Control Flow

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 8 is out, due July 19.
  - Final exam, 1:00pm, Friday, August 2
- Readings: Read Chapter 6.1 - 6.5
- Lecture Contents
  - Introduction to Control Flow
  - Statements and Expressions
  - Variables
  - Short Circuit Evaluation
  - Sequencing
  - Selection
  - Iteration (Time permitting)

Introduction

- Control flow is the sequence of steps or computation that your program performs
- How we specify control flow matters because
  - The more expressive a language is, the easier it is for a programmer to implement an algorithm
  - The more control flow features a language provides, the easier it is to write and understand the code.

Control Flow

- Example: Which is easier to understand and faster to write?
  - **Snippet 1**

```
for( i = list.first(); !list.end();
                      i = list.next() ) {
  printf( “%s\n”, i );
}
```

  - **Snippet 2**

```
for i in list:
  print i
```

Prologue

- Before discussing control flow, we first need to understand what happens as the program executes
- A program execution (computation)
  - Executes statements
  - Evaluates expressions
  - Uses variables to
    - Store values
    - Refer to objects
  - Modifies variables by performing assignments
  - Makes decision using Boolean expressions
- This occurs regardless of the language
- We need to discuss these things first

Statements and Expressions

- Most languages have both statements and expressions
- Some languages are based on expression evaluation
  - Functional languages
  - E.g., Scheme
- Other languages are based on executing statements
  - Imperative languages
  - E.g., Java

Expressions

Are pieces of code that:

- Yield a result

  E.g. the result of 3 + 2 is 5
- Have no side-effects (usually)

  E.g., (+ a (* b c))
- A **side-effect** is the change of value of a variable

  E.g., (set! a 42)
  - Instantiation, where variables are initialized, is not a side-effect
- Expression evaluation is the basis of functional languages
- Examples:
  - Arithmetic expressions
  - Boolean or relational expressions
  - Function calls
  - Variable access or array indexing

Statements

Are pieces of code that

- Modify the value of variables
- **Have side-effects**
- Do not yield any values
- Are the basis of imperative languages
- Example:
  - Assignment statement
  - Conditional statement
  - Etc
- If a statement does not modify any variables, is it useful?

Some statements may also be recast as expressions
E.g., a conditional expression

Assignment Statements

- Assignment is the simplest (and most fundamental) type of computation (with side-effect)
- Consists of 2 parts:
  - R-Value : An expression that will yield a value to be stored
  - L-Value : The location where the value is to be stored
- Very important in imperative programming languages
- Examples:
  - FORTRAN, PL/1, SNOBOL4, C, C++, Java `A = 42`
  - Pascal, Ada, Icon, ML, Modula-3, ALGOL 68 `A := 42`
  - Smalltalk, Mesa, APL : `A <- 42`
  - J `A =. 42`
  - BETA `42 -> A`
  - COBOL `MOVE 42 TO A`
  - LISP `( SETQ A 42 )`

References and Values

- Expressions that yield values are referred to as r-values
- Expressions that yield memory locations are referred to as l-values
- Idea: Meaning of a variable name normally differs depending on the side of an assignment statement it appears on:
  - Right-hand side name refers to variable's value
  - Left-hand side name refers to variable's location

  Example: `a=b` vs `b=a`
- Some languages explicitly distinguish between l-values and r-values:
  - BLISS: X := **.X** + 1
  - ML: X := **!X** + 1
  - C/C++ on right hand side
    - X is the value
    - &X is the location
- In some languages, a function can return an l-value

  e.g., ML or C++

The & makes all the difference

```
int a[10];

int& f( int i ) {
  return a[i % 10];
}
…
for( int i = 0; i < 100; i++ )
  f( i ) = i;
```

Models of Variables

- Models of Variables
  - *Value Model*: Assignment copies the value
  - *Reference Model*:
    - A variable is always a reference
    - Assignment makes both variables refer to the same memory location
    - There is a big difference between:
      - Variables referring to the same object and
      - Variables referring to different but identical objects.
- Example: Java
  - Value model for built-in types
  - Reference model for classes and arrays
- Example: C
  - Default is value unless explicitly declared as pointer

Short Circuit Evaluation

- Idea: Most languages do not always evaluate the full Boolean expression
  - **a** and **b** : If **a** is false, **b** is not evaluated (hence no side-effect)
  - **a** or **b** : If **a** is true, **b** is not evaluated (hence no side-effect)
- Useful for optimization
- Creates problems if programmer expects **b** to be evaluated for side effects
- Some languages provide both regular and short-circuit Boolean operators

  E.g., Ada
  - `and` : short circuit
  - `and then` : full evaluation
  - `or` : short circuit
  - `or else` : full evaluation
- Example in C

```
while( p != NULL && p->val != val ) {
    p = p->next;
}
```

Types of Control Flow

- **Sequencing** : Ordering operations
- **Selection or alternation** : Conditionals
- **Iteration** : Loops
- Procedural abstraction : Functions / methods / subroutines
- Recursion
- Concurrency : Multithreading
- Exception handling and speculation: rolling back executions
- Nondeterminism : Implemented using search / backtracking

Sequencing

- In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.
  - Each statement follows the next
- Mixed imperative/functional languages (LISP, Scheme, . . .) often provide special constructs for sequencing.
- Issue: What's the value of a sequence of expressions/statements?

  Example: What is the value of this Scheme expression?

```
( let ( ( x 7 ) )
    ( + ( set! x 10 ) ( * 2 x ) )
)
```

- Typically, it's the last expression in the sequence

  e.g., C, LISP, Scheme, ...
- Some languages allow you to select which value to use

  e.g. LISP

```
(prog2 (+ 1 2) (* 2 3) (- 4 5))
```

  6

Gotos and Breaks

- A `goto` allows the program to jump to a new location in the instruction sequence
  - This is how all control structures are implemented at the machine level.
- The destination is denoted by a label (C/C++) or line # (BASIC)
  - What does the example on the right do?
- Use of goto is bad programming practice
  - Why?
  - Always?
- Sometimes we need to use gotos to break out of loops!

```
again:
   goto infinity
and:
   printf("and");
   goto beyond
infinity:
   printf("To infinity");
   goto and
beyond:
   printf("beyond.\n");
   goto again
```

Break statements

- Break statements break out of the current loop or switch statement
- Sometimes gotos are unavoidable:
  - Break out of nested loops
  - Break out of a subroutine
  - Break out of a deeply nested context
  - Handle error conditions
- Many languages provide alternatives:
  - Labeled breaks
  - return statement
  - Structured exception handling

```
// Java allows this
outer:
  for( … ) {
    for( ... ) {
      for( ... ) {
        ...
        break outer;
      }
    }
  }
...
```

Using goto to Handle Errors

- Some languages do not have exceptions
  - E.g., C
- In this case, gotos are a possible "good" use to separate common path code from error handling code.
- Idea:
  - Have a single error label at bottom of procedure
  - If error occurs during the common code, jump to error label

```
// Common path code
...
if( oops ) {
  goto error;
}
...
if( big oops ) {
  goto error;
}
...
return ...;
error:
  // handle error ...
```

Selection or Alternation

- Idea: Allow program to select a sequence based on a condition
- Standard if-then-else statement :
```
if ... then ...
        else ...
```
- Multi-way if-then-else:
```
if ... then ...
elif ... then ...
elif ... then ...
else ...
```
- Modern languages use elif, why?
- Avoids
  - Bunching of end markers
  - Unnecessary indenting

```
if cond_a:
  X()
else:
  if cond_b:
    Y()
  else:
    if cond_c:
      Z()
```

```
if cond_a:
  X()
elif cond_b:
  Y()
elif cond_c:
  Z()
```

Dangling Else Problem: Which of these is Correct?

```
if cond_a then
  if cond_b then
    X()
  else
    Y()
```

```
if cond_a then
  if cond_b then
    X()
else
  Y()
```

Use braces to avoid this ambiguity.

Switch Statements

- Switch statements are a special case of if/then/elsif/else

```
switch ... of
case ... : ...
case ... : ...
...
```

- Principal motivation:
  - Make code easier to read
  - Generate more efficient code
- Compiler can use different methods to generate efficient code:
  - Sequential testing (most common)
  - Jump table (common)
  - Binary search (less common)
  - Hash table (mostly interpreters)
- Choice depends on cases

Switch Implementation Using Sequential Testing

```
SWITCH i OF
CASE 1: clause_A
CASE 2, 7: clause_B
CASE 3..5: clause_C
CASE 10: clause_D
DEFAULT: clause_E
END
```

Suppose i is 4.

```
     move i to r1
     if r1 != 1 goto L1
     clause_A
     goto L6
L1:  if r1 == 2 goto L2
     if r1 != 7 goto L3
L2:  clause_B
     goto L6
L3:  if r1 < 3 goto L4
     if r1 > 5 goto L4
     clause_C
     goto L6
L4:  if r1 != 10 goto L5
     clause_D
     goto L6
L5:  clause_E
L6:  ...
```

Switch Implementation Using Jump Table

```
SWITCH i OF
CASE 1: clause_A
CASE 2, 7: clause_B
CASE 3..5: clause_C
CASE 10: clause_D
DEFAULT: clause_E
END
```

Suppose i is 7.

```
     move i to r1
     if r1 < 1 goto L5
     if r1 > 10 goto L5
     r1 := r1 - 1
     r1 := T[r1]
     goto *r1
T:  &L1, &L2, &L3, &L3, &L3,
    &L5, &L2, &L5, &L5, &L4
L1: clause_A
    goto L6
L2: clause_B
    goto L6
L3: clause_C
    goto L6
L4: clause_D
    goto L6
L5: clause_E
L6: ...
```

T is an array of pointers to jump locations

Switch Implementation Trade-Offs

- Jump table
  - Fast: one table lookup to find the right branch
  - Overhead for jump table storage
  - Potentially large table: one entry per possible value
  - Should be used with ”dense” switch statements

    I.e., Many cases in a given range
- Sequential Testing
  - Potentially slow
  - No storage overhead
  - Used for ”sparse” switch statements

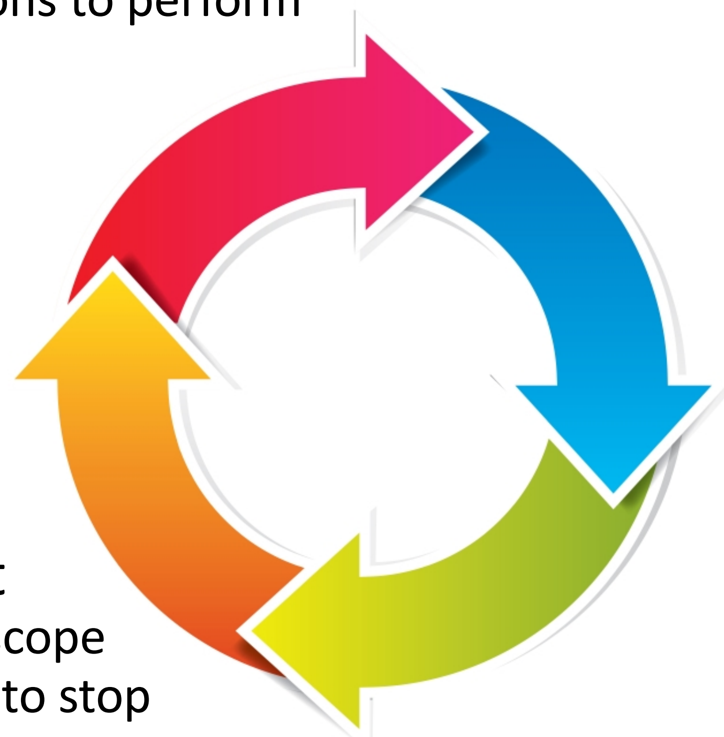
    I.e., small number of cases in a given range

Switch Implementation Trade-Offs

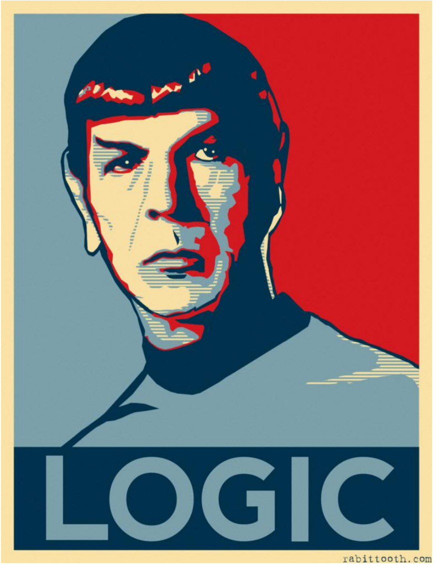
- Hash table
  - Fast: one hash table access to find the right branch
  - More complicated
  - Elements in a range need to be stored individually
  - Possibly large table
  - Used when case labels are strings or non-ordinal values
- Binary search
  - / Faster (special case of sequential testing
    - Slower than table lookup for ”dense” switch statements
  - No storage overhead
- No single implementation is best in all circumstances.
- Compilers often use different strategies based on the specific code.

Motivation for Iteration

- To be useful, programs need to
  - Repeat sequences of instructions
  - Decide at run-time how many iterations to perform
- E.g.,
  - Process arrays
  - Walk lists
  - Read arbitrary sized input
  - etc
- There are two approaches
  - Iteration
  - Recursion
- Iteration uses a loop construct that
  - Performs all repetitions in the same scope
  - Uses side-effects to determine when to stop



Iteration (Looping)



- Two types of loops
  - Logically controlled loops
    - Example: while-loop
    - Executed until a Boolean condition changes
    - The number of iterations is not known in advance
  - Enumeration-controlled loops
    - Example: for-loop
    - One iteration per element in a finite set
    - The number of iterations is known in advance

- Some languages do not have loop constructs

  E.g., Scheme, which uses tail recursion instead

Logically Controlled Loops

- Pre-loop test

  ```
  while ( condition ) do
     ...
  end
  ```

  - The loop may be executed 0 or more times
  - Test occurs before the iteration
- Post-loop test :

  ```
  do
     ...
  while ( condition )
  ```

  - Loop is executed at least once
  - Test is performed at end of iteration
- Mid-loop test or one-and-a-half loop

  ```
  loop
     ...
     if ( condition ) break
     ...
     if ( condition ) break
     ...
  end
  ```

  - Conditions are tested inside the loop and may break out
- Modern languages provide a `break` statement to use first two loop constructs in this way

Do-While Loop Implementation

DO

statements

WHILE cond

```
L1:
  statements
  r1 := cond
  if r1 goto L1
...
```

While Loop Implementation

WHILE cond do

 statements

END

```
   goto L2
L1:
   statements
L2: r1 := cond
    if r1 goto L1
...
```

For Loop Implementation

```
for( init;  cond; step ) {
  statements
}

// A "for" loop is a
// "while" loop with
// extra stuff
```

```
    [init]
    goto L2
L1:
    statements
    [step]
L2: r1 := [cond]
    if r1 goto L1
...
```

Enumeration Controlled Loops

- Use an index variable to count up or down the number of iterations
- The index variable can be incremented / decremented by a step other than 1

```
FOR i = start TO end BY step DO:
        ...
END
```

  - This loop
    - Initializes *i* to *start*
    - Adds *step* to i at the end of each iteration
    - Tests if *i* is less than *end*
    - If so, performs another iterations

For (Enumeration) Implementation

FOR i = start TO end BY step DO
  statements
END

```
     r1 := start
     r2 := end
     r3 := step
L1:  if r1 > r2 goto L2
     statements
     r1 := r1 + r3
     goto L1
L2:  ...
```

For (Enumeration) Implementation If the index is not used in the loop

```
FOR i = start TO end BY step DO
  statements
END
```

```
    r1 := end - start
    r1 := r1 / step
    inc r1
L1: if r1 == 0 goto L2
    statements
    dec r1
    goto L1
L2: ...
```

Trade-Offs

- Logically controlled loops are very flexible but expensive.
  - May have arbitrarily expensive condition
  - Cannot be unrolled
  - May have arbitrarily expensive step
- Enumeration-Controlled loops
  - Have a single int or float comparison
  - Can be unrolled to improve pipelining
  - Have a simple step,
    e.g., increment / decrement
- for-loop in C/C++/Java is syntactic sugar for init-test-step idiom of logic-controlled while loops.