Theory of Programming Languages

Week 6

Data Types

Evolution of Data Types

FORTRAN I (1956) - `INTEGER`, `REAL`, arrays
...
Ada (1983) - User can create a unique type for every category of variables in the problem space and have the system enforce the types

Design Issues

1. **What is the syntax of references to variables?**
2. **What operations are defined and how are they specified?**

Primitive Data Types

(those not defined in terms of other data types)

- ***Integer*** **-Almost always an exact reflection of the hardware, so the mapping is trivial**
- **There may be as many as eight different integer types in a language**

Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada)

```
type SPEED is digits 7 range 0.0..1000.0;
type VOLTAGE is delta 0.1 range -12.0..24.0;
```

Decimal

- For business applications (money)
- Store a fixed number of decimal digits (coded)
- *Advantage:* accuracy
- *Disadvantages:* limited range, wastes memory

Boolean

- Could be implemented as bits, but often as bytes
- *Advantage:* readability

Character String Types

- Values are sequences of characters

Design issues:

1. Is it a primitive type or just a special kind of array?
2. Is the length of objects static or dynamic?

Operations:

- Assignment
- Comparison (=, >, etc.)
- Concatenation
- Substring reference
- Pattern matching

Examples:

- *Pascal*
  - Not primitive; assignment and comparison only (of packed arrays)

```
Name: Array [1..5] of Char;
Name: PACKED Array [1..5] of Char;
```

- Ada, FORTRAN 77, FORTRAN 90 and BASIC
  - Somewhat primitive
  - Assignment, comparison, catenation, substring reference

e.g. (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

- Java - `String` class (not arrays of `char`)

String Length Options:

1. *Static* - FORTRAN 77, Ada, COBOL
   e.g. (FORTRAN 90)
   `CHARACTER (LEN = 15) NAME;`

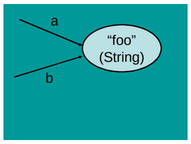
2. *Limited Dynamic Length* - C and C++ actual length is indicated by a null character

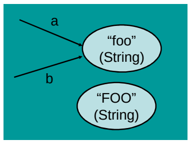
3. *Dynamic* - SNOBOL4

– **Java Strings**

– what does this do?

```
String a = “foo”;
String b = a;
a.toUpperCase ();
System.out.println (b);
```





- Now What?

```
String a = “foo”;
String b = a;
a = “FOO”;
```

Implementation:

- Static length - compile-time descriptor
- Dynamic length - need run-time descriptor; allocation/deallocation is the biggest implementation problem
- Limited dynamic length

Static String
Length
Address

Dynamic String
Length
Address

Limited Dynamic String
Maximum Length
Current Length
Address

***Evaluation* (of character string types):**

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

Ordinal Types (user defined)

An *ordinal type* is one in which the range of possible values can be easily associated with the set of positive integers

1. **Enumeration Types - one in which the user enumerates all of the possible values, which are symbolic constants**

***Design Issue:* Should a symbolic constant be allowed to be in more than one type definition?**

Enumeration Type

Examples:

Pascal

- cannot reuse constants;
- they can be used for array subscripts, `for` variables, `case` selectors;
- NO input or output;
- can be compared

Ada

- constants can be reused (overloaded literals);
- disambiguate with context or type_name ‘ (one of them);
- can be used as in Pascal;
- CAN be input and output

C and C++

- like Pascal, except they can be input and output as integers

Java does not include an enumeration type

Evaluation (of enumeration types):

a. Aid to readability
-- e.g. no need to code a color as a number

b. Aid to reliability
-- e.g. compiler can check operations and ranges of values

2. *Subrange Type*

- an ordered contiguous subsequence of an ordinal type

Pascal

- Subrange types behave as their parent types;

can be used as `for` variables and array indices

e.g. `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

e.g.

```
subtype POS_TYPE is
     INTEGER range 0 ..INTEGER'LAST;
```

Implementation of user-defined 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 *array* is 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:

1. What types are legal for subscripts?
2. Are subscripting expressions in element references range checked?
3. When are subscript ranges bound?
4. When does allocation take place?
5. What is the maximum number of subscripts?
6. Can array objects be initialized?
7. Are any kind of slices allowed?

Arrays

Indexing is a mapping from indices to elements

map(array_name, index_value_list) → an element

Syntax
- FORTRAN, PL/I, Ada use parentheses
- Most others use brackets

Subscript Types:

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

Four Categories of Arrays

(based on subscript binding and binding to storage)

1. *Static* - range of subscripts and storage bindings are static
   e.g. FORTRAN 77, some arrays in Ada

   Advantage: execution efficiency (no allocation or deallocation)

2. *Fixed stack dynamic* - range of subscripts is statically bound, but storage is bound at elaboration time
   e.g. Pascal locals and, C locals that are not `static`

   Advantage: space efficiency

3. *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 the array is about to be used

4. ***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)

- In APL & Perl, arrays grow and shrink as needed
- In Java, all arrays are objects (heap-dynamic)

Number of subscripts

- FORTRAN I allowed up to three
- FORTRAN 77 allows up to seven
- C, C++, and Java allow just one
- 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:

1. FORTRAN - uses the `DATA` statement, or put the values in `/.../` on the declaration

2. C and C++ - put the values in braces; can let the compiler count them e.g. `int stuff [] = {2, 4, 6, 8};`

3. Ada - positions for the values can be specified e.g.
```
SCORE : array (1..14, 1..2) :=
    (1 => (24, 10), 2 => (10, 7),
    3 =>(12, 30), others => (0, 0));
```

4. Pascal and Modula-2 do not allow array initialization

Array Operations

1. Ada
- assignment; RHS can be an aggregate constant or an array name

- concatenation; for all single-dimensioned arrays

- relational operators (= and `/=` only)

2. FORTRAN 90
- intrinsics (subprograms) for a wide variety of array operations (e.g., matrix multiplication, vector dot product)

Slices

- A slice is some substructure of an array
- nothing more than a referencing mechanism

Slice Examples:

1. FORTRAN 90

```
INTEGER MAT (1 : 4, 1 : 4)
    MAT(1 : 4, 1) - the first column
    MAT(2, 1 : 4) - the second row
```

More complex versions in FORTRAN95

2. Ada - single-dimensioned arrays only

```
LIST(4..10)
```

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Row major or column major order

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*

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

Records

A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names

Design Issues:
1. What is the form of references?
2. What unit operations are defined?

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

Record Field References

1. COBOL
   field_name OF record_name_1 OF ... OF record_name_n

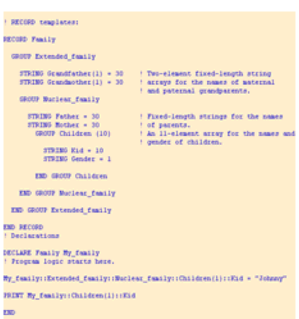
2. Others (dot notation)
   record_name_1.record_name_2. ... .record_name_n.field_name

Fully qualified references must include all record names

Elliptical references allow leaving out record names as long as the reference is unambiguous

Pascal and Modula-2 provide a `with` clause to abbreviate references

Elliptical Reference Example BASIC Records



```
! RECORD templates:

RECORD Family

  GROUP Extended_family

    STRING Grandfather(1) = 30    ! Two-element fixed-length string
    STRING Grandmother(1) = 30    ! arrays for the names of maternal
                                  ! and paternal grandparents.
    GROUP Nuclear_family

      STRING Father = 30          ! Fixed-length strings for the names
      STRING Mother = 30          ! of parents.
        GROUP Children (10)       ! An 11-element array for the names and
                                  ! gender of children.
          STRING Kid = 10
          STRING Gender = 1

        END GROUP Children

    END GROUP Nuclear_family

  END GROUP Extended_family

END RECORD
! Declarations

DECLARE Family My_family
! Program logic starts here.

My_family::Extended_family::Nuclear_family::Children(1)::Kid = "Johnny"

PRINT My_family::Children(1)::Kid

END
```

Record Operations

1. Assignment
   Pascal, Ada, and C allow it if the types are identical. In Ada, the RHS can be an aggregate constant

2. Initialization
   - Allowed in Ada, using an aggregate constant (basically a constant data vector)

3. Comparison
   - In Ada, = and /=; one operand can be an aggregate constant

4. `MOVE CORRESPONDING`
   - In COBOL - it moves all fields in the source record to fields with the same names in the destination record

Comparing records and arrays

1. Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
2. Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower
   Why can't we index structs?

Sets

A *set* is 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?

Examples (continued)

1. Pascal
   - No maximum size in the language definition (not portable, poor writability if max is too small)
   - Operations: union (+), intersection (*), difference (-), =, <>, superset (>=), subset (<=), in
2. Ada - does not include sets, but defines `in` as set membership operator for all enumeration types
3. Java includes a class for set operations
4. C++ provides an STL set class, with dynamic 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 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:

1. Addressing flexibility
2. Dynamic storage management

Fundamental Pointer Operations:

1. Assignment of an address to a pointer
2. References (explicit versus implicit dereferencing)

Design Issues:

1. What is the scope and lifetime of pointer variables?
2. What is the lifetime of heap-dynamic variables?
3. Are pointers restricted to pointing at a particular type?
4. Are pointers used for dynamic storage management, indirect addressing, or both?
5. Should a language support pointer types, reference types, or both?
6. Pointer arithmetic?

Evaluation of pointers:

1. Dangling pointers and dangling objects are problems, as is heap management
2. Pointers are like goto's--they widen the range of cells that can be accessed by a variable
3. Pointers are necessary--so we can't design a language without them

Unions

A *union* is a type whose variables are allowed to store different type values at different times during execution

Design Issues for unions:

1. What kind of type checking, if any, must be done?
2. Should unions be integrated with records?

Examples:

1. FORTRAN - with `EQUIVALENCE`

2. Algol 68 - discriminated unions
   - Use a hidden tag to maintain the current type
   - Tag is implicitly set by assignment
   - References are legal only in conformity clauses
   - This runtime type selection is a safe method of accessing union objects

Examples:

3. Pascal
   - both discriminated and non-discriminated unions

   e.g.

```
type intreal = record tag : Boolean of
      true  : (blint : integer);
      false : (blreal : real);
end;
```

Problem with Pascal's design: type checking is ineffective

Reasons:

a. User can create inconsistent unions (because the tag can be individually assigned)

```
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 }
```

b. The tag is optional!

4. Ada - discriminated unions

   - Reasons they are safer than Pascal & Modula-2:
     a. Tag <u>must</u> be present

     b. It is impossible for the user to create an inconsistent union (because tag cannot be assigned by itself
        - <u>All</u> assignments to the union <u>must</u> include the tag value)

5. C and C++ - free unions (no tags)
   - Not part of their records
   - No type checking of references

6. Java has neither records nor unions

Evaluation - potentially unsafe in most languages (not Ada)

Programming Languages

Arithmetic Expressions

Arithmetic Expressions

- Their evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Design issues for arithmetic expressions:

1. What are the operator precedence rules?
2. What are the operator associativity rules?
3. What is the order of operand evaluation?
4. Are there restrictions on operand evaluation side effects?
5. Does the language allow user-defined operator overloading?
6. What mode mixing is allowed in expressions?

Conditional Expressions

- C, C++, and Java (`?:`)
e.g.

```
average = (count == 0)? 0 : sum / count;
```

Operand evaluation order

1. *Variables:* just fetch the value

2. *Constants:* sometimes a fetch from memory; sometimes the constant is in the machine language instruction

3. *Parenthesized expressions:* evaluate all operands and operators first

4. *Function references:* The case of most interest!

- Order of evaluation is crucial

- If there are no side effects, the operand evaluation order is irrelevant.

5. Associativity

***Functional side effects* - when a function changes a two-way parameter or a nonlocal variable**

The problem with functional side effects:

- When a function referenced in an expression alters another operand of the expression e.g., for a parameter change:

```
int fun(int &x)
{
   x = x + 1;
   return x*2;
}

a = 10;
b = a + fun(a);
```

Two Possible Solutions to the Problem:

1. Write the language definition to disallow functional side effects

- No two-way parameters in functions
- No non-local references in functions

- *Advantage:* it works!
- *Disadvantage:* Programmers want the flexibility of two-way parameters and non-local references

2. Write the language definition to demand that operand evaluation order be fixed

- *Disadvantage*: limits some compiler optimizations

Operator Overloading

- Some are common (e.g., `+` for `int` and `float`)
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)

Operator Overloading

- C++ and Ada allow user-defined overloaded operators

Potential problems?

- Users can define nonsense operations
- Readability may suffer

Implicit Type Conversions

Def: A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type

Def: A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type

Def: A *mixed-mode* expression is one that has operands of different types

Def: A *coercion* is an implicit type conversion

- *The disadvantage of coercions:*
  - They decrease in the type error detection ability of the compiler
  - In most languages, all numeric types are coerced in expressions, using widening conversions
  - In Modula-2 and Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Often called *casts*

e.g.
Ada:
```
FLOAT(INDEX)  -- INDEX is INTEGER type
```
C:
```
(int)speed    /* speed is float type */
```

Errors in Expressions

- Caused by:
  - Inherent limitations of arithmetic
    e.g. division by zero
  - Limitations of computer arithmetic
    e.g. overflow
- Such errors are often ignored by the run-time system

Relational Expressions

- Use relational operators and operands of various types
- Evaluate to some boolean representation
- Operator symbols used vary somewhat among languages (`!=`, `/=`, `.NE.`, `<>`, `#`)

Boolean Expressions

- Operands are boolean and the result is boolean
- *Operators:*

FORTRAN 77	*FORTRAN 90*	*C*	*Ada*
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`
			`xor`

Boolean Expressions

- One odd characteristic of C's expressions:

```
a < b < c
```

Short Circuit Evaluation

Pascal: does not use short-circuit evaluation
Problem: table look-up

```
index := 1;
  while (index <= length) and
         (LIST[index] <> value) do
    index := index + 1
```

C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (`&&` and `||`), but also provide bitwise Boolean operators that are not short circuit (`&` and `|`)

Ada: programmer can specify either (short-circuit is specified with `and then` and `or else`)

FORTRAN 77: short circuit, but any side-affected place must be set to undefined

Short-circuit evaluation exposes the potential problem of side effects in expressions
e.g. `(a > b) || (b++ / 3)`

Assignment Statements

The operator symbol:

1. `=` FORTRAN, BASIC, PL/I, C, C++, Java
2. `:=` ALGOLs, Pascal, Modula-2, Ada

`=` can be bad if it is overloaded for the relational operator for equality

e.g. (PL/I) `A = B = C;`

Note difference from C

More complicated assignments:

1. Multiple targets **(PL/I)**

```
A, B = 10
```

2. Conditional targets **(C, C++, and Java)**

```
(first = true) ? total : subtotal = 0
```

3. Compound assignment operators **(C, C++, and Java)**

```
sum += next;
```

4. Unary assignment operators **(C, C++, and Java)**

```
a++;
```

Assignment as an Expression

C, C++, and Java treat = as an arithmetic binary operator

e.g.

```
a = b * (c = d * 2 + 1) + 1
```

This is inherited from ALGOL 68

- So, they can be used as operands in expressions

e.g. `while ((ch = getchar() != EOF) { ... }`

Disadvantage
- Another kind of expression side effect

Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done

- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded)

- In Java, only widening assignment coercions are done

- In Ada, there is no assignment coercion

Summary

- Expressions consist of constants, variables, parentheses, function calls, and operators.
- Assignment statements include target variables, assignments, operators, and expressions.
- The semantic of and expression is determined in large part by the order of evaluation of operators – which is determined by the associativity and precedence rules.
- Operand evaluation order in important if function side effects are possible.
- Type conversions can be widening or narrowing.
- Some narrowing conversions produce erroneous results.
- Coercion in expression is common but reduces error detection and hence reduce reliability.