CS 536 Announcements for Monday, April 17, 2023

Last Time

- parameter passing
- terminology
- different styles
  - what they mean
  - how they look on the stack
  - compare and contrast

Today

- how do we deal with variables and scope?
- how do we organize activation records?
- how do we retrieve values of variables from activation records?

Next Time

- code generation

Accessing variables at runtime

local variables

- declared and used in the same function
- further divided into "block" scope in brevis

global variables

- declared at the outermost level of the program

In C/C++/Brevis - globals integer x;

In Java - class (static) data members

↑ Java keyword

non-local variables (i.e., from nested scopes)

- for static scope: variables declared in an outer scope
- for dynamic scope: variables declared in the calling context

Compile versus runtime

nested classes (Java)

nested procedures (Pascal)

Accessing local variables at runtime

Local variables

- includes parameters and all local variables in a function
- stored in activation record of function in which they are declared
- accessed using offset from frame pointer

Accessing the stack

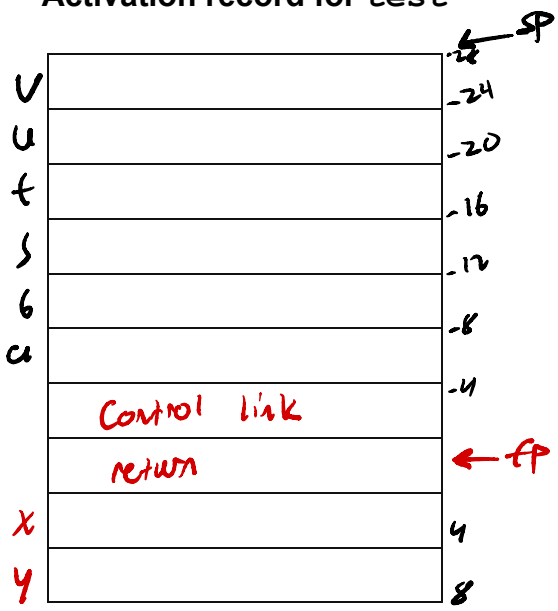
- general anatomy of MIPS instruction

  opcode operand1 operand2

- use "load" and "store" instructions
  - every memory cell has an address
  - calculate that memory address, then move data from/to that address

```
void test(int x, int y) {
    int a, b;
    ...
    if (...) {
        int s;
        ...
    }
    else {
        int t, u, v;
        ...
        u = b + y;
    }
}
```

Activation record for `test`



MIPS code for `u = b + y`

comment symbol

```
lw $t1, -12($fp) #load b
```

load contents of a given address into given register

q: rey, offset, framePointer

load word

```
lw $t2, 8($fp) #load y
add $t3, $t1, $t2 #b+y
sw $t3, -24($fp) #store u
```

Simple memory-allocation scheme

Reserve a slot for each variable in the function

Algorithm

For each function

```
set offset = +4

for each parameter
    add name to symbol table
    offset += size of parameter

offset = -4

offset -= size of callee saved registers

for each local
    offset -= size of variable
    add name to symbol table
```

Implementation

- add an offset field to each symbol table entry
- during name analysis, add the offset along with the name
- walk the AST performing decrements at each declaration node

Example

```
void test(int x, int y) {
    int a, b;
    if (...) {
        int s;
    }
    else {
        int t, u, v;
        u = b + y;
    }
}
```

Accessing global variables at runtime

Place in static data area

- in MIPS, handled with a special storage directive
- variables referred to by name, not address

Note

- space allocated directly at compile time
- never needs to be deallocated

Example

```
.data
_x: .word 10
_y: .byte 1
_z: .asciiz "this is a string"

.text
lw $t0, _x  # load from x into $t0
sw $t0, _x  # store from $t0 into x
```

Accessing non-local variables at runtime

Two situations

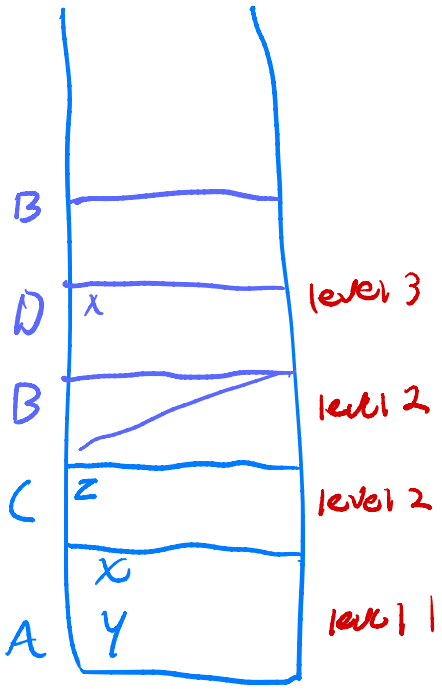
- static scope
  - variable declared in one procedure and accessed in a nested one
- dynamic scope
  - any variable x that is not declared locally resolves to instance of x in the AR closest to the current AR

Example: static non-local scope

```
function main() {
    int a = 0;

    function subprog() {
       a = a + 1;
    }
}
```

Example: static non-local scope

```
void procA() {//level 1
   int x, y;
   void procB() {
      print x; x1 (always)
   }
   void procC() { //level 2
      int z;
      void procD() { //level 3
         int x;
         x = z + y; x3 = z2 + y1
         procB();
      }

      x = 4; x1 = 4
      z = 2; z2 = 2
      procB();
      procD();
   }
   x = 3;
   y = 5;
   procC();
}
```



Access links

Add additional field in the AR (called access link, or static link)

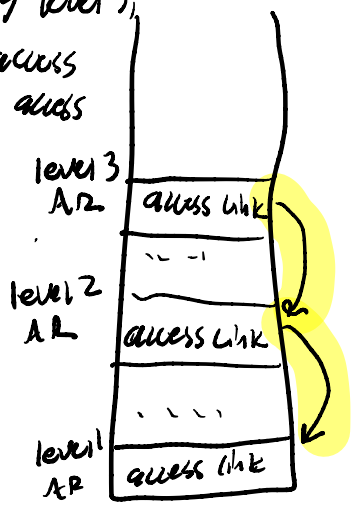
Points to local envt (or fp) of enclosing function

How access links work

- we know how many *levels* to traverse statically

if the current scope is at nesting level 3, and the variable we want to access is at level 1, we go back 2 access links.

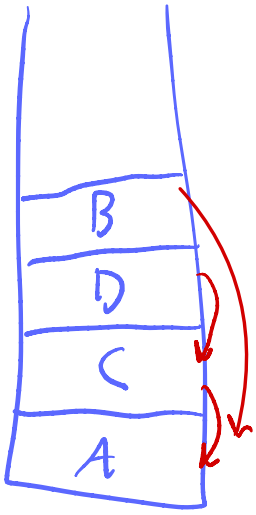
3 − 1 = 2 levels



Access links (cont.)

Setting up access links

```
void procA() {  // level 1
   int x, y;
   void procB() {  // level 2
      print x;
   }
   void procC() {  // level 3
      int z;
      void procD() {
         int x;
         x = z + y;
         procB();
      }

      x = 4;
      z = 2;
      procB();
      procD();
   }
   x = 3;
   y = 5;
   procC();
}
```



Handling use of non-local variable x (at compile time)

- each variable keeps track of nesting level in which it is declared
- when x is used in procedure P
  - follow predetermined # of links to get to AR for procedure in which x is declared

Lx = level of x's declaration Lp = level of p

links to follow if Lp - Lx

Make $fp be location of access link

MIPS

```
lw $to, 0($fp)   # 1 link followed
lw $to, ($to)    # 2 links followed
lw $to, -12($to) # use x's offset in AR of declaring procedure
```

Using a display

Idea: avoid run-time overhead of following access links by having a global array (called the display) containing links to the procedures that lexically enclose the current procedure

How it works

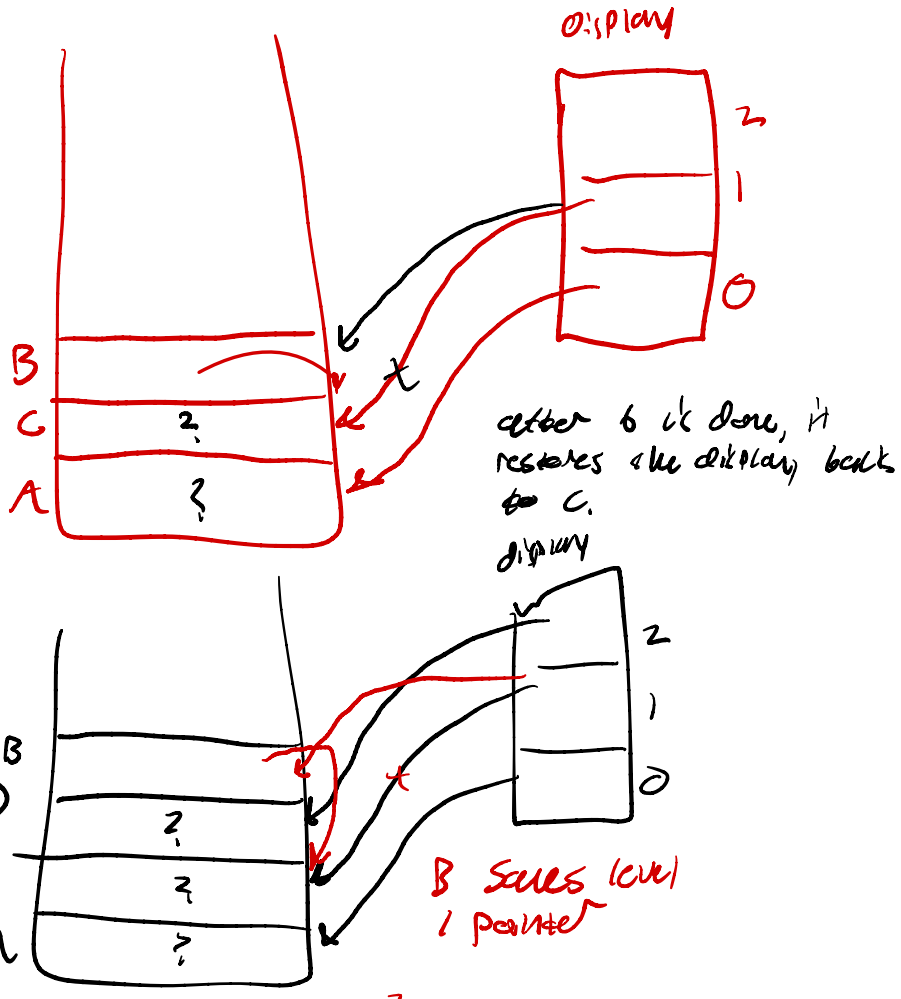
- given procedure `P` at nesting level `k` is currently executing
- `display[0], display[1], ..., display[k-2]` hold pointers to ARs of the most recent activations of the `k-1` procedures that enclose `P`
- `display[k-1]` holds pointer to `P` 's AR
- to access non-local variable `x` declared in nesting level `n`
  - use `display[n-1]` to get to AR that holds `x`
  - then use regular offset (within AR) to get to `x`

How to maintain the display in the code

- add new "save-display" field to AR
- when procedure `P` at nesting level `k` is called
  - save current value of `display[k-1]` in save-display field of `P` 's AR
  - set `display[k-1]` to point to save-display field of `P` 's AR
- when procedure `P` is ready to return
  - restore `display[k-1]` using value in save-display field

Example

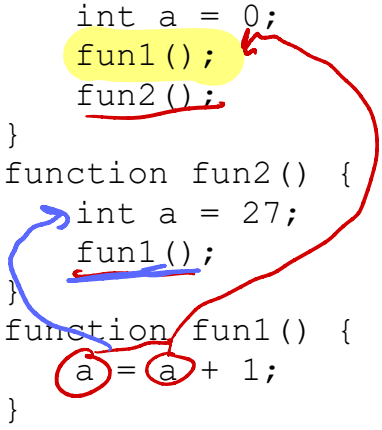
```
void procA() {
   int x, y;
   void procB() {
      print x;
   }
   void procC() {
      int z;
      void procD() {
         int x;
         x = z + y;
         procB();
      }

      x = 4;
      z = 2;
      procB();
      procD();
   }
   x = 3;
   y = 5;
   procC();
}
```



Dynamic non-local scope

Example

```
function main() {
    int a = 0;
    fun1();
    fun2();
}
function fun2() {
    int a = 27;
    fun1();
}
function fun1() {
    a = a + 1;
}
```



Key point – we don't know *which* non-local variable we are refering to (can compile time)

Two ways to set up dynamic access

- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

Deep access

- if the variable isn't local
  - follow control link to caller's AR
  - check to see if it defines the variable
  - if not, follow the next control link down the stack
- note that we need to know if a variable is defined *with that name* in an AR
  - usually means we'll have to associate a name with a stack slot

Shallow access

- keep a table with an entry for each variable declaration
- compile a direct reference to that entry
- at function call on entry to function F
  - F saves (in its AR) the current values of all variables that F declares itself
  - F restores these values when it finishes