Chapter 2

Types

What is computation? Computation can be viewed as a transformation of input values to output values.

```
input -> output
```

Computation is *operations* on a *collection* of values. Not all operations are meaningful on all values. So it makes sense to group values which share a collection of operations. Such groups are called types. So a type is a collection of values and the associated operations.

Consider, for example, the values "Jan 01, 2000", "Aug 01, 1999", etc. What distinguishes these values from aribirary values like "PL", "3.14149", etc. is that the following operations are meaningful on these values: equality, next value, prev value, difference of values, etc. Note that addition of two such values is meaningless, whereas an integer can be added to such values. So a collection of such values and the associated operations constitutes the type *date*.

Types used in a program depend on the *application*. What should a PL provide?

primitive types: These types are built into the PL itself. Applications can use these types to build their own types. Since these are application independent, the primitive types stay close to the machine.

facility for definition of types: Most of the PLs provide a notation for the users to define their own types.

2.1 Primitive types

What is a primitive type depends on the language. In some languages, arrays are considered "primitive". Most PLs provide the following primitive types: integer, real, boolean, and character. Sounds simple? Not really!

In most languages, the sizes of reals and ints are left to the implementor. So machine dependence starts early! In JAVA, the sizes are specified and are not implementation dependent.

Some languages make the implementation details available to the programmer. For example, Ada has predefined constants `MAX_INT` and `MIN_INT`. The programmer can test for the range of availble values using these constants. Ada also has a facility to define number of digits and precision.

```
type COEF is digits 10 range -1.0 .. 1.0;
type HUNDREDTHS is delta 0.01 range 0.0 .. 1.0;
```

The result of the first type definition is that `COEF` has 10 digit precision. The second says that `HUNDREDTHS` has 100 values 0.00, 0.01, 0.02, $\cdots$, 1.0.

User can also get other information for a ordered numeric type `T` using `T'DIGITS, T'SMALL, T'LARGE`.

Some languges provide multiple numerical types:

```
byte, short, int, long
float, double
```

Why? Appropriate use of these types results in memory savings. The programmer uses the knowledge of the problem domain to decide on the types of variables used.

A comparison of the operations on integers and reals is interesting. Of the arithmetic operations, `+,-,*,/`, there are two types of division. Integer division results in an integer value and real division results in a real value. Some languages have two different notations. Pascal, for example, has `div` for integer division and the usual `/` for real division. C has just one operator and the meaning is context dependent.

```
Pascal: 3/2 = 1.5, 3 div 2 = 1
C:      3/2 = 1.5, (float) 3/2 = 1.5
```

Most of the new implementations of floating point libraries use the IEEE notation which can represent infinity, as shown below.

```
0/0 = NaN    // not a number (IEEE floating point arithmetic)
1/0 = Inf    // infinity (IEEE floating point arithmetic)
```

Types tend to get mixed during computation. For example, it is natural to multiply an int and a real. What is the type of the result? What we expect that the result has the type real. Note that the sizes of the types obey the following relation

```
|short| < |int| < |long|
|int| < |real| < |double|
```

The compiler automatically decides the type of such mixed expressions to be the most general or *widest* type.

A more tricky situation arises when the programmer makes an assignment in the other way.

```
int x; float y;

x =  y; // should this be allowed?
```

This can be intentional or an oversight. Most of the languages require that the programmer assert his/her intention by explicitly *casting* the variables involved as shown below.

```
x =  (int) y; // the programmer takes responsibility
```

Another interesting operation is test for equality. Two numbers can be tested if they are equal. Can we test real numbers for equality? By definition, a real number consists of an infinite number of digits. The finite representation is only an approximation. So it is not correct to test real numbers for equality. ML takes such a position.

Java 1.1 has types (read "classes") `BigInteger` and `BigDecimal` that can do arbitrary precision arithmetic. Are these classes useful? These classes are useful for performing cryptographic operations, for example.

2.2 Boolean

The values are `true` and `false`. The basic operations are `and`, `or`, and `not`. (Other operations can be built using these.) In Algol, Pascal, and Ada booleans can't be mixed with other types. In Pascal and Ada `false < true`. C doesn't have an explicit type boolean. Anything that evaluates to 0 is false and everything else is true.

Most of the languages take an optimized approach for evaluating boolean expressions. Suppose `A` and `B` are boolean variables. When evaluating `A and B`, if `A` evaluates to `false`, we need not evaluate `B` and declare the value of the expression to `false`. (A similar statement hold for `A or B` and when `A` evaluates to `true`.) This is called *short circuit evaluation.* ML makes this explicit by naming the operators as `andalso` and `orelse`.

Perl programmers make heavy and obvious use of this feature in the following Perl idiom.

```
open(file, HANDLE) || die "can't open file";
```

2.3 Characters

The underlying character set of most languages used to be ASCII. Java has the character set as Unicode. Unicode is a 16-bit character set and has codes for all the languages of the world.

It is often useful to know the position of a given character in the character set and also the character corresponding to a given position. Pascal has two functions `int ord(char)` and `char chr(int)` for this. In C characters and integers are distinguished only based on the context.

A string is a sequence of characters. The evolution of the status of strings in PLs is interesting. In early languages, strings were of fixed length and can be used in only certain contexts (eg, printing). Why?

1. There was no need for strings except as descriptions in print outs.
2. Handling dynamic sized strings requires dynamic memory allocation. Dynamic memory allocation was considered expensive.

As the needs changed and the technologies matured, the following features are available in most PLs.

1. Arbitrary size strings
2. Relational operators on strings
3. substring testing, string concatenation are usually handled by libraries.

2.4 Enumerated data types

Enumerated data type is an example of user defined data type. Suppose that the user wants to use a collection of values, say names of colors or names of days of the week. These values can be declared as as values of an enumerated data type as

```
type days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Note that the values are "literals" and not strings. What are the operations that are permitted on this type? Equality and assignment are the natural operations on this type. Some languages, provide additional functionality. Pascal, for example, provides ordering. In the above example,

```
Mon < Tue < Wed < Thu < Fri < Sat < Sun
```

Pascal also provides predecessor and successor functions. For example, `pred(Sat) = Fri` and `succ(Thu)=Fri`.

In C, one can assign arbitrary integers as values of these values.

```
enum int16 = {
   sign = 0x8000;
   msb = 0x4000;
   lsb = 0x0001;
}
```

Again, enumerated data types are not without their own issues. How do we treat multiple occurrences of the same symbol? Consider the example (in Ada)

```
type DAYS is (MON, TUE, WED, THU, FRI, SAT, SUN);
type WEEKENDS is (SAT, SUN);
```

Note that the literals `SAT, SUN` occur in both the types. Should it be allowed? In Pascal, this is not allowed. Ada disambiguates the lierals by providing a context. In Ada, we write `DAYS'ord(SUN)` and `WEEKENDS'ord(SUN)` to distinguish the two. This also reflects a general feature of Ada: the use of qualifications (`DAYS'`) for disambiguation.

2.5 Subranges

Some languages allow subranges. For example, in Pascal we can write

```
type weekends = (Sat, Sun);  (* Pascal *)
type years = (1990..2010);
```

This is a useful notation. Such subranges can be used as array indexes, ranges of `for` loops, etc.

The above discussion illustrates that PL specification issues. Can you reason out why the decisions for the languages are different?

At this point, there are two branches. To discuss pointers, which in some sense is primitive or native. Or, to discuss structured data types, which are user constructed data types. To discuss about pointers, we need to really discuss variables and assignment.

2.6 Variables

Consider the following statements.

```
x = 1.0;
y = x;
```

What should the computer/compiler know about the variable `x`?

1. name: `x`
2. attributes: `float`
3. storage location or reference: memory address
4. contents of the storage location: 1.0

Note that in the first statement (3) is important and not (4). But in the second statement both are important.

So (3) is called an *l-value* and (4) *r-value.* (Why?)

We now need new operatiors for

1. Given a variable, how to use its l-value on the right side? More technically, how to obtain its reference? We need to use the reference operator. The reference operator in C is `&`.
2. Once a language provides the reference operator, *pointer variables* are required. Pointer variables hold references as their value.
3. The value of a pointer variable is reference. How to denote the contents of the reference? In other words, how to dereference the reference? The deferenencing operator in C is `*`.

Consider the example.

```
int a=1, b;
int *c; // pointer variable

c = &a; // reference
b = *c; // dereference
```

The advantage of reference variables is that they help sharing of values. Had `a` been a HUGE value, `c` can also refer to the same value without copying. Java uses only references to objects.

The ugly side of sharing is called *aliasing*. Aliasing means that there are multiple references to the same location. Aliasing can lead to unexpected side effects. For example, the statement `*c = 10` changes the value of `a`.

2.7 Declarations

How to introduce names to compilers? Declarations introduce names to compilers.

```
int i;   // declaration, a reference is created
i = 10;  // the reference is bound
```

Constants have a value, but no reference.

```
const float pi = 3.14; // a preprocessor can't introduce a type!
```

There are a host of issues with constants.

1. Can constants be introduced for structured data?
2. Can a language have predefined constants? Pascal has `maxint` and Ada has `MAX_INT`.
3. What kind of initialization is allowed? In Ada expressions which are evaluated at runtime are allowed.
4. At what time is the constant created? Compile time or run time? In Ada constants are created and initialized at runtime.

2.8 Pointers and references

As mentioned before a pointer is a reference to an object and a pointer variable is an identifier whose value is reference to some object. Why are pointers supported in PLs? Pointers provide a notation which facilitates sharing (read aliasing!). Also there are applications in which the number and size of values can be determined only at runtime. For example, the number of characters in a name can be determined only at runtime. Values can be created at runtime and pointers provide a way to refer to those values. Another important use of pointers is in the creation of recursive types (section 2.10.1).

Creating objects at runtime needs special support from the language. Most languages have support for runtime memory allocation and reclamation. C++, for example, has the functions, `new` and `delete` for this. So pointer variable operations follow the sequence

1. declare the variable. `int* a;`
2. allocate storage. `a = new int[10];`
3. reclaim storage. `delete a;`

Because of the possibility of aliasing, the programmer can not be really sure that there are no references to the location before before deleting it. Because of these uncertainties, "safe" programming does not use pointers.

Java does not support pointers. The programmer is responsible for object creation but not reclamation. The system does *garbage collection* – identifying unreferenced storage areas are reclamining the storage. Since GC can kick in any time, performance guarantees are not possible.

Pointers have types. Let `T` be a type and let `p` be a pointer to type `T`. The meaning of `p++` is that `p` is incremented by `sizeof(T)`. This is a good navigational notation.

C++ supports the notion of *reference variable.* A reference is an *alias* for an object. As mentioned before, aliasing is highly undesirable. The main use of references is to define functions that can be used on both left and right side of an assignment. Consider the following example.

```
char greet[] = "hello";

char& charAt(int i) {
    return greet[i];
}

main(void) {
    char c;

    c = charAt(0); // c == 'h'
    charAt(0) = 'H'; // greet == "Hello"
}
```

2.9 Structured data types

The data types discussed so far have direct correspondence to data types available in the hardware. As mentioned before, a PL should allow the programmer to construct values applicable to the problem domain. A more important requirement is that these values should be "first class". We will discuss notion of first-class objects later.

There are two structured data types available in most of the programming languages: arrays and records. An array is an aggregate of homogenous data types; the components are identified by their position. A record is an aggregate of heterogenous data types; the components are identified by name.

2.9.1 Arrays

Consider the following array declarations in Pascal.

```
var a array [1996..2000] of real;
var b array [(Mon..Fri)] of integer;
var c array[char] of token; (* token is a user defined type *)
```

An array is characterized by

name: a, b, c

dimensions: all the above arrays are one dimensional

element type: real, integer, or token

index set: subrange, enumerated, or integer

The notation used for arrays varies between languages.

```
A(1, 2, 3) // Fortran, Ada
A[1, 2, 3] // Algol, Pascal
A[1][2][3] // C, Java
```

Is the difference in notation accidental or is it fundamental? Note that it is easy to confuse `A(1, 2, 3)` for a function call. That's why `A[1, 2, 3]` is more readable. What about the difference between the second and the third? The notation `A[1][2][3]` emphasizes the fact that the array may not be rectangular and the storage may not be contiguous. In this case, `A[1][2]` makes sense.

For example, in Java, one can write

```
int[][] a = new int[2][3];
```

This is equivalent to

```
int[][] a;

a = new int[2][];
a[0] = new int[3];
a[1] = new int[3]; // need not be 3
```

Note that the scheme is more flexible but also takes more time for element access.

There are other issues with arrays.

array bounds specification and allocation: Can the array bounds be specified at run time? Consider the two declarations

```
int a[10];  // allocated at compile time
int[] a = new int[size]; // allocated at runtime
```

Algol allows dynamic (runtime) allocation, but allocation is done on entry into a procedure.

index values: In Pascal, enumerated data types can be used as index values.

```
freeTime: array[days] of integer;

for i := Mon to Sun do println(freeTime[i]);
```

array initialization: Arrays can be initialized as a whole.

```
int coin[] = {1, 5, 10, 25, 100};
```

array storage layout: Array elements can be stored in row major or column major order, if they are stored contiguously.

slices: If `A` is an array of size 10, can we use `A[3:7]` to refer to `A[3], A[4], A[5], A[6],` and `A[7]`? In Perl, the notation is `@A[3..7]` where `@A` is the name of the array. (In Perl, arrays names should start with `@`.)

2.9.2 Records

A record denotes the grouping of a possibly heterogenous collection of types. For example, a book has an author, title, and price. It can be handled as a record as in

```
type book = record
   author: array[1..30] of char;
   title: array[1..50] of char;
   price: real
end

var textBook: book;
```

The record is said to be composed of *fields*. In the above example, `author, title`, and `price` are fields. How to access the fields of the record? There are two notations. The *functional notation* uses the notation `textBook(price) = 24.99` where as the *qualified name* form uses `textBook.price = 24.99`. Algol uses the functional notation (`price of textBook`) where as most of the other languages use the other notation.

The other issue is that of initialization. Can records be initialized in a single statement as in

```
textBook = {"Sethi", "Programming Languages", 24.99};
```

When working with the fields of a record, the record name has to be mentioned repeatedly. Pascal provides a shorthand

```
with textBook do
  begin
    author := "Sethi";
    title := "PL";
    price := 24.99;
  end
```

In Pascal, one can use records in assignment statement. The individual fields are assigned. For example, one can write

```
refBook := textBook; (* fieldwise assignment *)
```

In a record, all values should have all the fields. When grouping information, it sometimes happens that mutually exclusive fields are needed. In other words, the record should have just one of the fields listed. For example, a book might be available in hardcopy or in downloadable form. We need the field, say `location`, for hardcopy and the field `url` for the electronic version. These fields are mutually exclusive. Instead of storing both the fields separately, one can optimize on storage by storing these fields in the same memory area. For example,

```
type medium = (electronic, paper);

type book = record
   author: array[1..30] of char;
   title: array[1..50] of char;
   price: real
   case bookform: medium of  (* called the "tag field" *)
     electronic: (url: char[100])
     paper:      (location: address)  (* address is type record *)
   end
end

var textBook: book;
```

This is called a *variant record*. A *union* is a variant record with an empty common part.

Danger lurks because the tag field is optional. For example, in C, one can write

```
union pun {
   int i;
   float f;
} a;

a.i = 1;
float b = a.f;  // reading int as float
```

Also checking for consistency of tags values and fields should be done at runtime. So it is easy to make mistakes in accessing the fields. Some languages have made the tag fields write once. For example, in Euclid, one can write the following.

```
var x: book(electronic);
var y: book(paper);
var z: book(any);

z := y; // OK
y := z; // not permitted
```

2.10 A little bit of mathematics

The discussion so far has been informal. Let us now formalize the notion of type a bit. This will also motivate to the notation used by ML. So far we have seen primitive types and composite types. A type, as discussed before, is a collection of values and the associated operations. In mathematics, the notion of *set* formalizes a collection. So we can consider primitive types to be sets. We can cosider a composite type to be a *Cartesian product*. The Cartesian product of sets A and B is defined as

$$A \times B = \{< a, b >: a \in A, b \in B\}$$

The Cartesian product corresponds to a *record* or a *tuple*. So the record `book` discussed above corresponds to the product type `string*string*real`.

Other operations on sets are also possible. For example, union is defined as

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

So union corresponds to unions!

Special cases of Cartesian products are interesting. Consider the notion of a function. A function, F, from set A to set B is a subset of the Cartesian product ($F \subseteq A \times B$) such that

For each $a \in A$, there is a unique $b \in B$ such that $< a, b > \in F$.

A is called the domain of the function and B is called the range of the function. This is also denoted by

$$F: \; A \rightarrow B$$

Conceptually the notation $A \rightarrow B$ stands for the set of *all* functions from A to B. Since it is a set, it can be considered a type.

Note that arrays satisfy the definition of a function. The index set corresponds to set A and the base type corresponds to B. For an array, A should be finite. What happens when A is infinite? Such an array cannot be stored. Usually such functions are *computed* in a program. That is why we have the notion of functions in PLs.

Consider the sine function. We can write `y = sin(x)`. `x` can take (potentially) infinite number of values. What is the type of `sin`? Since `sin` maps reals to reals, it is a member of `real->real`.

Since functions are also a type, can they be used in conjunction with other types? For example, what is the meaning of `real*(int->real)`? This is a product type the first component of which is a real number and the second component is a function mapping integers to reals. What is the meaning of `real->(real->int)`? This is a function taking a real value as input and returning a function of the type `real->int`.

How many languages allow such operations? In many languages, operations on functions are restricted and hence functions are said to be not *first-class*. A first-class value is one which is on par with a primitive type. It can be assigned to a variable, passed as a paramter, returned as a result of a function, can be input and output, just like built-in values.

What are the operations permitted on functions? One type of operation natural to functions is *function composition*. Suppose $f : A \rightarrow B$ and $g : B \rightarrow C$, we can define the composition of f and g as

$$g \circ f = g(f(x))$$

What is the type of $g \circ f$? It belongs to the type $A \rightarrow C$.

Another useful idea associated with functions is the notion of *partial instantiation*. Consider the power function $f(x, y) = x^y$. This function takes two arguments. What happens when we supply only one argument? We get a function back!

$$f(x, 2) = x^2$$

This is the square function.

A functional language is one in which functions are first class as well as operations natural to functions are supported.

2.10.1 Recursive types

The types discussed so far have a fixed number of components. It is useful to define types that have an indefinite number of components. Consider the notion of a "list". A list can have any number of elements, including zero elements. How can we describe this? We can consider a list of integers as an infinite union of tuples.

$$list = \varphi \cup integer \cup (integer \times integer) \cup (integer \times integer \times integer) \cdots$$

It is difficult to deal with infinities. So a recursive defintion is used. A recursive defintion of a list of integers is

1. `nil` is an integer list
2. If `L` is an integer list and `a` is an integer, then `a::L` is an integer list. Here `::` is the *constructor* for a list.

In this notation the list `[1, 2, 3, 4]` is written as `1::2::3::4::nil`.

We can write the integer list type as

$$integerlist = integer \cup integer :: integerlist$$

We can write a binary tree of strings as

$$stringtree = string \cup (stringtree, string, stringtree)$$

Can we consider an array as an example of a recursive type?

2.10.2 Function types

Functions can be classified in different ways. A function is said to be *total* on a given set if it returns a value for all inputs from the set. A function is said to be *partial* if the value on some of its inputs is undefined. The classic example of a partial function is division on input 0. (It is now made total by the addition of `Inf`.) How can we represent partial functions in PLs? The notion of exceptions is used.

From the point of view of PLs, there is another classification: pure and impure functions. A pure function just returns a value. It does not change the state of computation in any other way. A function can be impure because it has side effects. Side effects include desirable ones (printing, for example) or undesirable ones (changing the value of a global variable).

2.11 Back to PLs

How can we implement the recursive expression

```
(string, stringtree, string)
```

in a programming language? There are two different types of implementations: explicit and implicit. Pascal, C, and related languages need explicit implementations. In this, the programmer explicitly constructs the representation. Languages like Lisp, ML, etc. provide language support for such representations. So handling such types is natural and less error prone.

Consider the C implementation of a string tree.

```
struct stringTree {
     struct stringTree *left;
     char data[30];
     struct stringTree *right;
};
```

This is not a completely faithful representation (Why?). But the idea is that the programmer has to keep track of pointers (allocation, assignment, etc.). Languages like Lisp and ML automatically do this for the programmer.