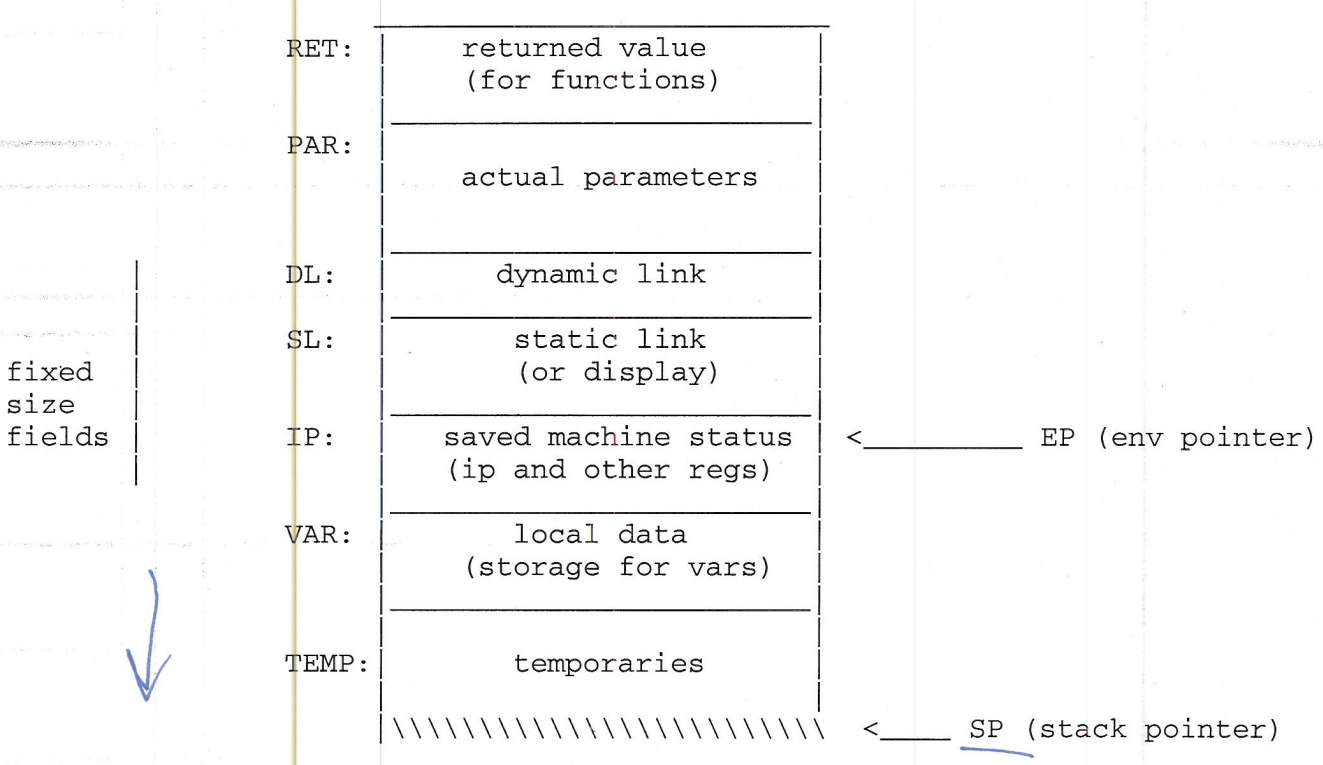
How to access the values of local identifiers in the environment?

Aho and Ullman's design for Activation Record (using static links):



---

does saved part save information about caller or callee?
How would this be used in making a call?
How would this be used in a return?

ii. Last call optimization (2.5.1)

What is a last call optimization?

```
% Fully recursive
fun {Length Lst}
  case Lst of
    _|T then 1+{Length T}
  else 0
  end
end
```

pending computation

Tracing this:

{Length 1|2|3|4|nil}

= 1 + {Length 2|3|4|nil}
= 1 + (1 + {Length 3|4|nil})
= 1 + (1 + (1 + {Length 4|nil}))
= 1 + (1 + (1 + (1 + {Length nil})))
= 1 + (1 + (1 + (1 + 0)))
= 1 + (1 + (1 + 1))
= 1 + (1 + 2)
= 1 + 3
= 4

```
fun {Length Lst}
   {LengthIter Lst 0}
end
% Tail recursive
fun {LengthIter Lst N}
   case Lst of
      _|T then {LengthIter T N+1}
   else N
   end
end
```

an accumulator

tail recursion

returns accumulator

N:=0
while (Lst \= nil)
do N:=@N+1
   Lst:=@Lst.2
end

N,Lst := N+1, Lst.2

Tracing this:

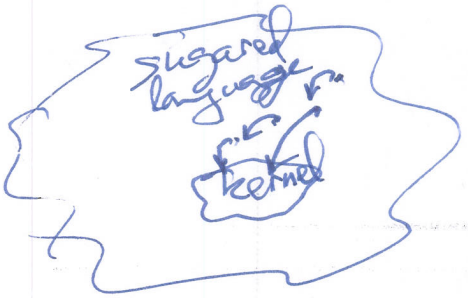
{Length 1|2|3|4|nil}
= {LengthIter 1|2|3|4|nil 0}
= {LengthIter 2|3|4|nil 1+0}
= {LengthIter 3|4|nil 2}
= {LengthIter 4|nil 3}
= {LengthIter nil 4}
= 4

What is it useful for?
Does the semantics already to this?
Do C, C++, and Java require this optimization?
What does that say about using recursion in these languages?
iii. Garbage collection (2.5.2-4)
Why is garbage collection useful?
Does C do garbage collection? C++? Java?
What kinds of error does garbage collection prevent?
Does garbage collection prevent memory leaks?

* dangling reference
- memory leaks
- security issue access to old memory

II. Extensions for a Practical Language (2.6-2.8)
Why not just program in the kernel language?
A. Syntactic conventions (2.6.1)

SYNTACTIC SUGARS FOR VALUES,
DECLARATIONS, AND PATTERNS



Let E1, ..., En be expressions,
lit, f1, ..., fn be literals,
X1, ..., Xn be identifiers

```
   lit(f1:E1, ..., fn:En)   % sugared
==>
   local X1 = E1 in         % desugared
      ...
      local Xn = En in
         lit(f1:X1, ..., fn:Xn)
      end
      ...
   end
```

foo(y:3 x:2+z)

```
   local X = E in S end
==>
```

local X in X=E S end

```
   local X = E1 in E2 end
==>
   local X in X=E1 E2 end
```

R = local X in X=E1 E2 end
⇒ local X in X=E1 R=E2 end

```
   local X1 ... Xn in S end
==>
   local X1 in
     ...
       local Xn in
          S
       end
     ...
   end
```

```
   {P E}
==>
   local X in X=E {P X} end
```

```
e.g.,    {{CAdd 3} 5}
      ==>
```

local Five in Five=5 local Th in Th=3
local F in F={CAdd Th} {F Five}
end end end

```
   local <pattern> = <expression>
   in <statement> end
==>
```

local X1 ... Xn X0 in
X0 = <expression>
X0 = <pattern>
<statement> → {X1 ... Xn} = FVE(<pattern>)
end
declared Hard and X0 is fresh

```
   where
```

```
e.g., local H|T = [7 99] in R=(H+1)|T end
   ==> local H T X0 in
          X0 = [7 99]
          X0 = '|'(1:H 2:T)
          local HP1 = H+1 in
             R='|'(1:HP1 2:T)
          end
       end

e.g., [7 99]
   ==>
     local Seven=7 NN=99 Nil=nil in
       local Tail='|'(1:NN 2:Nil) in
         '|'(1:Seven 2:NN)
       end
     end
-----------------------------------------
   Any conditions on the new identifiers in the local <pattern> =  ... one?
   How does this get back to kernel syntax?
   Why do they preserve meaning?
-----------------------------------------
   PATTERNS IN FUNCTION DEFINITIONS


In general:
```

```
 proc {P ... <pattern>1 ... <pattern>n ...} S end
==>
 proc {P ... X1 ... Xn ...}
    case X1 of <pattern>1 then
       ...
       case Xn of <pattern>n then
          S
       end
       ...
    end
 end

    where X1 ... Xn are fresh
```

Example

```
 fun {First A#_} A end
==>
```

fun {First X1}
case X1 of A#_ then A end
end

How would you call First? {First 40#20}
How could you use this to define a Tail function for lists?
What sugars would help with if statements?

IF-RELATED SUGARS

if <E> then <S1> else <S2> end
⇒ local X=<E> in if X then <S1> else <S2> end end

if <E1> then <S1> elseif <E2> then <S2> ... else <Sn> end
⇒ if <E1> then <S1>
else if <E2> then <S2>
...
else <Sn>
end
end

if <E> then <S> end
⇒ if <E> then <S> else skip end

) only for statements.

How could you desugar the short-circuiting andthen and orelse?
What sugars help with case statements?

CASE-RELATED SUGARS

case <X> of
<pattern> then <S> end
⇒ case <X> of
<pattern> then <S>
else raise error(kernel(noElse ...)) end
end

case <E> of
<P1> then <S1>
[] <P2> then <S2>
[] <P3> then <S3>
...
else <Sn>
end

⇒ local X=<E> in
case X of
<P1> then <S1>
else case X of
<P2> then <S2>
else case X of
<P3> then <S3>
else ...
else <Sn>
end end end

How would you desugar the use of "andthen" in case clauses?
How would you desugar the use of constants in case clauses?
What are these like in C, C++, and Java?

NESTING MARKERS

Use '$'
to turn a statement into an expression

Examples:

would be a statement
written as {Obj get(R)}

```
  R = {Obj get($)}
==>
  {Obj get(R)}
```

≈ local X in {Obj get(X)} end R=X where X is fresh

What's the general rule for this?
How would you translate proc {P X} X=Y end ? ⇒ P = proc {$ X} X=Y end

B. Expressions and Functions (2.6.2)
What's the desugaring rule for fun declarations?
What rule desugars calls to functions?

EXPRESSIONS AND FUNCTIONS

How to translate into kernel syntax:

fun {F X1 ... Xn} E end
⇒ F = proc {$ X1 ... Xn R} R=E end (where R is fresh)

```
   fun {Add1 X} X+3 end
```

⇒ Add1 = proc {$ X R} local Th in Th=3 {Number.'+' X Th R} end end

```
   R = {Add1 3}
```

⇒ local Th in Th=3 {Add1 Th R} end

R = {F E1 ... En}
⇒ local X1=E1 ... Xn=En in {F X1 ... Xn R} end