CS:314 Fall 2024

Section **04**

Recitation **7 + 8**

chris.tu@rutgers.edu

Office hours: 2-3pm Thursday CoRE 335

Topics Covered

- Scoping (could be on Final!)
  - Lexical vs. Dynamic
  - Global Scope Stack and Access vs. Control
  - Level Offset Pairs
- Parameter Passing
- Functional Programming Introduction
- Bonus: Project 1 Help

Example 1: Scoping - Lexical vs Dynamic

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

Assume **A** is dynamically scoped, and **b** is lexically scoped.

What will this program print?

Example 1: Scoping - Lexical vs Dynamic

the syntax

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

- **Visualize the stack for dynamic variables, and look at syntax for lexical variables.**
- **Follow along in a notebook.**

Runtime stack

Example 1: Scoping - Lexical vs Dynamic

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

- call main, add **main** to runtime stack
- **A and b** declared in main
- foo() and bar() are defined.
- We are still in main, and **A and B** get updated to 10 and 10.

main

Example 1: Scoping - Lexical vs Dynamic

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

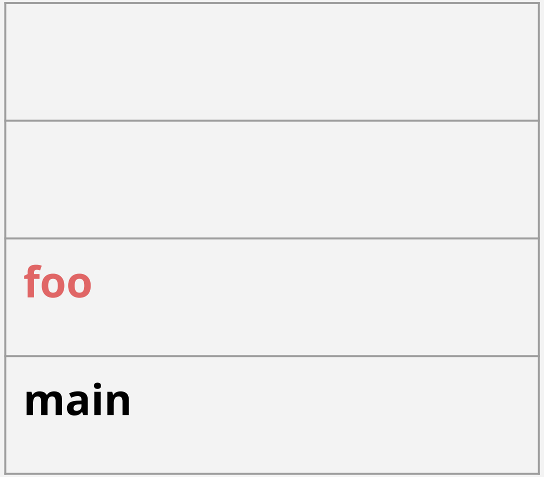
- call foo, add **foo** to stack
- inside foo, **A and b get declared as 0.**

foo
main

Example 1: Scoping - Lexical vs Dynamic

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

- For dynamic A, the most recent process is **foo** right now. And within foo, A was declared and defined to be 0. **So print 0 for A.**
- For lexical B, is B defined in foo? Yes. It's set to 0. **So print 0 for B.**



Example 1: Scoping - Lexical vs Dynamic

- Call bar, add **bar** to stack.

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

bar
foo
main

- We are in bar now.
- For dynamic A, the most recent process with a declaration for A is **foo()**. So take from foo() and A is 0. 0 + 1 = 1. **So print 1 for A**.
- For lexical b, does bar() declare b? **No**. Now look in ENCLOSING syntax, which is **main**. Does main() **declare** b? **Yes**. And then main() updates it as 10. So B is 10. 10+1 =1. **So print 11 for B.**

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
dynamic A   ------>  A = A + 1;
lexical B   ------>  b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

bar
foo
main

Example 1: Scoping - Lexical vs Dynamic

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, b = 0;
                print A, b;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```

You follow along okay?

You should see that the program outputted:

0, 0

1, 11

Try doing it from the start without following the steps in these slides :)

Example 2: **How Scope is fixed at compile vs run**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

How do you think lexical vs dynamic scoping determines the outcome of this source code here?

Example 2: **Lexical Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

compile-time fixes lexical scope

- **while we are in function b:**
  is x declared in b()? No
  Then check enclosing syntax.
  is x declared in a()? **Yes**
- So b() updates x1 to be 2.
- (even if it is called within function c, and there is a declaration of a variable x in c.)
- Therefore, the final printed result is 2.

Example 2: **Dynamic Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

run-time fixes dynamic scope. while the program runs, a global scope stack is maintained for each variable name.

When variable is accessed, take from top of stack (just like in example 1)

Example 2: Dynamic Scoping

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- When the program executes at the point where x1 is declared, the compiler notices the declaration of the variable x and since there is no global scope stack for the variable name x yet, it creates a global scope stack for x and pushes x => x1 onto it.
- At this point, the scope stack for variable x looks like this: [x => x1].

Example 2: Dynamic Scoping

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- As the program executes at the point where x2 is declared, the compiler finds another declaration of the variable x and pushes x => x2 onto the scope stack for x.
- Now the scope stack for variable x looks like this: [x => x1, x => x2].

Example 2: **Dynamic Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- Function b starts to execute and attempts to modify the variable x. The compiler reads the current scope of the variable x from the top of x's scope stack, which is x => x2.
- Thus, what really gets modified is x2, while the value of x1 remains unchanged.
- Then function b finishes executing and returns to function c.

Example 2: **Dynamic Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- Function b starts to execute and attempts to modify the variable x. The compiler reads the current scope of the variable x from the top of x's scope stack, which is x => x2.
- Thus, what really gets modified is x2, while the value of x1 remains unchanged.
- Then function b finishes executing and returns to function c.

Example 2: **Dynamic Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- Subsequently, function c also finishes executing, and a pop operation is performed on the scope stack for the variable x.
- At this time, the scope stack for x looks like this: [x => x1].

Example 2: **Dynamic Scoping**

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

- Finally, when printing the variable x, the compiler looks at the top of the scope stack for x and determines that the variable being accessed is actually x1.
- Since the value of x1 has not changed (it was x2 that was modified earlier), the result printed is 1.

Implementation of Scopes

- Procedures are executed in stack memory
  - Last in, first out
  - Begins when control enters activation (call)
  - Ends when control returns from call

Run-time stack has frames for each procedure
Each frame includes:

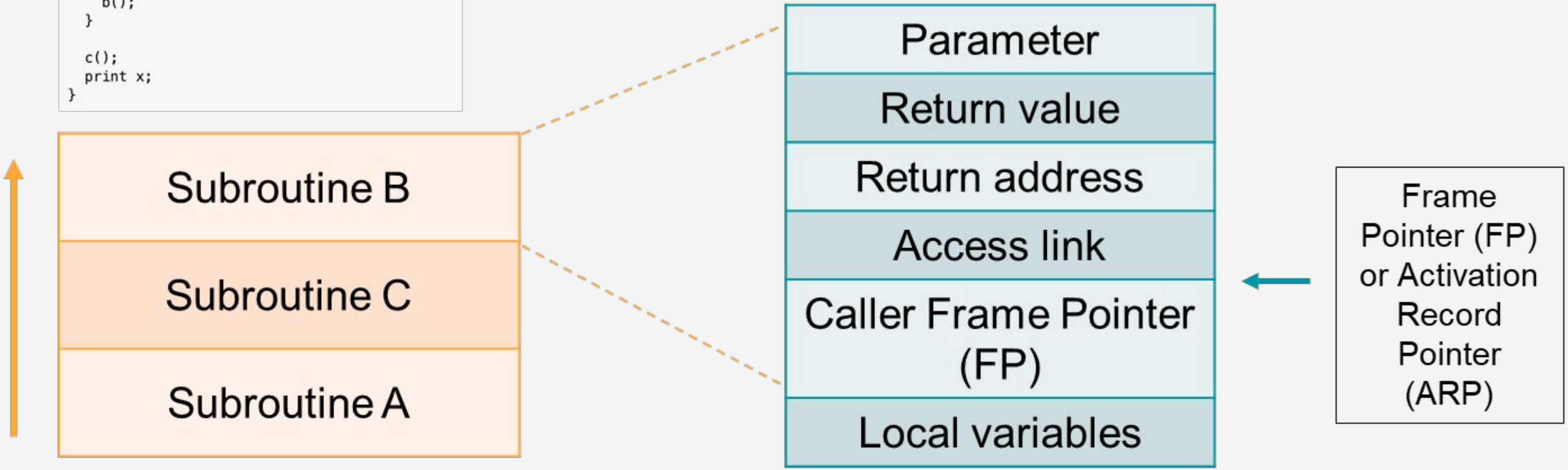
- Pointer to stack frame of caller (**control link**)
- Return address
- Mechanism to find non-local variables (**access link**)
- Storage for parameters, local variables, final values
- Intermediate values & saved register

Implementation of Scopes

```
program a() {
  x: integer; // "x1" in discussions below
  x = 1;

  procedure b() {
    x = 2; // <-- which "x" do we write to?
  }

  procedure c() {
    x: integer; // "x2" in discussions below
    b();
  }

  c();
  print x;
}
```

Calling chain:
A → C → B

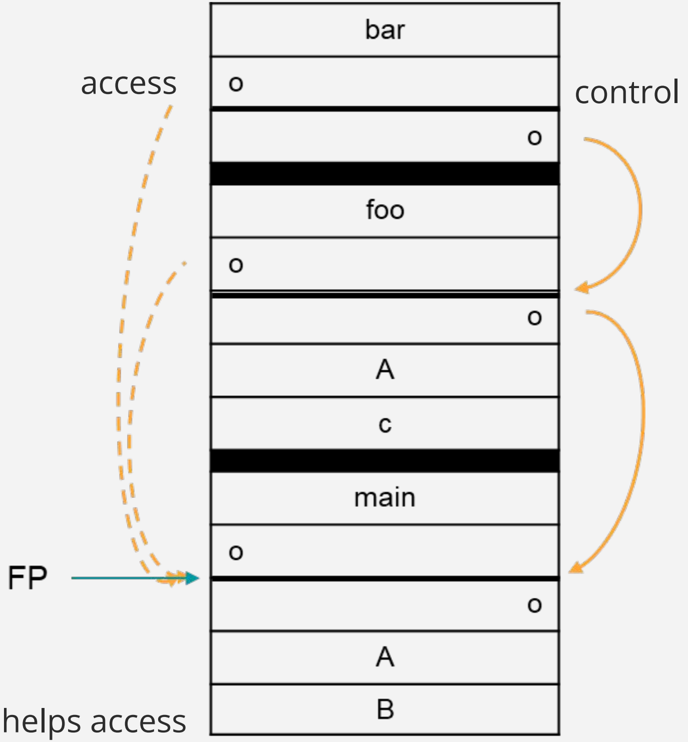
Visualization of what procedure, or subroutine frames contain



Runtime Stack with frames for each procedure.

Example of Global Scope Stack

```
program main():
        int A = 0, b = 0;
        procedure foo():
                int A = 0, c = 0;
                print A, c;
                bar();
        end foo;
        procedure bar():
                A = A + 1;
                b = b + 1;
                print A, b;
        end bar;
        A = A + 10;
        b = b + 10;
        foo();
end main;
```



the frame pointer is fixed, helps access syntax ancestor's variables for lexical scoping in a predictable way

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        (1, 1), (1, 2): integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        (1, 1), (1, 2): integer
        procedure (1, 3)
                y, z: real
        begin
                …
                call C
        end;
        procedure (1, 4)
                w: real
        begin
        end;
begin
        call B
end
```

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        (1, 1), (1, 2): integer
        procedure (1, 3)
                y, z: real
        begin
                …
                call (1, 4)
        end;
        procedure (1, 4)
                w: real
        begin
        end;
begin
        call (1, 3)
end
```

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        (1, 1), (1, 2): integer
        procedure (1, 3)
                (2, 1), (2, 2): real
        begin
                …
                call (1, 4)
        end;
        procedure (1, 4)
                w: real
        begin
        end;
begin
        call (1, 3)
end
```

Level Offset Pairs - Example

```
program main():
        x, y: integer
        procedure B
                y, z: real
        begin
                …
                call C
        end;
        procedure C
                w: real
        begin
        end;
begin
        call B
end
```

```
program main():
        (1, 1), (1, 2): integer
        procedure (1, 3)
                (2, 1), (2, 2): real
        begin
                …
                call (1, 4)
        end;
        procedure (1, 4)
                (2, 1): real
        begin
        end;
begin
        call (1, 3)
end
```

Parameter Passing

Parameter Passing Modes:

- **Pass-by-Value**: Copies the value of the argument into the function. Changes made in the function do not affect the original variable.
- **Pass-by-Reference**: Passes the address of the argument, so changes affect the original variable. This can lead to aliasing, where multiple names refer to the same data.
- **Pass-by-Result**: The function doesn't receive the original variable initially, but any changes made are written back to the original variable after the function finishes.
- **Pass-by-Value-Result**: Combines pass-by-value and pass-by-result. A copy of the argument is passed in, changes are made to this copy, and it is then written back to the original variable. Avoids aliasing.

Parameter Passing Example

```
procedure main
    x = 5; y = 3;
    procedure modify(a, b)
        a = a + 1; // modifies a
        b = b + 2; // modifies b
    end modify
    modify(x, y); // pass parameters
    print(x, y);
end main
```

Pass by Value:
Prints 5 and 3.

Pass by Reference:
x and y become 6 and 5.
Prints 6 and 5.

Parameter Passing Example

```
procedure main
    x = 5; y = 3;
    procedure modify(a, b)
        a = 1; // modifies a
        b = 2; // modifies b
    end modify
    modify(x, y); // pass parameters
    print(x, y);
end main
```

Pass by Result:
Prints 1 and 2.

Parameter Passing Example

```
procedure main
    x = 5; y = 3;
    procedure modify(a, b)
        a = a + 1;
        b = b + 2;
    end modify
    modify(x, y); // pass parameters
    print(x, y);
end main
```

Pass by Value Result: Avoids aliasing.

Prints 6 and 5.

Parameter Passing Example

```
procedure main
    x = 5; y = 3;
    procedure modify(a, b)
        a = a + 1;
        b = b + 2;
    end modify
    modify(x, x); // pass parameters
    print(x, y);
end main
```

Pass by Value-Result:
Passing same variable?

The result is implementation-dependent, some languages might copy 6 back to x first, or they might copy 7 back first.

Therefore, x could end up as 6 or 7.

Aliasing is avoided, but copyback order is depended on by language design.

Functional Programming Overview

Functional programming is a programming paradigm where computation is treated as the evaluation of mathematical functions (hence the name).

Instead of focusing on changes in program state (like in imperative programming),
functional programming emphasizes:

- **immutability**—no changes to data after it's created
- **pure functions**—functions that always produce the same output given the same inputs, with no side effects.