EECS 390 – Lecture 6

Control Flow

1/30/24

Review: Static and Dynamic Scope

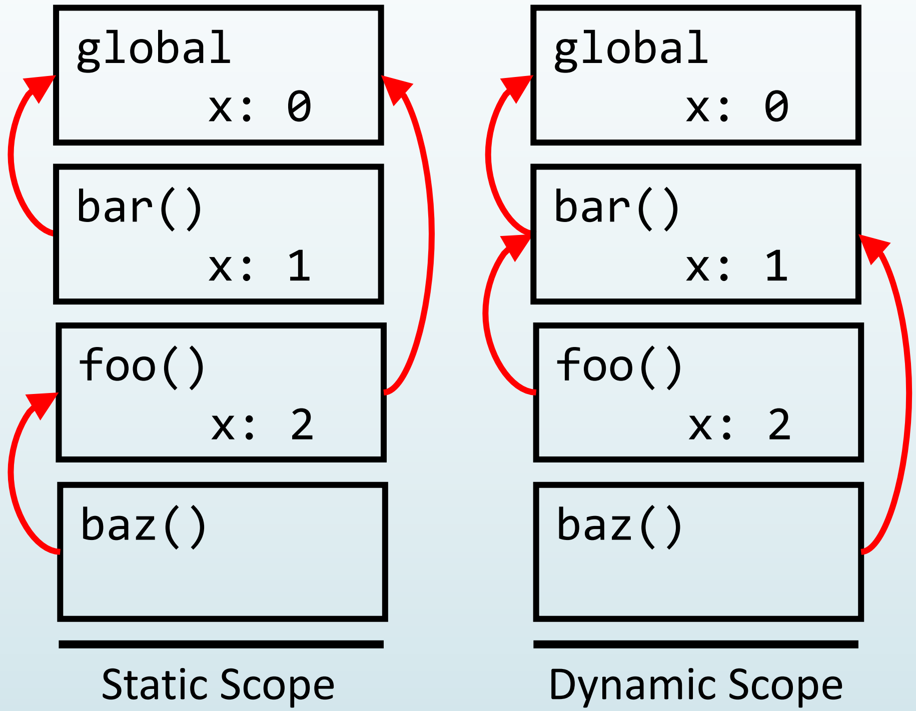
- **Static scope:** non-local environment is the *definition* environment
- **Dynamic scope:** non-local environment is the *caller's* environment

```
x = 0

def foo():
    x = 2
    def baz():
        print(x)
    return baz

def bar():
    x = 1
    foo()()

bar()
```



Static vs. Dynamic Scope

- Static scope
  - **Pro:** Compiler can resolve name lookups, so this enables offset-based implementations
  - **Con:** Cannot use stack-based memory management in the presence of nested function definitions
  - **Pro:** Static scope and nested functions enable higher-order functional patterns that rely on the non-local environment (more on this in the next unit)
  - **Pro:** Better abstraction boundaries – the author of a function need not worry about the caller's environment
- Dynamic scope
  - **Pro:** Can use stack-based memory management even with nested function definitions
  - **Pro:** Enables some patterns that are particularly important to error handling
  - **Con:** Raises more semantic issues (e.g. binding policy – more on this next week)

Review: Assignments in Python

- Python assumes that an assignment to a variable is intended to target a local variable
- Furthermore, the scope of a local variable starts at the beginning of a function
- Using a variable before it is initialized is an error

```
x = 2

def foo():
    print(x)
    x = 3
```

Scope of local x starts here

Error: use before initialization

Defines local variable x

`global` and `nonlocal` in Python

- A programmer can specify that a name is meant to refer to a global or non-local variable using the `global` and `nonlocal` statements

```
x = 2

def foo():
    global x
    print(x)
    x = 3

foo()
print(x)
```

Specifies that x refers to the global variable

Prints 2

Assigns 3 to the global variable x

Prints 3

Agenda

- Sequencing
- Unstructured Control
- Structured Control
- Avoiding Control Flow

Expression Sequences

- Some languages provide syntax for explicitly sequencing the evaluation of expressions
  - Generally, the result of the overall expression is the result of the last one

- C++ example:

```
int x = (1 + 3, 4 / 2);
cout << x;
```

Evaluates 1+3, throws it away; evaluates 4/2, initializes x to 2

- Scheme example:

```
(define x (begin (+ 1 3) (/ 4 2)))
(display x)
```

Sets x to 2

Statement Sequences

- Statements generally have side effects, so they must execute in some well-defined sequence[1]
- ***Blocks*** and ***suites*** consist of sequences of statements
- The language syntax determines how statements are separated or terminated
  - Separated by semicolon:

    `S_1; S_2; ... ; S_N`
  - Terminated by semicolon:

    `S_1; S_2; ... ; S_N;`

Trailing semicolon required

[1]Compilers/interpreters can reorder statements if they can guarantee that it won't change the semantics.

Gotos

- Some languages provide a mechanism for direct transfer of control in the form of a ***goto***

Label

```
int x = 0;
LOOP: printf("%d\n", x);
x++;
goto LOOP;
```

Go to statement at given label

- Correspond to machine-level direct jumps

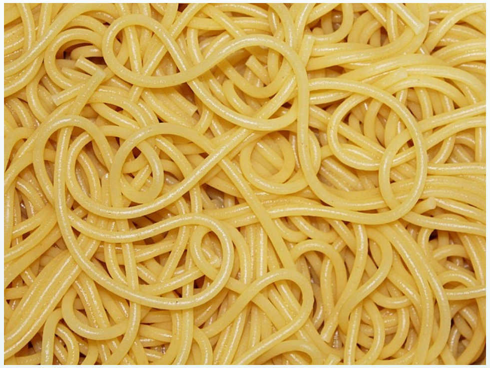
- Some languages provide a variant that can also branch

```
goto (10, 20, 30) i
```

Go to the ith label

Goto Problems

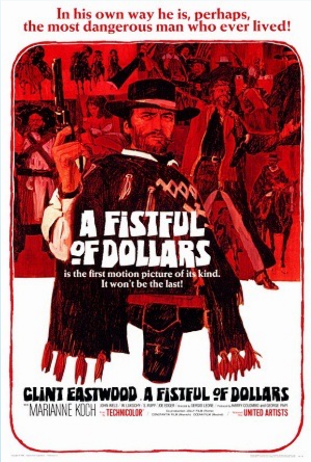
- Gotos are criticized for resulting in ***spaghetti code***, code with a complex control structure that is difficult to follow



```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
```

vs.

```
10 FOR i = 1 TO 10
20     PRINT i; " squared = "; i * i
30 NEXT i
40 PRINT "Program Completed."
```



Dangling Else

- In many languages, the syntax of conditionals results in a potential ambiguity

  ```
  if <test1> if <test2> then <stmt1> else <stmt2>
  ```

- Which if does the else belong to? This is called a ***dangling else***

- The usual resolution is that an else belongs to the innermost possible if

Switch Statements

- A ***switch*** or ***case*** statement allows branching based on the value of a non-boolean expression

```
switch <expression>:
  case <value1>: <statement1>
  case <value2>: <statement2>
  ...
  case <valueN>: <statementN>
  default: <statementN+1>
```

Generally must be compile-time constant

- Many differences between languages
  - Can a default case be defined
  - Do the cases have to cover all possible values
  - Does execution “fall through” from one case to another
  - Can a single case cover multiple values

The example is pseudocode and not from a real language.

Loop Termination

- Sometimes it can be useful to terminate a loop early

```
bool found = false;
for (size_t i = 0; i < size; i++) {
  if (array[i] == value) {
    found = true;
    goto end; break;
  }
}
end: cout << "found? " << found;
```

- `break`: terminate loop and move to code after loop
- `continue`: terminate loop iteration and move to next iteration

Termination in Nested Loops

- What if we want to terminate an outer loop (or iteration) from an inner loop?
- In C or C++, must either use goto or refactor code
- Java has ***labelled*** `break/continue`

```
outer: for (...) {
  for (...) {
    if (...) break outer;
  }
}
```

Exceptions

- Separate job of *detecting* errors from task of *handling* errors
  - May not have enough context at detection point to be able to recover
- Provide a *structured* mechanism for handling errors
  - Make it apparent in code what code an error handler covers and what kinds of errors it can handle
- Language provides:
  - Syntax for specifying what region of code a set of error handlers covers
  - Syntax for defining the error handlers for a region of code, and the kinds of exceptions each one can handle
  - A mechanism for ***throwing*** or ***raising*** an exception
  - Optionally: a mechanism for defining new kinds of exceptions

Scoping of Exception Handlers

- Exception handlers are *dynamically* scoped

```
def foo():
    try:
        bar()
    except NotImplementedError:
        print('caught exception')

def bar():
    baz()

def baz():
    raise NotImplementedError('baz')
```

- If exception reaches top level, program terminates

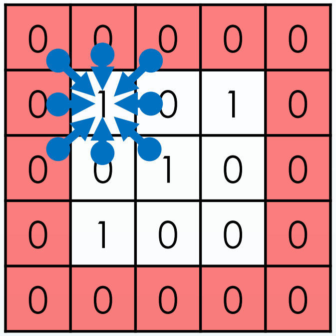
Avoiding Control Flow

- Algorithms can sometimes be implemented more cleanly or efficiently by avoiding control flow in favor of more abstract patterns

- Common tools
  - Lookup tables
    - Example: bit-counting example from a past lecture
  - Abstract data types
  - Functional patterns
    - Example: using `map` or `for-each` instead of a loop
    - We'll see these in a couple of weeks

Example: Lookup Table

- Recall Conway's game of life from Lab 1:
  - If a grid cell is alive and has 2 or 3 live neighbors, it stays alive
  - If a grid cell is dead and has 3 live neighbors, it becomes alive
  - Otherwise the cell becomes (or remains) dead

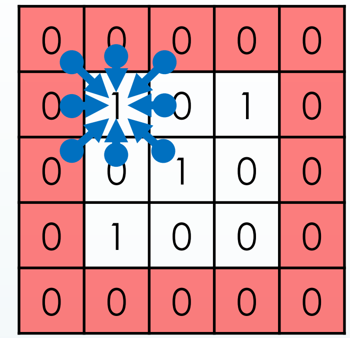


- Code with conditionals:

```
if grid[i,j] == 1 and neighbor_count in (2, 3):
    new_grid[i,j] = 1
elif grid[i,j] == 0 and neighbor_count == 2:
    new_grid[i,j] = 1
else:
    new_grid[i,j] = 0
```

Example: Lookup Table

- Recall Conway's game of life from Lab 1:
  - If a grid cell is alive and has 2 or 3 live neighbors, it stays alive
  - If a grid cell is dead and has 3 live neighbors, it becomes alive
  - Otherwise the cell becomes (or remains) dead



- Code with lookup table:

```
table = [[0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 1, 1, 0, 0, 0, 0]]
...
new_grid[i,j] = table[grid[i,j]][neighbor_count]
```

Example: ADTs

- Many "trick-taking" card games (e.g. Whist, Euchre, Bridge) have complicated rules about card values
- Example:
  - Gameplay-dependent: trump suit > led suit > other suits
  - Suit ordering: hearts > diamonds > clubs > spades
  - Rank ordering: A > K > Q > ... > 3 > 2
- Conditional-based comparator:

```
def Card_less(card1, card2, trump, led):
    if card1.suit == trump:
        return (card2.suit == trump
                and card1.rank.value < card2.rank.value)
    if card2.suit == trump:
        return True
    if card1.suit == led:
        return (card2.suit == led
                and card1.rank.value < card2.rank.value)
    if card2.suit == led:
        return True
    return (card1.suit.value < card2.suit.value
            or (card1.suit == card2.suit
                and card1.rank.value < card2.rank.value))
```

Example: ADTs

- Many “trick-taking” card games (e.g. Whist, Euchre, Bridge) have complicated rules about card values
- Example:
  - Gameplay-dependent: trump suit > led suit > other suits
  - Suit ordering: hearts > diamonds > clubs > spades
  - Rank ordering: A > K > Q > ... > 3 > 2
- ADT-based comparator:

```
def Card_value(card, trump, led):
    return (card.suit == trump, card.suit == led,
            card.suit.value, card.rank.value)

def Card_less(card1, card2, trump, led):
    return (Card_value(card1, trump, led)
            < Card_value(card2, trump, led))
```

Tuple of values that encode details about the card

Make use of lexicographic tuple ordering