Principles of Programming Languages

Topic: Scope and Memory

Professor Louis Steinberg

Fall 2004
Review

Functions as first-class objects

• **What can you do with an integer?**
  – Use as an argument to a function
  – Return as value of a function
  – Write a constant

• **What can you do with a function?**
  – Use as an argument to a function
  – Return as value of a function
  – Write a constant

higher order functions
map

• Higher order function used to apply another function to every element of a list
• Takes 2 arguments a function \( f \) and a list \( lst \) and builds a new list by applying the function to every element of the (argument) list

\[
(\text{map } \text{abs} \ (-1 \ 2 \ -3 \ -4)) \text{ returns } (1 \ 2 \ 3 \ 4)
\]

\[
(\text{map } (\lambda (x) (+ 1 x)) \ '(1 \ 2 \ 3)) \text{ returns } (2 \ 3 \ 4)
\]
Using map

Define \texttt{atomcnt3} which uses map to calculate the number of atoms in a list. \texttt{atomcnt3} creates a list of the count of atoms in every sublist and apply of + calculates the sublist sum.

\begin{verbatim}
(define (atomcnt3 s) (cond ((atom? s) 1)
                           (else (apply + (map atomcnt3 s))))
\end{verbatim}

(\texttt{atomcnt3 \texttt{`(1 2 3)}}) returns 3
(\texttt{atomcnt3 \texttt{`((a b) d)}}) returns 3
(\texttt{atomcnt3 \texttt{`(1 ((2) 3) (((3) (2) 1))}}) returns 6

How does this function work?
apply

apply is a built-in function whose first argument f is a function and whose second argument ll is a list of arguments for that function.

Evaluation of apply applies f to ll.

(apply + '(1 2 3)) returns 6
(apply zero? 2) returns #f
(apply (lambda (n) (+ 1 n)) '(3)) returns 4

The power of apply is that it lets your program build an S-expression to evaluate during execution, and then lets it be evaluated.
A Common Recursive Pattern

(define (sum lst)
  (if (null? lst) 0
      (+ (car lst)
          (sum (cdr lst)))))

(define (prod lst)
  (if (null? lst) 1
      (* (car lst)
          (sum (cdr lst)))))

A Common Recursive Pattern
foldr

- Higher order function that takes a binary, associative operation and uses it to “roll-up” a list

```scheme
(define (foldr op lst id)
  (if (null? lst) id
     (op (car lst) (foldr op (cdr lst) id))))

(foldr + '(10 20 30) 0) yields
(+!10 (foldr + (20 30) 0) )
(+ 10 (+ 20 (foldr + (30) 0) ))
(+ 10 (+ 20 (+ 30 (foldr + () 0))))
(+ 10 (+ 20 (+ 30 0))) yields 60
```
Reverse

(define (reverse lst)
  (if (null? lst) ()
    (append
     (reverse (cdr lst))
     (list (car lst))))
)

(define (reverse lst)
  (foldr ()
    lst
    (lambda (e l)
      (append l (list e)))))
The Power of Higher Order

- Can compose higher order functions to form compact powerful functions

```
(define (sumfn f lst) (foldr + (map f lst) 0))
```

- `sumfn` takes a function `f` and a list `lst`
- `sum` applies `f` to each element of the list and then sums the results

```
(sumfn (lambda (x) (* 2 x)) '(1 2 3)) yields 12  
(sumfn square '(2 3)) yields 13
```
Scope

• The scope of a variable binding is the region of the program in which you can refer to that binding
Scope in Scheme

(let* ((x 2) (y 3) (x 4)) (+ x y))

(let ((x 2))
  (let ((y 3))
    (let ((x 4))
      (+ x y))))

What result?
→ 5 ?
→ 7 ?
Scope in Scheme

(((lambda (x)
  (((lambda (y)
    (((lambda (x)
      (+ x y))
    4))
  3))
  2)

What result?

→ 5 ?
→ 7 ?
Scope in Scheme

(let ((x 2))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 4))
      (+ x (f 3))))))

What result?

→ 9 ?
→ 11 ?
→ error message ?
Consider:

\[(\lambda y \ (\ + \ x \ y))\]

Where does \(x\) get its value?

- From the outermost \texttt{let}, where \(x = 2\)?
- From the innermost \texttt{let}, where \(x = 4\)?
Dynamic Scope

A variable has “dynamic scope” if:
- it can be referred to anywhere in the program, and
- at the time of reference, it gets whatever binding is currently in effect.

(let ((x 2))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 4))
      (+ x (f 3)))))

Under dynamic scope rules  \rightarrow  11
Lexical Scope

A variable has “lexical scope” if:

– it can only be referred to within the textual construct in which it was created, and
– it gets its binding from that context.

(let ((x 2))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 4))
      (+ x (f 3)))))

Under lexical scope rules → 9
Closures

Variables in Scheme are lexically scoped!
In addition, their bindings persist for as long as the possibility of reference remains.

> (define (adder x) (lambda (y) (+ x y)))

> ((adder 3) 5)
8
> (map (adder 5) '(1 3 5 7 9))
(6 8 10 12 14)
Closures

A closure is a function plus a binding environment in which it is to be evaluated.

- \((\text{adder } 3)\) returns a closure in which \(x \rightarrow 3\)
- \((\text{adder } 5)\) returns a closure in which \(x \rightarrow 5\)
- These are, in effect, different functions!

When a closure is applied to its argument(s), the binding environment provides values for variables in the function definition.

- Thus \(((\text{adder } 3) \ 5)\) is, in effect, equivalent to:

\[
\begin{align*}
\text{(let ((x 3))} \\
\text{ ((lambda (y) (+ x y))} \\
\text{ 5))}
\end{align*}
\]
program \textit{L};
    \begin{verbatim}
var \textit{n} : \texttt{char}; \{ \textit{n} declared in \textit{L} \}
procedure \textit{W};
    \begin{verbatim}
begin
    writeln(\textit{n}) \{ occurrence of \textit{n} in \textit{W} \}
end;
procedure \textit{D};
    \begin{verbatim}
var \textit{n} : \texttt{char}; \{ \textit{n} redeclared in \textit{D} \}
begin
    \textit{n} := 'D';
    \textit{W} \{ \textit{W} called within \textit{D} \}
end;
begin \{ \textit{L} \}
    \textit{n} := 'L';
    \textit{W}; \{ \textit{W} called from the main program \textit{L} \}
end
\end{verbatim}
\end{verbatim}
end.
\end{verbatim}

\textbf{Figure 5.6} The output of this program is different under lexical and dynamic scope.
Scope in C (Sethi)

int main(···)
{
    int i;
    for(···)
    {
        int c;
        if(···)
        {
            int i;
            ...
        }
        ...
    }
    while(···)
    {
        int i;
        ...
    }
    ...
}

(b) Scope of $i_1$.

(a) Nested compound statements.

Figure 5.8
Block-Structured Languages

Examples:
Algol, Pascal, Ada, ...

C has nested blocks, but not nested procedure declarations
Visibility of Identifiers

program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c, d: integer;
            procedure R(x: integer) {
                ...; @5; ...}
                ...; @4; ...}
                ...; @2; ...}
            procedure R(x: integer) {
                a: integer;
                procedure Q {...; @6; ...}
                ...; @3; ...}
                ...; @1; ...}
program Main {
    a,b,c: integer;
    procedure P {
        c: integer;
        procedure S {
            c,d: integer;
            procedure R(x: integer) {
                ...; @5; ...}
            ...; @4; ...}
            ...; @2; ...}
        ...; @2; ...}
    procedure R(x: integer) {
        a: integer;
        procedure Q {...; @6; ...}
        ...; @3; ...}
    ...; @1; ...}
Visibility of Procedures

program Main {
  a,b,c: integer;
  procedure P {
    c: integer;
    procedure S {
      c,d: integer;
      procedure R(x: integer) {
        ...; @5; ...}
        ...; @4; ...
        ...; @2; ...
    }procedure R(x: integer) {
      a: integer;
      procedure Q {...; @6; ...}
      ...; @3; ...
      ...; @1; ...
    }procedure R(x: integer) {
      a: integer;
      procedure Q {...; @6; ...}
      ...; @3; ...
      ...; @1; ...
    }
  }
}

Which procedures are visible/callable at location
@1?
@2?
@3?
@4?
@5?
@6?

What is the Rule?
Binding Times

• “Static” Bindings:
  – Compile Time
  – Link Time
  – Load Time

• “Dynamic” Bindings:
  – Run Time

• “Lifetime” or “Extent” of a Binding:
  – When is a binding initiated?
  – When is a binding terminated?
Memory Allocation

Classical Layout:

<table>
<thead>
<tr>
<th>CODE</th>
<th>STATIC</th>
<th>STACK</th>
<th>free memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>Address Space</td>
<td>low</td>
<td>H E A P</td>
</tr>
</tbody>
</table>

free memory
Runtime Stack

• Used to manage block-structured storage, as well as procedure calls and returns.
• Stack contains *frames* of all blocks (or procedures) which have been entered but not yet exited.
• Frame contains:
  – All information necessary to update stack and resume execution when a block (or procedure) is exited.
  – Addresses of all local variables (and all parameters), encoded as an offset from the base address of the frame.
Heap

• Used for dynamically created objects whose lifetime may not coincide with procedure calls and returns.
• Allows allocation and deallocation of indeterminate sized blocks of memory during execution.
• Allocation and deallocation can be:
  – Explicit (e.g., in Pascal, C, C++)
    Problems: dangling references, memory leaks, ...
  – Implicit (e.g., in Java, Scheme, Prolog)
    Requires: automatic garbage collection
Implementing Lexical Scope

• Use stack to manage nesting

• Stack frame includes:
  – space for parameters, local variables, return value
  – saved state of caller (registers, return address, etc.)
  – control link (or dynamic link): return pointer to stack frame of caller
  – access link (or static link): to access nonlocal variables
Implementing Lexical Scope

• Prologue
  – push new stack frame, reset CurrentFramePointer
  – initialize fixed length fields
  – evaluate argument expressions and place in parameters
    (if language uses call-by-value, e.g., Scheme, C, Java)

• Execute code in the called procedure or function

• Epilogue
  – restore caller’s registers
  – pop stack, and reset CurrentFramePointer
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...; R(c); ...
            ...
        }
        R(2);
        S; ...
    }
    procedure R(x: integer) {
        a: integer;
        ...
        ...; print(a, b, c, x); ...
        ...
    }
    ...
    P; ...
}
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...
            R(2);
            S; ...
        }
        ...
        procedure R(x: integer) {
            a: integer;
            ...
            print(a, b, c, x); ...
        }
        ...
    }
}
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...}
                ...; R(c); ...}
        ...; R(2);...
        S; ...}
    procedure R(x: integer) {
        a: integer;
        ...; print(a, b, c, x); ...}
    ...; P; ...
}
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...
        }
        ...
    }
    P;
}

procedure R(x: integer) {
    a: integer;
    ...
    print(a, b, c, x); ...
}

R(2);
P;
S; ...

procedure R(x: integer) {
    a: integer;
    ...
    print(a, b, c, x); ...
}

...; P; ...

CS 314, LS.BR,LTM: Scope and Memory 34
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...
        }
    }
    ...
    R(2);
    S; ...
    procedure R(x: integer) {
        a: integer;
        ...
        print(a, b, c, x); ...
    }
    ...
}
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...
        }
        ...
    }
    ...
    R(2);
    S; ...
}
procedure R(x: integer) {
    a: integer;
    ...
    print(a, b, c, x); ...
}
...
...
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
        ...
        ...
        R(c);
    }
    ...
    R(2);
    S; ...
    procedure R(x: integer) {
        a: integer;
        ...
        print(a, b, c, x); ...
    }
    ...
    P; ...
}
program Main {
    a, b, c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a, b, c, x); ...
            }
            ...
            R(c); ...
            ...
            R(2);
        S; ...
            procedure R(x: integer) {
                a: integer;
                ...
                print(a, b, c, x); ...
            }
        }
        ...
        P; ...
    }
}
program Main {
    a,b,c: integer;
    procedure P {
        c: integer;
        procedure S {
            c: integer;
            procedure R(x: integer) {
                print(a,b,c,x); ...}
                ...; R(c); ...}
                ...;
                R(2);
                S; ...}
            procedure R(x: integer) {
                a: integer;
                ...; print(a,b,c,x); ...}
                ...; P; ...}
Implementing Lexical Scope

• *Dynamic chain*: sequence of control links
  – Where a procedure appears on the dynamic chain depends on the calling context.

• *Static chain*: sequence of access links
  – Where a procedure appears on the static chain depends only on the lexical nesting of declarations, independent of the calling context.
  – We can determine at compile time from the program text the *nesting depth* $n$ of every procedure.
  – Use this to implement “displays” [Sethi, pp. 196-198].
Displays

• Display:
  – An array of pointers to the currently active frames on the stack that correspond to lexically enclosing blocks of the currently executing procedure
  – $\text{display}[j]$ = pointer to some frame of nesting depth $j$

• To encode the address of a variable:
  – Determine statically (i.e., at compile time) the block whose binding is in effect, and its nesting depth $j$
  – Variable is stored at address $\text{display}[j] + \text{offset}$

• Manage display in procedure prologue and epilogue
program Main {
  a₁,b₁,c₁: integer;
  procedure P₁ {
    c²: integer;
    procedure S² {
      d³: integer;
      procedure R³(x⁴: integer) {
        c²=a₁+c²+d³+x⁴; ...}
      ...; R³(b₁); ...}
    ...;
    R¹(2);
    S²; ...}
  procedure R¹(x²: integer) {
    a²,c²: integer;
    ...; print(a²,b₁,c²,x²); ...}
  ...; P¹; ...}
Display

<table>
<thead>
<tr>
<th>1</th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>proc P</td>
</tr>
<tr>
<td>3</td>
<td>proc S</td>
</tr>
</tbody>
</table>

dtop = 3

\begin{align*}
\mathbf{b}^1 & \text{ is at display}[1] + 2 \\
\mathbf{c}^2 & \text{ is at display}[2] + 1 \\
\mathbf{d}^3 & \text{ is at display}[3] + 1 \\
\mathbf{R}^1 & \text{ is at display}[1] + 5 \\
\mathbf{R}^3 & \text{ is at display}[3] + 2
\end{align*}
Implementing Displays

• Display is managed as a stack itself.

• Prologue:
  – Push frame onto stack.
  – Save old value of display[].
  – Update display[] and adjust dtop, if needed.

• Epilogue:
  – Pop frame from stack.
  – Restore display[] and adjust dtop, if needed.

• Two versions:
  – How much information is saved in each frame?
NonRecursive Example

```
program Main {
    a: integer;
    procedure P {
        b: integer;
        b=a;
        Q;
        ...
    }
    procedure Q {
        b: integer;
        b=a;
        ...
    }
    a=10;
    P;}
```

```
1 Main
dtop = 1
```

```
1 Main
dtop = 2
```

```
1 Main
dtop = 2
```

```
1 Main
dtop = 1
```

```
1 Main
dtop = 2
```

```
1 Main
dtop = 2
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```

```
<Main>
 a == 10
```
program Main {
    a: integer;
    procedure P {
        c: integer;
        procedure S {
            b: integer;
            P;
            ...
        }
        S;
        ...
    }
    P;
}

Recursive Example
Recursive Example
Garbage Collection

• Automatic heap memory management (e.g., in Java, Scheme, Prolog, ...)

• Goal: Reclaim memory that is no longer accessible.

• Methods:
  – Reference Counting
  – Mark and Sweep
  – Stop and Copy
  – Generational