Outline

- Control Flow
- Sequencing
- Selection
- Iteration

Sources:

PLP, 6.1 - 6.5

Control Flow

Control flow determines the order in which things get done in a program.

Primary mechanisms for control flow

- **Sequencing**: execute statements or evaluate expressions in sequential (or other explicitly specified) order
- **Selection or alternation**: make a choice based on some condition at run-time
  - if statements
  - case statements
- **Iteration**: execute a piece of code repeatedly
  - iterate until some condition is met (e.g. while loop)
  - iterate a fixed number of times (e.g. for loop)
  - iteration over collections
Control Flow

Control flow determines the order in which things get done in a program.

Additional mechanisms for control flow

• **Procedural abstraction**: use subroutine to parameterize and encapsulate a collection of control constructs
• **Recursion**: self-referencing expressions or subroutines
• **Concurrency**: execution of two or more program fragments “at the same time”
• **Exception handling** and **speculation**: program execution is interrupted and execution is transferred to a special handler
• **Nondeterminacy**: order or choice of statements is deliberately unspecified
• **Continuation**: save and later return to a specific point in a computation

Sequencing

Broad term including several specific sub-categories

• Expression evaluation (dominant form of control in functional languages)
• Execution of consecutive statements (imperative languages)
• Explicit goto statements (unstructured flow)

Expressions

What is an expression?

• **simple object**
  – literal constant
  – named constant
  – named variable
• **function application**
  – applied to one or more arguments, each of which is an expression
  – built-in functions called **operator**
  – arguments of operators called **operands**
  – notations include prefix, postfix, infix, mixfix

Most imperative languages use infix operator notation.

What are advantages and disadvantages of infix?

Precedence and Associativity

Consider this expression in FORTRAN:

\[ 6 + 2 \times 4 \times 2 \times 3 / 64 \]

Determined by operator **precedence**:

\[ 6 + ((2 \times (4 \times (2 \times 3))) / 64) = 2054 \]

What about

\[ 9 - 3 - 2 \]

Determined by operator **associativity**:

\[ 9 - 3 - 2 = 4 \]

Precedence and associativity vary among languages. For best results, check the manual.

Grammars can be used to enforce precedence and associativity.
Side Effects

If the evaluation of an expression influences subsequent computation in some other way besides returning a value, this is called a side effect.

Imperative languages

- Include expressions whose sole purpose is their side effect
- These are called statements (e.g. assignment)
- Imperative programming also called computing by means of side effects

Purely functional languages

- No side effects
- Said to be referentially transparent

Examples

- Imperative: C, JAVA, PASCAL
- Mostly imperative: C#, PYTHON, RUBY
- Mostly functional: ML, LISP
- Purely functional: HASKELL, MIRANDA

Structured and Unstructured Flow

The Infamous goto

- In machine language, there are no if statements or loops.
- We only have branches, which can be either unconditional or conditional (on a very simple condition).
- With this, we can implement loops, if statements, and case statements. In fact, we only need
  1. increment
  2. decrement
  3. branch on zero
  to build a universal machine (one that is Turing complete).
- We don’t do this in high-level languages (any more) because unstructured use of the goto can lead to confusing programs. See Go To Statement Considered Harmful by Edgar Dijkstra.

L-values and R-values

Expressions that denote locations are called l-values
Expressions that denote values are called r-values

Value Model

- variable is used as a name for the value stored in that variable
- same expression can be an l-value or r-value depending on its context

Example

\[
a = b + c;
\]

Here, \(a\) is an l-value because it refers to the location of the variable \(a\). Both \(b\) and \(c\) are r-values.

Reference model

- Every variable is an l-value
- To get a value, the variable must be dereferenced
- Dereferencing can be automatic (based on context, e.g. C) or explicit (e.g. ML)

Structured and Unstructured Flow

Structured alternatives to goto

- Iteration: general-purpose iteration constructs
- Exit from subroutine: explicit return statements
- Exit from loop: explicit break or continue statements
- Return from nested subroutine: Some languages support this explicitly
- Exceptions Language mechanisms for throwing exceptions
**Selection**

- **if** Condition then Statement – **PASCAL, ADA**
- **if** (Condition) Statement – **C/C++, JAVA**

To avoid ambiguities, use end marker: `end if, "}"`

To deal with multiple alternatives, use keyword or bracketing:

```
if Condition then
  Statements
elsif Condition then
  Statements
else
  Statements
end if;
```

**Nesting**

```
if Condition1 then
  if Condition2 then
    Statements1
  end if;
else
  Statements2
end if;
```

**Statement Grouping**

- **PASCAL** introduces begin-end pair to mark sequence
- **C/C++/JAVA** abbreviate keywords to `{ }`
- **ADA** dispenses with brackets for sequences, because keywords for the enclosing control structure are sufficient
- `for J in 1..N loop ... end loop`
  - More writing but more readable
- Another possibility – make indentation significant (e.g., **ABC, PYTHON, HASKELL**)

**Short-circuit evaluation**

```
if x/y > 5 then z := ... -- what if y = 0?
if y /= 0 and x/y > 5 then z := ...
```

But binary operators normally evaluate both arguments. Solutions:

- a lazy evaluation rule for logical operators (**LISP, C**)
  
  ```
  C1 && C2     // don’t evaluate C2 if C1 is false
  C1 || C2     // don’t evaluate C2 if C1 is true
  ```

- a control structure with a different syntax (**ADA**)
  
  ```
  -- don’t evaluate C2
  if C1 and then C2 then -- if C1 is false
  if C1 or else C2 then  -- if C1 is true
  ```
Multiway selection

Case statement needed when there are many possibilities “at the same logical level” (i.e. depending on the same condition)

```plaintext
case Next_Char is
  when 'I' => Val := 1;
  when 'V' => Val := 5;
  when 'X' => Val := 10;
  when 'C' => Val := 100;
  when 'D' => Val := 500;
  when 'M' => Val := 1000;
  when others => raise Illegal_Numeral;
end case;
```

Can be simulated by sequence of if-statements, but logic is obscured.

Implementation of case

A possible implementation for C/C++/JAVA/ADA style case:

(If we have a finite set of possibilities, and the choices are computable at compile-time.)

- build table of addresses, one for each choice
- compute value
- transform into table index
- get table element at index and branch to that address
- execute
- branch to end of case statement

This is not the typical implementation for a ML/HASKELL style case.

Complications

```plaintext
case (x+1) is
  when integer'first..0 ⇒ Put_Line ("negative");
  when 1 ⇒ Put_Line ("unit");
  when 3 | 5 | 7 | 11 ⇒ Put_Line ("small prime");
  when 2 | 4 | 6 | 8 | 10 ⇒ Put_Line ("small even");
  when 21 ⇒ Put_Line ("house wins");
  when 12..20 | 22..99 ⇒ Put_Line ("manageable");
  when others ⇒ Put_Line ("irrelevant");
end case;
```

Implementation would be a combination of tables and if statements.

C style case

```plaintext
switch (Next_Char) {
  case 'I': Val = 1;
  case 'V': Val = 5;
  case 'X': Val = 10;
  case 'C': Val = 100;
  case 'D': Val = 500;
  case 'M': Val = 1000;
  default: Illegal_Numeral = true;
}
```

What's wrong with this code?
C style case

```c
switch (Next_Char) {
    case 'I': Val = 1; break;
    case 'V': Val = 5; break;
    case 'X': Val = 10; break;
    case 'C': Val = 100; break;
    case 'D': Val = 500; break;
    case 'M': Val = 1000; break;
    default: Illegal_Numeral = true;
}
```

Unstructured Flow (Duff’s device)

```c
void send (int *to, int *from, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
        case 0: do {*to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
            } while (--n > 0);
    }
}
```

Discovered by Tom Duff in 1983; discovery announced with “a combination of pride and revulsion”.

Indefinite loops

- All loops can be expressed as while-loops
  - good for invariant/assertion reasoning
- condition evaluated at each iteration
- if condition initially false, loop is never executed

```c
while condition loop ... end loop;
```

is equivalent to

```c
if condition then
    while condition loop ... end loop;
end if;
```

if condition has no side-effects

Executing while at least once

Sometimes we want to check condition at end instead of at beginning; this will guarantee loop is executed at least once.

- `repeat ... until condition;` (PASCAL)
- `do { ... } while (condition);` (C)

while form is most common can be simulated by while + a boolean variable:

```c
first := True;
while (first or else condition) loop
    ...
    first := False;
end loop;
```
Breaking out

A more common need is to be able to break out of the loop in the middle of an iteration.

- **break** (C/C++, JAVA)
- **last** (PERL)
- **exit** (ADA)

```plaintext
loop
  ... part A ...
  exit when condition;
  ... part B ...
end loop;
```

Breaking way out

Sometimes, we want to break out of several levels of a nested loop

- give names to loops (ADA, PERL)
- use a goto (C/C++)

```plaintext
Outer: while C1 loop ... 
  Inner: while C2 loop ... 
    Innermost: while C3 loop ... 
      exit Outer when Major_Failure;
      exit Inner when Small_Annoyance;
      ... 
    end loop Innermost;
  end loop Inner;
end loop Outer;
```

Definite Loops

Counting loops are iterators over discrete domains:

- **for J in 1..10 loop ... end loop;**
- **for (int i = 0; i < n; i++) { ... }**

Design issues:

- evaluation of bounds
- scope of loop variable
- empty loops
- increments other than 1
- backwards iteration
- non-numeric domains

Evaluation of bounds

```plaintext
for J in 1..N loop 
  ... 
  N := N + 1;
end loop; -- terminates?
```

Yes – in ADA, bounds are evaluated once before iteration starts. C/C++/JAVA loop has hybrid semantics:

```plaintext
for (int j = 0; j < last; j++) {
  ... 
  last++; -- terminates?
}
```

No – the condition “\(j < last\)” is evaluated at the end of each iteration.
The loop variable

- is it mutable?
- what is its scope? (i.e. local to loop?)

Constant and local is a better choice:

- **constant**: disallows changes to the variable, which can affect the loop execution and be confusing
- **local**: don’t need to worry about value of variable after loop exits

```
Count: integer := 17;
...
for Count in 1..10 loop
  ...
end loop;
... -- Count is still 17
```

Non-numeric domains

**ADA** form generalizes to discrete types:

```
for M in months loop ... end loop;
```

Basic pattern on other data types:

- define primitive operations: `first, next, more_elements`
- implement for loop as:

  ```
  iterator = Collection.Iterate();
  element thing = iterator.first;
  for (element thing = iterator.first;
          iterator.more_elements();
          thing = iterator.next()) {
    ...
  }
  ```

Different increments

**ALGOL 60**:

```
for j from exp1 to exp2 by exp3 do ...
```

- too rich for most cases; typically, `exp3` is `+1` or `-1`
- what are semantics if `exp1 > exp2` and `exp3 < 0`?

**C/C++**:

```
for (int j = exp1; j <= exp2; j += exp3) ...
```

**ADA**:

```
for J in 1..N loop ...
for J in reverse 1..N loop ...
```

Pre- and Post-conditions

How can we prove that a loop does what we want? **pre-conditions** and **post-conditions**:

```
{P} S {Q}
```

If proposition `P` holds before executing `S`, and the execution of `S` terminates, then proposition `Q` holds afterwards.

Need to formulate:

- pre- and post-conditions for all statement forms
- syntax-directed rules of inference

```
{P and C} S {P}
{P and C} while C do S endloop {P and not C}
```
Efficient exponentiation

function Exp (Base: Integer; Expon: Integer) return Integer is
N: Integer := Expon; -- successive bits of exponent
Res: Integer := 1; -- running result
Pow: Integer := Base; -- successive powers: $Base^{2^i}$
begin
  while N > 0 loop
    if N mod 2 = 1 then
      Res := Res * Pow;
    end if;
    Pow := Pow * Pow;
    N := N / 2;
  end loop;
  return Res;
end Exp;

Adding invariants

function Exp (Base: Integer; Expon: Integer) return Integer is
N: Integer := Expon; -- successive bits of exponent
Res: Integer := 1; -- running result
Pow: Integer := Base; -- successive powers: $Base^{2^i}$
begin
  i := 0; -- count iterations
  while N > 0 loop
    if N mod 2 = 1 then -- ith bit of Expon from left
      Res := Res * Pow; -- $Res := Base^{(Expon \mod 2^i)}$
    end if;
    Pow := Pow * Pow; -- $Pow := Base^{2^i}$
    N := N / 2; -- $N := Expon/(2^i)$
  end loop;
  i = lg Expon; Res = Base^{Expon}; N = 0
  return Res;
end Exp;