Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 18

- Control Flow
  - Assignment: Value Model and Reference Model
  - Structured and unstructured flow
  - Sequencing and selection
  - Logically- and enumeration-controlled iteration
  - Iterators

Assignments and Expressions

- Fundamental difference between imperative and functional languages
- **Imperative languages**: "computing by means of side effects"
  - Computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing values of variables
- Expressions in (pure) **functional language** are *referentially transparent*:
  - All values used and produced depend on the local referencing environment of the expression
  - A function is *idempotent* in a functional language: it always returns the same value given the same arguments because of the absence of side-effects

L-Values vs. R-Values and Value Model vs. Reference Model

- Consider the assignment of the form: ***a* := *b***
  - The left-hand side *a* of the assignment is an *l-value* which is an expression that should denote a location, e.g. array element **a[2]** or a variable **foo** or a dereferenced pointer ***p** or a more complex expression **(f(a)+3)->b[c]**
  - The right-hand side ***b*** of the assignment is an *r-value* which can be any syntactically valid expression with a type that is compatible to the left-hand side
- Languages that adopt the *value model* of variables copy the value of *b* into the location of *a* (e.g. Ada, Pascal, C)
- Languages that adopt the *reference model* of variables copy references, resulting in shared data values via multiple references
  - Clu, Lisp/Scheme, ML, Haskell, Smalltalk adopt the reference model. They copy the reference of *b* into *a* so that *a* and *b* refer to the same object
  - Most imperative programming languages use the value model
  - **Java** is a mix: it uses the value model for built-in types and the reference model for class instances

Assignment in Value Model vs. Reference Model

```
b := 2;
c := b;
a := b + c
```

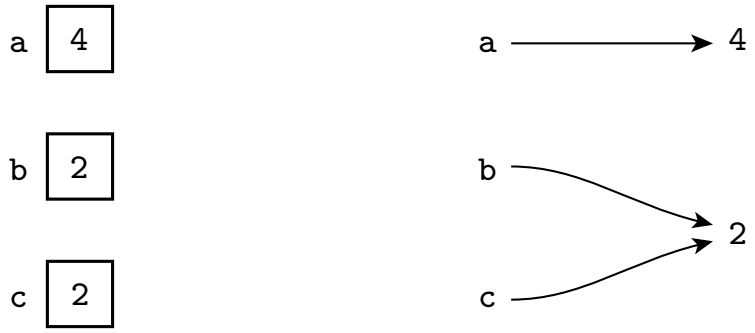


Figure 6.2 The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

Denotational semantics of value model and reference model

- A PL with **value model** has the usual Env and Store semantic domains
  - Env = Ide → Dval (Dval = … + Loc + …)
  - Store = Loc → Sval
- "r-values" are expressions that evaluate to elements of domain Sval (storable values)
- "l-values" are expressions **e** that evaluate to locations: (E{**e**} r s **as Loc**)
- In a PL with **reference model**, conceptually there is no Store, but only
  - Env = Ide → Dval thus *E*: Exp → Env → Eval
- The main binding operator is **let**

  Exp = … | let Ide = Exp in Exp

with semantics

E{ let x = e in e1 } r = *E*{ e1 } r[*E*{e}r / x]

- Note: let x = e in e1 is just syntactic sugar for (λ x. e1) e

References and pointers

- Most **implementations** of PLs have as target architecture a Von Neumann one, where memory is made of cells with addresses
- Thus implementations use the ***value model*** of the target architecture
- Assumption: every data structure is stored in memory cells
- We "define":
  - A **reference to X** is the address of the (base) cell where X is stored
  - A **pointer to X** is a location containing the address of X
- Value model based implementation can mimic the ***reference model*** using *pointers* and standard assignment
  - Each variable is associated with a location
  - To let variable **x** refer to data **X**, the address of (reference to) **X** is written in the location of **x**, which becomes a pointer.
  - Can be modeled by requiring that **Loc** is contained in **Sval**
  - Expressions of "reference types" must return a location

Denotational Semantics of Reference Memory Model on Value Memory Model

Semantic interpretation functions

D: Decl → Env → Store → (Env x Store)

C: Cmd → Env → Store → Store

***E*: Exp → Env → Store → (Eval x Store)**

Env = Ide → Dval

Store = Loc → Sval

Dval = … + Loc + …

Eval = … + Loc + Sval + …

Sval = … + Loc + …

Semantics: declaration

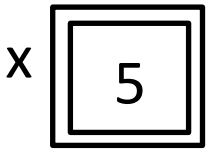
$D\{\textbf{var } x =_{ref} e\}$ r s = (r[l/x], s1[n/l])

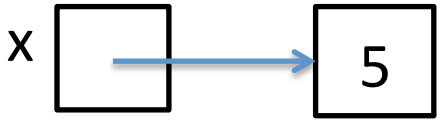
where l = *newloc*(s)

and (n,s1) = *E*{*e*} r s

and (n ***as Loc***)

Allocates a new location bound to x and referring to n





Special Cases of Assignments

- Assignment by *variable initialization*
  - Use of *uninitialized variable* is source of many problems, sometimes compilers are able to detect this but with programmer involvement e.g. *definite assignment* requirement in Java
  - Implicit initialization, e.g. 0 or NaN (not a number) is assigned by default when variable is declared
- Combinations of *assignment operators* (`+=, -=, *=, ++, --`…)
  - In C/C++ `a+=b` is equivalent to `a=a+b` (but `a[i++]+=b` is different from `a[i++]=a[i++]+b`, !)
  - Compiler produces better code, because the address of a variable is only calculated once
- *Multiway assignments* in Clu, ML, and Perl
  - `a,b := c,d // ` assigns `c` to `a` and `d` to `b` simultaneously,
    - e.g. `a,b := b,a` swaps `a` with `b`
  - `a,b := f(c) // f` returns a pair of values

Structured and Unstructuted Flow

- *Unstructured flow*: the use of ***goto*** *statements* and *statement labels* to implement control flow
  - Close correspondence with conditional/unconditional branching in assembly/machine code
  - Merit or evil? Hot debate in 1960's. Dijkstra "GOTO Considered Harmful"
  - Böhm-Jacopini theorem: goto's are not necessary
  - Generally considered bad: programs are hardly understandable
  - Sometimes useful for jumping out of nested loops and for coding the flow of exceptions (when a language does not support exception handling)
  - Java has no goto statement (supports labeled loops and breaks) but **goto** is a reserved word

Structured and Unstructuted Flow

- *Structured flow*:
  - Statement sequencing
  - Selection with "if-then-else" statements and "switch" statements
  - Iteration with "for" and "while" loop statements
  - Subroutine calls (including recursion)
  - All of which promotes "structured programming”
- Structured alternatives to goto
  - *break* to escape from the middle of a loop
  - *return* to exit a procedure
  - *continue* to skip the rest of the current iteration of a loop
  - *raise (throw)* an exception to pass control to a suitable handler
  - *multilevel return* with *unwinding* to repair the runtime stack (e.g. **return-from** statement in Common Lisp)
- Cannot jump into *middle* of block or function body

Sequencing

- A list of statements in a program text is executed in top-down order
- A *compound statement* is a delimited list of statements
  - A compund statement is called a *block* when it includes variable declarations
  - C, C++, and Java use { and } to delimit a block
  - Pascal and Modula use **`begin ... end`**
  - Ada uses **`declare ... begin ... end`**
- Special cases: in C, C++, and Java expressions can be inserted as statements
- In pure functional languages sequencing is impossible (and not desired!)
- In some (non-pure) functional languages a sequence of expression has as value the last expression's value

Selection

- If-then-else selection statements in C and C++:
  - **`if`** (<expr>) <stmt> [**`else`** <stmt>]
  - Condition is a bool, integer, or pointer
  - Grouping with { and } is required for statement sequences in the *then clause* and *else clause*
  - Syntax ambiguity is resolved with "*an else matches the closest if*" rule
- Conditional expressions, e.g. **`if`** and **`cond`** in Lisp and **`a?b:c`** in C
- Java syntax is like C/C++, but condition must be Boolean
- Ada syntax supports multiple **`elsif`**'s to define nested conditions:
  - **`if`** <cond> **`then`**
    <statements>
    `elsif` <cond> **`then`**
    ...
    `else`
    <statements>
    `end if`

Selection (cont'd)

- Case/switch statements are different from if-then-else statements in that an expression can be tested against multiple constants to select statement(s) in one of the arms of the case statement:
  - C, C++, and Java:
    `switch` (<expr>)
    { **`case`** <const>: <statements> **`break`**;
    `case` <const>: <statements> **`break`**;
    ...
    `default`: <statements>
    }
  - A **`break`** is necessary to transfer control at the end of an *arm* to the end of the switch statement
  - Most programming languages support a switch-like statement, but do not require the use of a break in each arm

Selection (cont'd)

- The allowed types of <exp> depends on the language: e.g. int, char, enum, strings (in C# and Java)
- Some languages admit label ranges
- A switch statement is much more efficient compared to nested if-then-else statements
- Several possible implementation techniques with complementary advantages/disadvantages:
  - Jump tables
  - Sequential testing (like if … then … elseif …)
  - Hash tables
  - Binary search

Iteration

- An **iterative command** (or *loop*) repeatedly executes a subcommand, which is called the **loop body**.
- Each execution of the loop body is called an **iteration**.
- Classification of iterative commands:
  - **Indefinite iteration**: the number of iterations is not predetermined.
  - **Definite iteration**: the number of iterations is predetermined.
- Note: sequencing, selection and **definite** iteration are not sufficient to make a language Turing complete: either **indefinite** iteration or **recursion** is needed

Iteration

- ***Enumeration-controlled loops*** (aka ***bounded/definite iteration***) repeat a collection of statements a number of times, where in each iteration a *loop index variable* (*counter, control variable*) takes the next value of a set of values specified at the beginning of the loop

- ***Logically-controlled loops*** (aka ***unbounded/indefinite iteration***) repeat a collection of statements until some Boolean condition changes value in the loop
  - *Pretest loops* test condition at the begin of each iteration
  - *Posttest loops* test condition at the end of each iteration
  - *Midtest loops* allow structured exits from within loop with exit conditions

Logically-Controlled Pretest loops

- *Logically-controlled pretest loops* check the exit condition before the next loop iteration
- Not available in Fortran-77
- Pascal:
  `while` <cond> **`do`** <stmt>
  where the condition is a Boolean-typed expression
- C, C++:
  `while` (<expr>) <stmt>
  where the loop terminates when the condition evaluates to 0, NULL, or false
  - Use **`continue`** and **`break`** to jump to next iteration or exit the loop
- Java is similar C++, but condition is restricted to Boolean

Logically-Controlled Posttest Loops

- *Logically-controlled posttest loops* check the exit condition after each loop iteration
- Not available in Fortran-77
- Pascal:
  `repeat` <stmt> [; <stmt>]* **`until`** <cond>
  where the condition is a Boolean-typed expression and the loop terminates when the condition is true
- C, C++:
  `do` <stmt> **`while`** (<expr>)
  where the loop terminates when the expression evaluates to 0, NULL, or false
- Java is similar to C++, but condition is restricted to Boolean

Logically-Controlled Midtest Loops

- Ada supports *logically-controlled midtest loops* check exit conditions anywhere within the loop:

```
loop
   <statements>
exit when <cond>;
   <statements>
exit when <cond>;
   ...
end loop
```

- Ada also supports labels, allowing exit of outer loops without gotos:

```
outer: loop
   ...
   for i in 1..n loop
      ...
      exit outer when a[i]>0;
      ...
   end loop;
end outer loop;
```

- Java allows **labeled breaks** to exit of outer loops

Enumeration-Controlled Loops

General form:

```
for I = start to end by step do
   body
```

- Informal operational semantics…

Some critical issues

- Number of iterations?
- What if **`I`**, **start** and/or **end** are modified in body?
- What if **step** is negative?
- What is the value of **`I`** after completion of the iteration?

Enumeration-Controlled Loops

- Some failures on design of enumeration-controlled loops
- Fortran-IV:

```
          DO 20 i = 1, 10, 2
          ...
     20   CONTINUE
```

which is defined to be equivalent to

```
          i = 1
     20   ...
          i = i + 2
          IF i.LE.10 GOTO 20
```

- Problems:
  - Requires positive constant loop bounds (1 and 10) and step size (2)
  - If loop index variable i is modified in the loop body, the number of iterations is changed compared to the iterations set by the loop bounds
  - GOTOs can jump out of the loop and also from outside into the loop
  - The value of counter **i** after the loop is implementation dependent
  - The body of the loop will be executed at least once (no empty bounds)

Enumeration-Controlled Loops (cont'd)

- Fortran-77:
  - Same syntax as in Fortran-IV, but many dialects support **ENDDO** instead of **CONTINUE** statements
  - Can jump out of the loop, but cannot jump from outside into the loop
  - Assignments to counter **i** in loop body are not allowed
  - Number of iterations is determined by
    $$\max(\lfloor (H - L + S) / S \rfloor, 0)$$
    for lower bound *L*, upper bound *H*, step size *S*
  - Body is not executed when $(H\text{-}L\text{+}S)/S < 0$
  - Either integer-valued or real-valued expressions for loop bounds and step sizes
  - Changes to the variables used in the bounds *do not affect* the number of iterations executed
  - Terminal value of loop index variable is the most recent value assigned, which is
    $$L + S * \max(\lfloor (H\text{-}L\text{+}S)/S \rfloor, 0)$$

Enumeration-Controlled Loops (cont'd)

- Algol-60 combines logical conditions in *combination loops*:

  `for` <id> := <forlist> **`do`** <stmt>

  where the syntax of <forlist> is

  <forlist> ::= <enumerator> [, <enumerator>]*
  <enumerator> ::= <expr>
  | <expr> **`step`** <expr> **`until`** <expr>
  | <expr> **`while`** <cond>

- Not orthogonal: many forms that behave the same:

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i+2 while i < 10 do ...
```

Enumeration-Controlled Loops (cont'd)

- Pascal's enumeration-controlled loops have simple and elegant design with two forms for *up* and *down*:
  `for` <id> := <expr> **`to`** <expr> **`do`** <stmt>
  and
  `for` <id> := <expr> **`downto`** <expr> **`do`** <stmt>
- Can iterate over any discrete type, e.g. integers, chars, elements of a set
- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable cannot be assigned in the loop body
- Final value of loop counter after the loop is undefined

Enumeration-Controlled Loops (cont'd)

- Ada's for loop is much like Pascal's:

```
for <id> in <expr> .. <expr> loop
    <statements>
end loop
```

and

```
for <id> in reverse <expr> .. <expr> loop
    <statements>
end loop
```

- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable has a local scope in the loop body
  - Not accessible outside of the loop
- Counter variable cannot be assigned in the loop body

Enumeration-Controlled Loops (cont'd)

- C and C++ **do not have true enumeration-controlled loops**, they have *combination loops*
- A "for" loop is essentially a logically-controlled loop

- ```
  for (i = first; i <= last; i += step) {
      ...
  }
  ```

is equivalent to

```
{
    i = first;
    while (i <= last) {
        ...
        i += step;
    }
}
```

- Java's standard **for** statement is as in C/C++, but the **enhanced for** is almost a true enumeration-controlled loop (see later)
```

Enumeration-Controlled Loops (cont'd)

- Why is C/C++/Java **for** not enumeration controlled?
  - Assignments to counter `i` and variables in the bounds are allowed, thus it is the programmer's responsibility to structure the loop to mimic enumeration loops
- Use **`continue`** to jump to next iteration
- Use **`break`** to exit loop
- C++ and Java also support local scoping for counter variable

```
for (int i = 1; i <= n; i++) ...
```

- In this case the look index variable is not accessible after the loop

Enumeration-Controlled Loops (cont'd)

- Other problems with C/C++ for loops to emulate enumeration-controlled loops are related to the mishandling of bounds and limits of value representations
  - This C program never terminates (do you see why?)

```
#include <limits.h> // INT_MAX is max int value
main()
{ int i;
  for (i = 0; i <= INT_MAX; i++)
    printf("Iteration %d\n", i);
}
```

  - This C program does not count from 0.0 to 10.0, why?

```
main()
{ float n;
  for (n = 0.0; n <= 10; n += 0.01)
    printf("Iteration %g\n", n);
}
```

Enumeration-Controlled Loops (cont'd)

- How is loop iteration **counter overflow** handled?
- C, C++, and Java: nope
- Fortran-77
  - Calculate the number of iterations in advance
  - For **`REAL`** typed index variables an exception is raised when overflow occurs
- Pascal and Ada
  - Only specify step size 1 and -1 and detection of the end of the iterations is safe
  - Pascal's final counter value is undefined (may have wrapped)

Iterators

- *Containers (collections)* are aggregates of homogeneous data, which may have various (topo)logical properties
  - Eg: arrays, sets, bags, lists, trees,…
- Common operations on containers require to iterate on (all of) its elements
  - Eg: search, print, map, …
- *Iterators* provide an abstraction for iterating on containers, through a sequential access to all their elements
- Iterator objects are also called *enumerators* or *generators*

Iterators in Java

- Iterators are supported in the Java Collection Framework: interface **`Iterator<T>`**
- They exploit generics (as collections do)
- Iterators are usually defined as *nested classes* (*non-static private member classes*): each iterator instance is associated with an instance of the collection class
- Collections equipped with iterators have to implement the **`Iterable<T>`** interface

```
class BinTree<T> implements Iterable<T> {
    BinTree<T> left;
    BinTree<T> right;
    T val;
    ...
    // other methods: insert, delete, lookup, ...
    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }
```

Iterators in Java (cont'd)

```
class BinTree<T> implements Iterable<T> {
    …
    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>();
        TreeIterator(BinTree<T> n) {
            if (n.val != null) s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {      //preorder traversal
            if (!hasNext()) throw new NoSuchElementException();
            BinTree<T> n = s.pop();
            if (n.right != null) s.push(n.right);
            if (n.left != null) s.push(n.left);
            return n.val;
        }
        public void remove() {
            throw new UnsupportedOperationException();
  } }
}
```

Iterators in Java (cont'd)

- Use of the iterator to print all the nodes of a BinTree:

```
for (Iterator<Integer> it = myBinTree.iterator();
                                    it.hasNext();   )
    {   Integer i = it.next();
        System.out.println(i);
    }
```

- Java provides (since Java 5.0) an *enhanced for* statement (*foreach*) which exploits iterators. The above loop can be written:

```
for (Integer i : myBinTree)
        System.out.println(i);
```

- In the *enhanced for*, **`myBinTree`** must either be an array of integers, or it has to implement **`Iterable<Integer>`**
- The enhanced for on arrays is a **bounded iteration.** On an arbitrary iterator it depends on the way it is implemented.

Iterators in C++

- C++ iterators are associated with a container object and used in loops similar to pointers and pointer arithmetic
- They exploit the possibility of overloading primitive operations.

```
vector<int> V;
…
for (vector<int>::iterator it = V.begin(); it !=
V.end(); ++it)
    cout << *it << endl;
```

An in-order tree traversal:

```
tree_node<int> T;
…
for (tree_node<int>::iterator it = T.begin(); it !=
T.end(); ++it)
    cout << *it << endl;
```

True Iterators

- While Java and C++ use *iterator objects* that hold the state of the iterator, Clu, Python, Ruby, and C# use "*true iterators*" which are functions that run in "parallel" (in a separate thread) to the loop code to produce elements
  - The *yield* operation in Clu returns control to the loop body
  - The loop returns control to the generator's last yield operation to allow it to compute the value for the next iteration
  - The loop terminates when the generator function returns

True Iterators (cont'd)

- Generator function for pre-order visit of binary tree in Python
- Since Python is dynamically typed, it works automatically for different types

```
class BinTree:
    def __init__(self):     # constructor
        self.data = self.lchild = self.rchild = None
    ...
    # other methods: insert, delete, lookup, ...
    def preorder(self):
        if self.data != None:
            yield self.data
        if self.lchild != None:
            for d in self.lchild.preorder():
                yield d
        if self.rchild != None:
            for d in self.rchild.preorder():
                yield d
```

Iterators in some functional languages

- Exploting “in line” definitions of functions, the ***body*** of the iteration can be defined as a function having as argument the loop index
- Then the body is passed as last argument to the **iterator** which is a function realising the loop
- Simple iterator in Scheme and sum of 50 odd numbers:

```
(define uptoby
    (lambda (low high step f)
        (if (<= low high)
            (begin
              (f low)
      (uptoby (+ low step) high step f))
            '())))
```

```
(let ((sum 0))
    (uptoby 1 100 2
        (lambda (i)
        (set! sum (+ sum i))))
    sum)
```