Principles of Programming Languages
Lecture 18

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

Function Type

- **Function types**
  - A type that allow an object to be invoked or called as if it were an ordinary function
  - E.g. Sorting an array containing a user defined type

```
// Declaration of C sorting function
void sort (int [] itemlist, int numitems,  int (*cmpfunc)(item*, item*)  );
...

// Callback function
int compare_function( item* A, item* B)
{
  //……
  //……
}
.
sort( itemlist, numitems, compare_function);
```

Function Type

- **Function types**
  - A type that allow an object to be invoked or called as if it were an ordinary function
  - E.g. using different sorting algorithm

```
  // Declaration of C sorting function
void bubblesort (int [] itemlist, int numitems,  int (*cmpfunc)(int*, int*)  ) {
  ….
}
void quicksort (int [] itemlist, int numitems,  int (*cmpfunc)(int*, int*)  ) {
….
}

void doSomething(….) {
    ...
    func1(itemlist , numitems , bubblesort);
    …
    func2(itemlist , numitems , quicksort);
}
```

```
void func1(….,sortfunc ) {

    sortfunc(itemlist,numitems)

}
void func2(….,sortfunc ) {

    sortfunc(itemlist,numitems)

}
```

Data types Summary

- **Primitive types**
  - **Integer, Float, Boolean, Char**
- **Structured Types**
  - **Strings, Ordinal, Arrays, Associative Arrays, Records, Union, Lists**
- **Object Type**
- **Class Type**
- **Function Type**

Note: Don’t confuse data types with API data structures (BST, Graphs,…)

Type Conversions

- Often want to write expressions that are mixed mode (contain operands of more than one type)

  real + integer

- This implies a need to convert one type to another so the expression can be evaluated
  - coercion is *implicit conversion*, done automatically by the compiler
    - implies semantics that define rules for determining type to convert to from operands
    - problem is loss of error detection
  - casts are *explicit conversions* specified by the programmer
    - can lead to very clumsy expressions if doing a lot of mixed mode expressions

```
int (Index);        // Ada and Python
(int) Index;        // C and Java
```

Type Conversions - cont'd

- Whether implicit or explicit, conversions can be
  - widening
    - convert to a type with a greater representation range
      - although perhaps with a loss of precision
    - integer → real
  - narrowing
    - convert to a type with a more restricted range
    - double precision → real
  - promoting
    - convert to a type with additional semantic information
    - integer → character e.g. NewChar = chr(IntValue)
  - demoting
    - strip away semantic information
    - character → integer

Type Conversion – examples

- PL/I allows coercion between almost any types

```
DCL  A, B, C  INT;
if (A <= B <= C) then ...
```

  - A <= B yields a Boolean value, which is a single bit 0 or 1
  - convert bit to integer to compare to C
  - bit <= C is true for any positive values of C

- Ada allows no coercion
  - all conversions must be casts
  - conversions are allowed between all numeric types
  - other conversions only allowed between derived types that share an ancestor

```
type foo is new Boolean;
type bar is new Boolean;
A: foo;  B:  bar;
A := foo(B);
```

Data Types Questions

- What are the data types in the language?
- How is the data type declared?
- What operations are allowed on each data type?
- How to reference the data type?
- How is the data type implemented by the compiler/interpreter?
- What conversion rules exist for each datatype?
- What type checking does the language support?

Data types: how many should a language have?

- **Early programming languages:**
  - Many data types
  - Support large range of applications

- **Modern programming languages:**
  - Few basic types and few basic data structure
  - Allow a programmer to design complex structures for every need

Components of an Imperative Language

- Data types

→ Variables & Expressions

- Assignment construct
- Iteration construct
- Branching construct
- Function construct
- Container construct

Names: design decisions

- **How long can a name be? What characters can be used? Are connecters allowed?**
  - 30-60 max characters are most practical and used by most languages
- **Are names case sensitive?**
  - Disadvantage: names that look alike are different
  - Java and Modula-2 are the worst because predefined function names are mixed (E.g. java.lang.Integer.parseInt(…)) You just have to remember that…
- **What is the set of keyword vs. reserved words?**
  - *Keywords*: special only in certain contexts
    - E.g. In Fortran

```
REAL TEMP        // ok
REAL = 3.4       // also ok!
```

  - *Reserved words*: cannot be used by programmer as names
    - E.g. In Java `double int = 5;   //  error`

Variables: introduction

- **A variable is an abstraction of a computer memory cell**
  - A variable is a not a name!

- **A variable can be characterized by a sixtuple of attributes:**
  - Name
  - Address:
    - A variable may have different addresses at different times during execution
    - A variable may have different addresses at different places in a program
    - If two variable names can be used to access the same memory location, they are called *aliases* *(aliases are harmful to readability)*
  - Value:
    - The content of the location with which the variable is associated.
  - Type:
    - Determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determine the precision.
  - Lifetime:
    - It is the time during which the variable is bound to a particular memory cell
  - Scope

Variables: value vs. reference

- **How will the variables be used in statements and how they are represented in memory?**

- **Example:**

```
// Java language
int b = 3;
int a = b;
```

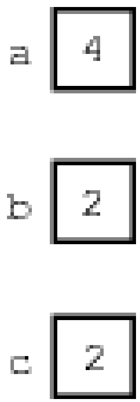
```
//  C language
int   c    = 3;
int   b   = &c ;
int   a   = b;

int*  z ;
z    = (int)malloc(size(int))
*z  = 10
```

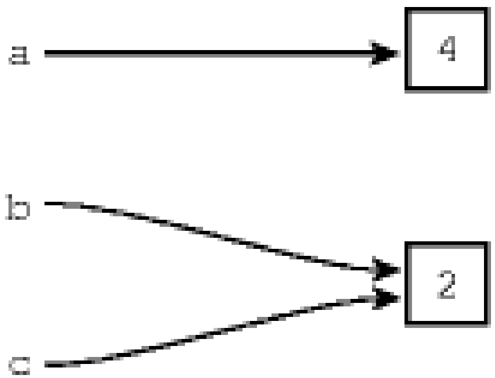
Variables: value vs. reference

- **Example:**

b = 2
c = b
a = b + c



Value model



*b = 2
c = b
*a = *b + *c

Reference model

Variables: variations

- **Value-oriented**
  - C/C++, Pascal, Basic, Ada
- **Reference-oriented**
  - C/C++, Scheme/ML (functional languages)
  - Clu, Smalltalk
- **Hybrids**
  - Algol-68, C/C++
  - Java
    - built-in types are values (int, float, double..)
    - user-defined types are objects (i.e., references)

Variables: variations

- **Encoded variables**

- **E.g. Perl**
  - scalars starts with $

    $numberOfRooms = 23;
  - Arrays starts with @

    @stringArray = ("This", "is", 'an', "array", 'of', "strings");
  - Associative Array starts with %

    %associativeArray = ("Jack A.", "Dec 2", "Joe B.", "June 2", "Jane C.", "Feb 13");

Binding: introduction

- **A binding is an association between two things, such as a name and the thing it names.**
  - E.g.

```
int x;          // for this to work the compiler must bind a memory
                // cell (that is sufficient to hold an integer) with the
                // identifier x so later on the programmer can  write
x = 5;
```

- **The** *binding time* **is the point at which a binding is created or, more generally, the point at which any implementation decision is made.**
  - When will the name bind to the value?

Binding: variables to storage

- **Binding creation/allocation:**
  - Getting a (memory) cell from some pool of available (memory) cells and establishing an association between it a specific variable
- **Binding destruction/de-allocation:**
  - Putting a (memory) cell back into the pool and destroying the association between it and the variable.
- **Binding lifetime:**
  - The period of time from creation of a binding to its destruction

Binding: when?

- **During language design**
  - E.g.: In Pascal, % operator is bound to mod operation
- **During language implementation**
  - E.g. In Java, data type double is bound to certain range
- **At compile time**
  - E.g. In Java, int x; x is bound to particular data type (integer)
- **When linking**
  - E.g. A call to subprogram *foo* in a separate library (.dll / .lib / .so) is bound to the subprogram code
- **At load time**
  - E.g. In C, a variable may be bound to a storage cell when the program is loaded
- **At run time**
  - E.g. In C++, a variable may be bound to a storage cell after the program is loaded (using pointers).

Binding: classification

- **Static vs. Dynamic:**
  - The terms *static* and *dynamic* are generally used to refer to things bound before run time and at run time, respectively.
  - Static binding:
    - Can mean many different times (e.g., language design, compile time, etc.).
  - Dynamic binding:
    - Generally referring to binding times such as when variable values are bound to variables.
    - Advantage: flexibility
    - Disadvantage: High cost, Type error detection by the compiler is difficult

Type Checking

- **It is the activity of ensuring that the operands of an operator are of compatible types:**
  - Subprograms are considered as operators
  - A compatible type is one that is either legal for the operator or is allowed under language rules to be implicitly converted to a legal type:
    - E.g.

```
float R  = (float)10;        // valid in most languages
int X    = (int)10.5;        // invalid in most languages
```

- **Type error:**
  - It is an attempt to apply a function to an argument of the wrong type
  - E.g.

```
int X =  java.lang.StrictMath.round( “nancy”);
```

Type Checking

- **Type checking depends on binding:**
  - *Static type checking:* if all bindings of variables are done at compile type, then type checking can be done statically.

  - *Dynamic type checking:* if some bindings of variables are done at run time, then type checking will be for thoese variables dynamically, i.e. when the program is running.

Type Checking: classification

- **Strongly typed language:**
  - It is one in which each name in a program in the language has a single type associated with it, and that type is known at compile time (i.e. statically bound).
  - E.g.: ML, Pascal, Ada

- **Not-Strongly typed language:**
  - It is one in which variable types may be known but the storage location to which it is bound may store values of different types at different times.
  - E.g.: C, C++, Fortran

- **Weakly typed language:**
  - It is one in which a name in a program in the language can change the type associated with it during run time and type checking is
  - E.g.: Basic, Perl, Python

Components of an Imperative Language

- Data types
- Variables

→ Operators & Expressions

- Iteration construct
- Branching construct
- Function construct
- Container construct

Operators & Expressions

- **Mathematical operators**
- **Logical operators**
- **Bitwise operators**
- **User-defined operators**

Mathematical & Logical Operators

- **Precedence rules**
  - **Norm: respect mathematical precedence**
  - **Force evaluation of specific terms using ( )**
  - **E.g. (x + y) * z**

    vs.

    x + (y * z)

- **Short circuit evaluation**
  - **E.g.**

    if (X < 5 && foo() == 100)

    vs.

    if (foo() == 100 && X < 5)
  - **E.g.**

    if (X < 5 || foo() == 100)

    vs.

    if (foo() == 100 || X < 5)

Bitwise Operators

- Low level operations supported directly by CPU
- Performed on register content
- Bitwise AND, Bitwise OR, Bitwise, Shift left, Shift right, XOR,…

```
  0 1 0 1 0 1 1 0
& 0 0 1 1 0 0 1 0
  ---------------
  0 0 0 1 0 0 1 0
```

```
  0 1 0 1 0 1 1 0
| 0 0 1 1 0 0 1 0
  ---------------
  0 1 1 1 0 1 1 0
```

```
~ 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0
  -------------------------------
  1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1
```

```
    0 1 0 1 0 1 1 0 << 2
-------------------
0 1 0 1 0 1 1 0 0 0
```

```
0 1 0 1 0 1 1 0 >> 1
---------------
  0 1 0 1 0 1 1
```

Bitwise Operators

- **Why care?**
- **Languages: C/C++, PhP**
- **E.g.**

```
// C language. left shifting is the equivalent of multiplying by
// a power of two
int mult_by_pow_2(int number, int power) {
        return number<<power;
}
```