Class Announcements

- Fourth homework: Will be posted soon.
- Updated project description with clarifications; updated sample solution for strengthreduc.sol; deadcode.sol now available; two more test cases; please see files/projects on canvas.
- Office hours today: 2:00pm - 3:00pm

Project 1 - Dead Code Elimination Pass

An ILOC instruction is dead and can be removed if it does not contribute to the output of the program. Output means I/O behavior, so memory accesses are not I/O.

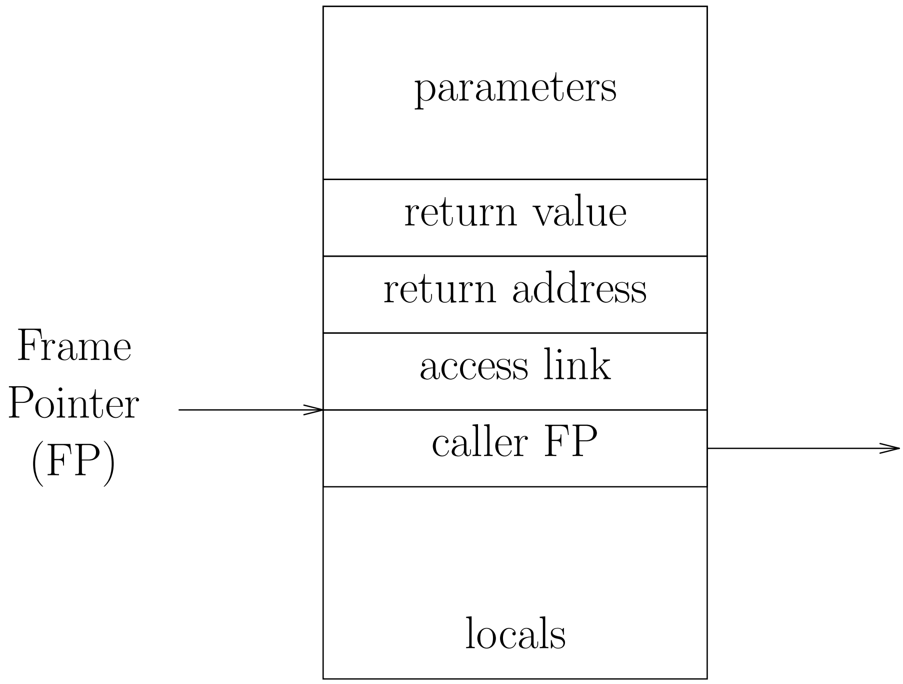
A POSSIBLE ALGORITHM

1. Mark all `outputAI` instructions as critical, and all other instructions as non-critical.
2. For each critical instruction, find the instruction(s) that determine(s) the value or outcome of that instruction. For operations on registers (e.g. `add, lshiftI`), this will be the instructions that compute the values of the operand registers. For `loadAI`, it is the instruction that writes a value into the memory location. For `storeAI`, it is the instruction that computes the register value to be written to memory.
3. Once no more critical instructions can be found, delete all non-critical instructions.

Stack Frame, Activation Record

Scott: Chap. 8.1 - 8.2; ALSU Chap. 7.1 - 7.3

- Run-time stack contains frames for main program and each active procedure.
- Each stack frame includes:
  1. Pointer to stack frame of caller (control link for stack maintainance and dynamic scoping)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (access link for lexical scoping)
  4. Storage for parameters, local variables, and final values



Context of Procedures

Two contexts:

- *static* placement in source code (same for each invocation)
- *dynamic* run-time stack context (different for each invocation)

Scope Rules

Each variable reference must be associated with a single declaration (ie, an offset within a stack frame).

Two choices:

1. Use static and dynamic context: *lexical scope*
2. Use dynamic context: *dynamic scope*

- Easy for variables declared locally, and same for *lexical* and *dynamic* scoping
- Harder for variables not declared locally, and not same for *lexical* and *dynamic* scoping

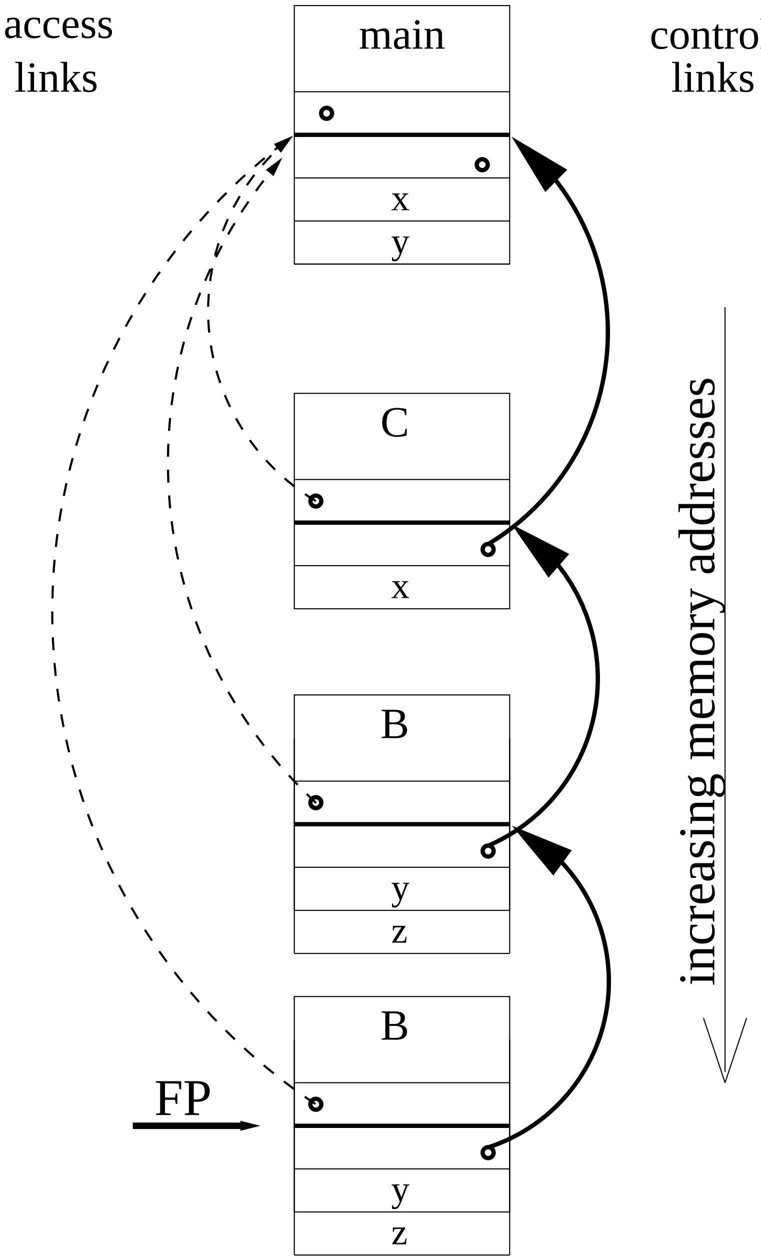
Lexical Scoping Example

scope of a declaration: Portion of program to which the declaration applies

```
Program
   x, y:  integer    // declarations of x and y
   begin
      Procedure B    // declaration of B
         y, z:  real    // declaration of y and z
         begin
            ...
            y = x + z    // occurrences of y, x, and z
            if (...)  call B         // occurrence of B
         end
      Procedure C    // declaration of C
         x:  real       // declaration of x
         begin
            ...
            call B    // occurrence of B
         end
      ...
      call C    // occurrence of C
      call B    // occurrence of B
   end
```

Lexical Scoping Example

Calling chain: MAIN $\Rightarrow$ C $\Rightarrow$ B $\Rightarrow$ B



Scoping and the Run-time Stack

Access links and **control links** may be used to look for non-local variable references.

Static Scope:

Access link points to stack frame of the most recently activated lexically enclosing procedure

⇒ Non-local name binding is determined at *compile time*, and implemented at *run-time*

Dynamic Scope:

Control link points to stack frame of caller

⇒ Non-local name binding is determined and implemented at *run-time*

Lexical scoping (de Bruijn notation)

Symbol table matches declarations and occurrences.

⇒ Each $\boxed{\text{variable name}}$ can be represented as a pair

(nesting_level, local_index).

```
Program
   (1,1), (1,2):  integer    // declarations of x and y
   begin
      Procedure B         // declaration of B
         (2,1), (2,2):  real   // declaration of y and z
         begin
            ...    // occurrences of y, x, and z
(*)         (2,1) = (1,1) + (2,2)
            if (...)  call B      // occurrence of B
         end
      Procedure C    // declaration of C
         (2,1):  real      // declaration of x
         begin
            ...
            call B    // occurrence of B
         end
      ...
      call C    // occurrence of C
      call B    // occurrence of B
   end
```

Access to non-local data

How does the code find non-local data at *run-time*?

Real globals

- visible *everywhere*
- translated into an address at compile time

Lexical scoping

- view variables as (*level,offset*) pairs (**compile-time symbol table**)
- look-up of (*level,offset*) pair uses chains of access links (**at run-time**)
- optimization to reduce access cost: display

Dynamic scoping

- variable names must be preserved
- look-up of variable name uses chains of control links (**at run-time**)
- optimization to reduce access cost: reference table

Access to non-local data (lexical scoping)

What code (ILOC) do we need to generate for statement ($*$)?

```
(2,1) = (1,1) + (2,2)
```

What do we know?

1. The nesting level of the statement is **level 2**.
2. Register r_0 contains the current FP (frame pointer).
3. **(2,1) and (2,2) are local variables**, so they are allocated in the activation record that current FP points to; **(1,1) is a non-local variable**.

Compiler-generated code for the statements in a procedure must work for all possible, valid runtime stacks/environments

```
(1,1) | loadAI r0, -4 => r1 // get access link; r1 now
      |                     // contains ‘‘one-level-up’’ FP
      |                     // in frame (bytes)
      | loadAI r1,  4 => r2 // get content of first local variable
      |                     // in ‘‘one-level-up’’ frame (bytes)
      |
(2,2) | loadAI r0, 8 => r3  // content of second local variable
      |                     // current frame (bytes)
      |
  +   | Add r2, r3 => r4    // (1,1) + (2,2)
      |
(2,1) | storeAI r4 => r0, 4 // store value into first local variable
  =   |                     // in current frame (bytes)
```

Access to non-local data (lexical scoping)

Two important problems arise

1. *How do we map a name into a* (level,offset) *pair?*

   We use a block structured symbol table
   (**compile-time**)

   - when we look up a name, we want to get the most recent declaration for the name
   - the declaration may be found in the current procedure or in any nested procedure

2. *Given a* (level,offset) *pair, what's the address?*

   Two classic approaches
   (**run-time**)

   ⇒ access links (*static links*)

   ⇒ displays

Managing non-local data (lexical scoping)

To find the value specified by (l, o)

- need current procedure level, k
- if $k = l$, is a local value
- if $k > l$, must find l's activation record
  $\Rightarrow$ follow $k - l$ access links
- $k < l$ cannot occur

Maintaining access links:

If procedure p is nested immediately within procedure q, the access link for p points to the activation record of the most recent activation of q.

- calling level $k + 1$ procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level $l \leq k$
  1. find my link to level $l - 1$ and pass it
  2. its access link will work for lower levels

The display

To improve run-time access costs, use a *display*.

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame

Access with the display

assume a value described by (l, o)

- find slot as DP[l] in display pointer array
- add offset to pointer from slot

"setting up the activation frame" now includes display manipulation.

Display management

Single global display: *simple method*

on entry to a procedure at level l

save the level l display value

push FP into level l display slot

on return

restore the level l display value

Procedures

- Modularize program structure
  - **Argument**: information passed from caller to callee (actual parameter)
  - **Parameter**: local variable whose value (usually) is received from caller (formal parameter)
- Procedure declaration
  - procedure name, formal parameters, procedure body with local declarations and statement lists, optional result type
    example: `void translate(point *p, int dx)`

Scott: Chapter 8.3

Parameter Association

- **Positional association**: Arguments associated with formals one-by-one; example: C, Pascal, Scheme, Java.
- **Keyword association**: formal/actual pairs; mix of positional and keyword possible; example: Ada

  procedure plot(x, y: in real; penup: in boolean)

  . . . plot (0.0, 0.0, penup ⇒ true)

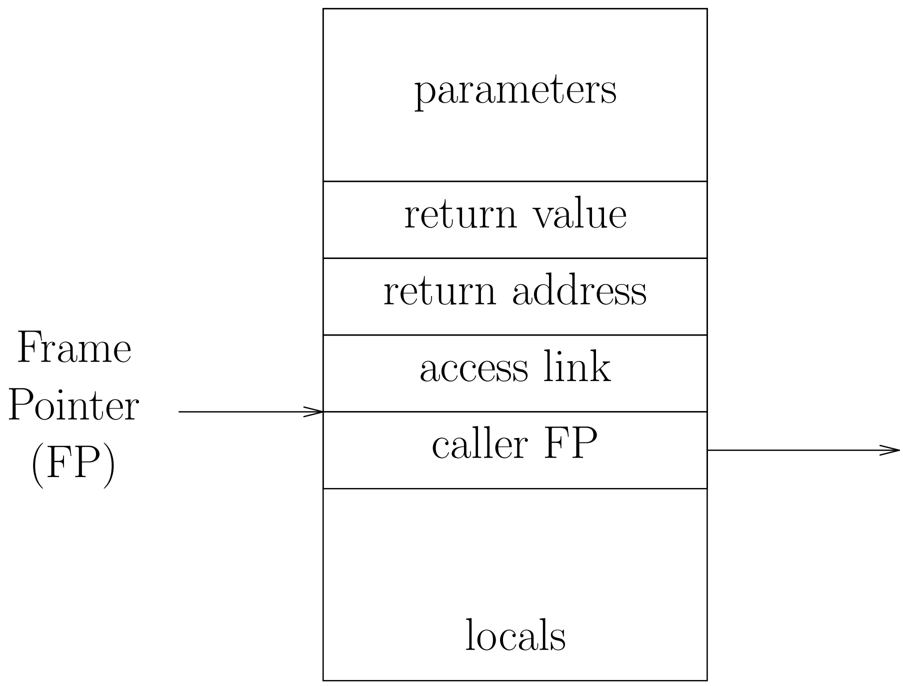
  . . . plot (penup ⇒ true, x ⇒ 0.0, y ⇒ 0.0)

Parameter Passing Modes

- **pass-by-value**: C, Pascal, Ada (`in` parameter), Scheme, Algol 68
- **pass-by-result**: Ada (`out` parameter)
- **pass-by-value-result**: Ada (`in out` parameter)
- **pass-by-reference**: Fortran, Pascal (`var` parameter)
- **pass-by-name** (not really used any more): Algol60

Review: Stack Frames

- Run-time stack contains frames for main program and each active procedure.
- Each stack frame includes:
  1. Pointer to stack frame of caller (control link)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (access link)
  4. Storage for parameters
  5. Storage for local variables
  6. Storage for final values



Pass-by-?????

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output?:

Pass-by-value

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

5 3

Advantage: Argument protected from changes in callee

Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.:arrays, structs).

Pass-by-reference

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

6 5

Advantage: more efficient than copying

Disadvantage: leads to **aliasing**: there are two or more names for the same storage location; hard to track side effects

Pass-by-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;          ==> ERROR: CANNOT USE PARAMETERS
    j := j+2;              WHICH ARE UNINITIALIZED
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output: **program doesn't compile or has runtime error**

Pass-by-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1;          ==> HERE IS ANOTHER PROGRAM
    j := 2;              THAT WORKS
  end r;
...
  m := 5;
  n := 3
  r(m,m);            ==> NOTE: CHANGED THE CALL
  write m,n;
end
```

Output: 1 or 2?

Problem: order of copy-back makes a difference; implementation dependent.

Pass-by-value-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

6 5

Problem: order of copy-back can make a difference; implementation dependent.

Pass-by-value-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  /* set c[m] = m */
  m := 2;
  r(m,c[m]); ==> WHAT ELEMENT OF ``c'' IS ASSIGNED TO?
  write c[1], c[2], ... c[10];
end
```

Output:

1 4 3 4 5 ... 10 on entry

1 2 4 4 5 ... 10 on exit

Problem: When is the address computed for the copy-back operation? At procedure call (procedure entry), just before procedure exit, somewhere inbetween? (Example: ADA on entry)

Next Lecture

HAVE A GREAT SPRING BREAK

Functional programming

Please see our website for an online Scheme textbook