CS536 Lecture 18

Thursday 9 April 2015

Last class:

- Runtime Environment
    - Storage

Today:

- Variable Access

Variable Access at Runtime

Three kinds of variables:

Local:

Global:

Non-local:

Accessing Local Variables at Runtime

Includes parameters, declarations in all block scopes.

Stored in the AR of the current function.

Accessed using offsets from FP (negative, since stack grows upwards).

Note: The type of variable is important because the offset is in number of bytes.

In MIPS:

```
opcode Operand1 Operand2
```

Basic memory operations: Use “load” and “store” instructions, referencing memory cell *address*.

```
lw register memoryAddress

sw register memoryAddress
```

Example:

```
lw   $t1   -8($fp)



sw   $t1   0($fp)
```

Simple Memory Allocation Algorithm

Scheme: Reserve a slot for each variable in the function

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

For each function:

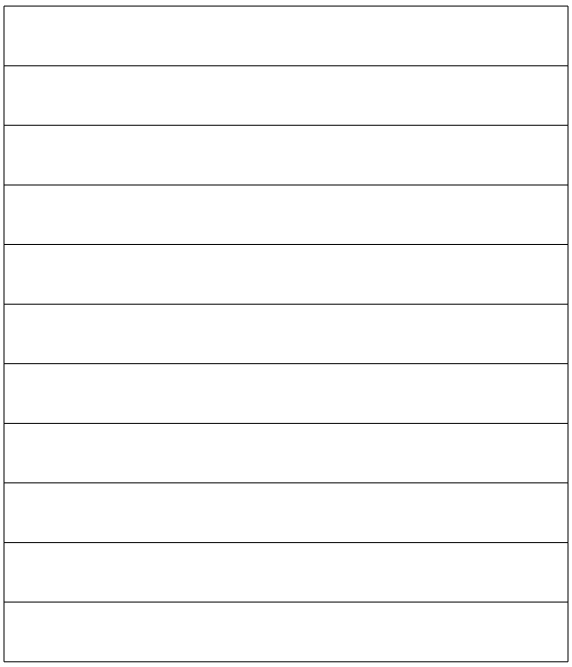
```
set offset = 0
for each parameter:
    add name to symbol table
    offset -= size of parameters
offset -= size of return address
offset -= size of control link
offset -= size of callee-saved registers

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

Implementation:

- Add an offset to each symbol table entry
- Add offset along with name during name analysis (P6)
- Walk AST, perform decrements at each declaration node

Algorithm Example

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



MIPS code:

Global Variables

Less involved, easier to handle

- Space allocated at compile-time
- No de-allocation needed

MIPS stores them in a different area of memory altogether (static data area) labeled `.data .`

Example:

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

Format: `label: type initial_value`

Type can be `.word`, `.byte`, `.ascii`, or `.asciiz`

They are then accessed by name: `lw reg, label`

```
.text
lw $t0, _x # Load from x into register t0
sw $t0, _x # Store from register t0 into x
```

Non-local variables

Dependent on scoping method:

- In static scoping, this refers to variables declared in a nested procedure (in languages that allow it)

  Example:

```
function main() {
    a = 0;
    function sub() {
        a = a + 1;
    }
}
```

- In dynamic scoping, it refers to any variable not locally declared

(Non-local) Static Scope Example

Each function has its own AR, and the "inner" function can access the outer AR.

Example:

```
void procA() { // level 1
    int x, y; // x1
    void procB() { // level 2
        print x; // always prints ____
    }

    void procC() { // level 2
        int z;
        void procD() { // level 3
            int x;
            x = z + y; // _________________
            procB();
        }

        x = 4; // refers to ____

        z = 2; // refers to ____
        procB();
        procD();
    }
    x = 3; // _______

    y = 5; // _______
}
```

But how?

Access Links

Add another field to the AR, pointing to locals area of outer function.

Keep track of “nesting level”

One access link:

```
lw   $t0,-4($fp)
lw   $t0,-12($t0)
```

Two access links:

```
lw   $t0,-4($fp)
lw   $t0,($t0)
lw   $t0,-12($t0)
```

Displays:

Idea: Cache the frame pointers for each nesting level in display registers.

Dynamic non-local scope

Example:

Deep Access

Shallow Access