CSE 452: Programming Languages

Data Types

Where are we?

Machine Language
High-level Programming Languages
Assembly Language
Logic
Object Oriented
Imperative
Functional

You are here

Concepts
• specification (syntax, semantics)
• variables (binding, scoping, types, …)
• statements (control, selection, assignment, …)

Implementation
• compilation (lexical & syntax analysis)

Types: Intuitive Perspective

- Behind intuition:
  - Collection of values from a “domain” (denotational perspective)
  - Internal structure of data, described down to small set of fundamental types (structural view)
  - Equivalence class of objects (implementor’s approach)
  - Collection of well-defined operations that can be applied to objects of that type (abstraction approach)

- Utility of types:
  - Implicit context
  - Checking: ensure that certain meaningless operations do not occur. (type checking can’t catch all).
Terminology

- Strong typing: language prevents you from applying an operation to data on which it is not appropriate.
- Static typing: compiler can do all the checking at compile time.
- Examples:
  - Common Lisp is strongly typed, but not statically typed.
  - Ada is statically typed.
  - Pascal is almost statically typed.
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically.

Type System

Has rules for:

- Type equivalence
  - (when are the types of two values the same?)
- Type compatibility
  - (when can a value of type A be used in a context that expects type B?)
- Type inference
  - (what is the type of an expression, given the types of the operands?)

Type compatibility/equivalence

- Compatibility: tells you what you can do
- More useful concept of the two
- Erroneously used interchangeably
- Equivalence:
  - What are important differences between type declarations?
  - Format does not matter:
    ```c
    struct { int a, b; }  
    Same as
    struct { int a, b; AND int a; }  
    ```
Equivalence: two approaches

- Two types: name and structural equivalence
- Name Equivalence: based on declarations
  - More commonly used in current practice
  - Strict name equivalence:
    - Types are equivalent if refer to same declaration
  - Loose name equivalence:
    - Types are equivalent if they refer to same outermost constructor
      (refer to same declaration after factoring out any type aliases)
- Structural Equivalence: based on meaning/semantics behind the declarations.
  - Simple comparison of type descriptions
  - Substitute out all names;
  - Expand all the way to built-in types

Data Types

- A data type defines
  - a collection of data objects, and
  - a set of predefined operations on the objects
  
  type: integer
  operations: +, -, *, /, %, ^

- Evolution of Data Types
  - Early days:
    all programming problems had to be modeled using only a few data types
    FORTRAN I (1957) provides INTEGER, REAL, arrays
  - Current practice:
    Users can define abstract data types (representation + operations)

Data Types

- Primitive Types
- Strings
- Records
- Unions
- Arrays
- Associative Arrays
- Sets
- Pointers
Primitive Data Types
- Those not defined in terms of other data types
- Numeric types
  - Integer
  - Floating point
  - Decimal
- Boolean types
- Character types

Numeric Types
- Integer
  - There may be as many as eight different integer types in a language (can you name them?)
  - Negative numbers
    - How to implement them in hardware?

Representing Negative Integers

\[ 1 + (-1) = ? \]

1. Ones complement, 8 bits
   - +1 is 0000 0001
   - -1 is 1111 1110
   - If we use natural method of summation we get sum 1111 1111

2. Twos complement, 8 bits
   - +1 is 0000 0001
   - -1 is 1111 1111
   - If we use the natural method we get sum 0000 0000 (and carry 1 which we disregard)
Floating Point

- Floating Point
- Approximate real numbers
- Note: even 0.1 cannot be represented exactly by a finite number of binary digits!
- Loss of accuracy when performing arithmetic operation
- Languages for scientific use support at least two floating-point types; sometimes more
  
  $1.63245 \times 10^5$

- Precision: accuracy of the fractional part
- Range: combination of range of fraction & exponent
- Most machines use IEEE Floating Point Standard 754 format

Floating Point Puzzle

**True or False?**

```
int x = 1;
float f = 0.1;
double d = 0.1;
```

- $x == (int)(float) x$ True
- $x == (int)(double) x$ True
- $f == (float)(double) f$ True
- $d == (float) d$ False
- $f == -(-f)$ True
- $d > f$ False
- $-f > -d$ False
- $f > d$ True
- $-d > -f$ True
- $d == f$ False
- $(d+f)-d == f$ True

Floating Point Representation

**Numerical Form**

$-1^s \times M \cdot 2^E$
- Sign bit $s$ determines whether number is negative or positive
- Significant $M$ normally a fractional value in range $[1.0,2.0)$
- Exponent $E$ weights value by power of two

**Encoding**

<table>
<thead>
<tr>
<th>$s$</th>
<th>$exp$</th>
<th>$frac$</th>
</tr>
</thead>
</table>

- $s$ is sign bit
- $exp$ field encodes $E$
- $frac$ field encodes $M$
Floating Point Representation

- Encoding
  - \[ x \exp \frac{ac}{} \]
  - MSB is sign bit
  - \( \exp \) field encodes \( E \)
  - \( \frac{ac}{} \) field encodes \( M \)
- Sizes
  - Single precision: 8 \( \exp \) bits, 23 \( \frac{ac}{} \) bits
    - 32 bits total
  - Double precision: 11 \( \exp \) bits, 52 \( \frac{ac}{} \) bits
    - 64 bits total
  - Extended precision: 15 \( \exp \) bits, 63 \( \frac{ac}{} \) bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits
    - 1 bit wasted

Decimal Types

- For business applications (\$\$\$) – e.g., COBOL
  - Store a fixed number of decimal digits, with the decimal point at a fixed position in the value
  - Advantage
    - can precisely store decimal values
  - Disadvantages
    - Range of values is restricted because no exponents are allowed
    - Representation in memory is wasteful
      - Representation is called binary coded decimal (BCD)

Boolean Types

- Could be implemented as bits, but often as bytes
- Introduced in ALGOL 60
- Included in most general-purpose languages designed since 1960
- ANSI C (1989)
  - all operands with nonzero values are considered true, and zero is considered false
  - Advantage: readability
Character Types

- Characters are stored in computers as numeric codings
- Traditionally use 8-bit code ASCII, which uses 0 to 127 to code 128 different characters
- ISO 8859-1 also use 8-bit character code, but allows 256 different characters
- Used by Ada
- 16-bit character set named Unicode
  - Includes Cyrillic alphabet used in Serbia, and Thai digits
  - First 128 characters are identical to ASCII
  - Used by Java and C#

Character String Types

- Values consist of sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of character array?
  - Is the length of objects static or dynamic?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
- Examples:
  - Pascal
    - Not primitive; assignment and comparison only
  - Fortran 90
    - Somewhat primitive; operations include assignment, comparison, catenation, substring reference, and pattern matching

Character Strings

- Examples
  - Ada
    - N := N1 & N2 (catenation)
    - N(2..4) (substring reference)
  - C and C++
    - Not primitive; use char arrays and a library of functions that provide operations
  - SNOBOL4 (a string manipulation language)
    - Primitive; many operations, including elaborate pattern matching
  - Perl and JavaScript
    - Patterns are defined in terms of regular expressions; a very powerful facility
  - Java
    - String class (not arrays of char); Objects are immutable
    - StringBuffer is a class for changeable string objects
Character Strings

- String Length
  - Static – FORTRAN 77, Ada, COBOL
    - e.g. (FORTRAN 90) CHARACTER (LEN = 15) NAME;
  - Limited Dynamic Length – C and C++
    - actual length is indicated by a null character
  - Dynamic – SNOBOL4, Perl, JavaScript

- Evaluation (of character string types)
  - Aid to wrtability
  - As a primitive type with static length, they are inexpensive to provide
  - Dynamic length is nice, but is it worth the expense?

- Implementation

---

Ordinal Data Types

- Range of possible values can be easily associated with the set of positive integers

- Enumeration types
  - User enumerates all the possible values, which are symbolic constants
    - enum days (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  - Design Issue:
    - Should a symbolic constant be allowed to be in more than one type definition?
  - Type checking
    - Are enumerated types coerced to integer?
    - Are any other types coerced to an enumerated type?

---

Enumeration Data Types

- Examples
  - Pascal
    - cannot reuse constants; can be used for array subscripts, for variables, case selectors; can be compared
  - Ada
    - constants can be reused (overloaded literals); disambiguate with context or type_name'(one of them) (e.g., Integer'Last)
  - C and C++
    - enumeration values are coerced into integers when put in integer context
  - Java
    - does not include an enumeration type, but provides the Enumeration interface
    - can implement them as classes
class colors {
  public final int red = 0;
  public final int blue = 1;
}
Subrange Data Types

- An ordered contiguous subsequence of an ordinal type
- e.g., 12..14 is a subrange of integer type
- Design Issue: How can they be used?
- Examples:
  - Pascal
    - subrange types behave as their parent types;
    - can be used as for variables and array indices
      type pos = 0..MAXINT;
  - Ada
    - Subtypes are not new types, just constrained existing types (so
      they are compatible); can be used as in Pascal, plus case
      constants
      subtype POS_TYPE is INTEGER range 0..INTEGER'LAST;
- Evaluation
  - Aid to readability - restricted ranges add error detection

Implementation of Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are the parent types with code
  inserted (by the compiler) to restrict assignments to
  subrange variables

Arrays

- An aggregate of homogeneous data elements in which an
  individual element is identified by its position in the
  aggregate, relative to the first element
- Design Issues:
  - What types are legal for subscripts?
  - Are subscripting expressions in element references
    range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?
  - Are any kind of slices allowed?
Arrays

- Indexing is a mapping from indices to elements
  - map(array_name, index_value_list) ↝ an element

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses: A(3)
  - most other languages use brackets: A[3]

- Subscript Types:
  - FORTRAN, C - integer only
  - Pascal - any ordinal type (integer, boolean, char, enum)
  - Ada - integer or enum (includes boolean and char)
  - Java - integer types only

Five Categories of Arrays
(based on subscript binding and binding to storage)
- Static
- Fixed stack dynamic
- Stack dynamic
- Fixed Heap dynamic
- Heap dynamic

Static
- range of subscripts and storage bindings are static
- e.g. FORTRAN 77, some arrays in Ada
- Arrays declared in C and C++ functions that include the static modifier are static
- Advantage: execution efficiency (no allocation or deallocation)

Fixed stack dynamic
- range of subscripts is statically bound, but storage is bound at elaboration time
- Elaboration time: when execution reaches the code to which the declaration is attached
- Most Java locals, and C locals that are not static
- Advantage: space efficiency
Arrays

- Stack-dynamic
  - Range and storage are dynamic, but fixed from then on for the variable's lifetime
  - E.g., Ada declare blocks
    ```
    declare
      STUFF : array (1..N) of FLOAT;
    begin
      ...
    end;
    ```
  - Advantage: flexibility - size need not be known until array is about to be used

- Fixed Heap dynamic
  - Binding of subscript ranges and storage are dynamic, but are both fixed after storage is allocated
  - Binding done when user program requests them, rather than at elaboration time and storage is allocated on the heap, rather than the stack
  - In Java, all arrays are objects (heap-dynamic)
  - C# also provides fixed heap-dynamic arrays

- Heap-dynamic
  - Subscript range and storage bindings are dynamic and not fixed
  - E.g., (FORTRAN 90)
    ```
    INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT
    (Declares MAT to be a dynamic 2-dim array)
    ALLOCATE (MAT (10, NUMBER_OF_COLS))
    (Allocates MAT to have 10 rows and NUMBER_OF_COLS columns)
    DEALLOCATE MAT
    (Deallocates MAT’s storage)
    ```

  - Perl and JavaScript support heap-dynamic arrays
    - Arrays grow whenever assignments are made to elements beyond the last current element
    - Arrays are shrunk by assigning them to empty array
      ```
      Perl: @myArray = ();
      ```
Arrays

- Number of subscripts (dimensions)
  - FORTRAN I allowed up to three
  - FORTRAN 77 allows up to seven
  - Others - no limit

- Array Initialization
  - Usually just a list of values that are put in the array in the order in which the array elements are stored in memory
  - Examples:
    - FORTRAN - uses the DATA statement
      
      ```
      Integer List(3)
      Data List /0, 5, 5/
      ```
    
    - C and C++ - put the values in braces; let compiler count them
      ```
      int stuff [] = {2, 4, 6, 8};
      ```
    
    - Ada - positions for the values can be specified
      ```
      SCORE : array (1..14, 1..2) :=
      (1 => (24, 10), 2 => (10, 7),
      3 => (12, 30), others => (0, 0));
      ```
    
    - Pascal does not allow array initialization

Array Initialization

- Implementation of Arrays
  - Access function maps subscript expressions to an address in the array
  - Single-dimensioned array
    ```
    address(list[k]) = address(list[lower_bound]) + (k-1)*element_size
    ```
  
  - Multi-dimensional arrays
    ```
    row major order: 3, 4, 7, 6, 2, 5, 1, 3, 8
    column major order 3, 6, 1, 4, 2, 3, 7, 5, 8
    ```

Arrays: Operations

- Ada
  - Assignment; RHS can be an aggregate constant or an array name
  - Concatenation between single-dimensioned arrays

- FORTRAN 95
  - Includes a number of array operations called elementals because they are operations between pairs of array elements
    - E.g., add (+) operator between two arrays results in an array of the sums of element pairs of the two arrays

- Slices
  - A slice is some substructure of an array

- FORTRAN 90
  - INTEGER MAT (1 : 4, 1 : 4)
  - MAT(1 : 4, 1) - the first column
  - MAT(1 : 4) - the second row

- Ada - single-dimensional arrays only
  ```
  LIST(1..10)
  ```

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Single-dimensioned array
  ```
  address(list[k]) = address(list[lower_bound]) + (k-1)*element_size
  ```
  ```
  row major order: 3, 4, 7, 6, 2, 5, 1, 3, 8
  column major order 3, 6, 1, 4, 2, 3, 7, 5, 8
  ```
Associative Arrays

- An unordered collection of data elements that are indexed by an equal number of values called keys
  - also known as hashes

- Design Issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?

Structure and Operations in Perl

- Names begin with %
- Literals are delimited by parentheses
- `%hi_temps = ("Monday" => 77, "Tuesday" => 79,...);`
- Subscripting is done using braces and keys
  - e.g., `$hi_temps("Wednesday") = 83;`
- Elements can be removed with delete
  - e.g., delete $hi_temps("Tuesday");

Records

- A (possibly heterogeneous) aggregate of data elements in which the individual elements are identified by names

- Design Issues:
  - What is the form of references?
  - What unit operations are defined?
Record Definition Syntax

- COBOL uses level numbers to show nested records; others use recursive definitions
- COBOL

```plaintext
01 EMPLOYEE-RECORD.
  02 EMPLOYEE-NAME.
    05 FIRST PICTURE IS X(20);
    05 MIDDLE PICTURE IS X(10);
    05 LAST PICTURE IS X(20).
  02 HOURLY-RATE PICTURE IS 99V99.
```

Level numbers (01, 02, 05) indicate their relative values in the hierarchical structure of the record
PICTURE clause show the formats of the field storage locations
X(20): 20 alphanumeric characters
99V99: four decimal digits with decimal point in the middle

Records

Ada:

```plaintext
Type Employee_Name_Type is record
  First: String (1..20);
  Middle: String (1..10);
  Last: String (1..20);
end record;
type Employee_Record_Type is record
  Employee_Name: Employee_Name_Type;
  Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

References to Record Fields

- COBOL field references

```plaintext
field_name OF record_name_1 OF … OF record_name_n
```
- e.g., MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE_RECORD

- Fully qualified references must include all intermediate record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous
- e.g., the following are equivalent:
  ```plaintext
  FIRST, FIRST OF EMPLOYEE-NAME, FIRST OF EMPLOYEE-RECORD
  ```
**Records**

- Operations
  - Assignment
    - Pascal, Ada, and C allow it if the types are identical
    - In Ada, the RHS can be an aggregate constant
  - Initialization
    - Allowed in Ada, using an aggregate constant
  - Comparison
    - In Ada, = and /=; one operand can be an aggregate constant
  - MOVE CORRESPONDING
    - In COBOL - it moves all fields in the source record to fields with the same names in the destination record

**Comparing Records to Arrays**

- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

**Unions**

- A type whose variables are allowed to store different type values at different times during execution
- Design Issues for unions:
  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?
- Examples:
  - FORTRAN - with EQUIVALENCE
    - No type checking
  - Pascal
    - both discriminated and nondiscriminated unions
      
      ```pascal
      type intreal = record
tag : Boolean of 
true : (blint : integer);
false : (breal : real);
end;
      ```

      Problem with Pascal's design: type checking is ineffective
Unions

- Example (Pascal)...
  - Reasons why Pascal’s unions cannot be type checked effectively:
    - User can create inconsistent unions
      (because the tag can be individually assigned)
      ```pascal
      var blurb : intreal;
      x : real;
      blurb.tagg := true;   { it is an integer }
      blurb.blint := 47;    { ok }
      blurb.tagg := false;  { it is a real }
      x := blurb.blreal;    { assigns an integer to a real }
      ```
      - The tag is optional!
    - Now, only the declaration and the second and last assignments
      are required to cause trouble

- Ada
  - discriminated unions
  - Reasons they are safer than Pascal:
    - Tag must be present
      - It is impossible for the user to create an inconsistent union (because
        tag cannot be assigned by itself – All assignments to the union
        must include the tag value, because they are aggregate values)
  - C and C++
    - free unions (no tags)
    - Not part of their records
    - No type checking of references
  - Java has neither records nor unions

- Evaluation - potentially unsafe in most languages (not Ada)

Sets

- A type whose variables can store unordered collections of distinct values from some ordinal type
- Design Issue:
  - What is the maximum number of elements in any set base type?
- Example
  - Pascal
    - No maximum size in the language definition
      (not portable, poor writability if max is too small)
    - Operations: in, union (+), intersection (\), difference (-), =, <>,
      superset (=>), subset (<=)
  - Ada
    - does not include sets, but defines in as set membership operator
      for all enumeration types
  - Java
    - includes a class for set operations
Sets

Evaluation
- If a language does not have sets, they must be simulated, either with enumerated types or with arrays.
- Arrays are more flexible than sets, but have much slower set operations.

Implementation
- Usually stored as bit strings and use logical operations for the set operations.

Pointers

A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null).

Uses:
- Addressing flexibility
- Dynamic storage management

Design Issues:
- What is the scope and lifetime of pointer variables?
- What is the lifetime of heap-dynamic variables?
- Are pointers restricted to pointing at a particular type?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should a language support pointer types, reference types, or both?

Fundamental Pointer Operations:
- Assignment of an address to a pointer
- References (explicit versus implicit dereferencing)

Problems with pointers:
- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
  - Creating one (with explicit deallocation):
    - Allocate a heap-dynamic variable and set a pointer to point at it
    - Set a second pointer to the value of the first pointer
  - Dealocate the heap-dynamic variable, using the first pointer
- Lost Heap-Dynamic Variables (wasteful)
  - A heap-dynamic variable that is no longer referenced by any program pointer
  - Creating one:
    - Pointer p1 is set to point to a newly created heap-dynamic variable
    - p1 is later set to point to another newly created heap-dynamic variable
- The process of losing heap-dynamic variables is called memory leakage

Fundamental Pointer Operations:
- Assignment of an address to a pointer
- References (explicit versus implicit dereferencing)
Pointers

- Examples:
  - Pascal
    - used for dynamic storage management only
    - Explicit dereferencing (postfix `^`)
    - Dangling pointers are possible (dispose)
    - Dangling objects are also possible
  - Ada
    - a little better than Pascal
    - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
    - All pointers are initialized to null
    - Similar dangling object problem (but rarely happens, because explicit deallocation is rarely done)

Pointers

- Examples...
  - C and C++
    - Used for dynamic storage management and addressing
    - Explicit dereferencing and address-of operator
    - Can do address arithmetic in restricted forms
    - Domain type need not be fixed (void ^)
    - float stuff[100];
      float *p;
      p = stuff;
    - *(p+i) is equivalent to stuff[i] and p[i]
    - (Implicit scaling)

Pointers

- Examples...
  - FORTRAN 90 Pointers
    - Can point to heap and non-heap variables
    - Implicit dereferencing
    - Pointers can only point to variables that have the TARGET attribute
    - The TARGET attribute is assigned in the declaration, as in:
      INTEGER, TARGET :: NODE
    - A special assignment operator is used for non-dereferenced references
      REAL, POINTER :: ptr (POINTER is an attribute)
      ptr = target (where target is either a pointer or a non-pointer with the TARGET attribute)
    - This sets ptr to have the same value as target
Pointers

- Examples...
  - C++ Reference Types
    - Constant pointers that are implicitly dereferenced
    - Used for parameters
    - Advantages of both pass-by-reference and pass-by-value
  - Java
    - Only references
    - No pointer arithmetic
    - Can only point at objects (which are all on the heap)
    - No explicit deallocator (garbage collection is used)
    - Means there can be no dangling references
    - Dereferencing is always implicit

- Evaluation
  - Dangling pointers and dangling objects are problems, as is heap management
  - Pointers are like goto's--they widen the range of cells that can be accessed by a variable
  - Pointers or references are necessary for dynamic data structures--so we can't design a language without them