CENG 242 - Chapter 2: Values and Types

Burak Metehan Tunçel - March 2022

1 Types

A ***value*** is any entity that can be manipulated by a program. Values can be *evaluated, stored, passed* as *arguments, returned as function results*, and so on. Different programming languages support different types of values: `C` supports integers, real numbers, structures, arrays, unions, pointers to variables, and pointers to functions; `C++` supports all the above types of values plus objects.

Most programming languages group values into *types*. For instance, nearly all languages make a clear distinction between integer and real numbers. Most languages also make a clear distinction between booleans and integers: integers can be added and multiplied, while booleans can be subjected to operations like not, and, and or.

- **C Types:**
  - int, char, long,...
  - float, double
  - pointers
  - structures: struct, union
  - arrays
- **Haskell Types:**
  - Bool, Int, Float, ...
  - Char, String
  - tuples,(N-tuples), records
  - lists
  - functions

What exactly is a type? The most obvious answer, perhaps, is that a type is a set of values. When we say that v is a value of type T, we mean simply that $v \in T$. When we say that an expression E is of type T, we are asserting that the result of evaluating E will be a value of type T.

However, not every set of values is suitable to be regarded as a type. We insist that each operation associated with the type behaves uniformly when applied to all values of the type. Thus $\{false, true\}$ is a type because the operations not, and, and or operate uniformly over the values $false$ and $true$. Also, $\{..., -2, -1, 0, +1, +2, ...\}$ is a type because operations such as addition and multiplication operate uniformly over all these values.

But {13, $true$, Monday} is not a type, since there are no useful operations over this set of values. Thus we see that a type is characterized not only by its set of values, but also by the operations over that set of values.

Therefore we define a ***type*** to be a set of values, equipped with one or more operations that can be applied uniformly to all these values.

Every programming language supports both primitive types and composite types. Some languages also have recursive types, a recursive type being one whose values are composed from other values of the same type.

2 Primitive Types

A ***primitive value*** is one that cannot be decomposed into simpler values. A ***primitive type*** is one whose values are primitive.

Every programming language provides built-in primitive types. Some languages also allow programs to define new primitive types.

2.1 Built-in Primitive Types

One or more primitive types are built-in to every programming language. The choice of built-in primitive types tells us much about the programming language's intended application area.

Nevertheless, certain primitive types crop up in a variety of languages, often under different names. For the sake of consistency, we shall use `Boolean, Character, Integer`, and `Float` as names for the most common primitive types:

- `Boolean` = $\{false, true\}$
- `Character` = {..., 'a', ..., 'z', ..., '0', ..., '9', ..., '?', ...}
- `Integer` = {..., -2, -1, 0, +1, +2, ...}
- `Float` = {..., -1.0, ..., 0.0, ..., +1.0, ...}

The ***cardinality*** of a type T, written #T, is the number of distinct values in T. For example:

- #Boolean = 2
- #Character = 256 (ISO LATIN character set)
- #Character = 65 536 (UNICODE character set)

If some types are implementation-defined, the behavior of programs may vary from one computer to another, even programs written in high-level languages. This gives rise to portability problems: a program that works well on one computer might fail when moved to a different computer.

One way to avoid such portability problems is for the programming language to define all its primitive types precisely. As we have seen, this approach is taken by JAVA.

2.2 Defined Primitive Types

Another way to avoid portability problems is to allow programs to define their own integer and floating-point types, stating explicitly the desired range and/or precision for each type. For example, the following declaration can be used in `ADA`: '`type Population is range 0 ... 1e10;`'. In ADA, we can define a completely new primitive type by enumerating its values. Such a type is called an ***enumeration type***, and its values are called ***enumerands***.

2.3 Discrete Primitive Types

A ***discrete primitive type*** is a primitive type whose values have a *one-to-one relationship with a range of integers.*

This is an important concept in `ADA`, in which values of any discrete primitive type may be used for array indexing, counting, and so on. The discrete primitive types in `ADA` are `Boolean, Character`, integer types, and enumeration types.

Most programming languages allow only integers to be used for counting and array indexing. `C` and `C++` allow enumerands also to be used for counting and array indexing, since they classify enumeration types as integer types.

3 Composite Types

A ***composite value*** (or *data structure*) is a value that is composed from simpler values. A ***composite type*** is a type whose values are composite.

Programming languages support a huge variety of composite values: tuples, structures, records, arrays, algebraic types, discriminated records, objects, unions, strings, lists, trees, sequential files, direct files, relations, etc. The variety might seem bewildering, but in fact nearly all these composite values can be understood in terms of a small number of structuring concepts, which are:

- Cartesian products (tuples, records)
- mappings (arrays)
- disjoint unions (algebraic types, discriminated records, objects)
- recursive types (lists, trees).

Each programming language provides its own notation for describing composite types. Here, mathematical notation that is concise, standard, and suitable for defining sets of values structured as Cartesian products, mappings, and disjoint unions will be used.

3.1 Cartesian Products, Structures, and Records

In a ***Cartesian product***, values of several (possibly different) types are grouped into tuples.

We use the notation $(x,\ y)$ to stand for the pair whose first component is x and whose second component is y. We use the notation $S \times T$ to stand for the set of all pairs $(x,\ y)$ such that x is chosen from set S and y is chosen from set T. Formally:

$$S \times T = \{(x,\ y) \mid x \in S;\ y \in T\} \tag{1}$$

The basic operations on pairs are:

- ***construction*** of a pair from two component values;
- ***selection*** of the first or second component of a pair.

We can easily infer the cardinality of a Cartesian product:

$$\#(S \times T) = \#S \times \#T \tag{2}$$

This equation motivates the use of the notation "$\times$" for Cartesian product.

We can extend the notion of Cartesian product from pairs to tuples with any number of components. In general, the notation $S_1 \times S_2 \times ... \times S_n$ stands for the set of all n-tuples, such that the first component of each n-tuple is chosen from S_1, the second component from S_2, ..., and the n^{th} component from S_n.

C `struct`, Pascal `record`, functional languages `tuple`

in C: `string` $\times$ `int`

```
struct Person {
    char name[20];
    int no;
} x = {" Osman Hamdi " ,23141};
```

in Haskell: `string` $\times$ `int`

```
type People = (String, Int)
...
x = ("Osman Hamdi", 23141)::People
```

in Python: `string` $\times$ `int`

```
x = ("Osman Hamdi", 23141)
type (x)
<type 'tuple'>
```

A special case of a Cartesian product is one where all tuple components are chosen from the same set. The tuples in this case are said to be ***homogeneous***. For example:

$$S^2 = S \times S \tag{3}$$

means the set of homogeneous pairs whose components are both chosen from set S. More generally we write:

$$S^n = S \times S \times \cdots \times S \tag{4}$$

to mean the set of homogeneous n-tuples whose components are all chosen from set S.

The cardinality of a set of homogeneous n-tuples is given by:

$$\#(S^n) = (\#S)^n \tag{5}$$

This motivates the superscript notation.

Finally, let us consider the special case where $n = 0$. Equation 5 tells us that S_0 should have exactly one value. This value is the empty tuple $()$, which is the unique tuple with no components at all. We shall find it useful to define a type that has the empty tuple as its only value:

$$\text{Unit} = \{()\} \tag{6}$$

This type's cardinality is:

$$\#\text{Unit} = 1 \tag{7}$$

Note that Unit is *not* the empty set (whose cardinality is 0). Unit corresponds to the type named **`void`** in `C`, `C++`, and `JAVA`, to the type **null record** in `ADA`, and to the **`None`** in `Python`.

3.2 Mappings, Arrays, and Functions

The notion of a ***mapping*** from one set to another is extremely important in programming languages. This notion in fact underlies two apparently different language features: *arrays* and *functions*.

We write:

$$m : S \to T \tag{8}$$

to state that m is a mapping from set S to set T. In other words, m maps every value in S to a value in T. (Read the symbol "$\to$" as "maps to".)

If m maps value x in set S to value y in set T, we write $y = m(x)$. The value y is called the image of x under m.

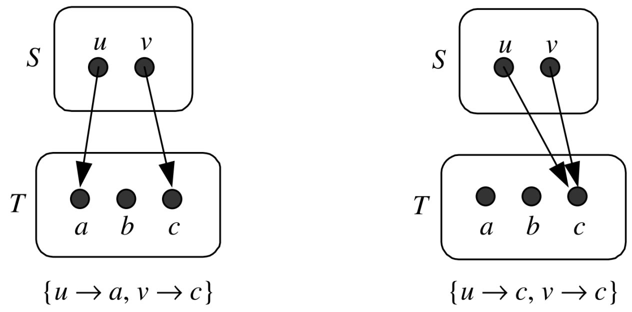

Figure 1: Two different mappings in $S \to T$

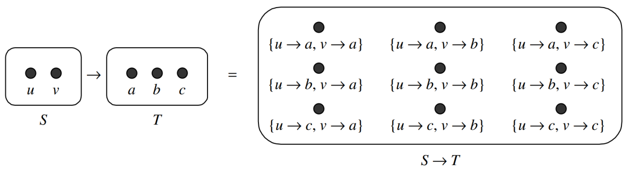

Figure 2: Set of all mappings in $S \to T$

Two different mappings from $S = \{u,\ v\}$ to $T = \{a,\ b,\ c\}$ are illustrated in Figure 1. We use notation such as $\{u \to a,\ v \to c\}$ to denote the mapping that maps u to a and v to c.

The notation $S \to T$ stands for the set of all mappings from S to T. Formally:

$$S \to T = \{m \mid x \in S \Rightarrow m(x) \in T\} \tag{9}$$

This is illustrated in Figure 2.

Let us deduce the cardinality of $S \to T$. Each value in S has $\#T$ possible images under a mapping in $S \to T$. There are $\#S$ such values in S. Therefore there are $\#T \times \#T \times \cdots \times \#T$ possible mappings ($\#S$ copies of $\#T$ multiplied together). In short:

$$\#(S \to T) = (\#T)^{\#S} \tag{10}$$

An ***array*** is an indexed sequence of components. An array has one component of type T for each value in type S, so the array itself has type $S \to T$. The ***length*** of the array is its number of components, which is $\#S$. Arrays are found in all imperative and object-oriented languages.

The type S must be finite, so an array is a *finite* mapping. In practice, S is always a range of consecutive values, which is called the array's ***index range***. The limits of the index range are called its ***lower bound*** and ***upper bound***.

The basic operations on arrays are:

- ***construction*** of an array from its components;
- ***indexing***, i.e., selecting a particular component of an array, given its index.

The index used to select an array component is a computed value. Thus array indexing differs fundamentally from Cartesian-product selection (where the component to be selected is always explicit).

`C` and `C++` restrict an array's index range to be a range of integers whose lower bound is zero. (S is integers whose lower bound is 0.)

Mappings occur in programming languages, not only as arrays, but also as ***function procedures*** (more usually called simply *functions*). We can implement a mapping in $S \to T$ by means of a function procedure, which takes a value in S (the ***argument***) and computes its image in T (the ***result***). Here the set S is not necessarily finite.

3.2.1 Array and Function Difference

Arrays:

- Values stored in memory
- Restricted: only integer domain
- double $\to$ double ?

Functions:

- Defined by algorithms
- Efficiency, resource usage
- All types of mappings possible
- Side effect, output, error, termination problem.

3.3 Disjoint Unions, Discriminated Records, and Objects

Another kind of composite value is the ***disjoint union***, whereby a value is chosen from one of several (usually different) sets.

We use the notation $S + T$ to stand for a set of disjoint-union values, each of which consists of a ***tag*** together with a ***variant*** chosen from either set S or set T. The tag indicates the set from which the variant was chosen. Formally:

$$S + T = \{left\ x \mid x \in S\} \cup \{right\ y \mid y \in T\} \tag{11}$$

Here $left\ x$ stands for a disjoint-union value with tag $left$ and variant x chosen from S, while $right\ x$ stands for a disjoint-union value with tag $right$ and variant y chosen from T. This is illustrated in Figure 2.4.

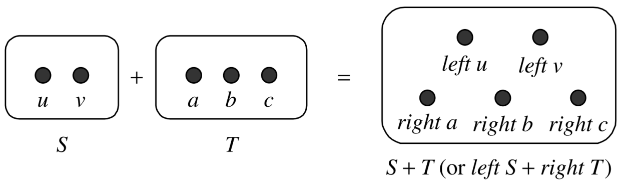

Figure 3: Disjoint union of sets S and T

When we wish to make the tags explicit, we will use the notation $left\ S + right\ T$:

$$left\ S + right\ T = \{left\ x \mid x \in S\} \cup \{right\ y \mid y \in T\} \tag{12}$$

When the tags are irrelevant, we will still use the simpler notation $S + T$.

Note that the tags serve only to distinguish the variants. They must be distinct, but otherwise they may be chosen freely.

The basic operations on disjoint-union values in $S + T$ are:

- ***construction*** of a disjoint-union value, by taking a value in either S or T and tagging it accordingly;
- ***tag test***, determining whether the variant was chosen from S or T;
- ***projection*** to recover the variant in S or the variant in T (as the case may be).

For example, a tag test on the value $right\ b$ determines that the variant was chosen from T, so we can proceed to project it to recover the variant b.

We can easily infer the cardinality of a disjoint union:

$$\#(S + T) = \#S + \#T \tag{13}$$

This motivates the use of the notation "$+$" for disjoint union.

We can extend disjoint union to any number of sets. In general, the notation $S_1 + S_2 + \cdots + S_n$ stands for the set in which each value is chosen from one of $S_1,\ S_2,\ \cdots$, or S_n.

The functional language HASKELL has ***algebraic types***, which we can understand in terms of disjoint unions. In fact, the HASKELL notation is very close to our mathematical disjoint-union notation.

How should we understand ***objects***? Simplistically we could view each object of a particular class as a tuple of components. However, any object-oriented language allows objects of different classes to be used interchangeably (to a certain extent), and therefore provides an operation to test the class of a particular object. Thus each object must have a tag that identifies its class. So we shall view each object as a *tagged* tuple.

It is *important not to confuse disjoint union with ordinary set union.* The tags in a disjoint union $S + T$ allow us to test whether the variant was chosen from S or T. This is not necessarily the case in the ordinary union $S \cup T$. In fact, if $T = \{a,\ b,\ c\}$, then:

$$T \cup T = \{a,\ b,\ c\} = T \tag{14}$$

$$T + T = \{left\ a,\ left\ b,\ left\ c,\ right\ a,\ right\ b,\ right\ c\} \neq T \tag{15}$$

The **unions** of C and C++ are not disjoint unions, since they have no tags. This obviously makes tag test impossible, and makes projection unsafe. In practice, therefore, C programmers enclose each union within a structure that also contains a tag.

3.4 Power Set

The set of all subsets. $\mathcal{P}(S) = \{T \mid T \subseteq S\}$. Cardinality of set $\#\mathcal{P}(S) = 2^{\#S}$

4 Recursive Types

A ***recursive type*** is one defined in terms of itself. In this section we discuss two common recursive types, lists and strings, as well as recursive types in general.

4.1 Lists

A ***list*** is a sequence of values. A list may have any number of components, including none. The number of components is called the ***length*** of the list. The unique list with no components is called the ***empty list***.

A list is ***homogeneous*** if all its components are of the same type; otherwise it is ***heterogeneous***.

Typical list operations are:

- length
- emptiness test
- head selection (i.e., selection of the list's first component)
- tail selection (i.e., selection of the list consisting of all but the first component)
- concatenation

Suppose that we wish to define a type of *integer-lists*, whose values are lists of integers. We may define an integer-list to be a value that is either empty or a pair consisting of an integer (its head) and a further integer-list (its tail). This definition is recursive. We may write this definition as a set equation:

$$\texttt{Integer-List} = nil\ \texttt{Unit} + cons(\texttt{IntegerInteger-List}) \tag{16}$$

or, in other words:

$$\texttt{Integer-List} = \{nil()\} \cup \{cons(i, l) \mid i \in \texttt{Integer};\ l \in \texttt{Integer-List}\} \tag{17}$$

where we have chosen the tags *nil* for an empty list and *cons* for a nonempty list. Henceforth we shall abbreviate *nil*() to *nil*.

Polymorphic lists: a single definition defines lists of many types. So, the general idea:

$$L = \text{Unit} + (T \times L) \qquad\qquad \texttt{List}\alpha = \alpha \times (\texttt{List}\alpha) + \{\textit{empty}\} \tag{18}$$

has a *least solution* for L that corresponds to the set of all finite lists of values chosen from T. Every other solution is a superset of the least solution.

Lists (or sequences) are so ubiquitous that they deserve a notation of their own: T^* stands for the set of all finite lists of values chosen from T. Thus:

$$T^* = \text{Unit} + (T \times T^*) \tag{19}$$

In imperative languages (such as `C, C++`, and `ADA`), recursive types must be defined in terms of pointers. In functional languages (such as `HASKELL`) and in some object-oriented languages (such as `JAVA`), recursive types can be defined directly.

4.2 Strings

A ***string*** is a sequence of characters. A string may have any number of characters, including none. The number of characters is called the ***length*** of the string. The unique string with no characters is called the ***empty string***.

Strings are supported by all modern programming languages. Typical string operations are:

- length
- equality comparison
- lexicographic comparison
- character selection
- substring selection
- concatenation

How should we classify strings? No consensus has emerged among programming language designers.

- One approach is to classify strings as *primitive values*.
- Another approach is to treat strings as *arrays of characters*.
- A slightly different and more flexible approach is to treat strings as *pointers to arrays of characters*.
- In a programming language that supports lists, the most natural approach is to treat strings as *lists of characters*.
- In an object-oriented language, the most natural approach is to treat strings as *objects*.

4.3 Recursive Types in General

As we have seen, a *recursive type* is one defined in terms of itself. Values of a recursive type are composed from values of the same type.

In general, the set of values of a recursive type, *R*, will be defined by a recursive set equation of the form:

$$R = ... + (...R...R...) \tag{20}$$

A recursive set equation may have many solutions. Fortunately, a recursive set equation always has a *least solution* that is a subset of every other solution. In computation, the least solution is the one in which we are interested.

The cardinality of a recursive type is *infinite*, even if every individual value of the type is finite.

5 Type Systems

A programming language's ***type system*** groups values into types. This allows programmers to describe data effectively. It also helps prevent programs from performing nonsensical operations, such as multiplying a string by a boolean. Performing such a nonsensical operation is called a ***type error***.

5.1 Static vs Dynamic Typing

Types are required to provide data processing, integrity checking, efficiency, access controls. Type compatibility on operators is essential. Before any operation is to be performed, the types of its operands must be checked in order to prevent a type error. For example, before an integer multiplication is performed, both operands must be checked to ensure that they are integers. Such checks are called ***type checks***.

The type check must be performed before the operation itself is performed. However, there may still be some freedom in the timing: the type check could be performed either at *compile-time* or at *run-time*. This seemingly pragmatic issue in fact underlies an important classification of programming languages, into statically typed and dynamically typed languages.

- In a ***statically typed*** language, each variable and each expression has a *fixed type* (which is either explicitly stated by the programmer or inferred by the compiler). Using this information, all operands can be type-checked *at compile-time*.
- In a ***dynamically typed*** language, values have fixed types, but variables and expressions have no fixed types. Every time an operand is computed, it could yield a value of a different type. So operands must be type-checked after they are computed, but before performing the operation, *at run-time*.

The choice between static and dynamic typing is essentially pragmatic:

- *Static typing is more efficient.* Dynamic typing requires (possibly repeated) run-time type checks, which slow down the program's execution. Static typing requires only compile-time type checks, whose cost is minimal (and one-off). Moreover, dynamic typing forces all values to be tagged (in order to make run-time type checks possible), and these tags take up storage space. Static typing requires no such tagging.
- *Static typing is more secure*: the compiler can certify that the program contains no type errors. Dynamic typing provides no such security. (This point is important because type errors account for a significant proportion of programming errors.)
- *Dynamic typing provides greater flexibility*, which is needed by some applications where the types of the data are not known in advance.

In practice the greater security and efficiency of static typing outweigh the greater flexibility of dynamic typing in the vast majority of applications. It is no coincidence that most programming languages are statically typed.

5.2 Type Equivalence

Consider some operation that expects an operand of type T_1. Suppose that it is given instead an operand whose type turns out to be T_2. Then we must check whether T_1 is equivalent to T_2, written $T_1 \equiv T_2$. What exactly this means depends on the programming language.

One possible definition of type equivalence is ***structural equivalence***: $T_1 \equiv T_2$ if and only if T_1 and T_2 have the same set of values.

Structural equivalence is so called because it may be checked by comparing the *structures* of the types T_1 and T_2. (It is unnecessary, and in general even impossible, to enumerate all values of these types.)

The following rules illustrate how we can decide whether types T_1 and T_2, defined in terms of *Cartesian products*, *disjoint unions*, and *mappings*, are structurally equivalent or not.

- If T_1 and T_2 are both primitive, then $T_1 \equiv T_2$ if and only if T_1 and T_2 are identical. For example:

  $\texttt{Integer} \equiv \texttt{Integer}$
  $\texttt{Integer} \not\equiv \texttt{Float}$
  (The symbol "$\not\equiv$" means "is not equivalent to".)

- If $T_1 = A_1 \times B_1$ and $T_2 = A_2 \times B_2$, then $T_1 \equiv T_2$ if and only if $A_1 \equiv A_2$ and $B_1 \equiv B_2$. For example:

  $\texttt{Integer} \times \texttt{Float} \equiv \texttt{Integer} \times \texttt{Float}$
  $\texttt{Integer} \times \texttt{Float} \not\equiv \texttt{Float} \times \texttt{Integer}$

- If $T = A_1 \mapsto B_1$ and $T_2 = A_2 \mapsto B_2$, then $T_1 \equiv T_2$ if and only if $A_1 \equiv A_2$ and $B_1 \equiv B_2$. For example:

  $\texttt{Integer} \mapsto \texttt{Float} \equiv \texttt{Integer} \mapsto \texttt{Float}$
  $\texttt{Integer} \mapsto \texttt{Float} \not\equiv \texttt{Integer} \mapsto \texttt{Boolean}$

- If $T_1 = A_1 + B_1$ and $T_2 = A_2 + B_2$, then $T_1 \equiv T_2$ if and only if either $A_1 \equiv A_2$ and $B_1 \equiv B_2$, or $A_1 \equiv B_2$ and $B_1 \equiv A_2$. For example:

  $\texttt{Integer} + \texttt{Float} \equiv \texttt{Integer} + \texttt{Float}$
  $\texttt{Integer} + \texttt{Float} \equiv \texttt{Float} + \texttt{Integer}$
  $\texttt{Integer} + \texttt{Float} \not\equiv \texttt{Integer} + \texttt{Boolean}$

- Otherwise $T_1 \not\equiv T_2$

Although these rules are simple, it is not easy to see whether two recursive types are structurally equivalent. In other words, it is harder to implement structural equality, especially in recursive cases. Consider the following:

$$T_1 = \text{Unit} + (S \times T_1)$$
$$T_2 = \text{Unit} + (S \times T_2)$$
$$T_3 = \text{Unit} + (S \times T_3)$$

Intuitively, these three types are all structurally equivalent. However, the reasoning needed to decide whether two arbitrary recursive types are structurally equivalent makes type checking uncomfortably hard.

Another possible definition of type equivalence is ***name equivalence***: $T_1 \equiv T_2$ if and only if T_1 and T_2 were defined in the same place.

The following summarizes the advantages and disadvantages of structural and name equivalence:

- *Name equivalence forces each distinct type to be defined in one and only one place.* This is sometimes inconvenient, but it helps to make the program more maintainable. (If the same type is defined in several places, and subsequently it has to be changed, the change must be made consistently in several places.)

- *Structural equivalence allows confusion between types that are only coincidentally similar.*

Most programming languages use name equivalence.

5.3 The Type Completeness Principle

First order values needs to have the following:

- Assignment
- Function parameter
- Take part in compositions
- Return value from a function

Most imperative languages (`Pascal`, `Fortran`) classify functions as second order value (`C` represents function names as pointers). Functions are first order values in most functional languages like `Haskell` and `Scheme`.

We can characterize a language's class-consciousness in terms of its adherence to the ***Type Completeness Principle***:

> No operation should be arbitrarily restricted in the types of its operands.

For another definition,

> First order values should take part in all operations above, no arbitrary restrictions should exist.

The word *arbitrarily* is important here. Insisting that the operands of the and operation are booleans is not an arbitrary restriction, since it is inherent in the nature of this operation. But insisting that only values of certain types can be assigned is an arbitrary restriction, as is insisting that only values of certain types can be passed as arguments or returned as function results.

C Types	Primitive	Array	Struct	Functions
Assigment	✓	✗	✓	✗
Functions parameter	✓	✗	✓	✗
Function return	✓	✗	✓	✗
In compositions	✓	✓	✓	✗

Haskell Types	Primitive	Array	Struct	Functions
Assigment	✓	✓	✓	✓
Functions parameter	✓	✓	✓	✓
Function return	✓	✓	✓	✓
In compositions	✓	✓	✓	✓

Pascal Types	Primitive	Array	Struct	Functions
Assigment	✓	✓	✓	✗
Functions parameter	✓	✓	✓	✗
Function return	✓	✗	✗	✗
In compositions	✓	✓	✓	✗

6 Expression

An ***expression*** is a construct that will be ***evaluated*** to yield a value. Expressions may be formed in various ways. In this section we shall survey the fundamental forms of expression:

- Literals
- Constructions
- Function Calls
- Conditional Expression
- Iterative Expression
- Constant and Variable Accesses
- Aggregates
- Block Expression

6.1 Literals

The simplest kind of expression is a ***literal***, which denotes a fixed value of some type.

6.2 Constructions

A ***construction*** is an expression that constructs a composite value from its component values. In some languages the component values must be literals; in others, the component values are computed by evaluating subexpressions.

6.3 Function Calls

A ***function call*** computes a result by applying a function procedure (or method) to one or more arguments. The function call typically has the form "$F(E)$", where F determines the function procedure to be applied, and the expression E is evaluated to determine the argument.

An ***operator*** may be thought of as denoting a function. Applying a unary or binary operator to its operand(s) is essentially equivalent to a function call with one or two argument(s): $\oplus E$ is essentially equivalent to $\oplus(E)$ (where $\oplus$ is a unary operator) $E_1 \otimes E_2$ is essentially eqivalent to $\otimes(E_1, E_2)$ (where $\otimes$ is a binary operator.

6.4 Conditional Expressions

A ***conditional expression*** computes a value that depends on a condition. It has two or more subexpressions, from which exactly one is chosen to be evaluated.

6.5 Iterative Expressions

An ***iterative expression*** is one that performs a computation over a series of values (typically the components of an array or list), yielding some result. Iterative expressions are rather more unusual, but they are a prominent feature of the functional language `HASKELL`, in the form of list comprehensions.

6.6 Constant and Variable Accesses

A ***constant access*** is a reference to a named constant, and yields the value of that constant. A ***variable access*** is a reference to a named variable, and yields the current value of that variable. ("`#define`" can be example of that in `C`)

6.7 Aggregates

Used to construct composite values without any declaration/definition.

```
x = (12 , "ali", True)
y = {name = "ali", no = 12}
f = \x -> x*x
l = [1 ,2 ,3 ,4]
```

```
x = (12 , "ali", True)
y = [1, 2, [2, 3], "a"]
f = {"name": "ali", "no": "12"}
l = lambda x:x+1
```

```
struct Person {
    char name[20];
    int no;
} p = {"Ali Cin", 332314};
```

6.8 Block Expressions

Some languages allow multiple/statements in a block to calculate a value. GCC extension for compound statement expressions:

```
double s, i, arr[10];
s = ( { double t = 0;
        for (i = 0; i < 10; i++)
            t += arr[i];
        t;} ) + 1;
```

Value of the last expression is the value of the block.