CSc 553

Principles of Compilation

5 : Procedure Calls

Department of Computer Science
University of Arizona

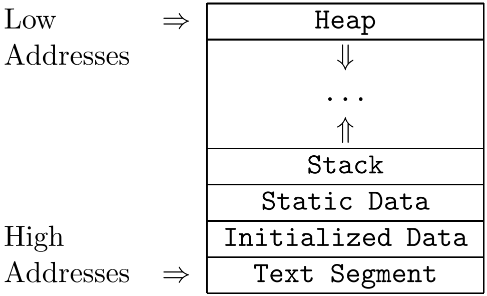
collberg@gmail.com

Introduction

Procedure Calls

- How do we deal with recursion? Every new recursive call should get its own set of local variables.
- How do we pass parameters to a procedure?
    - Call-by-Value or Call-by-Reference?
    - In registers or on the stack?
- How do we allocate/access local and global variables?
- How do we access non-local variables? (A variable is non-local in a procedure `P` if it is declared in procedure that statically encloses `P`.)
- How do we pass large structured parameters (arrays and records)?

Run-Time Memory Organization



- The `Text Segment` holds the code (instructions) of the program. The `Initialized Data` segment holds strings, etc, that don't change. `Static Data` holds global variables. The `Stack` holds procedure activation records and the `Heap` dynamic data.

Storage Allocation I

Global Variables are stored in the `Static Data` area.

Strings (such as `"Bart!"`) are stored in the `Initialized Data` section.

Own Variables are stored in the `Static Data` area. An **Own** variable can only be referenced from within the procedure in which it is declared. It retains its value between procedure calls.

```
PROCEDURE P (X : INTEGER);
   OWN W : INTEGER;
   VAR L : INTEGER;
BEGIN W := W + X; END P
```

Dynamic Variables are stored on the `Heap`:

```
PROCEDURE P ();
   VAR X : POINTER TO CHAR;
BEGIN NEW(X) END P
```

Global Variables – MIPS I

- How do we allocate space for and access global variables? We'll examine three ways.

Running Example:

```
PROGRAM P;
   VAR X : INTEGER;   (* 4 bytes.  *)
   VAR C : CHAR;      (* 1 byte.  *)
   VAR R : REAL;      (* 4 bytes.  *)
END.
```

Global Allocation by Name:

- Allocate each global variable individually in the data section. Prepend an underscore to each variable to avoid conflict with reserved words.
- Remember that every variable has to be aligned on an address that is a multiple of its size.

Global Variables – MIPS II

```
            .data
_X:         .space 4
_C:         .space 1
            .align 2     # 4 byte boundary.
_R:         .space 4
            .text
main:       lw $2, _X
```

Global Allocation in Block:

- Allocate one block of static data (called `_Data`, for example), holding all global variables. Refer to individual variables by offsets from `_Data`.

```
            .data
_Data:      .space 48
            .text
main:       lw     $2, _Data+0     # X
```

Global Variables – MIPS III

Global Allocation on Stack:

- Allocate global variables on the bottom of the stack. Refer to variables through the **Global Pointer** $gp, which is set to point to the beginning of the stack.

```
main:     subu $sp,$sp,48
          move $gp,$sp
          lw    $2, 0($gp)     # X
          lb    $3, 4($gp)     # C
          l.s   $f4, 8($gp)    # R
```

Pros and Cons

`_X: .space 4` Easy, but slow. Each access `lw $2, _X` takes 2 cycles.

`_Data: .space 48` Same as above.

`subu $sp,$sp,48` 1 cycle to access the first 64K global variables.

Storage Allocation II

LocalvVariables: stored on the run-time stack.

Actual parameters: stored on the stack or in special argument registers.

- Languages that allow recursion cannot store local variables in the `Static Data` section. The reason is that every **Procedure Activation** needs its own set of local variables.
- For every new procedure activation, a new set of local variables is created on the run-time stack. The data stored for a procedure activation is called an **Activation Record**.
- Each **Activation Record** (or **(Procedure) Call Frame**) holds the local variables and actual parameters of a particular procedure activation.

Storage Allocation III

- When a procedure call is made the **caller** and the **callee** cooperate to set up the new frame. When the call returns, the frame is removed from the stack.

returned value
actual parameter 1 actual parameter 2 ...
return address
static link
control link
saved registers, etc
local variable 1 local variable 2 ...

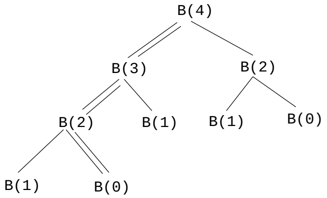
Recursion Examples

Example I (Factorial function):

- R_0 and R_1 are registers that hold temporary results.

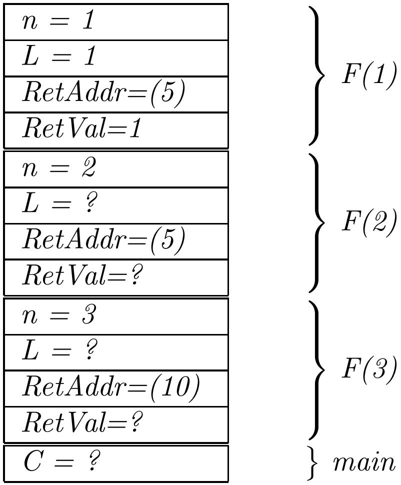
Example II (Fibonacci function):

- We show the status of the stack after the first call to `B(1)` has completed and the first call to `B(0)` is almost ready to return.
- The next step will be to pop `B(0)`'s AR, return to `B(2)`, and then for `B(2)` to return with the sum `B(1)+B(0)`.



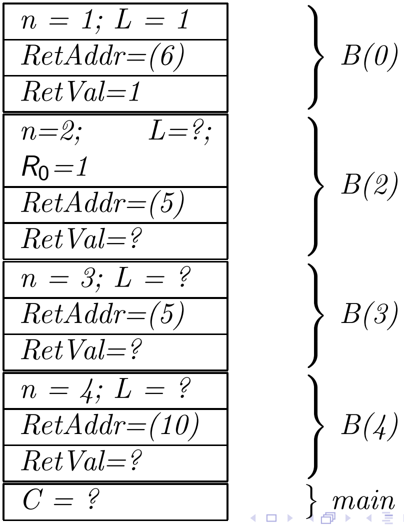
Recursion Example I

```
PROCEDURE F (
  n:INTEGER)
  :INTEGER;
  VAR L:INTEGER;
BEGIN
(1) IF n <= 1
(2) THEN L:=1;
(3) ELSE
(4)   $R_0$:=F(n-1);
(5)   $R_1$:=n;
(6)   L:=$R_0 * R_1$;
(7) ENDIF;
(8) RETURN L;
END F;
BEGIN
    (9) C:=F(3);
    (10)
```



Recursion Example II

```
PROCEDURE B (
 n:INTEGER)
 :INTEGER;
 VAR L:INTEGER;
BEGIN
(1)   IF n <= 1
(2)   THEN L:=1;
(3)   ELSE
(4)     $R_0$:=B(n-1);
(5)     $R_1$:=B(n-2);
(6)     L:=$R_0 + R_1$
(7)   ENDIF;
(8)   RETURN L;
END B;
BEGIN
      (9) C:=B(4);
      (10)
```



Calling Sequences

Procedure Call Conventions

- $\boxed{\textbf{Who}}$ does $\boxed{\textbf{what}}$ $\boxed{\textbf{when}}$ during a procedure call? Who pushes/pops the activation record? Who saves registers?
- This is determined partially the hardware but also by the conventions imposed by the operating system.
- Some work is done by the **caller** (the procedure making the call) some by the **callee** (the procedure being called).

Work During Call Sequence:

- Allocate Activation Record, Set up Control Link and Static Link. Store Return Address. Save registers.

Work During Return Sequence:

- Deallocate Activation Record, Restore saved registers, Return function result Jump to code following the call-site.

Example Call/Return Sequence

The Call Sequence

The **caller**:

- Allocates the activation record, Evaluates actuals, Stores the return address, Adjusts the stack pointer, and Jumps to the start of the **callee**'s code.

The **callee**:

- Saves register values, Initializes local data, Begins execution.

The Return Sequence

The **callee**:

- Stores the return value, Restores registers, Returns to the code following the `call` instr.

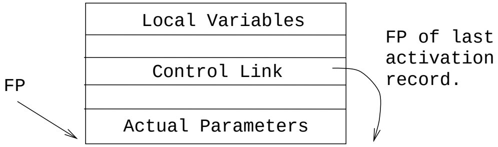
The **caller**:

- Restores the stack pointer, Loads the return value.

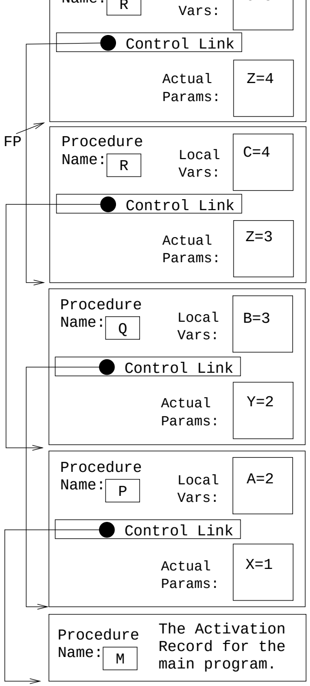
The Control Link

The Control Link I

- Most procedure calling conventions make use of a **frame pointer** (FP), a register pointing to the (top/bottom/middle of the) current activation record.
- Local variables and actual parameters are accessed relative the FP. The offsets are determined at compile time. MIPS example: $\boxed{\texttt{lw \$2, 8(\$fp)}}$.
- Each activation record has a **control link** (aka **dynamic link**), a pointer to the previous activation record on the stack. The control link is simply the stored FP of the previous activation.

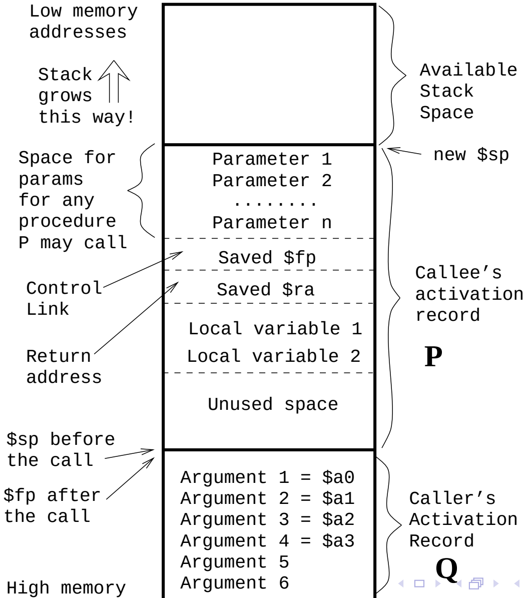


```
    LOCAL A;
      PROC Q(Y);
      LOCAL B;
        PROC R(Z);
        LOCAL C;
        BEGIN
          C:=Z+1;
          R(C);
        END R;
      BEGIN
        B:=Y+1;
        R(B);
      END Q;
    BEGIN
      A:=X+1;
      Q(A);
    END P;
  BEGIN
    P(0);
  END M.
```



Procedure Call on the MIPS

MIPS Activation Record



MIPS Procedure Call I

- Assume that a procedure **Q** is calling a procedure **P**. **Q** is the **caller**, **P** is the **callee**. **P** has **K** parameters.
- **Q** has an area on it's activation record in which it passes arguments to procedures that it calls. **Q** puts the first 4 arguments in registers (`$a0--$a3` $\equiv$ `$4--$7`). The remaining $\mathbf{K} - 4$ arguments **Q** puts in its activation record, at `16+$sp, 20+$sp, 24+$sp` etc. (We're assuming that all arguments are 4 bytes long).
- Note that there is space in **Q**'s activation record for the first 4 arguments, we just don't put them in there.
- We must know the max number of parameters of an call **Q** makes, to know how large to make its activation record.

MIPS Procedure Call II

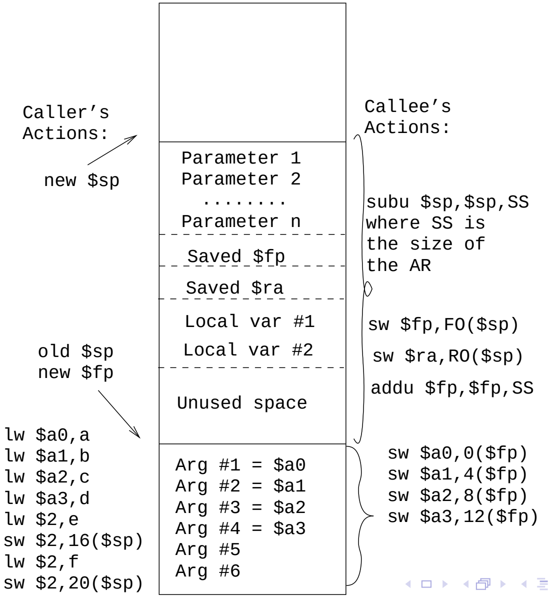
- Next, **Q** executes a `jal` (jump and link) instruction. This puts the return address (the address right after the `jal` instruction) into register `$ra` ($31), and then jumps to the beginning of **P**.
- Before **P** starts executing it's code, it has to set up it's stack frame (activation record). How much space does it need?
  1. Space for local variables,
  2. Space for the control link (old `$fp` 4 bytes).
  3. Space to save the return address `$ra` (4 bytes).
  4. Space for parameters **P** may want to pass when making calls itself.

  Furthermore, the size of the activation record must be a multiple of 8! This can all be computed at compile-time.

MIPS Procedure Call III

- Given the size of the stack frame (`SS`) we can set it up by subtracting from `$sp` (remember that the stack grows towards lower addresses!): `subu $sp,$sp,SS`. We also set `$fp` to point at the bottom of the stack frame.
- If **P** makes calls itself, it must save `$a0--$a3` into their stack locations.
- Procedures that don't make any calls are called **leaf routines**. They don't need to save `$a0--$a3`.
- Procedures that make use of registers that need to be preserved accross calls, must make room for them in the activation record as well.

MIPS Procedure Call IV



```
lw $a0,a
lw $a1,b
lw $a2,c
lw $a3,d
lw $2,e
sw $2,16($sp)
lw $2,f
sw $2,20($sp)
```

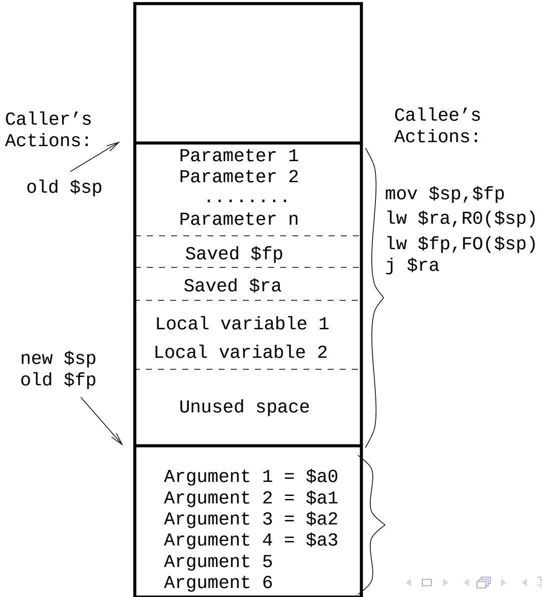
```
subu $sp,$sp,SS
where SS is
the size of
the AR

sw $fp,FO($sp)
sw $ra,RO($sp)
addu $fp,$fp,SS
```

```
sw $a0,0($fp)
sw $a1,4($fp)
sw $a2,8($fp)
sw $a3,12($fp)
```

MIPS Procedure Returns I

- When **P** wants to return from the call, it has to make sure that everything is restored exactly the way it was before the call.
- **P** restores `$sp` and `$fp` to their former values, by reloading the old value of `$fp` from the activation record.
- **P** then reloads the return address into `$ra`, and jumps back to the instruction after the call.

MIPS Procedure Returns II



```
mov $sp,$fp
lw $ra,R0($sp)
lw $fp,FO($sp)
j $ra
```

Parameter Passing

Parameter Passing I – Issues

Value Parameters:

```
PROG M;
   PROC P(X:INT);
   BEGIN X:=5 END P;
VAR S:INT;
BEGIN S:=6; P(S); END.
```

X=5
S=6

- Value parameters are (usually) copied by the caller into the callee's activation record. Changes to a formal won't affect the actual.

Reference (VAR) Parameters:

```
PROC P(VAR X:INT);
BEGIN X:=5 END P;
```

$$\begin{array}{|l|l} \hline X=5 & \} \; P(6) \\ \hline S=5 & \} \; M \\ \hline \end{array}$$

- Reference parameters are passed by passing the address (location, l-value) fo the parameter. Changes to a formal affects the actual also.

Call-by-Value Parameters

Algorithm:

1. The caller computes the arguments' *r-value*.
2. The caller places the *r-values* in the *callee*'s activation record.

Consequences:

- The caller's actuals are never affected by the call.
- Copies may have to be made of large structures.

Example:

```
TYPE T = ARRAY 10000 OF CHAR;
PROC P (a:INTEGER; b:T);
BEGIN a:=10; b[5]:="4" END P;

VAR r :  INTEGER; X : T;
BEGIN P(r, X) END
```

Call-by-Reference Parameters

Algorithm:

1. The caller computes the arguments' *l-value*.
2. Expression actuals (like $a + b$) are stored in a new location.
3. The caller places the *l-value*s in the *callee*'s activation record.

Consequences:

- The caller's actuals may be affected by the call.

Example:

```
TYPE T = ARRAY 10000 OF CHAR;
PROC P (VAR a:INT; VAR b:T);
BEGIN a:=10; b[5]:="4" END P;

VAR r :  INTEGER; X : T;
BEGIN P(5 + r, X) END
```

Call-by-Name Parameters I

- (Un-)popularized by Algol 60.
- A name parameter is (re-)evaluated
    - every time it is referenced,
    - in the callers environment.

Algorithm:

1. The caller passes a *thunk*, a function which computes the argument's *l-value* and/or *r-value*, to the callee.
2. The caller also passes a static link to its environment.
3. Every time the callee references the name parameter, the thunk is called to evaluate it. The static link is passed to the thunk.

Call-by-Name Parameters II

Algorithm:

4. If the parameter is used as an l-value, the thunk should return an l-value, otherwise an r-value.
5. If the parameter is used as an l-value, but the actual parameter has no l-value (it's a constant), the thunk should produce an error.

Consequences:

- Every time a callee references a name parameter, it may produce a different result.

Call-by-Name Parameters III

```
VAR i :  INTEGER;
VAR a :  ARRAY 2 OF INTEGER;

PROC P (NAME x:INTEGER);
BEGIN
   i := i + 1;
   x := x + 1;
END;

BEGIN
   i := 1; a[1] := 1; a[2] := 2;
   P(a[i]);
   WRITE a[1], a[2];
END
```

- $\boxed{\texttt{x := x + 1}}$ becomes $\boxed{\texttt{a[i] := a[i] + 1}}$.

Call-by-Name Parameters IV

Implementation:

```
VAR i : INTEGER;
VAR a : ARRAY 2 OF INTEGER;

PROC P (thunk : PROC());
BEGIN
   i := i + 1;
   thunk()↑ := thunk()↑ + 1;
END;

PROC thunk1 () : ADDRESS;
BEGIN RETURN ADDR(a[i]) END;

BEGIN
   i := 1; a[1] := 1; a[2] := 2;
   P(thunk1);
```

Call-by-Name V – Jensen's Device

```
PROC Sum (
   NAME Expr : REAL;
   NAME Idx : INTEGER;
   Max : INTEGER) : INTEGER;
VAR Result : REAL := 0;
BEGIN
   FOR i := 1 TO Max DO;
      Idx := i;
      Result := Result + Expr;
   ENDFOR;
   RETURN Result;
END;
VAR i : INTEGER;
BEGIN
   WRITE Sum(i, i, 5);        (* $\sum_{i=1}^{5} i$ *)
   WRITE Sum(i*i, i, 10);     (* $\sum_{i=1}^{10} i^2$ *)
```

Large Value Parameters I

- Large value parameters have to be treated specially, so that a change to the formal won't affect the actual. Example:

```
TYPE T = ARRAY [1..1000] OF CHAR;
   PROCEDURE P (x :  T);
   BEGIN x[5] := "f"; END P;
VAR L : T; BEGIN P(L); END
```

Algorithm 1: Callee Copy

```
PROCEDURE P (VAR x :  T);
VAR xT : T;
BEGIN copy(xT,x,1000);xT[5]:="f"; END P;
VAR L : T; BEGIN P(L); END
```

Large Value Parameters II

```
TYPE T = ARRAY [1..1000] OF CHAR;
   PROCEDURE P (x :  T);
   BEGIN x[5] := "f"; END P;
VAR L : T; BEGIN P(L); END
```

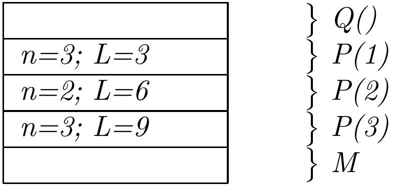
Algorithm 2: Caller Copy

```
PROCEDURE P (VAR x :  T);
BEGIN x[5] := "f"; END P;
VAR L : T; LT : T;
BEGIN copy(LT, L, 1000); P(LT); END
```

Access to Non-Local Names

Accessing Non-Local Variables I

```
PROGRAM M;
  PROC P(n);
  LOCAL L;
    PROC Q();
    BEGIN PRINT L; END Q;
  BEGIN
    L := n * 3;
    IF n >= 1
      THEN P(n-1);
      ELSE Q();
    ENDIF;
  END P;
BEGIN P(3); END M.
```



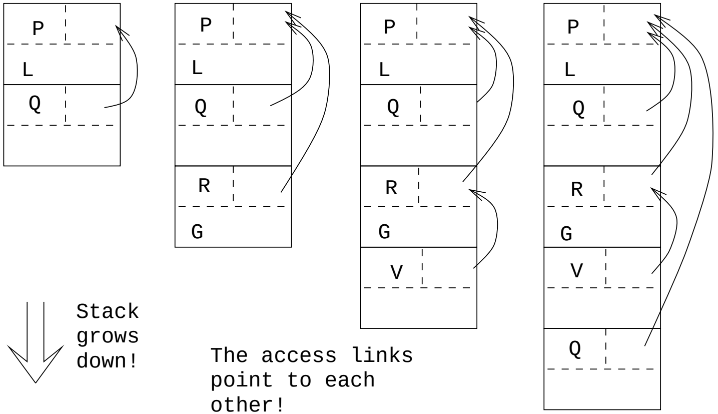
- Which `L` should `Q` print? There are three `L`s on the stack to choose from!
- `Q` should print the `L` from the topmost `P` on the stack.

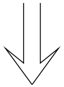
Accessing Non-Local Variables II

```
PROCEDURE P (a:INTEGER);
   VAR L : INTEGER:
   PROCEDURE Q (x:INTEGER);
   BEGIN R(16) END Q;

   PROCEDURE R (y:INTEGER);
      VAR G : INTEGER:
      PROCEDURE V (z:INTEGER);
      BEGIN Q(10) END V;
   BEGIN V(12) END R;

BEGIN Q (5); END P;
```

- We give each activation record an **Access Link** (aka **Static Link**).
- Assume that Q is nested within P (as above). Then Q's static

Accessing Non-Local Variables III





Stack
grows
down!

The access links
point to each
other!

Accessing Non-Local Vars. IV

```
PROC P ();
    VAR L:INTEGER;               ⇐ $n_L = 1$
    PROC R ();
        PROC V ();               $n_R - n_L = 2$
        BEGIN L:=...END V;       ⇐ $n_R = 3$
```

Access to non-local variable L:

- Assume that L is declared at nesting level n_L, and that the reference to L is at nesting level n_R (as above).
- Follow $n_R - n_L$ access links. We now point to the activation record for the most recent activation of P.

MIPS Example:

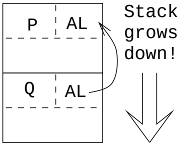
```
lw $2, AL($fp)  # AL is offset of access link.
lw $2, ($2)     # An access link points to
                # the previous access link.
lw $3, 12($2)   # Get the data in the AR
```

Setting up Access Links I

- Every time we make a procedure call we have to set up the access link for the new procedure activation.
- There are two cases to consider: (1) when the callee is nested within the caller, and (2) when the caller is nested within the callee.

Case (1): Callee Within Caller:

```
PROC P();         ⇐ $N_P = 1$
    PROC Q();     ⇐ $N_Q = 2$
        PROC V();
        ...
BEGIN Q (); END P;
```



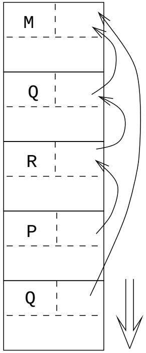
- P calls Q. P is at level N_P, Q is at level N_Q. $N_P = N_Q - 1$, since Q must be nested immediately within P.
- Make Q's access link point to the access link in P's activation record.

Setting up Access Links II

Case (2): Caller Within Callee:

```
PROG M;
  PROC Q();        ⇐ $N_Q = 1$
    PROC R();
      PROC P();    ⇐ $N_P = 3$
      BEGIN
        Q();
      END P;
```

$N_P - N_Q + 1 = 3$



- P calls Q. P is at level N_P, Q is at level N_Q. $N_P \geq N_Q$.
- Traverse the access links to find the most recent activation of the first procedure which statically encloses both P and Q.
- We need to follow $N_P - N_Q + 1$ links.

Summary

Summary I

- Read the Dragon Book:

Procedures	389–394
Storage Organiz.	396–397, 401–404
Activation Records	398–400
Calling Sequences	404–408
Lexical Scope	411, 415–418
Parameter Passing	424–427

- Each procedure call pushes a new activation record on the run-time stack. The AR contains local variables, actual parameters, a static (access) link, a dynamic (control) link, the return address, saved registers, etc.
- The frame pointer (FP) (which is usually kept in a register) points to a fixed place in the topmost activation record. Each local variable and actual parameter is at a fixed offset from FP.

Summary II

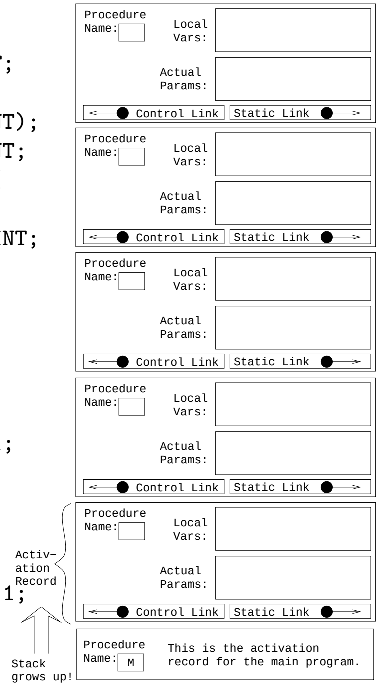
- The dynamic link is used to restore the FP when a procedure call returns.
- The static link is used to access non-local variables, i.e. local variables which are declared within a procedure which statically encloses the current one.
- A parameter is often passed by the caller copying it (or its address, in case of **VAR** parameters) into the callees activation record. On the MIPS, the caller has an area in its own activation record in which it puts actual parameters before it jumps to the callee. For each procedure `P` the compiler figures out the maximum number of arguments `P` passes to any procedure it calls. The corresponding amount of memory has to be allocated in `P`'s activation record.

Exam Problems

Exam Problem 415.330/96 (A)

- Show the status of the run-time stack when execution has reached point $\diamond$ **for the second time** in the program on the next slide. Fill in the name of each procedure invocation in the correct activation record. Also fill in the values of local variables and actual parameters, and show where the static links and control links are pointing. Assume that all actual parameters are passed on the stack rather than in registers.

```
PROGRAM M;
 PROC P (
  X:INT);
  VAR W:INT;
  PROC Q (
   VAR Z:INT);
   VAR N:INT;
   PROC R (
    V:INT);
    VAR L:INT;
   BEGIN
    L := W;
    P(23);
   END R;
  BEGIN
   N := W;
   Z := Z+1;
   ◇
   R(Z);
  END Q;
 BEGIN
  W := X + 1;
  Q(W);
 END P;
BEGIN
```



Homework

Homework I

- Draw the stack when control reaches point ◇ for **the third time**. Include all actual parameters, local variables, return addresses, and static and dynamic links.

```
PROGRAM M;
   PROCEDURE P(X:INTEGER);
   VAR A : INTEGER;
      PROCEDURE Q(Y : INTEGER);
      VAR B : INTEGER;
      BEGIN
         B := Y + 1; A := B + 2;
         ◇
         P(B);
      END Q;
   BEGIN
      A:= X + 1; Q(A);
   END P;
```