Control flow

Amer Diwan

What is control flow

- The order in which operations are executed in a program
- e.g. in C++ like language,
  - a = 1;
    b = a + 1;
  - if a > 100 then b = a - 1; else b = a + 1;
  - a - b + c

Organization

- This topic
  - A look at basic control-flow:
    - Within expressions
    - Between statements
- Next topic (chapter 8 of text)
  - A closer look at procedure calls
  - Advanced topics: exception handling and closures

Control flow within expressions: associativity and precedence

- Which operators group more tightly (and thus execute first)
- In Java all binary operators except assignments are left associative
  - 3 - 3 + 5
  - x = y = f()
    (assignments evaluate to the value being assigned)

Precedence and associativity (cont.)

- In C++ arithmetic operators (+, -, *, ...) have higher precedence than relational operators (<, ...)
  – x + y < z + w
  – x + y == z

What if a language does not specify detailed associativity and precedence

- E.g., in Smalltalk, all operators are of equal precedence
  – a + b * c results in (a+b)*c
  – Must use parenthesis liberally!
- E.g., in Ada exponentiation does not associate
  – 4 ** 3**2 is illegal: must write (4**3)**2 or 4**(3**2)

Ordering within expressions

- `x = a - f() - c * d` equivalent to:
  - ```
    t1 = a - f();
    t2 = c * d;
    x = t1 - t2;
    ```
    or
  - ```
    t2 = c * d;
    t1 = a - f();
    x - t1 - t2;
    ```
- What's the difference?

Compiler optimizations also like flexibility in reordering expressions

- ```
  t = c * d;
  x = a - f() - c * d;
  ```
- If the compiler has the freedom, it can rewrite the above as:
- ```
  t = c * d;
  x = a - f() - t;
  ```
- Most languages allow the compiler to reorder as needed
- Java is an exception: requires left-to-right evaluation

Control flow within expressions: short circuit evaluation

- `boolean b = very_unlikely_condition &&`
  `very_expensive_condition`

- `boolean b = very_expensive_condition &&`
  `very_unlikely_condition`

- An operator is short-circuit if the remainder of computation can be skipped as soon as the truth value is discovered

Short-circuit evaluation: not just for performance

- ```
  if (p != nil && p.next.value == 10) {
     ...
  }
  ```
- The compiler must respect short-circuit operations (i.e., "short-circuit" is not a performance hint!)
  ```

Side effects and control flow within expressions

- The order of evaluation of subexpressions is particularly important if the subexpressions have side-effects
  - Modify variables
  - Perform I/O
  - Cause errors/exceptions

Referential transparency

- An expression is referentially transparent if given the same inputs it always produces the same output
  - `4`
  - `int f(i) { i + 1; }`
  - `int f(i) { i + j; }` and `j` is not a constant
- An RT expression can be reordered easily without changing the meaning of the program

Control flow between statements: Structured versus unstructured

- Unstructured: goto, jeq, ...
  - Generally resembles assembly instructions
- Structured: if...then...else, while..., for...
- Structured statements encourage top-down programming:
  - programming by successive refinement

Why structured or unstructured?

Programs that use structured control flow are easier to read

- ```
  if x < y goto 10
  call f
  goto 20
  10:
  call g
  20:
  ```
- ```
  if x < y then
    call f
  else
    call g
  end
  ```

...but sometimes unstructured makes sense

- ```
  while !please_break
         && i < 10 do
    while !please_break
           && j < 10 do
      if a == NIL then
        please_break =
          true
    end
  end
  ```
- ```
  while i < 10 do
    while j < 10 do
      if a == NIL then
        goto exit;
      end
  end
  exit:
  ```

We will now look at a selection of control constructs in modern programming languages

Control flow: Sequencing

- One statement appearing after another
- i = 1;
  j = i * 7;
- Does sequencing make sense if there are no side effects?

Control flow: selection

- Selects which statements to execute next
- e.g.,
  - if-then-else
  - case/switch statement (next slide)
  - pattern matching (later in the term)

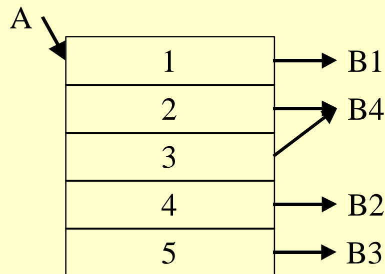
Case/switch statements

- Supports special case for cascaded if-then-else
  - Comparisons must compare to an ordinal constant (integer, enumeration type)
  -
```
switch f() {
   case 1: do_something1
   case 4: do_something2
   case 5: do_something3
   default: do_something4
}
```

Why case/switch statements?

- Compact
- May be more efficient than if-then-else

```
switch f() {
  case 1: B1
  case 4: B2
  case 5: B3
  default: B4
}
```




```
goto A[f()-1];
```

(assuming f()returns values between 1 and 5)

More on switch statements

- Efficiency is one reason why case labels need to be ordinal constants
- Another reason: simplicity

```
switch f() {
  case x: B1
  case y: B2
  case z: B3
  default: B4
}
```

- What if both x and y match?
- What if x, y, and z have side effects?

Guarded commands: a generalization of switch statements

- ```
  switch f() {
   ÿ x: B1
   ÿ y: B2
   ÿ z: B3
  }
  ```
- Guards can be arbitrary expressions (but side-effect free)
- If more than one matches, it picks one at random

Control flow: Iteration

- A conditional that keeps executing as long as the condition is true
- e.g: while, for, loop, repeat-until, ...
  ```

Logically controlled loops

- while, do-while, repeat-until, ...
- ```
  while i < 10 do
    a[i] := 0
    i := i + 1
  end
  ```
- WHILE checks condition at the top. REPEAT-UNTIL checks condition at the bottom

Enumeration controlled loops

- ```
  FOR i = first TO last DO
    ...
  END
  ```
- `i` is often automatically declared and often has the scope of the loop body

Access to index variable inside the loop

- ```
  FOR i := A'first TO A'last DO
    A[i] := 0;
  END
  ```
- ```
  FOR i = A'first TO A'last DO
    A[i] := 0;
    i := i + 1;
  END
  ```

Languages with enumeration controlled loops frequently do not allow modification to index variable

FOR statement example from Modula-3

- *FOR id := first TO last BY step DO S END*
- "The identifier id denotes a readonly variable whose scope is *S* and whose type is the common basetype of *first* and *last*"
- "...the statement steps id thorough the values first, first+step, first+2*step, ..., stopping when the value of id *passes* last... If step is negative the loop iterates downwards"

Example continued

- FOR i := 1 TO 10 BY 1
  DO
    A[i] := 0;
  END
- FOR i := 10 TO 1 BY -1
  DO
    A[i] := 0;
  END
- FOR i := 10 TO 1 BY 1
  DO
    A[i] := 0;
  END
- FOR i := x TO y BY z DO
    A[i] := 0;
  END

Some subtleties with FOR statements

- Overflow
  - FOR i := 1 TO n BY 1 DO
      print(i)
    END
  - What if ‘n’ is max-int?
- STEP sign determines whether the index variables needs to go above or below “upper”

Advantages and disadvantages of enumeration controlled loops

- Expressiveness
  - Can concisely and cleanly represent some kinds of loops
- Simplicity
  - All loop control information is in the header
  - Counting up/counting down is subtle
- Ease of implementation
  - Early implementations of M3 didn't get it right! (I recall it was an overflow problem)
- Efficiency
  - Provides more information to the compiler for optimization

Dealing with lack of gotos

- ```
  while i < 10 do
    while j < 10 do
      if a == NIL then
        goto exit;
      end
  end
  exit:
  ```
- Quite clean: how to handle it without gotos?
  - "Restricted gotos": "break", "continue", "exit",...
  ```

Labeled and unlabelled breaks

- ```
  while i < 10 do
    while j < 10 do
      if a == NIL
        break;
      end
  end
  ```
- ```
  outer:
  while i < 10 do
    while j < 10 do
      if a == NIL
        break outer;
      end
  end
  ```

Labeled breaks versus gotos

- Expressiveness
  - Can do more with gotos (but unless you are implementing a FSM you don't really need it)
- Simplicity
  - Breaks do not result in spaghetti code as easily as gotos
- Ease of implementation
  - Breaks (and structured control flow) gives more options to the compiler for analysis

Control flow: Recursion

- When a function may directly or indirectly call itself
- Can be used instead of loops
  – Functional languages frequently have no loops but only recursion

Recursion example

```
• while p != nil do
    p := p->next;
  end
```

```
• void trav(p) {
   if p != nil
     trav(p->next)
  }
```

•Some kinds of code are more readable with recursion; others are more readable with iteration
•Many more examples of iteration in the functional languages section of the course

Relationship to reading

- Reading has more discussion of
  - Different variations in for statements
  - Different variations in loop exit mechanisms
  - Suitability of recursion versus iteration

Next topic

- Closer look at calls, and exceptions
- Chapter 8 (see web page for section to read)