The ML Abstract Machine

The State of the Abstract Machine (AM) is determined by four quantities (together with their denotations):

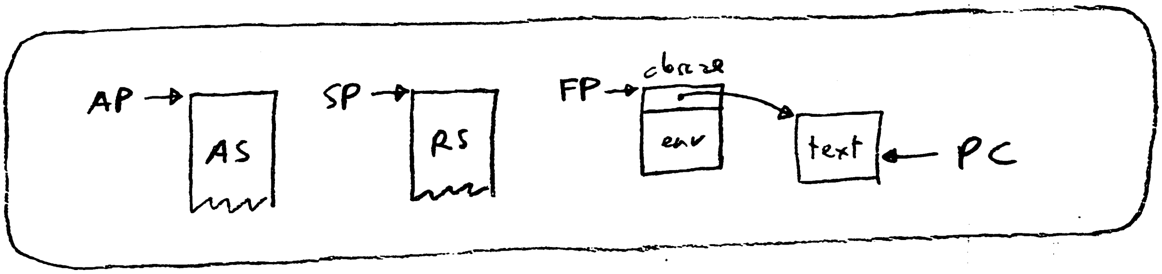
AP - Argument Pointer : Pointer to the top of the Argument Stack (AS), where arguments are loaded to be passed to functions, and results of functions are delivered. This stack is also used to store local and temporary values.

FP - Frame Pointer : Pointer to the current closure consisting of the text of the currently executed program, and of an environment for the free variables of the program.

SP - Stack Pointer : Pointer to the top of the Return Stack (RS), where program counter and frame pointer are saved during function calls.

PC - Program Counter : Pointer to the next instruction to be executed inside the current program.

Here is a snapshot of the AM:



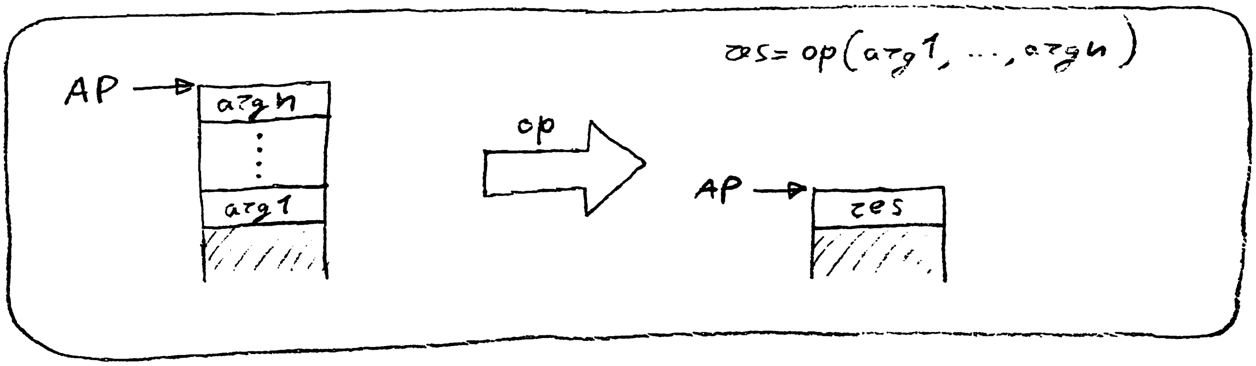
The AM assumes the existence of an infinite memory of cells of different sizes. Typical cells are null, bool, num, tok, ref, pair, list, injection, closure and text cells. The exact format of these cells is inessential, as long as the primitive operations on these cells respect the expected properties.

The AM does not assume these cells to contain any information about their type.

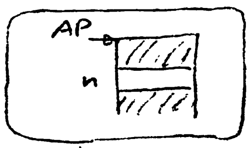
Stacks, closures, texts and data contain pointers to cells, and this convention will be strictly observed in these notes. However implementations may directly store certain cells, instead of their pointers, on stacks etc; this usually happens for null, empty, bool and num cells, but should never happen for ref cells.

Data Operations

These are operations which transfer data back and forth between the Argument Stack and data cells. With the exception of destructors (see below) they take n arguments ($n \geq 0$) from the top of AS (the first argument is the deepest one) popping AS n times. Then they push their result back on the top of AS.



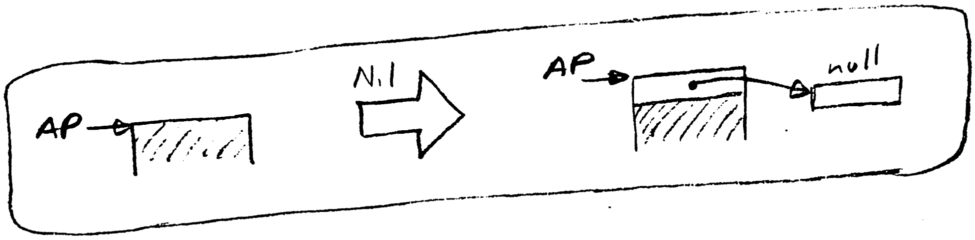
Note that the typechecking of the ML source program will guarantee against any misuse of AM operations.

The notation 

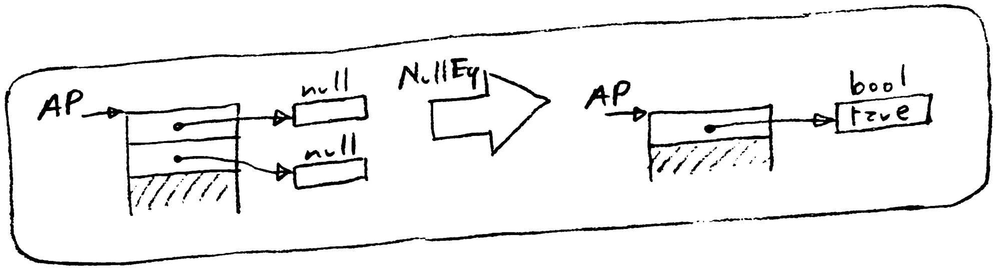
means that "n" is a displacement from the top of AS where the top has displacement 0.

Null Operations

- Nil



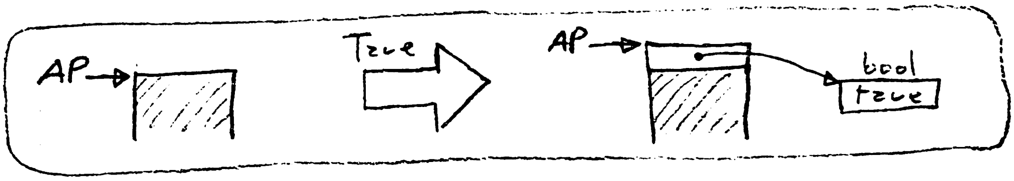
- NullEq



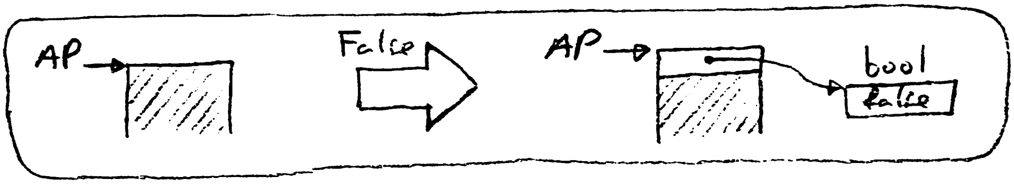
- NullPrint

Boolean Operations

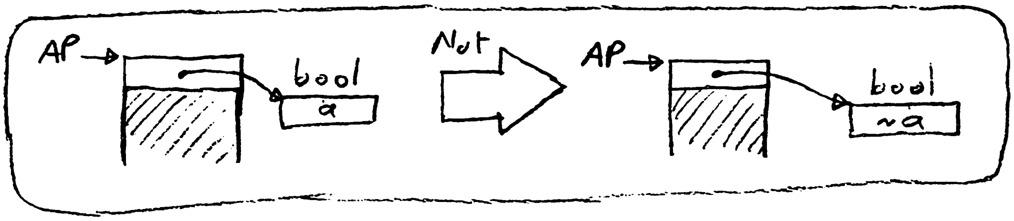
- True



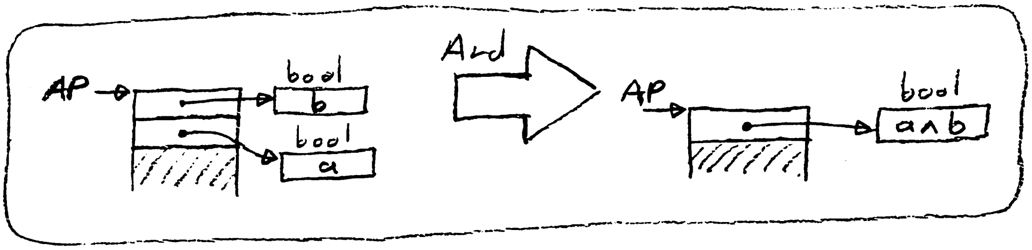
- False



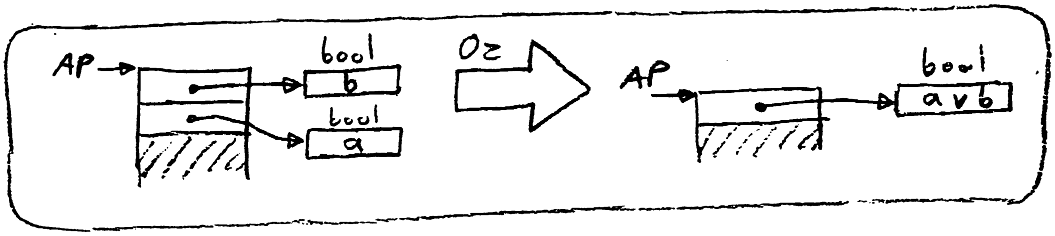
- Not



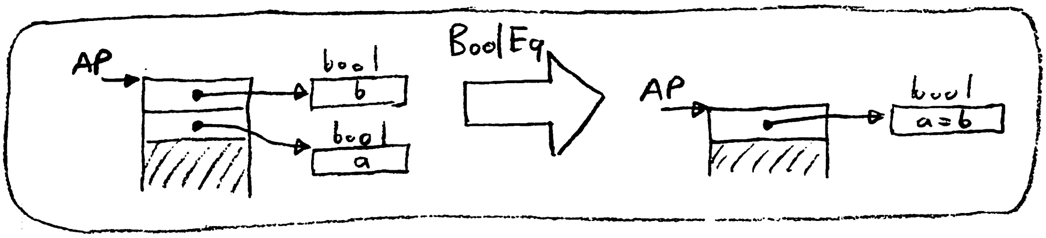
- And



- Or



- BoolEq



- Bool Print

Numeric Operations

All numbers are real numbers.

- Num n (push a <u>num</u> cell containing n on AS)
- Minus ($\text{Minus } a = -a$)
- Plus ($\text{Plus}(a,b) = a+b$)
- Diff ($\text{Diff}(a,b) = a-b$)
- Times ($\text{Times}(a,b) = a*b$)
- Divide ($\text{Divide}(a,b) = a/b$)
- IntDiv ($\text{IntDiv}(a,b) = \lfloor a/b \rfloor$)
- Module ($\text{Module}(a,b) = (a/b) - \lfloor a/b \rfloor$)
- Greater ($\text{Greater}(a,b) = a>b$)
- Less ($\text{Less}(a,b) = a<b$)
- GreatEq ($\text{GreatEq}(a,b) = a>=b$)
- LessEq ($\text{LessEq}(a,b) = a<=b$)
- NumEq ($\text{NumEq}(a,b) = a=b$)
- NumPrint

String Operations

- Tok \`...\` (push a tok cell containing \`...\` on AS)
- Conc (Conc(\`...\`, \`---\`) = \`...---\`)
- SubTok ($\mathrm{SubTok}(k, l, `c_1 \ldots c_n`) = `c_k \cdots c_{k+l-1}`$)
- TokLength ($\mathrm{TokLength}(`c_1 \cdots c_n`) = n$)
- Explode ($\mathrm{Explode}(`c_1 \ldots c_n`) = [`c_1`; \cdots ; `c_n`]$)
- Implode ($\mathrm{Implode}([`{-}{-}_1`; \cdots ; `{-}{-}_n`]) = `{-}{-}_1 \cdots {-}{-}_n`$)
- Search ($\mathrm{Search}(`p_0 \cdots p_k`, `\cdots q_n \; q_{n+1} \cdots`) = n$
  where $p_i = q_{n+i}$ (first occurrence from left)
  fails if no occurrence)
- TokEq ($\mathrm{TokEq}(`p_1 \cdots p_n`, `q_1 \cdots q_m`) = (n=m) \wedge (p_i = q_i)$)
- TokPrint
- TokUQPrint (print UnQuoted)

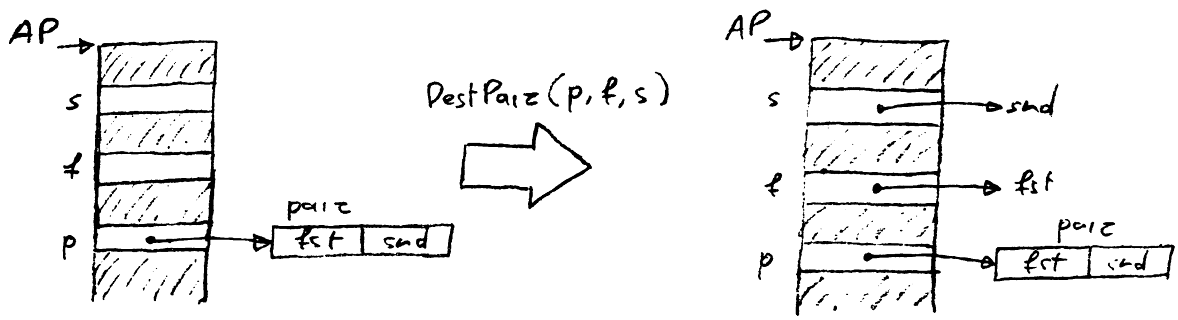
Reference Operations

- Fetch $(\text{Fetch}(r) = a \text{ where } r = \text{ref}(a))$
- Store $(\text{Store}(a, r) = (r := a;\ a))$
- RefEq $(\text{RefEq}(r_1, r_2) = (a_1 = a_2) \text{ where } r_1 = \text{ref}(a_1);\ r_2 = \text{ref}(a_2))$
- RefPrint

Pair Operations

- Pair (Pair(f,s) pushes a <u>pair</u> cell (f,s) on AS)
- Fst (Fst(p) = f where p = f,s)
- Snd (Snd(p) = s where p = f,s)

- DestPair



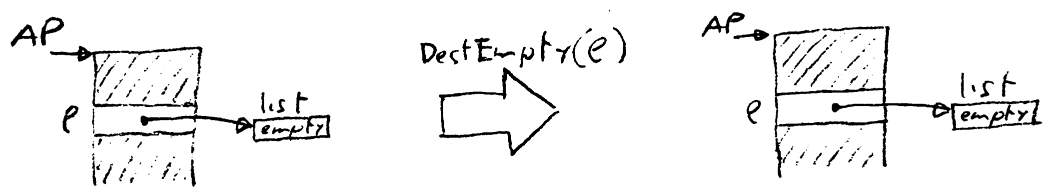
No constraints on the order or coincidence of p, f, s.

- Pair Print

List Operations

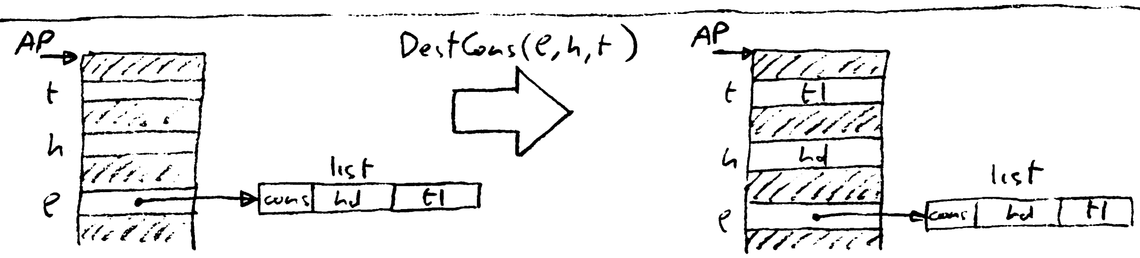
- Empty (push a list empty cell on AS)
- Cons (Cons(h, t) pushes a list cons cell (h_t) on AS)
- Hd (Hd(l) = h where l = h_t)
- Tl (Tl(l) = t where l = h_t)
- Null (Null(l) = l = [])

- DestEmpty



Fails if the list cell is not an empty cell.

- DestCons



Fails if the list cell is not a cons cell.
No constraints on the order or coincidence of l, h, t.

- ConsPrint

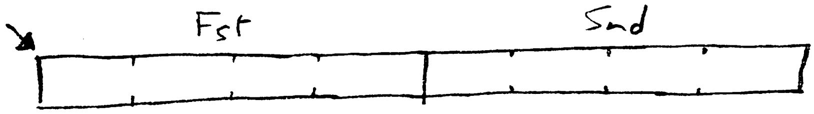
Disjunction Operations

- Inl (Inl(a) pushes a left <u>injection</u> cell (a) on AS)
- Inr (Inr(a) pushes a right <u>injection</u> cell (a) on AS)
- Outl (Outl(j) = a where j = inl(a))
- Outr (Outr(j) = a where j = inr(a))
- Isl (Isl(j) = (j = inl(a)) for some a)
- Isr (Isr(j) = (j = inr(a)) for some a)

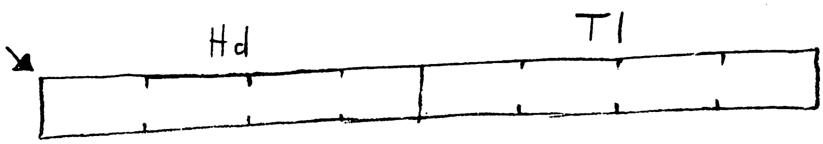
- DisjPrint

ML Run-Time Data Structures

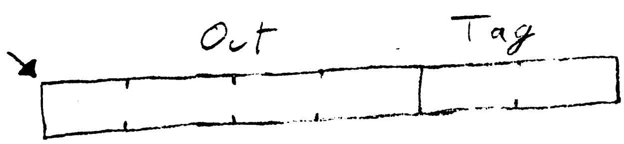
Pair



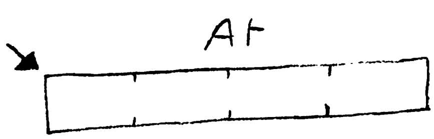
Cons



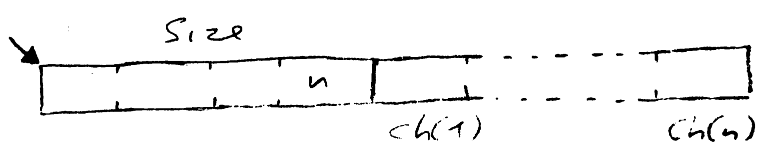
Injection



Reference



Token



Empty

0

True

1

False

0

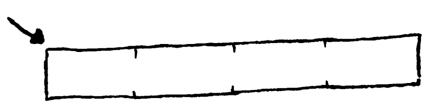
Nil

0

Integer

-32768 .. 32767

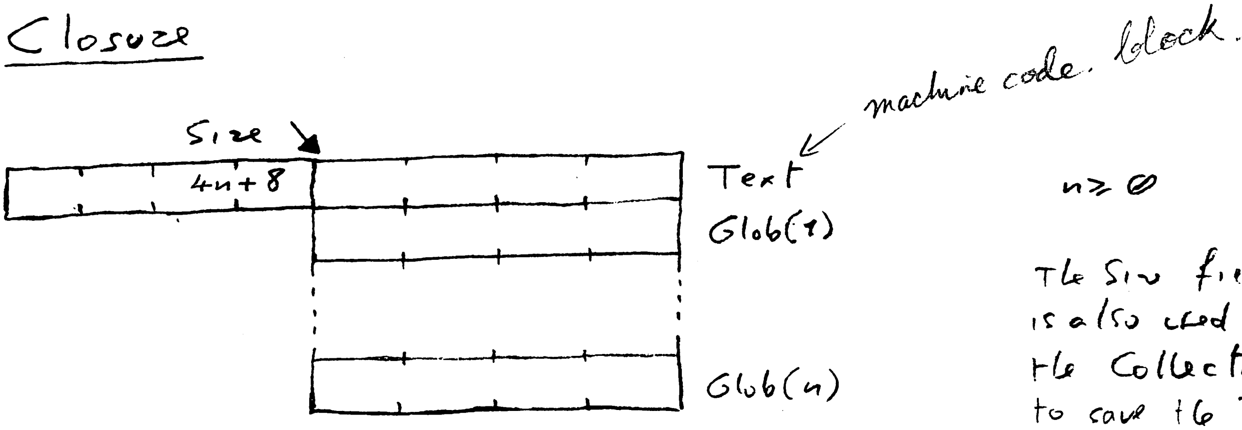
Number



(Contains a Floating Point)

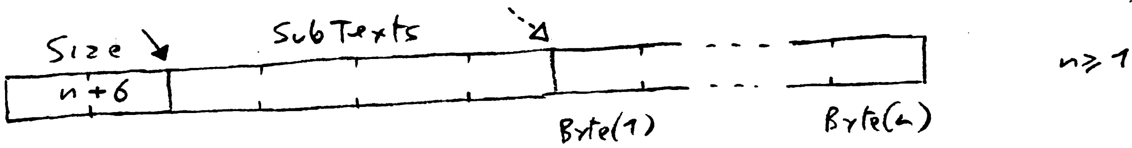
Note: Empty, True, False, Nil and Integers are <u>Unboxed</u> values. They are contained directly on the stack or inside data structures. When an unboxed value (occupying a word) is contained in the least significative part of a long word (e.g. in a pair) the other word must be zero. Pointers can be distinguished from unboxed values because the most significant word of a pointer cannot be zero.

Closure

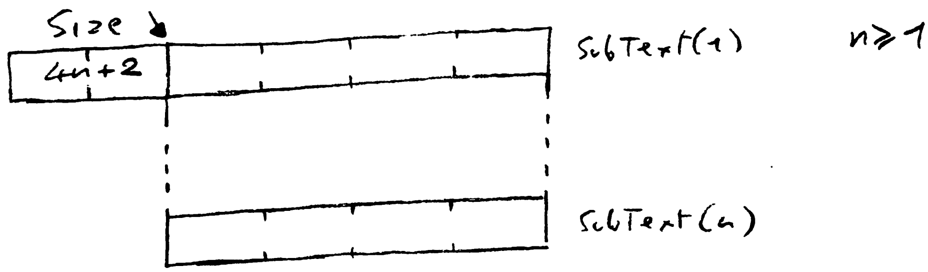


The Size field is also used by the Collector to save the Text pointer

Text[1]



SubText



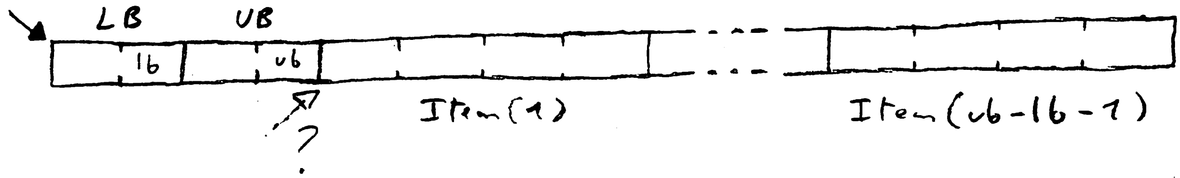
---

(1) The pointer "↘" is used when passing Texts around by themselves (it is convenient for garbage collection); the pointer "⇢" is used when referring to a Text from a closure (it is convenient for function application).

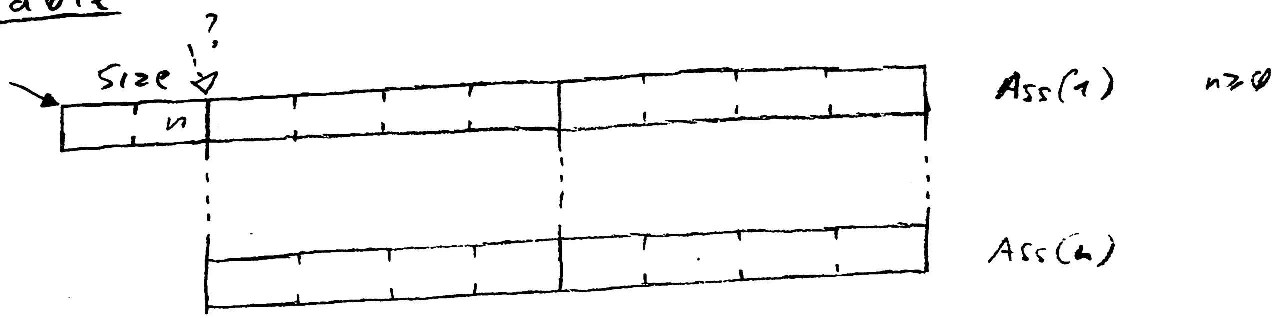
"Size" fields are placed before the arrow "↘" because they are only used during garbage collection.

Array

$ub+1 \geq lb$



Table



Abstract Syntax Operations

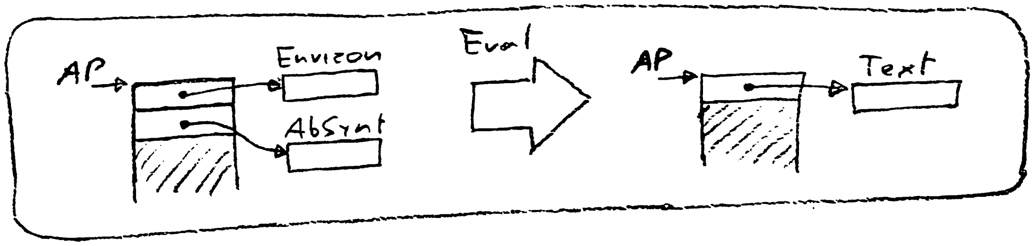
- Parse (read a string and push a parse tree on AS)

- - - - -
  - - - -
  - - - -
  - - - -

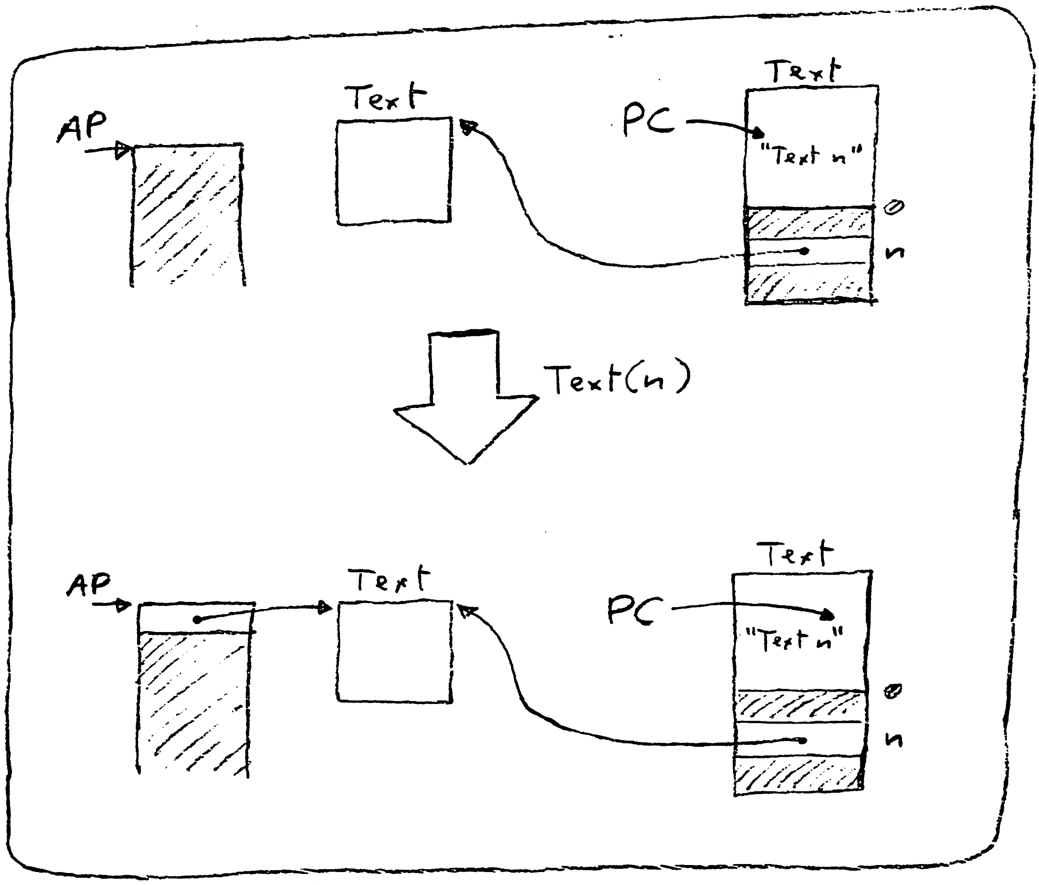
  } make-decl-is primitives for each syntactic clause

Text Operations

- Eval

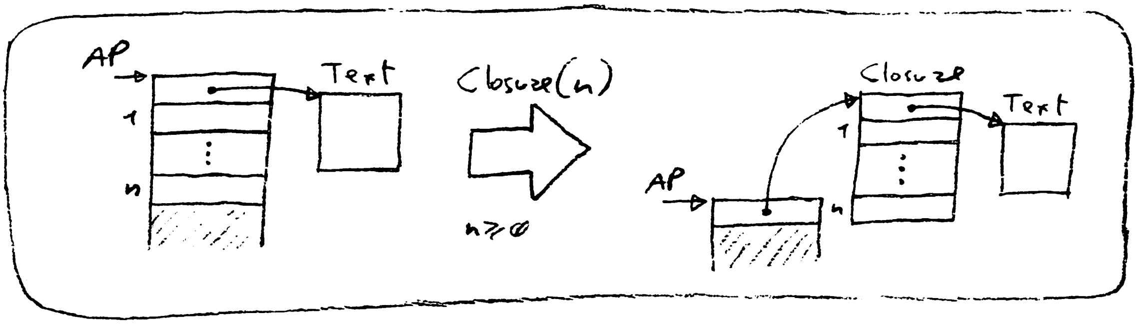


- Text

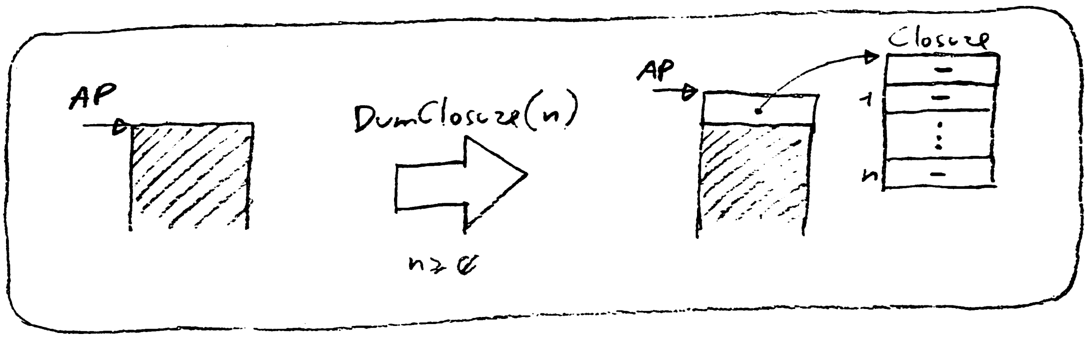


Closure Operations

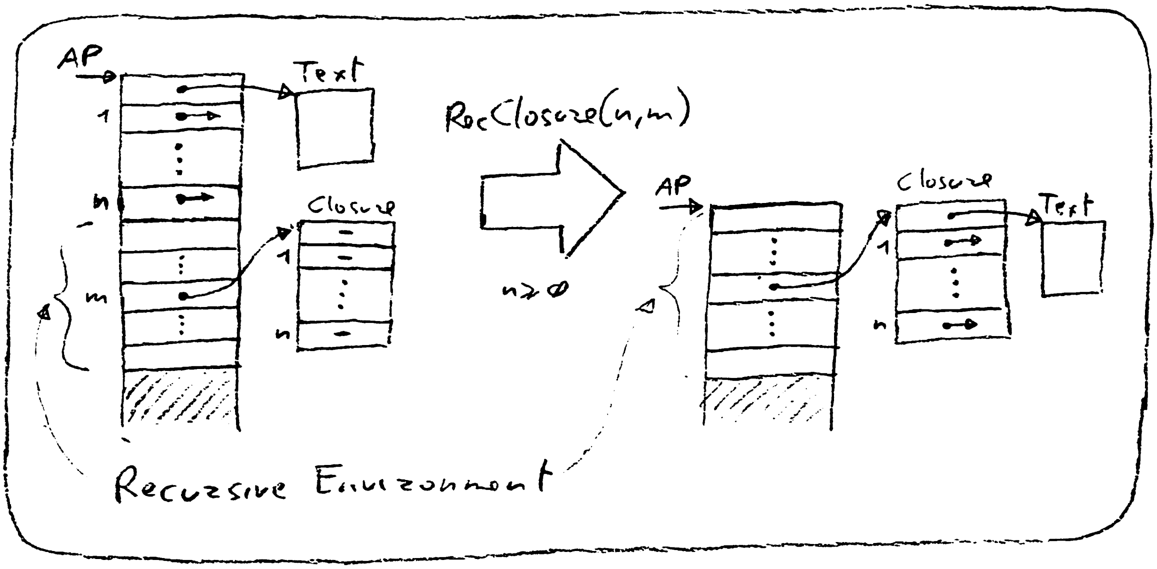
- Closure



- DumClosure



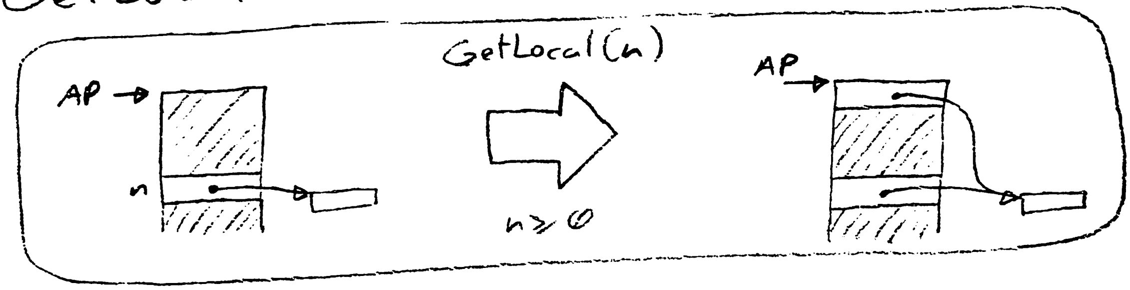
- RecClosure



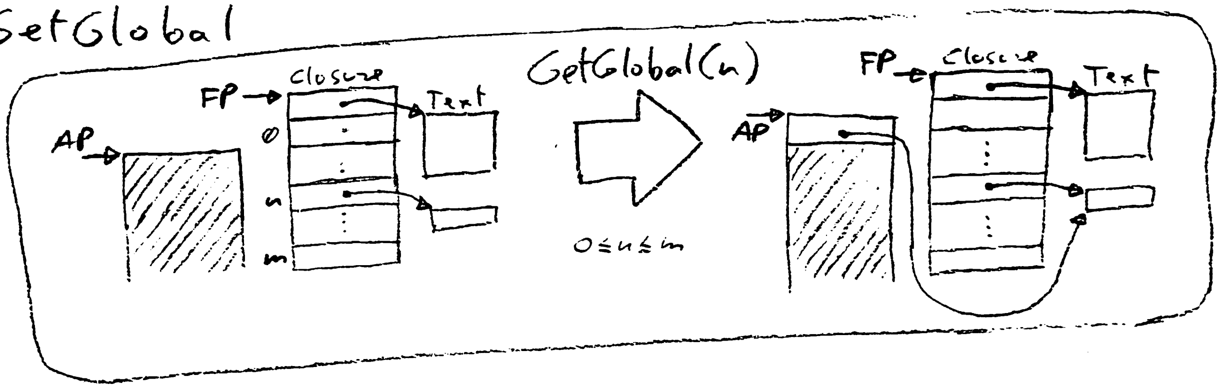
- ClosurePrint

Stack Operations

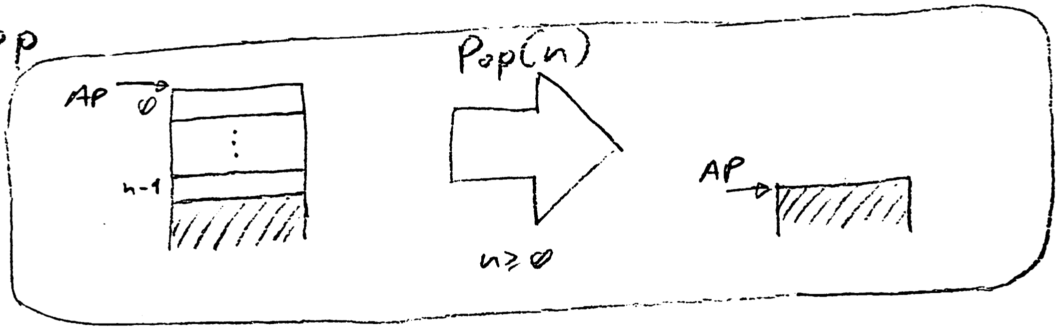
- GetLocal



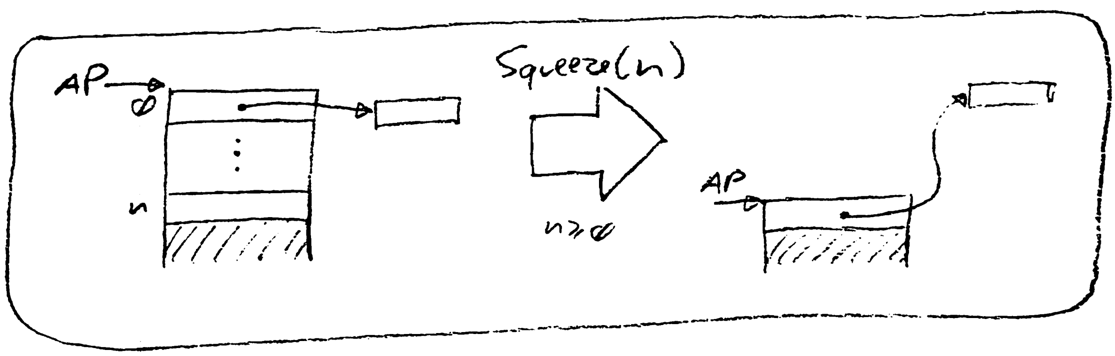
- GetGlobal



- Pop



- Squeeze



- Rise

Control Operations

These are operations affecting the Program Counter and the Stack Pointer.

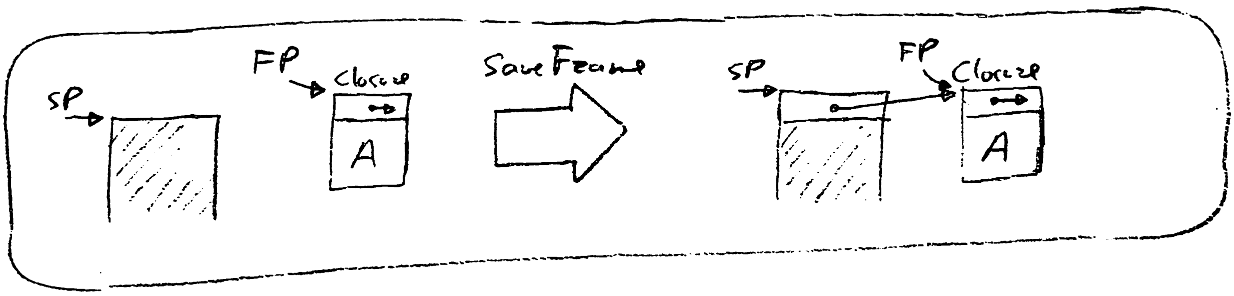
Jump Operations

Jump displacements are expressed in number of skipped instructions. Positive displacements are jumps forward.

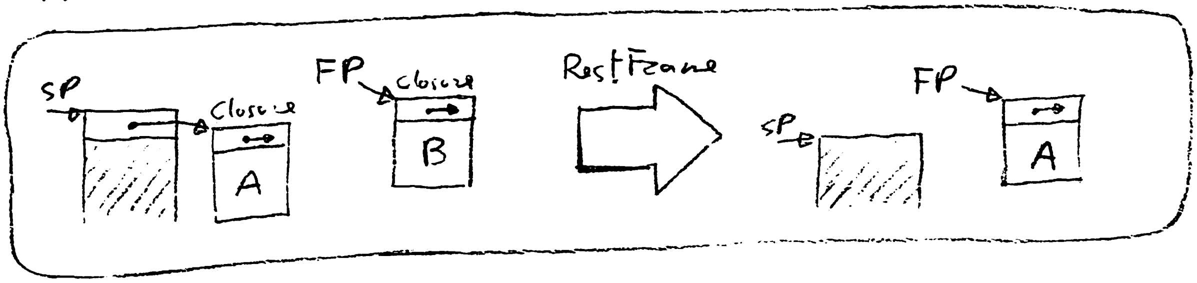
- Jump n (jump leaving AS unchanged)
- TrueJump n (pop AS and jump if the top was "true")
- FalseJump n (pop AS and jump if the top was "false")

Call Operations

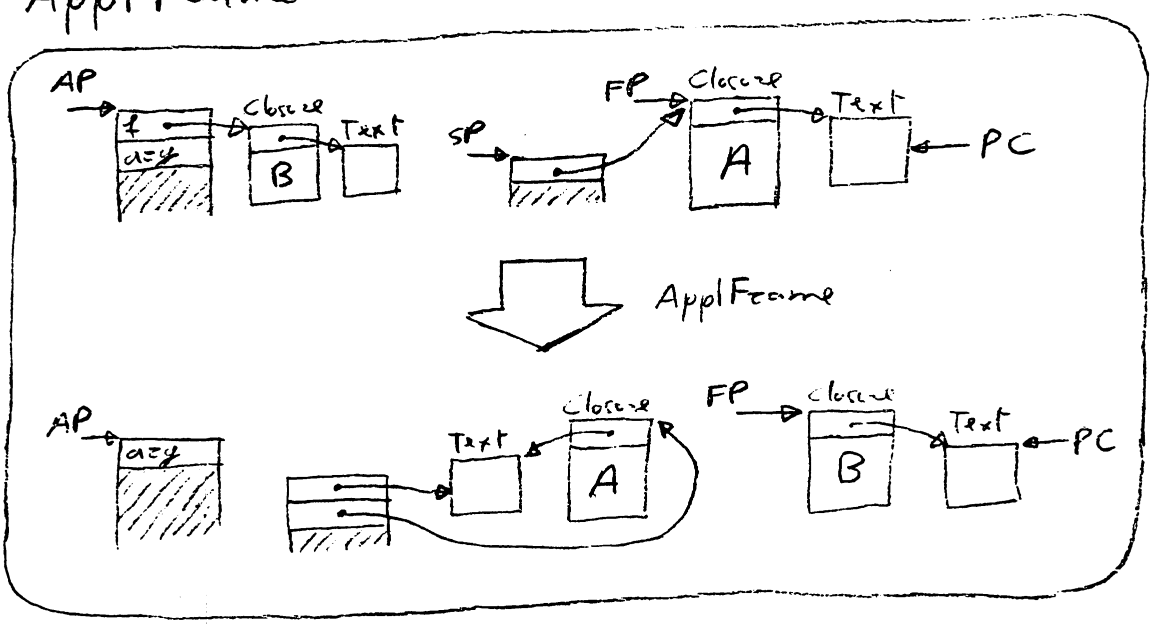
- Save Frame



- Rest Frame

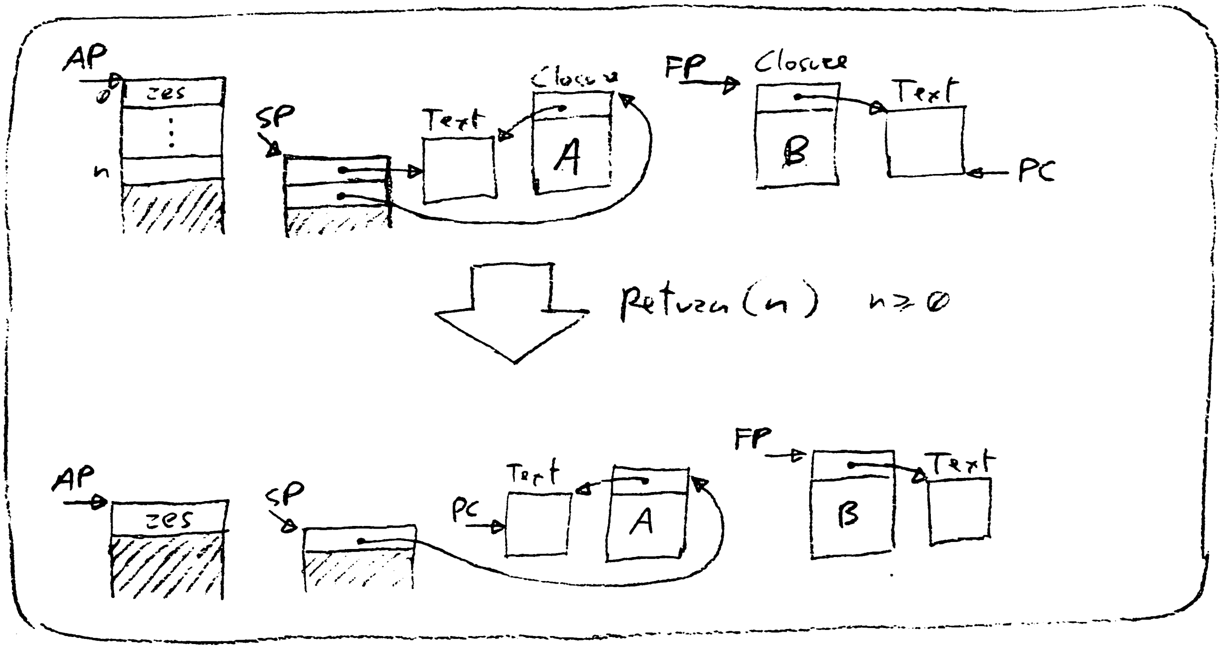


- Appl Frame

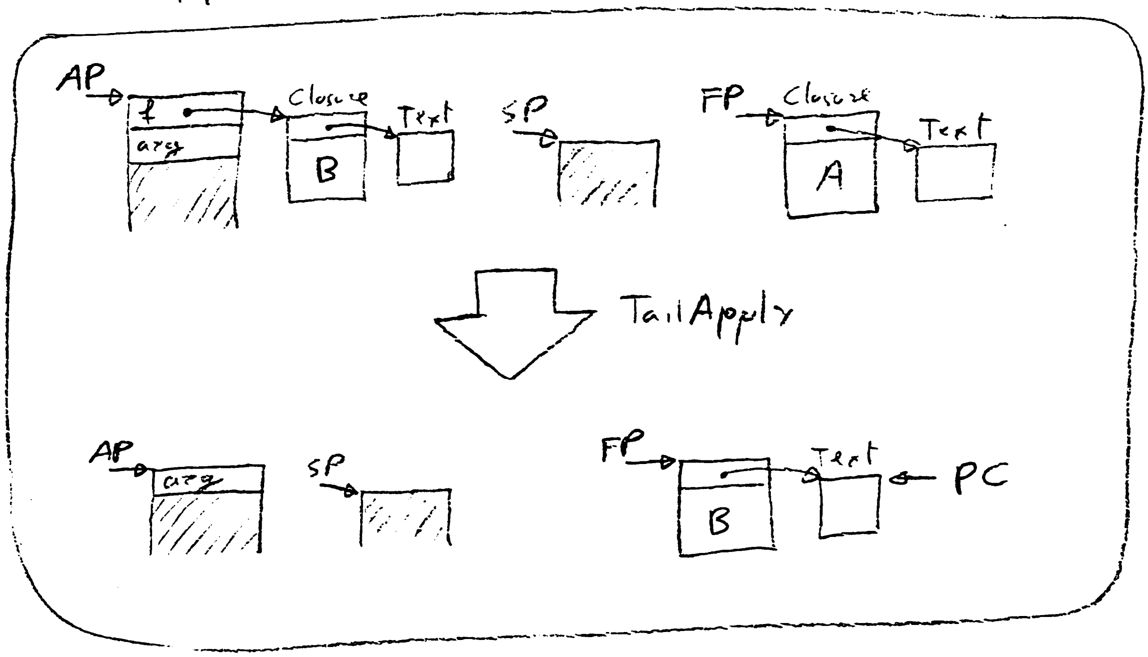


Note: the complete sequence is "SaveFrame; ApplFrame; RestFrame"; A is the callee and B is the called closure

- Return



- TailApply



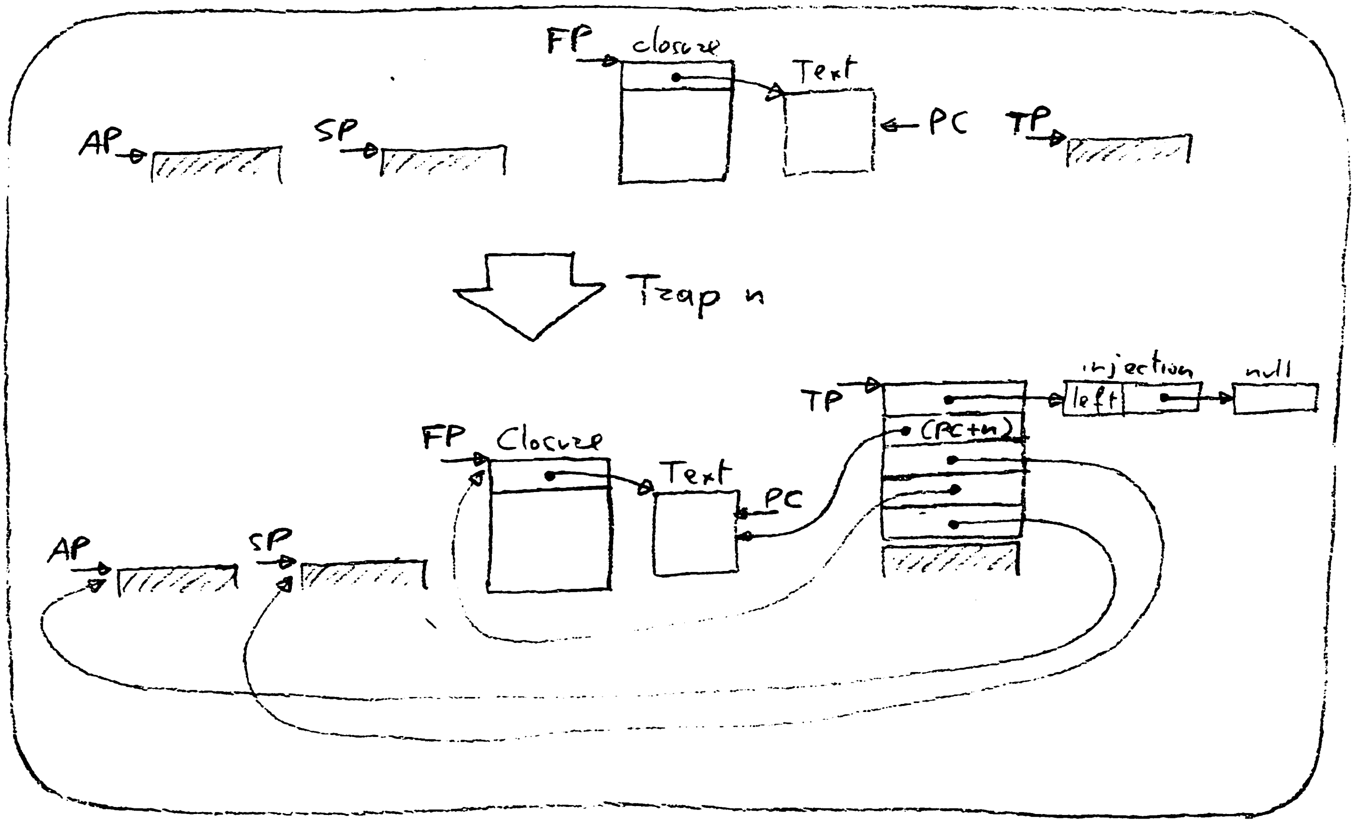
Note: "TailApply" is equivalent to "SaveFrame; ApplFrame; RestFrame; Return"

Fail Operations

A Trap Frame consists of five fields: the first one is a datum of type ".+ tok list" (the trap list) and the other four ones contain PC, SP, FP and AP.
Trap jump displacements are expressed in number of skipped instructions. A typical compilation is

"A ? B" → [Trap L1]"A"[UnTrap L2][L1: Pop 1]"B"[L2:]

- Trap

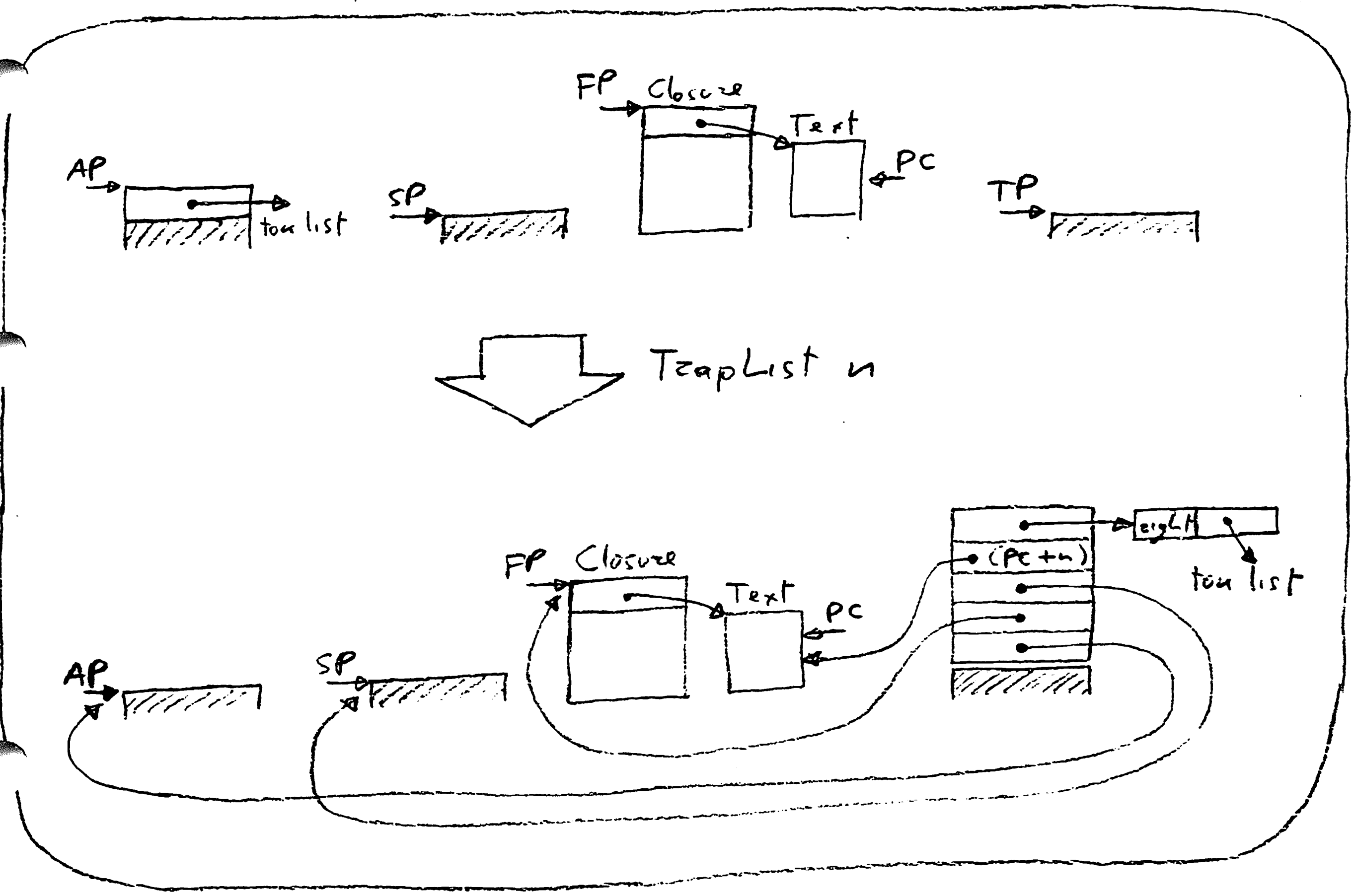


Also:

"A ??"..." B" → "..."[TrapList L1]"A"[UnTrap L2][L1: Pop 1]"B"[L2:]

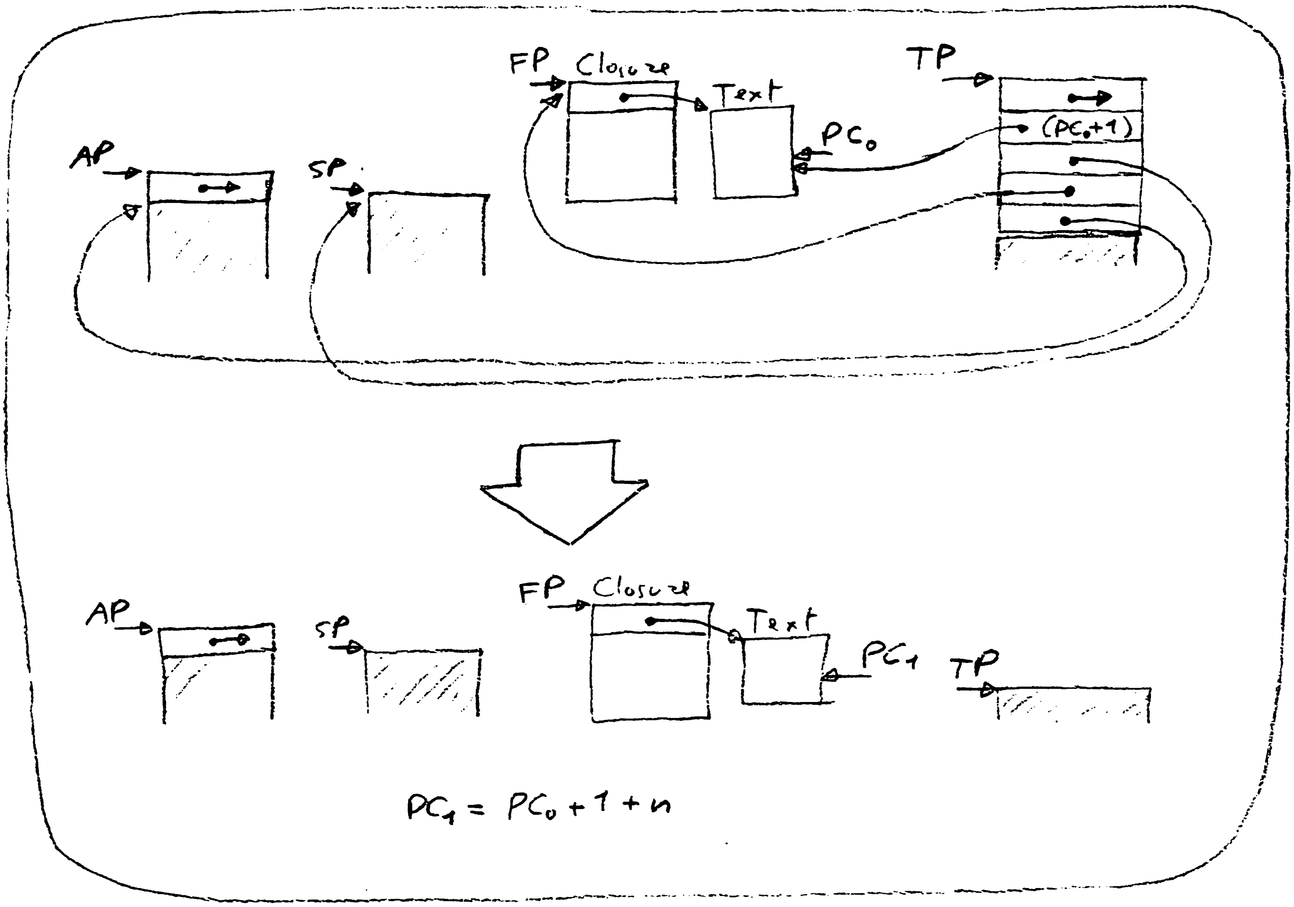
"A ?\x. B" → [Trap L1]"A"[UnTrap L2][L1:]"B"[L2: Squeeze 1]

• TrapList



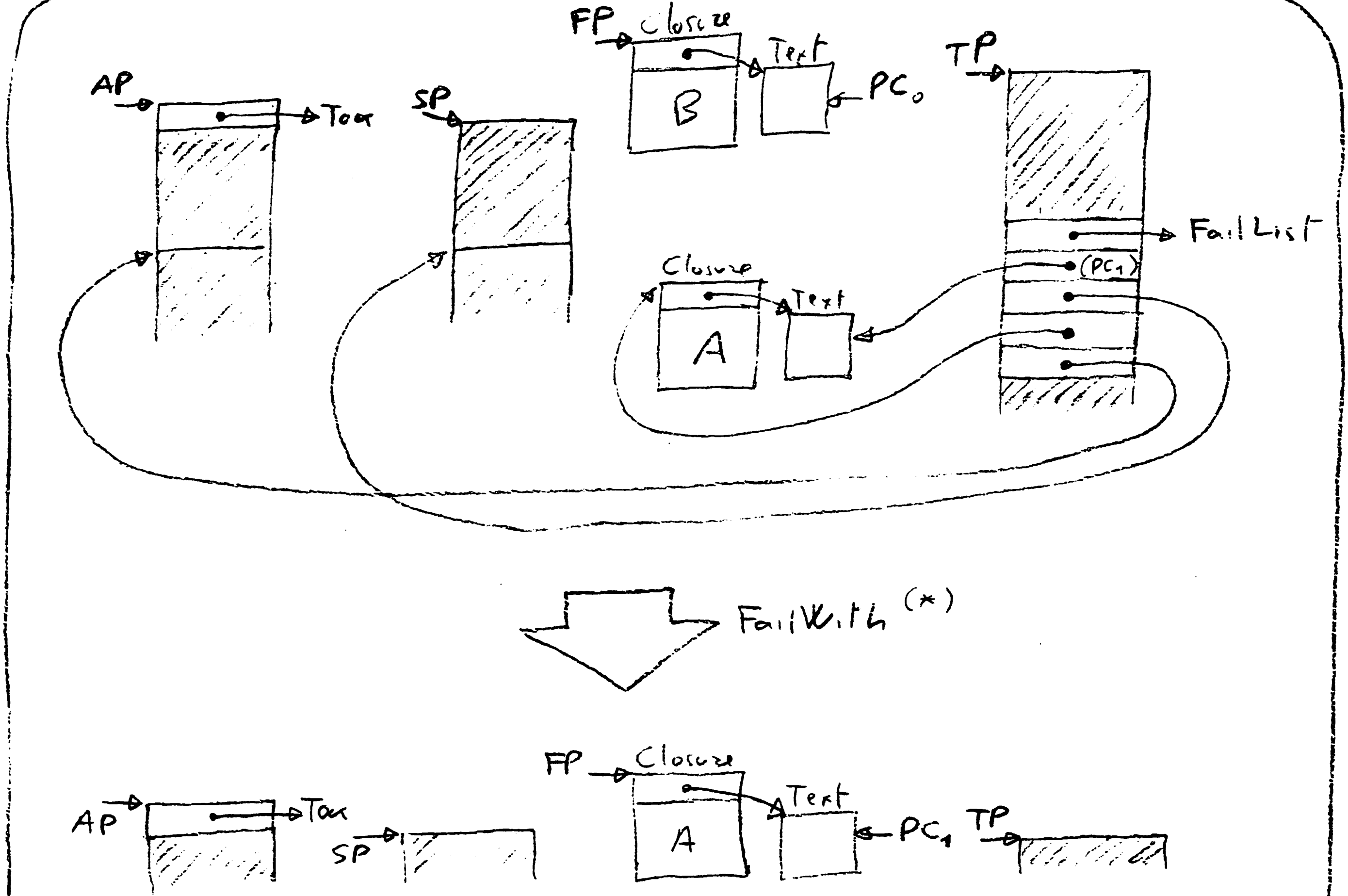
FP
Closure
Text
AP
PC
SP
TP
tan list
TrapList n
FP
Closure
Text
PC
AP
SP
(PC+n)
eighth
tan list

- UnTrap



Note: if execution reaches an UnTrap, then there have been no (untrapped) failures since the corresponding Trap; hence FP and SP are the same, AP has grown by 1 and PC_0 has almost reached the PC stored in the trap frame (the difference being the UnTrap instruction itself). It is then enough to remove the trap frame and skip the failure treatment, jumping to $PC_0 + 1 + n$.

• FailWith



(*) If FailList = nil() or FailList = inr(TouList) where Tou occurs in TouList.
If no such TrapFrame is found scanning the Trap Stack from top to bottom, a top-level failure is generated

Note: PC_1 points to the failure recovery text.