1. Value and Type
2. Primitive vs Composite Types
3. Cartesian Product
4. Disjoint Union
5. Mappings
   - Arrays
   - Functions
6. Powerset
7. Recursive Types
   - Lists
   - General Recursive Types
   - Strings
8. Type Systems
   - Static Type Checking
   - Dynamic Type Checking
   - Type Equality
9. Type Completeness
10. Expressions
    - Literals/Variable and Constant Access
    - Aggregates
    - Variable References
    - Function Calls
    - Conditional Expressions
    - Iterative Expressions
11. Summary
What are Value and Type?

- **Value** anything that exist, that can be computed, stored, take part in data structure. Constants, variable content, parameters, function return values, operator results...

- **Type** set of values of same kind. 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

Each type represents a set of values. Is that enough?
What about the following set? Is it a type?
{"ahmet", 1 , 4 , 23.453, 2.32, ’b’}

Values should exhibit a similar behavior. The same group of operations should be defined on them.
Primitive vs Composite Types

- **Primitive Types**: Values that cannot be decomposed into other sub values.
  - C: int, float, double, char, long, short, pointers
  - Haskell: Bool, Int, Float, function values

- **Cardinality of a type**: The number of distinct values that a datatype has. Denoted as: "#Type".
  - #Bool = 2  #char = 256  #short = 2^{16}
  - #int = 2^{32}  #double = 2^{32}, ... 

- What does cardinality mean? How many bits required to store the datatype?
User Defined Primitive Types

- **enumerated types**
  ```c
  enum days {mon, tue, wed, thu, fri, sat, sun};
  enum months {jan, feb, mar, apr, .... };
  ```

- **ranges** (Pascal and Ada)
  ```c
  type Day = 1..31;
  var g:Day;
  ```

- **Discrete Ordinal Primitive Types** Datatypes values have one to one mapping to a range of integers.
  - C: Every ordinal type is an alias for integers.
  - Pascal, Ada: distinct types

- **DOPT’s** are important as they
  1. can be array indices, switch/case labels
  2. can be used as for loop variable (some languages like pascal)
User defined types with composition of one or more other datatypes. Depending on composition type:

- Cartesian Product (struct, tuples, records)
- Disjoint union (union (C), variant record (pascal), Data (haskell))
- Mapping (arrays, functions)
- Powerset (set datatype (Pascal))
- Recursive compositions (lists, trees, complex data structures)
Cartesian Product

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

Example:
$S = \{a, b, c\}$  $T = \{1, 2\}$
$S \times T = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$

$\#(S \times T) = \#S \cdot \#T$
- C struct, Pascal record, functional languages **tuple**

- **in C:** string $\times$ int

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

- **in Haskell:** string $\times$ int

```haskell
type People=(String,Int)
...
(x :: People) = ("Osman Hamdi",23141)
```
Multiple Cartesian products:
C: string × int × {MALE,FEMALE}

```c
struct Person {
    char name[20];
    int no;
    enum Sex {MALE, FEMALE} sex;
} x = {"Osman Hamdi", 23141, FEMALE};
```

Haskell: string × int × float × String

```haskell
x = ("Osman Hamdi", 23141, 3.98, "Yazar")
```
Homogeneous Cartesian Products

\[ S^n = \underbrace{S \times S \times S \times \ldots \times S}_n \]

declare double^4:

```c
struct quad { double x, y, z, q; }
```

- \( S^0 = \{()\} \) is 0-tuple.
- not empty set. A set with a single value.
- terminating value (nil) for functional language lists.
Disjoint Union

- $S + T = \{left \, x \mid x \in S\} \cup \{right \, x \mid x \in T\}$
- Example:
  - $S = \{1, 2, 3\}$  $T = \{3, 4\}$
  - $S + T = \{left \, 1, left \, 2, left \, 3, right \, 3, right \, 4\}$
  - $\#(S + T) = \#S + \#T$
- Can union\'s be disjoint union?
- **C**: int + double:

  ```c
  union number { double real; int integer; } x;
  ```

- **C union’s are not safe! Same storage is shared. Valid field is unknown:**

  ```c
  x.real=3.14; printf("%d\n",x.integer);
  ```

- **Haskell**: Float + Int + (Int × Int):

  ```haskell
  data Number = RealVal Float | IntVal Int | Rational (Int,Int)
  x = Rational (3,4)
  y = RealVal 3.14
  z = IntVal 12     {-- You cannot access different values --}
  ```
Mappings

- The set of all possible mappings
- \( S \mapsto T = \{ V \mid \forall (x \in S) \exists (y \in T), (x \mapsto y) \in V \} \)
- Example: \( S = \{ a, b \} \quad T = \{ 1, 2, 3 \} \)

\[
S \mapsto T = \{ \{ a \mapsto 1, b \mapsto 1 \}, \{ a \mapsto 1, b \mapsto 2 \}, \{ a \mapsto 1, b \mapsto 3 \}, \\
\{ a \mapsto 2, b \mapsto 1 \}, \{ a \mapsto 2, b \mapsto 2 \}, \{ a \mapsto 2, b \mapsto 3 \}, \\
\{ a \mapsto 3, b \mapsto 1 \}, \{ a \mapsto 3, b \mapsto 2 \}, \{ a \mapsto 3, b \mapsto 3 \} \}
\]
- \( \#(S \mapsto T) = \#T \#S \)
Arrays

- `double a[3]=\{1.2, 2.4, -2.1\};`
  
  - `a \in (\{0, 1, 2\} \mapsto double)`
  - `a = (0 \mapsto 1.2, 1 \mapsto 2.4, 2 \mapsto -2.1)`

- Arrays define a mapping from an integer range (or DOPT) to any other type

- **C**: `T \ x[N] \Rightarrow x \in (\{0, 1, ..., N - 1\} \mapsto T)`

- Other array index types (Pascal):

```
type
  Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
var
  x : array Day of real;
  y : array Month of integer;
...

  x[Tue] := 2.4;
  y[Feb] := 28;
```
Functions

- **C function:**
  ```c
  int f(int a) {
    if (a%2 == 0) return 0;
    else return 1;
  }
  ```

- **f : int → {0, 1}**
  regardless of the function body: f : int → int

- **Haskell:**
  ```haskell
  f a = if mod a 2 == 0 then 0 else 1
  ```

- in C, f expression is a pointer type `int (*)(int)`
  in Haskell it is a mapping: `int → int`
Array and Function Difference

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

Cartesian mappings:
- double a[3][4];
- double f(int m, int n);
- int $\times$ int $\mapsto$ double and int $\mapsto$(int $\mapsto$ double)

Functions
- Defined by algorithms
- Efficiency, resource usage
- All types of mappings possible
- Side effect, output, error, termination problem.
Cartesian Mapping vs Nested mapping

Pascal arrays

```pascal
var
    x : array [1..3,1..4] of double;
    y : array [1..3] of array [1..4] of double;
...
    x[1,3] := x[2,3]+1;
    y[1,3] := y[2,3]+1;
```

Row operations:

Haskell functions:

\[ f(x, y) = x + y \]

\[ g(x, y) = x + y \]

\[ \ldots \]

\[ f(3 + 2) \]

\[ g(3, 2) \]

\[ g(3) \checkmark \]

\[ f(3) \times \]

Reuse the old definition to define a new function:

\[ \text{increment} = g(1) \]

\[ \text{increment } 1 \]

\[ 2 \]
Powerset

- \( \mathcal{P}(S) = \{ T \mid T \subseteq S \} \)
- The set of all subsets
- \( S = \{ 1, 2, 3 \} \)
- \( \mathcal{P}(S) = \{ \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \} \)
- \( \#\mathcal{P}(S) = 2^{\#S} \)
- Set datatype is restricted and special datatype. Only exists in Pascal and special set languages like SetL

- set operations (Pascal)

```pascal
type
color = (red, green, blue, white, black);
colorset = set of color;

var
  a, b : colorset;
...

a := [red, blue];
b := a*b; (* intersection *)
b := a+[green, red]; (* union *)
b := a-[blue]; (* difference *)
if (green in b) then ... (* element test *)
if (a = []) then ... (* set equality *)
```

- in C++ supported by library.
Recursive Types

- $S = \ldots S \ldots$
- Types including themselves in composition.

Lists

- $S = \text{Int} \times S + \{\text{null}\}$

$$S = \{ \text{right empty} \} \cup \{ \text{left } (x, \text{empty}) \mid x \in \text{Int} \} \cup \{ \text{left } (x, \text{left } (y, \text{empty})) \mid x, y \in \text{Int} \} \cup \{ \text{left } (x, \text{left } (y, \text{left } (z, \text{empty})))) \mid x, y, z \in \text{Int} \} \cup \ldots$$

- $S =$
  \{ right empty, left(1, empty), left(2, empty), left(3, empty), ..., left(1, left(1, empty)), left(1, left(2, empty)), left(1, left(3, empty)), ..., left(1, left(1, left(1, empty))), left(1, left(1, left(2, empty))), ... \}

```c
struct List {
    int x;
    List *next;
} a;
```

- Haskell lists.

```haskell
data List = Left (Int, List) | Empty

x = Left (1, Left(2, Left(3, Empty)))  -- [1,2,3] list --
y = Empty                              -- empty list, []
```
Polymorphic lists: a single definition defines lists of many types.

\[ \text{List } \alpha = \alpha \times (\text{List } \alpha) + \{\text{empty}\} \]

```haskell
data List alpha = Left (alpha, List alpha) | Empty

x = Left (1, Left(2, Left(3, Empty)))  {-- [1,2,3] list --}
y = Left ("ali", Left("ahmet", Empty))  {-- ["ali","ahmet"] --}
z = Left(23.1, Left(32.2, Left(1.0, Empty))) {-- [23.1,32.2,1.0] --}
```

Left(1, Left("ali", Left(15.23, Empty)) \in \text{List } \alpha? \text{ No.}

Most languages only permits homogeneous lists.
Haskell Lists

- binary operator "::" for list construction:
  \[
  \text{data}\ [\alpha] = (\alpha :\ [\alpha]) | []
  \]
  \[
  x = (1:(2:(3:[])))
  \]

- Syntactic sugar:
  \[
  [1,2,3] \equiv (1:(2:(3:[])))
  \]
  \[
  ["ali"] \equiv ("ali":[])
  \]
General Recursive Types

- $T = \ldots T\ldots$
- Formula requires a minimal solution to be representable:
  $$S = Int \times S$$
  Is it possible to write a single value? No minimum solution here!
- List example:
  $$x = \text{Left}(1, \text{Left}(2, x))$$
  $x \in S$? Yes
  can we process $[1, 2, 1, 2, 1, 2, \ldots]$ value?
- Some languages like Haskell lets user define such values. All iterations go infinite. Useful in some domains though.
- Most languages allow only a subset of $S$, the subset of finite values.
Tree $\alpha = empty + node \alpha \times Tree\alpha \times Tree\alpha$

$$Tree\ \alpha = \{ \text{empty} \} \cup \{ \text{node}(x, \text{empty, empty}) \mid x \in \alpha \}\cup \{ \text{node}(x, \text{node}(y, \text{empty, empty}), \text{empty}) \mid x, y \in \alpha \}\cup \{ \text{node}(x, \text{empty}, \text{node}(y, \text{empty, empty})) \mid x, y \in \alpha \}\cup \{ \text{node}(x, \text{node}(y, \text{empty, empty}), \text{node}(z, \text{empty, empty})) \mid x, y, z \in \alpha \} \cup \ldots$$

- **C++ (pointers and template definition)**
  ```cpp
  template<class Alpha>
  struct Tree {
    Alpha x;
    Tree *left,*right;
  } root;
  ```

- **Haskell**
  ```haskell
  data Tree alpha = Empty |
  Node (alpha , Tree alpha , Tree alpha)
  
x = Node (1, Node (2, Empty , Empty), Node (3, Empty , Empty))
y = Node (3, Empty , Empty)
  ```
Strings

Language design choice:

1. Primitive type (ML):
   Language keeps an internal table of strings

2. Character array (C, Pascal, ...)

3. Character list (Haskell, Prolog, Lisp)

- Design choice affects the complexity and efficiency of:
  concatenation, assignment, equality, lexical order, decomposition
Type Systems

- Types are required to provide data processing, integrity checking, efficiency, access controls. Type compatibility on operators is essential.

- Simple bugs can be avoided at compile time.

- Irrelevant operations:
  
  ```
  y=true * 12;
  x=12; x[1]=6;
  y=5; x.a = 4;
  ```

- When to do type checking? Latest time is before the operation. Two options:
  
  1. Compile time $\rightarrow$ static type checking
  2. Run time $\rightarrow$ dynamic type checking
- Compile time type information is used to do type checking.
- All incompatibilities are resolved at compile time. Variables have a fixed time during their lifetime.
- Most languages do static type checking
- User defined constants, variable and function types:
  - Strict type checking. User has to declare all types (C, C++, Fortran,...)
  - Languages with type inference (Haskell, ML, Scheme...)
- No type operations after compilation. All issues are resolved. Direct machine code instructions.
Dynamic Type Checking

- Run-time type checking. No checking until the operation is to be executed.
- Interpreted languages like Lisp, Prolog, PHP, Perl, Python.
- A hypothetical language:

```c
int whichmonth(input) {
    if (isinteger(input)) return input;
    else if (isstring(input))
        switch(input) {
        case "January": return 1;
        case "February": return 2;
        ...
        case "December": return 12;}
}

... read(input) /* user input at run time? */
ay=whichmonth(input)
```
Run time decision based on user's choice is possible.

- Has to carry type information along with variable at run time.
- Type of a variable can change at run-time (depends on the language).
Static vs Dynamic Type Checking

- Static type checking is faster. Dynamic type checking does type checking before each operation at run time. Also uses extra memory to keep run-time type information.
- Static type checking is more restrictive meaning safer. Bugs avoided at compile time, earlier is better.
- Dynamic type checking is less restrictive meaning more flexible. Operations working on dynamic run-time type information can be defined.
Type Equality

- \( S \equiv T \) How to decide?
  - **Name Equivalence**: Types should be defined at the same exact place.
  - **Structural Equivalence**: Types should have same value set. (mathematical set equality).

- Most languages use **name equivalence**.

- **C** example:

```c
typedef struct Comp { double x, y; } Complex;
struct COMP { double x, y; };

struct Comp a;
Complex b;
struct COMP c;

/* ... */
a=b; /* Valid, equal types */
a=c; /* Compile error, incompatible types */
```
Structural Equality

\[ S \equiv T \text{ if and only if:} \]

1. \( S \) and \( T \) are primitive types and \( S = T \) (same type),
2. if \( S = A \times B, T = A' \times B', A \equiv A' \), and \( B \equiv B' \),
3. if \( S = A + B, T = A' + B', (A \equiv A' \text{ and } B \equiv B') \text{ or } (A \equiv B' \text{ and } B \equiv A') \),
4. if \( S = A \mapsto B, T = A' \mapsto B', A \equiv A' \text{ and } B \equiv B' \),
5. if \( S = \mathcal{P}(A), T = \mathcal{P}(A') \), and \( A \equiv A' \).

Otherwise \( S \not\equiv T \)
- Harder to implement structural equality. Especially recursive cases.

- \[ T = \{ \text{nil} \} + A \times T, \quad T' = \{ \text{nil} \} + A \times T' \]
  \[ T = \{ \text{nil} \} + A \times T', \quad T' = \{ \text{nil} \} + A \times T \]

- struct Circle {
  double x, y, a;
}

- struct Square {
  double x, y, a;
}

Two types have a semantical difference. User errors may need less tolerance in such cases.

- Automated type conversion is a different concept. Does not necessarily conflicts with name equivalence.

```c
enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun} x;
x=3;
```
Type Completeness

- First order values:
  - 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.

- Arrays, structures (records)?

- **Type completeness principle**: First order values should take part in all operations above, no arbitrary restrictions should exist.
### C Types:

<table>
<thead>
<tr>
<th></th>
<th>Primitive</th>
<th>Array</th>
<th>Struct</th>
<th>Func.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>√</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Function parameter</td>
<td>√</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Function return</td>
<td>√</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>In compositions</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

### Haskell Types:

<table>
<thead>
<tr>
<th></th>
<th>Primitive</th>
<th>Array</th>
<th>Struct</th>
<th>Func.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable definition</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Function parameter</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Function return</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>In compositions</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Pascal Types:

<table>
<thead>
<tr>
<th></th>
<th>Primitive</th>
<th>Array</th>
<th>Struct.</th>
<th>Func.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Function parameter</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Function return</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>In compositions</td>
<td>√</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>
Program segments that gives a value when evaluated:

- Literals
- Variable and constant access
- Aggregates
- Variable references
- Function calls
- Conditional expressions
- Iterative expressions (Haskell)
Literals/Variable and Constant Access

- **Literals**: Constants with same value with their notation
  123, 0755, 0xa12, 12451233L, -123.342, -1.23342e-2, ’c’, ’\021’, "ayse", True, False

- **Variable and constant access**: User defined constants and variables give their content when evaluated.

```c
int x;
#define pi 3.1416
x = pi * r * r
```
Aggregates

- Used to compose composite values lexically.
  - `x=(12, "ali", True)`
  - `y={name="ali", no=12}`
  - `f=\x -> x*x`
  - `l=[1,2,3,4]`

- C only has aggregates at definition. There is no aggregates in the executable expressions!
  - `struct Person { char name[20], int no } p = {"Ali\_Cin", 332314}`
  - `double dizi[3][2] = {{0,1}, {1.2,4}, {12, 1.4}};`}
  - `p={"Veli\_Cin",123412};`
Variable References

- Variable access vs variable reference
- value vs l-value
- pointers are not references! You can use pointers as references with special operators.
- Some languages regard references like first order values (Java, C++ partially)
- Some languages distinguish the reference from the content of the variable (Unix shells, ML)
Function Calls

- $F(Gp_1, Gp_2, ..., Gp_n)$
- Function name followed by actual parameter list. Function is called, executed and the returned value is substituted in the expression position.
- **Actual parameters**: parameters send in the call
- **Formal parameters**: parameter names used in function definition
- Operators can be considered as function calls. The difference is the infix notation.
- $\oplus(a, b)$ vs $a \oplus b$
- Languages has built-in mechanisms for operators. Some languages allow user defined operators (operator overloading): C++, Haskell.
Conditional Expressions

- Evaluate to different values based on a condition.
- Haskell: if condition then exp1 else exp2.
  
  ```haskell```
  case value of p1 -> exp1 ; p2 -> exp2 ...
  ```haskell```
- C: `(condition)?exp1:exp2 ;`
- if .. else in C is **not** conditional expression but conditional statement. No value when evaluated!

```c
x = (a>b)?a:b;
y = ((a>b)?sin:cos)(x);
/* Does it work? try yourself... */
```
Haskell:

```haskell
x = if (a > b) then a else b
y = (if (a > b) then (+) else (*)) x y

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

convert a = case a of
  Left (x, rest) -> x : (convert rest)
  Empty -> []

daynumber g = case g of
  Mon -> 1
  Tue -> 2
  ... 
  Sun -> 7
```

case checks for a pattern and evaluate the RHS expression with substituting variables according to pattern at LHS.
Iterative Expressions

- Expressions that do a group of operations on elements of a list or data structure, and returns a value.

  \[
  [ \text{expr} \mid \text{variable} \leftarrow \text{list} , \text{condition} ]
  \]

- Similar to set notation in math:
  \[
  \{ \text{expr} \mid \text{var} \in \text{list}, \text{condition} \}
  \]


\[
\begin{align*}
  x &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \\
  y &= [a \times 2 \mid a \leftarrow x] \quad \{-- [2, 4, 6, 8, \ldots 24] --\} \\
  z &= [a \mid a \leftarrow x, \text{mod } a 3 == 1] \quad \{-- [1, 4, 7, 10] --\}
\end{align*}
\]
Summary

- Value and type
- Primitive types
- Composite types
- Recursive types
- When to type check
- How to type check
- Expressions