Class Information

• Midterm exam forum open in Sakai.
• HW4 and HW5 solutions posted in Sakai.
• This week’s recitation is a review for midterm. Attendance is encouraged!
• Reminder: midterm exam 3/26, in class, closed book, closed notes.
Review: Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

Program

```plaintext
x, y: integer   // declarations of x and y
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
   begin
      ...
      call B // occurrence of B
   end
begin
   ...
   call C  // occurrence of C
   call B // occurrence of B
end
```

Program

```plaintext
(1,1), (1,2): integer   // declarations of x and y
Procedure (1,3)   // declaration of B
   (2,1), (2,2): real  // declaration of y and z
   begin
      ...
      (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
      if (...) call (1,3) // occurrence of B
   end
Procedure (1,4)   // declaration of C
   (2,1): real
   begin
      ...
      call (1,3) // occurrence of B
   end
begin
   ...
   call (1,4)  // occurrence of C
   call (1,3) // occurrence of B
end
```
What code do we need to generate for statement (*)?

\[(2,1) = (1,1) + (2,2)\]

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

\[(1,1) \quad \text{LOADI} \ r1, \ #4 \quad \text{// offset of local variable (1,1) in frame} \]
\[\quad \text{LOADI} \ r2, \ #-4 \quad \text{// offset of access link in frame (bytes)} \]
\[\quad \text{ADD} \ r3 \ r0 \ r2 \quad \text{// address of access link} \]
\[\quad \text{LOAD} \ r4 \ r3 \quad \text{// get access link} \]
\[\quad \text{ADD} \ r5 \ r4 \ r1 \quad \text{// address of local variable (1,1) in frame} \]
\[\quad \text{LOAD} \ r6 \ r5 \quad \text{// get content of variable (1,1)} \]
\[\quad (2,2) \quad \text{LOADI} \ r7 \ #8 \quad \text{// offset of local variable (2,2) in frame} \]
\[\quad \text{ADD} \ r8 \ r0 \ r7 \quad \text{// address of local variable (2,2)} \]
\[\quad \text{LOAD} \ r9 \ r8 \quad \text{// get content of variable (2,2)} \]
\[\quad + \quad \text{ADD} \ r10 \ r6 \ r9 \quad \text{// (1,1) + (2,2)} \]
\[\quad (2,1) \quad \text{LOADI} \ r11 \ #4 \quad \text{// offset of local variable (2,1) in frame} \]
\[\quad \text{ADD} \ r12 \ r0 \ r11 \quad \text{// address of local variable (2,1)} \]
\[\quad = \quad \text{STORE} \ r12 \ r10 \quad \text{// (2,1) = (1,1) + (2,2)} \]

Assume “access link” field is 4 bytes. Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

(1,1)  LOADI r1, #4 \hspace{1cm} // offset of local variable (1,1) in frame
       LOADI r2, #-4 \hspace{1cm} // offset of access link in frame (bytes)
       ADD r3 r0 r2 \hspace{1cm} // address of access link
       LOAD r4 r3 \hspace{1cm} // get access link
       ADD r5 r4 r1 \hspace{1cm} // address of local variable (1,1) in frame
       LOAD r6 r5 \hspace{1cm} // get content of variable (1,1)

(2,2)  LOADI r7 #8 \hspace{1cm} // offset of local variable (2,2) in frame
       ADD r8 r0 r7 \hspace{1cm} // address of local variable (2,2)
       LOAD r9 r8 \hspace{1cm} // get content of variable (2, 2)
      +  ADD r10 r6 r9 \hspace{1cm} // (1,1) + (2,2)

(2,1)  LOADI r11 #4 \hspace{1cm} // offset of local variable (2,1) in frame
       ADD r12 r0 r11 \hspace{1cm} // address of local variable (2,1)

=  STORE r12 r10 \hspace{1cm} // (2,1) = (1,1) + (2,2)

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

\[(1,1)\]
\[
\text{LOADI r1, #4} \quad \text{// offset of local variable (1,1) in frame}
\]
\[
\text{LOADI r2, #-4} \quad \text{// offset of access link in frame (bytes)}
\]
\[
\text{ADD r3 r0 r2} \quad \text{// address of access link}
\]
\[
\text{LOAD r4 r3} \quad \text{// get access link}
\]
\[
\text{ADD r5 r4 r1} \quad \text{// address of local variable (1,1) in frame}
\]
\[
\text{LOAD r6 r5} \quad \text{// get content of variable (1,1)}
\]

\[(2,2)\]
\[
\text{LOADI r7 #8} \quad \text{// offset of local variable (2,2) in frame}
\]
\[
\text{ADD r8 r0 r7} \quad \text{// address of local variable (2,2)}
\]
\[
\text{LOAD r9 r8} \quad \text{// get content of variable (2,2)}
\]
\[
+ \quad \text{ADD r10 r6 r9} \quad \text{// (1,1) + (2,2)}
\]

\[(2,1)\]
\[
\text{LOADI r11 #4} \quad \text{// offset of local variable (2,1) in frame}
\]
\[
\text{ADD r12 r0 r11} \quad \text{// address of local variable (2,1)}
\]
\[
= \quad \text{STORE r12 r10} \quad \text{// (2,1) = (1,1) + (2,2)}
\]

Assume “access link” field is 4 bytes. Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

\[(1,1) \quad \text{LOADI } r1, \#4 \quad \text{// offset of local variable (1,1) in frame} \]
\[\text{LOADI } r2, \#-4 \quad \text{// offset of access link in frame (bytes)} \]
\[\text{ADD } r3 r0 r2 \quad \text{// address of access link} \]
\[\text{LOAD } r4 \quad r3 \quad \text{// get access link} \]
\[\text{ADD } r5 r4 r1 \quad \text{// address of local variable (1,1) in frame} \]
\[\text{LOAD } r6 r5 \quad \text{// get content of variable (1,1)} \]

\[(2,2) \quad \text{LOADI } r7 \#8 \quad \text{// offset of local variable (2,2) in frame} \]
\[\text{ADD } r8 r0 r7 \quad \text{// address of local variable (2,2)} \]
\[\text{LOAD } r9 r8 \quad \text{// get content of variable (2,2)} \]
\[\text{ADD } r10 r6 r9 \quad \text{// (1,1) + (2,2)} \]

\[(2,1) \quad \text{LOADI } r11 \#4 \quad \text{// offset of local variable (2,1) in frame} \]
\[\text{ADD } r12 r0 r11 \quad \text{// address of local variable (2,1)} \]
\[= \quad \text{STORE } r12 r10 \quad \text{// (2,1) = (1,1) + (2,2)} \]

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

What code do we need to generate for statement (\(^\ast\))?

\[(2,1) = (1,1) + (2,2)\]

\[(1,1)\]
\[
\text{LOADI r1, } #4 \quad /\text{ offset of local variable (1,1) in frame}\n\]
\[
\text{LOADI r2, } #-4 \quad /\text{ offset of access link in frame (bytes)}\n\]
\[
\text{ADD r3 } r0 \ r2 \quad /\text{ address of access link}\n\]
\[
\text{LOAD } r4 \ r3 \quad /\text{ get access link, fp for frame at level 1}\n\]
\[
\text{ADD r5 } r4 \ r1 \quad /\text{ address of local variable (1,1) in frame}\n\]
\[
\text{LOAD } r6 \ r5 \quad /\text{ get content of variable (1,1)}\n\]

\[(2,2)\]
\[
\text{LOADI r7, } #8 \quad /\text{ offset of local variable (2,2) in frame}\n\]
\[
\text{ADD r8 } r0 \ r7 \quad /\text{ address of local variable (2,2)}\n\]
\[
\text{LOAD } r9 \ r8 \quad /\text{ get content of variable (2,2)}\n\]
\[
+ \quad \text{ADD r10 } r6 \ r9 \quad / (1,1) + (2,2)\n\]

\[(2,1)\]
\[
\text{LOADI r11, } #4 \quad /\text{ offset of local variable (2,1) in frame}\n\]
\[
\text{ADD r12 } r0 \ r11 \quad /\text{ address of local variable (2,1)}\n\]
\[
= \quad \text{STORE } r12 \ r10 \quad / (2,1) = (1,1) + (2,2)\n\]

Assume “access link” field is 4 bytes. Each unit offset corresponds to 4 bytes.
What code do we need to generate for statement (*)?

\[(2,1) = (1,1) + (2,2)\]

\[(1,1)\]
- \(\text{LOADI } r1, \#4\)  // offset of local variable (1,1) in frame
- \(\text{LOADI } r2, \#-4\)  // offset of access link in frame (bytes)
- \(\text{ADD } r3 \ r0 \ r2\)  // address of access link
- \(\text{LOAD } r4 \ r3\)  // get access link, fp for frame at level 1
- \(\text{ADD } r5 \ r4 \ r1\)  // address of local variable (1,1) in frame
- \(\text{LOAD } r6 \ r5\)  // get content of variable (1,1)

\[(2,2)\]
- \(\text{LOADI } r7 \ #8\)  // offset of local variable (2,2) in frame
- \(\text{ADD } r8 \ r0 \ r7\)  // address of local variable (2,2)
- \(\text{LOAD } r9 \ r8\)  // get content of variable (2,2)
- \(\text{ADD } r10 \ r6 \ r9\)  // \((1,1) + (2,2)\)

\[(2,1)\]
- \(\text{LOADI } r11 \ #4\)  // offset of local variable (2,1) in frame
- \(\text{ADD } r12 \ r0 \ r11\)  // address of local variable (2,1)
- \(\text{STORE } r12 \ r10\)  // \((2,1) = (1,1) + (2,2)\)

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

\[(1,1)\]
- LOADI r1, #4  \quad // offset of local variable (1,1) in frame
- LOADI r2, #-4  \quad // offset of access link in frame (bytes)
- ADD r3 r0 r2  \quad // address of access link
- LOAD r4 r3  \quad // get access link, fp for frame at level 1
- ADD r5 r4 r1  \quad // address of local variable (1,1) in frame
- LOAD r6 r5  \quad // get content of variable (1,1)

\[(2,2)\]
- LOADI r7 #8  \quad // offset of local variable (2,2) in frame
- ADD r8 r0 r7  \quad // address of local variable (2,2)
- LOAD r9 r8  \quad // get content of variable (2,2)
+ ADD r10 r6 r9  \quad // (1,1) + (2,2)

\[(2,1)\]
- LOADI r11 #4  \quad // offset of local variable (2,1) in frame
- ADD r12 r0 r11  \quad // address of local variable (2,1)
= STORE r12 r10  \quad // (2,1) = (1,1) + (2,2)

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.
Procedures

• Modularize program structure
  - Actual parameter: information passed from caller to callee (*Argument*)
    Appears in caller code.
  - Formal parameter: *local* variable whose *value* (usually) is received from caller
    Appears in callee declaration

• Procedure declaration
  - Procedure names, formal parameters, procedure body with formal local declarations and statement lists, optional result type

Example: `void translate(point *p, int dx)`
Parameters

Parameter Association

• Positional association: Arguments associated with formals one-by-one; Example: C, Pascal, Java, Scheme
• Keyword association: formal/actual pairs; mix of positional and keyword possible; Example: Ada
  
  procedure plot(x, y: in real; z: in boolean)
  
  … plot (0.0, 0.0, z ⇒ true)
  
  … plot (z ⇒ true, x ⇒ 0.0, y ⇒ 0.0)

Parameter Passing Modes

• Pass-by-value: C/C++, Pascal, Java/C# (value types), Scheme
• Pass-by-result: Ada, Algol W
• Pass-by-value-result: Ada, Swift
• Pass-by-reference: Fortran, Pascal, C++, Ruby, ML
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: Advantage: Argument protected from changes in callee. Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.: structs, arrays)
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 storage location; hard to track side effects
Aliasing:

More than two ways to name the same object within a scope

Even without pointers, you can have aliasing through (global <-> formal) and (formal <-> formal) parameter passing.

begin
    j, k, m: integer;
    procedure r(a, b: integer)
    begin
        b := 3;
        m := m * a;
    end
    ...
    q(m, k);  → global/formal <m,a> ALIAS PAIR
    q(j, j);  → formal/formal <a,b> ALIAS PAIR
    write y;
end
Pass-by-result

begin
  c: array[1...10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1;
    j := 2;
  end r;
  ...
  m := 5;
  n := 3;
  r(m, m); → NOTE: CHANGE THE CALL
  write m, n;
end

Output: 1 or 2 for m?

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 makes a difference; implementation dependent.
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[3], ... c[10];
end

Output:
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[3], ... c[10];
end

Output:

Problem: When is the address computed for the copy-back operation? At procedure call (procedure entry), just before procedure exit, or somewhere in between? (Example: ADA on entry)
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[3], ... 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, or somewhere in between? (Example: ADA on entry)
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]);
    write c[1], c[2], c[3], ... 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, or somewhere in between? (Example: ADA on entry)
Comparison: by-value-result vs. by-reference

Actual parameters need to evaluate to L-values (addresses).

begin
  
y: integer;
  procedure p(x: integer)
  begin
    x := x + 1  \text{ ref: } x \text{ and } y \text{ are ALIASED}
    x := x + y  \text{ val-res: } x \text{ and } y \text{ are NOT ALIASED}
  end
  
  y := 2;
  p(y);
  write y;
end

Output:
  \begin{itemize}
    \item pass-by-reference: 6
    \item pass-by-value-result: 5
  \end{itemize}

Note: \textit{by-value-result}: Requires copying of parameter values (expansive for aggregate values); does not have aliasing, but copy-back order dependence.
**Imperative:**

Sequence of state-changing actions.
- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Fits the von Neumann architecture closely
- Key operations: Assignment and Control Flow
Functional:

Composition of operations on data.

- No named memory locations
- Value binding through parameter passing
- Key operations: *Function application* and *Function abstraction*
- Basis in *lambda calculus*
Pure Functional Languages

Fundamental concept: **application** of (mathematical) functions to values

1. **Referential transparency**: the value of a function application is independent of the context in which it occurs

   • value of $foo(a, b, c)$ depends only on the values of $foo$, $a$, $b$ and $c$
   • it does not depend on the global state of the computation

   $\Rightarrow$ all vars in function must be local (or parameters)

2. The concept of assignment is NOT part of function programming

   • no explicit assignment statements
   • variables bound to values only through the association of actual parameters to formal parameters in function calls
   • function calls have no side effects
   • thus no need to consider global states
3. Control flow is governed by function calls and conditional expressions
   ⇒ no iteration
   ⇒ recursion is widely used
4. All storage management is implicit
   • needs garbage collection
5. Functions are *First Class Values*
   • can be returned as the value of an expression
   • can be passed as an argument
   • can be put in a data structure as a value
   • (unnamed) function exists as a value
A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in scheme,

> (define length
   (lambda (x)
     (if (null? x)
       0
       (+ 1 (length (rest x))))))

> (length '(A LIST OF 5 THINGS))
5
LISP

• Functional language developed by John McCarthy in the mid 50’s
• Semantics based on *Lambda Calculus*
• All functions operate on lists or symbols called: “S-expression”
• Only five basic functions: list functions *con, car, cdr, equal, atom*, and one conditional construct: *cond*
• Useful for list-processing applications
• Program and data have the same syntactic form “S-expression”
• Originally used in Artificial Intelligence
SCHEME

- Developed in 1975 by G. Sussman and G. Steele
- A dialect of LISP
- Simple syntax, small language
- Closer to initial semantics of LISP as compared to COMMON LISP
- Provide basic list processing tools
- Allows functions to be first class objects
Things to do: