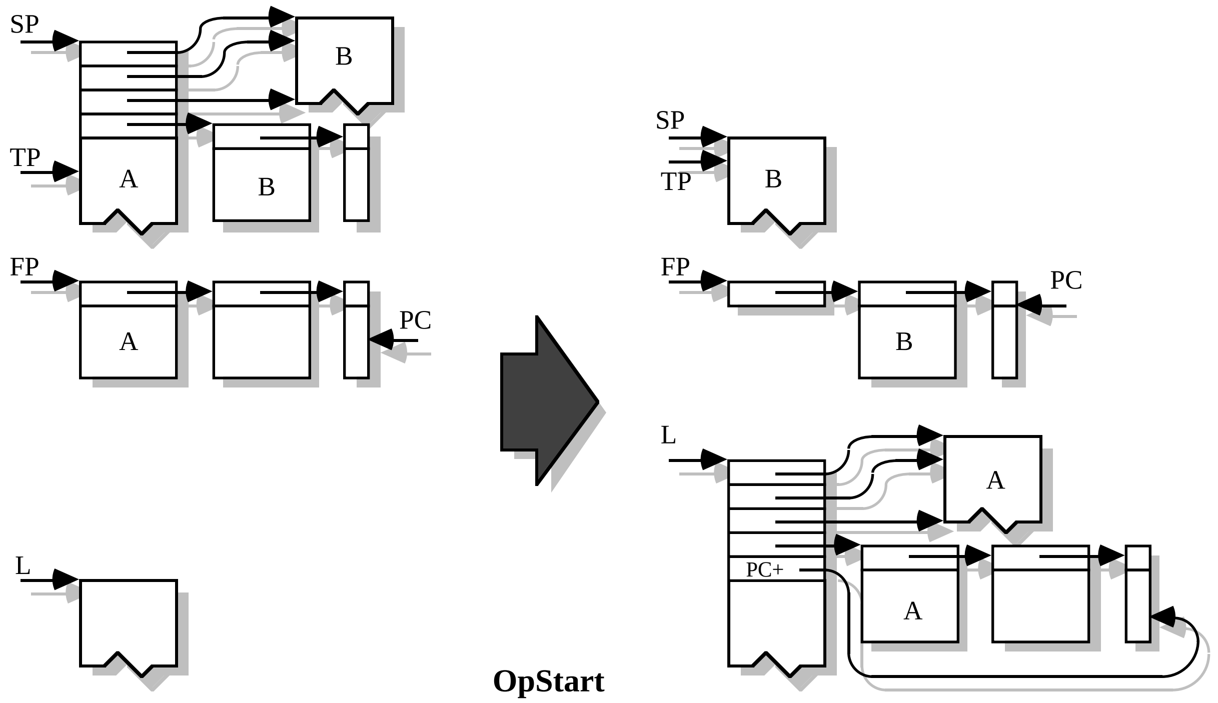
Signals can be generated by primitive operations, and by the **OpSignal** operation. Primitive operations push the name of the signal (a string) on the top of the stack, while OpSignal assumes that the name of the signal is already there. When a signal is issued, the following process takes place. The trap frames are scanned, starting from the current TP, looking for a trap for that signal. If no adequate trap is found, the effect is the same as an **OpStop** operation (see later), except that a message is printed reporting the occurrence of an untrapped signal. If an appropriate trap record is found somewhere down the stack, the current closure and PC are replaced by the closure and PC in the trap frame, the SP is set below that trap frame, and the TP is set to the next trap frame down the stack (if any).



***Meta operations***

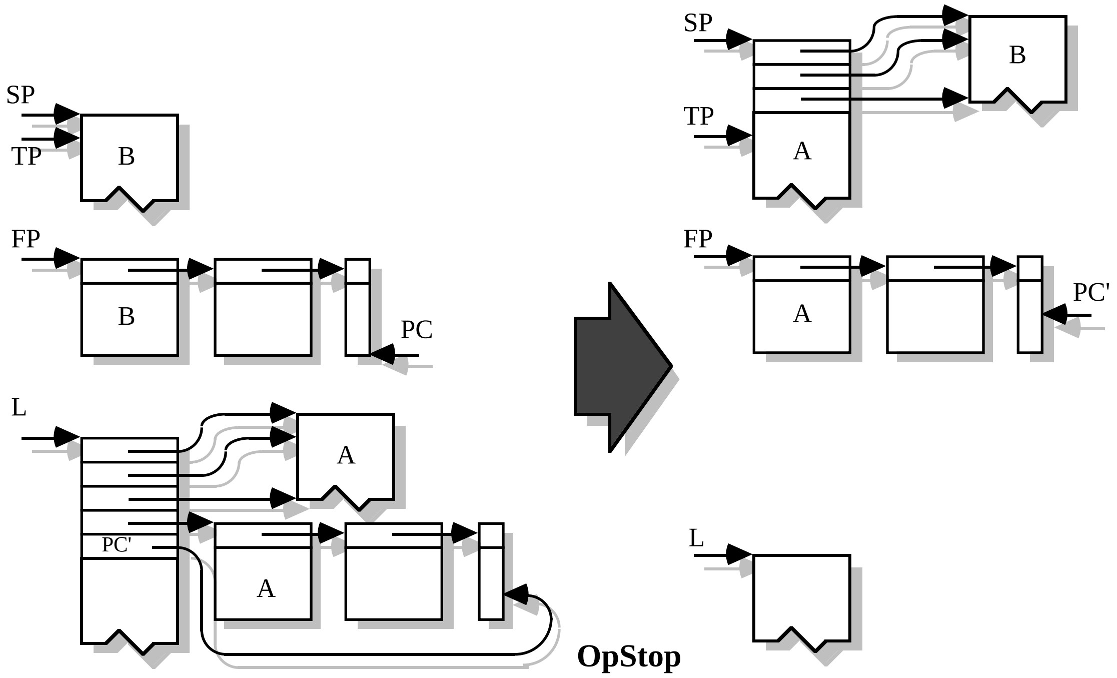
Meta operations are used by the Amber-in-Amber compiler to bootstrap itself, and to switch level between compilation and execution. Here we introduce a new component of machine state not shown before: the state stack. A complete machine state consisting of an execution stack and a closure can be saved on the state stack, and a fresh machine can be started. A complete machine state can be resumed later on.

**OpStart** saves a machine state on the state stack, and starts a new machine. The OpStart operation expects on the stack a text object, an execution stack and the initial setting of TP and SP for the new machine. The text object is converted into an empty closure and made current, with the PC set to its beginning. The new stack, TP and SP are made current. The old PC+, closure, stack, TP and SP are saved on the state stack, whose L (level) pointer is raised.



**OpStart**

**OpStop** restores a previously saved machine state, or stops the Amber machine if the state stack is empty. Here is the situation for a non-empty state stack. The saved closure, PC, stack, TP and SP are restored, and the current stack, TP and SP are pushed on the restored stack.



**OpStop**

**OpSetCoercion** takes one argument on the stack and returns none. The argument is a closure, and it is saved in a special place for later use by OpGetCoercion. This closure is to be used by the *coerce* construct of the Amber language, which involves a run-time call to the Amber typechecker.

**OpGetCoercion** takes no arguments and returns the closure set apart by OpSetCoercion (or signals "getcoercion" if no OpSetCoercion has been executed).

***Other operations***

**OpZot:** takes no arguments, and aborts the execution of the whole machine. This is for emergencies.

**OpSystem:** takes one argument (a string) and returns none. The string is interpreted to set system flags etc., and its meaning is implementation dependent.

**OpPoll:** takes no arguments and returns none. To be inserted in possibly infinite computation loops (e.g. while loops and recursive functions) to be able to stop them cleanly. The precise effect is implementation dependent.

## Bytecode Implementation

In this section we describe the current implementation in more detail. A code object is a sequence of bytecodes, each encoding a machine operation, and parameters to bytecodes.

**OpApply(n)**

code 1, followed by one unsigned byte (parameter n).

codes 160 .. 175, meaning OpApply(0) .. OpApply(15).

**OpArray**

code 2.

**OpArraySize**

code 3.

**OpAssign**

code 34.

**OpBool(b)**

code 9.

**OpBoolAnd**

code 72.

**OpBoolNot**

code 71.

**OpBoolOr**

code 73.

**OpCanInput**

code 70.

**OpCarveRegion**

code 91.

**OpCase(l(1),d(1),...,l(n),d(n),d)**

code 11, followed by one unsigned byte (n), followed by n pairs l(i) (one byte) d(i) (two bytes), followed by d (two bytes). The displacements of the d(i) and d are each relative to the value of the PC immediately after it.

**OpClosure(n)**

code 15, followed by one unsigned byte (n).

**OpDeRef**

code 16.

**OpDiff**

code 18.

**OpDiffRegion**

code 89.

**OpDiv**

code 19.

**OpDumClosure(n)**

code 21, followed by one byte (n).

**OpEqual**

code 22.

**OpEqualRegion**

code 94.

**OpExtern**

code 23.

**OpFalseJump(d)**

code 24, followed by two bytes constituting a signed integer (d). The displacement d is relative to the value of the PC immediately after d.

**OpGetAscii**

code 4.

**OpGetCoercion**

code 74.

**OpGetGlobal(n)**

code 26, followed by one byte (n).

codes 144 .. 159, meaning OpGetGlobal(0) .. OpGetGlobal(15).

**OpGetLiteral(n)**

code 51, followed by one unsigned byte (n).

**OpGetLocal(n)**

code 27, followed by two bytes constituting an unsigned integer (n).

code 224, followed by one unsigned byte (n).

codes 128 .. 143, meaning OpGetLocal(0) .. OpGetLocal(15).

**OpInfile**

code 30.

**OpIndex**

code 29.

**OpInsetRegion**

code 86.

**OpInput**

code 31.

**OpInt(n)**

code 32, followed by two bytes constituting a signed integer (n).

code 225, followed by one signed byte (n).

codes 192 .. 199, meaning OpInt(0) .. OpInt(7).

codes 200 .. 207, meaning OpInt(-8) .. OpInt(-1).

**OpIntern**

code 28.

**OpJump(d)**

code 33, followed by two bytes constituting a signed integer (d). The displacement d is relative to the value of the PC immediately after d.

**OpLength**

code 35.

**OpLess**

code 36.

**OpLessEqual**

code 37.

**OpMakeRegion**

code 84.

**OpMore**

code 76.

**OpMoreEqual**

code 77.

**OpMod**

code 39.

**OpMult**

code 40.

**OpNullRegion**

code 82.

**OpOffsetRegion**

code 85.

**OpOutfile**

code 41.

**OpOutput**

code 42.

**OpOutString**

code 65.

**OpPlus**

code 43.

**OpPointInRegion**

code 92.

**OpPoll**

code 81.

**OpPop(n)**

code 44, followed by an unsigned byte (n).

**OpPrint**

code 45.

**OpPutAscii**

code 13.

**OpRecClosure(n,m)**

code 46, followed by one unsigned byte (n), followed by one unsigned byte (m).

**OpRecord(l(0),...,l(n-1))**

code 47, followed by one unsigned byte (n), followed by n bytes (l(i)).

**OpRef**

code 14.

**OpReturn(n,m)**

code 48, followed by one unsigned byte (n), followed by one unsigned byte (m).

code 226, followed by one unsigned nibble (n), followed by one unsigned nibble (m).

codes 176 .. 179, meaning OpReturn(0,0) .. OpReturn(0,3).

codes 180 .. 183, meaning OpReturn(1,0) .. OpReturn(1,3).

codes 184 .. 187, meaning OpReturn(2,0) .. OpReturn(2,3).

codes 188 .. 191, meaning OpReturn(3,0) .. OpReturn(3,3).

**OpSame**

code 49.

**OpSearch**

code 25.

**OpSectRegion**

code 88.

**OpSelect(l,g)**

code 20, followed by one byte (l), followed by one byte (g).

**OpSetCoercion**

code 75.

**OpSetRecord(l,g)**
: code 52, followed by one byte (l), followed by one byte (g).

**OpSetSub**
: code 54.

**OpSetVariant(l)**
: code 59, followed by one unsigned byte (l).

**OpSignal**
: code 78.

**OpSqueeze(n,m)**
: code 55, followed by one unsigned byte (n), followed by one unsigned byte (m).

**OpStart**
: code 12.

**OpStop**
: code 56.

**OpString**
: code 57.

**OpStringBlit**
: code 5.

**OpSub**
: code 58.

**OpSystem**
: code 69.

**OpTileRegion**
: code 83.

**OpTileSectRegion**
: code 93.

**OpTrap(d)**
: code 79, followed by two bytes constituting a signed integer (d). The displacement d is relative to the value of the PC immediately after d.

**OpTrueJump(d)**
: code 61, followed by two bytes constituting a signed integer (d). The displacement d is relative to the value of the PC immediately after d.

**OpUnionRegion**
: code 87.

**OpUntrap**
: code 80.

**OpUpdate**
: code 63.

**OpVariant(l)**

code 64,followed by one unsigned byte (l).

**OpXorRegion**

code 90.

**OpZot**

code 0.

## References

[Apple 85] Apple Computer Inc. **Inside Macintosh**, Addison-Wesley, 1985.

[Cardelli 83] L.Cardelli: **The functional abstract machine**, Bell Labs Technical Report TR-107, 1983. Also in *Polymorphism* I.1, Jan.1983.

[Cardelli 84] L.Cardelli: **Compiling a functional language**, *Conference Record ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Cardelli 86] L.Cardelli: **Amber**, *Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP*, Le Val D'Ajol, Vosges (France), May 1985. Lecture Notes in Computer Science n. 242, Springer-Verlag, 1986.

[Curien 86] P-L.Curien: **Categorical combinators, sequential algorithms and functional programming**, Wiley, New York, 1986,

[Landin 64] P.J.Landin: **The mechanical evaluation of expressions**, *Computer Journal*, Vol. 6, No. 4, 1964, pp. 308-320.

[Pike Guibas Ingalls 84] R.Pike, L.Guibas, D.Ingalls: **Bitmap graphics**, *SIGGRAPH '84 course notes.*

[Turner 79] D.A.Turner: **A new implementation technique for applicative languages**, *Software Practice and Experience*, vol 9, 31-49, 1979.