Control Structures

Control Statements

- Statements that change the order in which other statements are executed
  - Involve a machine-level branch
- Originally emulated machine level
  - goto and conditional goto
- Goto Controversy
  - Bohm & Jacopini proved that all programs could be written using only selection and conditional loop
- Writeability & readability improved with a reasonable number of control statements
- Control structure is a control statement plus the statement(s) it controls
- Single-entry, single exit – improves readability/reliability
  - at least same statement always executed after control structure

Selection Statements

- Choice between alternative statement(s)
- Two-way and n-way
Two-way Selection

- if statement
- Design issues
  - Type/form of conditional expression
  - Specification of then and else parts
  - Semantics of nested selectors
- Conditional
  - Syntactic delimitation
  - Type
    - boolean
    - Other
- Clauses
  - Single statement or compound statement
  - Statement group (control structure has delimiters)

Nested Selectors

- Matching else clause with if
- Dangling else
  - Ambiguous grammar
  - Semantic rule
  - Unambiguous grammar
    - Disallow if-then in then part
    - Structure delimiters
Multiple-Selection

- case statement
- Design issues
  - Selector expression
  - Specification of selection groups
  - Single to multiple group selection
  - Selector values
  - Unrepresented selector values
- C switch
  - Selector expression and values are discrete type
  - default optional
  - Flow through from case to case unless break
- Ada
  - Selector expressions and values are discrete type
  - Others required unless cases exhaustive (all values must be covered)
  - No flow through
Implementation

- Multiple tests and branches
  - E.g. C switch
- Branch table
  - Values and branch addresses
    - Search and branch
      - Linear, hash
- Branch array
  - Array of branch addresses indexed by selector value
  - Only effective for small dense selector ranges

```c
switch (index) {
    case 1:
    case 3: odd = 1;
        sumod += index;
        break;
    case 2:
    case 6: even = 1;
        sumeven += index;
        break;
    default: printf("Error in switch, index = \%d", index);
}
```
Multiple-Selection Using if

- Nested ifs
- Special syntax
  - E.g., Python
  - `COND` in LISP & Scheme

```python
if count < 10:
  bag1 = True
elif count < 100:
  bag2 = True
elif count < 1000:
  bag3 = True
```

```lisp
(if (<= count 10) 
  (let ((bag1 true)) 
    ...)
   ... 
```
Loops

- Repetition of group of statements
- Counting vs conditional loop vs both
- Position of decision
  - Pretest
  - In-test
  - Posttest

Counting Loops

- Constituents
  - Control variable
  - Initial, final and step values
  - Loop parameters
  - Body
- Design issues
  - Type and scope of control variable
  - Can control variable and/or loop parameters be changed in body
  - How often are loop parameters evaluated
  - What is value of control variable after exit
Ada for:
- Control variable - discrete type defined by range
- Scope of control variable is for stmt
- Control variable is constant in body
- Loop parameters evaluated only once
- Change in body has no effect
- Cannot branch into for
- Operational semantics

C for:
- No explicit control variable or loop parameters
- Operational semantics
- Can branch into loop
- Each expression can be omitted
- Compound expressions
- C++ & Java allow declaration in first expr (scope is stmt)
- Java requires second expr to be boolean
- Functional languages use recursion

```plaintext
for variable in (reverse) discrete_range loop
  ... end loop;
```

```
Count : Float := 1.5;
for Count in 1..10 loop
  Sum := Sum + Count;
end loop;
```

```
[define for var (its type is that of the discrete range) [evaluate discrete range]
loop
  if there are no elements left in the discrete range] go out
  for var - [next element of discrete range]
  [loop body]
goto loop
end:
[undefined for var]
```

```
for (expression_1; expression_2; expression_3)
  loop body
```

```
for (count = 1; count <= 10; count++)
  ...
```

```
expression_1
  loop
    [expression_2 < 0 go out]
  [loop body]
expression_3
  goto loop
  out: ...
```
Conditional Loops

- Design Issues
  - Placement of test
    - Pretest, posttest, in-test
    - Included in counting or special statement
- C while and do
  - Condition is integer
  - Operational semantics
  - Can branch into loop
- In-test loop
  - Separate construct or built from other statements
    - Exit from multiple levels
    - C, Java
    - Ada
while (condition) {
    loop_body
    \(\text{if condition is false, \text{go to}}\) loop
}

while (condition) {
    loop_body
    \(\text{if condition is true, \text{go to}}\) loop
}

\(\text{while} (\text{sum} + 1000) \{
\text{get\textunderscore\text{next\textunderscore\text{value}}};
\text{if (value < 0) break; }
\text{sum} += \text{value};
\}

\(\text{while} \text{sum < 5000} \text{loop}
\text{get\textunderscore\text{value}};
\text{exit when value < 0}
\text{sum} += \text{sum} + \text{value};
\text{end loop}

let \text{foo \text{while}\text{loop} test body} =
\text{if test()} \text{then}
\text{body}()
\text{while\text{loop} test body}
\text{else}
(())
Iterators

- Data-based iteration
  - Traversing structures
- Range in Python
- Subrange types in Ada
- Array iterators in PHP
- For-each in Java (foreach in C#)
  - Iterable interface
  - Iterator class
- Blocks in Ruby
  - Iterator methods take a block
  - Optional parameters
  - times, each, upto

```python
for count in range [0, 9, 2]:
    subtype myrange is integer range 0..9;
    myArray : array (myrange) of integer;
    for index in myrange loop
        end loop;
    reset $list;
    print ("First number: ", current($list) . "+\r")
    while [$current_value = next($list)]
        print ("Next number: ", $current_value . "+\r")
    for (String myElement : $list) [ ... ]
```

```ruby
>> t.times { puts "Hey!" }  
Hey:  
Hey:  
Hey:  
1 4
>> list = [2, 4, 6, 8]  
>> list.each { |value| puts value }  
2  
4  
6  
8  
>> [2, 4, 6, 8]
 1.upto(5) { |x| print x, " " }  
1 2 3 4 5
```
Unconditional Branching

- `goto`
  - Reduces readability & reliability
  - Replaced by restricted `goto`
  - E.g., `break`
  - Still appear in most languages

Guarded Commands

- Non-deterministic programming
- Dijkstra (1975)
  - Basis for concurrency in CSP & Ada
- Selection
  - All guards evaluated
  - Execute one of the statements with a `true` guard
  - Can be non-deterministic
- Loop
  - All guards evaluated until all `false`
  - Execute one of the statements with a `true` guard
  - Can be non-deterministic

```plaintext
if <Boolean expression> -> <statement>
else <Boolean expression> -> <statement>
else <Boolean expression> -> <statement>
fi
```

```plaintext
if x < y -> max := x
else y < x -> max := y
fi
```
do <Boolean expression> -> <statement>
[] <Boolean expression> -> <statement>
[ ] ... 
[ ] <Boolean expression> -> <statement>
end

do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
do q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
do q5 > q6 -> temp := q5; q5 := q6; q6 := temp;
do q7 > q8 -> temp := q7; q7 := q8; q8 := temp;
do q9 > q10 -> temp := q9; q9 := q10; q10 := temp;
do q11 > q12 -> temp := q11; q11 := q12; q12 := temp;
do q13 > q14 -> temp := q13; q13 := q14; q14 := temp;
do q15 > q16 -> temp := q15; q15 := q16; q16 := temp;