Statements

Programming Languages

William Killian

Millersville University

Outline

- Statements vs. Expressions
- Sequenced Statements
- Selection Statements
  - Two-way
  - Multi-way
- Iterative Statements
  - Pre-test
  - Post-test
  - Counter controlled
  - Data-Structure controlled
- Control Mechanisms

Statements vs. Expressions

- Expressions will ***always*** have a **type**
- Expressions will ***always*** yield a **value**

- Statements may have no type or value

```
#  Python
print ("Hello, world!")

// C / C++ / C# / Java
if (x < min) {
    min = x;
}
```

Sequenced Statements

- Statements are said to be sequenced if they are evaluated/executed in a sequenced order
- Usually referred to as **blocks**

```
// C-like languages
{
    statement1;
    statement2;
    statement3;
}
```

```
# ruby
do
    statement1
    statement2
    statement3
end
```

```
(* F# / OCaml *)
begin
    statement1;
    statement2;
    statement3
end
```

```
# python
#   indentation
    statement1
    statement2
    statement3
```

```
# Lisp
(progn
   (statement1)
   (statement2)
   (statement3))
```

Sequenced Statements

Design Decisions

- Symbols or keywords used to denote a block
  - Usually curly braces or begin/end
- Should indentation matter?
  - Permitted in F#
  - Required in Python
- Statement separators?
  - Semicolons in most languages (optional in F#)
  - None for most scripting languages

Selection Statements

Selection Statements

- Selection provides the ability to choose between two or more paths of execution

- **Two-Way Selection**
  - Choosing between two options
  - Often based on a yes/no decision
- **Multi-Way Selection**
  - Choosing between more than two options
  - Often based on a value

Two-Way Selection

Commonly called an if-else statement

General Form:

```
if <control_expression>
  then <clause>
  else <clause>
```

- What's the form of the control expression?
- How are the clauses specified?
- Can we nest two-way selectors?

Two-Way Selection: Control

- The type of the control expression usually must evaluate to a Boolean type
  - Coerced from integral type in C, C++, and Python
- The control expression might be wrapped within parentheses. This is done in most C-like languages.

Two-Way Selection

C-Like Languages

```
if (cond)
     stmt;
else
     stmt;
```

Two-Way Selection

Python

```
if cond:
    stmt
else:
    stmt
```

Two-Way Selection

Ruby

```
if cond [then]
     stmt
else
     stmt
end
```

Two-Way Selection

OCaml

```
if cond then
     expr
else
     expr
```

Nested Selectors

```
if (cond)
     if (cond2) stmt1;
  else stmt2;
```

Question: Which if gets the else?

Nested Selectors

C-Like Languages

```
if (cond) {
    stmt1;
} else if (cond2) {
    stmt2;
} else {
    stmt3;
}
```

Nested Selectors

Python

```
if cond:
    stmt1
elif cond2:
    stmt2
else:
    stmt3
```

Nested Selectors

Ruby

```
if cond then
    stmt1
elif cond2 then
    stmt2
else
    stmt3
end
```

Multi-Way Selection

Allow the selection of one of any number of statements or statement groups

Design Issues:

- Form + type of control expression?
- Syntax for selectable segments?
- Execute multiple segments?
- Specification for case values?
  - Unrepresented values?

Multi-Way Selection

C, C++, Java, Javascript

```
switch (expr) {
    case val1: stmt1; break;
    case val2: stmt2; break;
    case val3: stmt3; // fall through
    [default:  stmtN;]
}
```

Fall through means that stmtN executes after stmt3

Multi-Way Selection

Ruby

```
case
    when cond1 then stmt1
    when cond2 then stmt2
    else stmt3
end
```

Multi-Way Selection

OCaml

```
match expr with
| pattern1                    -> expr1
| pattern2 [when cond] -> expr2
| pattern3                    -> expr3
```

The **first** matched pattern will return the corresponding expr

Multi-Way Selection

Lisp

```
(cond
  (cond1 expr1)
  (cond2 expr2)
  (cond3 expr3)
  (t     exprN))
```

The **first** truthy condition will return the corresponding expression

Iterative Statements

Iterative Statements

There are only three ways to perform the same statement more than once:

1. Manual repetition in code
2. Recursive
3. Iteration

How can we control iteration?

Infinite Loops

C-like Languages

```
while (true)
    <stmt>
```

Python

```
while True:
    <stmt>
```

Ruby

```
loop do
    <stmt>
end
```

F# / OCaml

```
while true do
    <expr>
done
```

Pre-test Loops

- Also known as a *while* loop

- Condition is checked before each iteration
  - If the condition evaluates to true, the loop body is executed
  - If the condition evaluates to false, the loop is done executing

Syntactically like an if-statement with no "else"

Pre-test Loops

Python

```
while cond:
    <stmt>
```

C-like Languages

```
while (cond)
    <stmt>
```

F#

```
while cond do
    <expr>
```

OCaml

```
while cond do
    <expr>
done
```

Pre-test Loops

Python (w/ else)

```
while cond:
    <stmt>
else:
    <stmt>
```

Ruby

```
while cond [do]
    <stmt>
end
```

Ruby (until)

```
until cond [do]
    <stmt>
end
```

Post-test Loops

- Also known as a *do-while* loop

- Condition is checked **after** each iteration
  - If the condition evaluates to true, the loop body is executed
  - If the condition evaluates to false, the loop is done executing

Execute the body at least once

Post-test Loops

C-like Languages

```
do
   <stmt>
while (cond)
```

Ruby

```
begin
   <stmt>
end while cond
```

Advantages?

Disadvantages?

Counter-Controlled Loops

- Also known as a *for-loop*

Three Components:

- Looping variable (with initial value)
- Exit condition (based on looping variable)
- Modifier for looping variable (usually increment)

Questions:

What is the type & scope of the variable?

Should we be able to change the variable?

Counter-Controlled Loops

C-like Languages

```
for (<init>; <test>; <update>)
    <stmt>
```

`<init>` - declaration with initializer or assignment

Evaluated only once

`<test>` - same as the condition for while

If omitted, infinite loop

`<update>` - expression that modifies the variable

Counter-Controlled Loops

C-like Languages

```
for (<init>; <test>; <update>) {
    <stmt>
}

{ // rewritten as a while
    <init>
    while <test> {
        <stmt>
        <update>
    }
}
```

Counter-Controlled Loops

OCaml / F#

```
for <var> = <low> to <high> do
  <expr>
done

for <var> = <high> downto <low> do
  <expr>
done
```

Emulated – not an actual loop. ***Use Recursion***

Counter-Controlled Loops

OCaml / F#

```
for <var> = <low> to <high> do
  <expr>
done
```

```
let <var> = <low> in <expr>;
let <var> = <low> + 1 in <expr>;
let <var> = <low> + 2 in <expr>;
...
```

Counter-Controlled Loops

OCaml / F#

```
for <var> = <high> downto <low> do
  <expr>
done
```

```
let <var> = <high> in <expr>;
let <var> = <high> - 1 in <expr>;
let <var> = <high> - 2 in <expr>;
...
```

Data Structure Controlled Loops

Traversal through an array or data structure is a common pattern across most languages

Case Studies

- PHP
- Java
- C#
- C++
- Python
- Ruby

Data Structure Controlled Loops

PHP

arr must model an *Iterator*

```
// traversing regular array
foreach (arr as $value)
    stmt

// traversing associative array
foreach (arr as $key => $value)
    stmt
```

Data Structure Controlled Loops

Java

arr must model *Iterable<E> (iterator())*

Called an enhanced for-loop

```
// traversing regular array
for (var x : arr)
    <stmt>
```

Data Structure Controlled Loops

Java

`iter` must model *`Iterator<E> (next(), hasNext())`*

The equivalent to the prior slide

```
var iter = arr.iterator();
while (iter.hasNext()) {
   var x = iter.next();
   <stmt>
}
```

Data Structure Controlled Loops

C#

`arr` must model *`IEnumerable<T> (GetEnumerator())`*

Called a foreach loop

```
foreach (var elem in arr)
    <stmt>
```

Data Structure Controlled Loops

C#

en must model *IEnumerator<T> (MoveNext(), Current)*

The equivalent to the prior slide

```
var en = arr.GetEnumerator();
while (en.MoveNext()) {
    var elem = en.Current;
    <stmt>
}
```

Data Structure Controlled Loops

C++

`obj` must model *`Container<T> (begin(), end())`*

Called a Range-based for loop

```
for (auto& elem : obj)
    <stmt>
```

`begin()` and `end()` must return an `Iterator<T>`

Data Structure Controlled Loops

C++

The equivalent to the prior slide

```
auto&& __range = obj;
auto __begin = begin(__range);
auto __end = end(__range);
for ( ; __begin != __end ; ++__begin)
{
    auto& elem = *__begin;
    <stmt>
}
```

Data Structure Controlled Loops

Python

`elems` must model *`iterator`* (`__iter__()`)

`__iter__()` must model incrementable (`__next__()`)

For loops rely on objects that can be iterated

```
for val in elems:
    <stmt>
```

Data Structure Controlled Loops

Python

The equivalent to the prior slide

```
obj = iter(elems)
try:
    while True:
        val = next(obj)
        <stmt>
except StopIteration:
    pass
```

Data Structure Controller Loops

Ruby

Three instances of *iterator methods*

- `times`

```ruby
10.times { puts "Hello" }
# executes the block 10 times
```

- `each`

```ruby
arr.each { |x| puts x }
# prints each element of an array
```

- `upto`

```ruby
330.upto(420) { |i| puts i }
# 330 <= i < 420
```

Control Mechanisms

Control Mechanisms

- Infinite loops can't run *forever*
- Complex logic can't always be expressed in a pre- or post-test
- There are times where we may want to:
  - Prematurely exit a loop / control structure
  - Prematurely advance to the next loop iteration

Control Mechanisms: **break**

- Used to prematurely exit a loop or control structure

```
int sum = 0;
for (int x : arr) {
    if (x > 10) break;
    sum += x;
}
// sum ?
```

Control Mechanisms: `continue`

- Used to prematurely advanced to the next iteration

```
int sum = 0;
for (int x : arr) {
    if (x > 10) continue;
    sum += x;
}
// sum ?
```

Ruby:
Called **`next`**
See also: **`redo`**

Control Mechanisms: `goto`

- Used to **arbitrarily transfer control**
- "Go To Statement Considered Harmful" – *Dijkstra*
- Direct mapping to low-level assembly instructions
- C / C++ / FORTRAN

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

A Note on Theory

(1960s) All algorithms represented by flow charts can be implemented with

- Two-way selection (if/else)
- Pre-test logical loops (while)

Which structures do you most commonly use?