

Programming Languages

Values and Types

Onur Tolga Şehitoğlu

Computer Engineering, METU



Outline

- 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
 - Block 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.

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

■ Python types

- bool, int, float, complex
- str, bytes, tuple, list, set, dict
- classes, functions

■ Each type represents a set of values. Is that enough?

■ Haskell types

- Bool, Int, Float, ...
- Char, String
- tuples, (N-tuples), records
- lists
- functions

■ Python types

- bool, int, float, complex
- str, bytes, tuple, list, set, dict
- classes, 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'}`

■ Haskell types

- Bool, Int, Float, ...
- Char, String
- tuples, (N-tuples), records
- lists
- functions

■ Python types

- bool, int, float, complex
- str, bytes, tuple, list, set, dict
- classes, 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
Python: bool, int, float, str, functions
- **cardinality of a type:** The number of distinct values that a datatype has. Denoted as: " $\#Type$ ".
 $\#Bool = 2$ $\#char = 256$ $\#short = 2^{16}$
 $\#int = 2^{32}$ $\#double = 2^{32}, \dots$
- What does cardinality mean?

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
Python: bool, int, float, str, functions
- **cardinality of a type:** The number of distinct values that a datatype has. Denoted as: " $\#Type$ ".
 $\#Bool = 2$ $\#char = 256$ $\#short = 2^{16}$
 $\#int = 2^{32}$ $\#double = 2^{32}, \dots$
- What does cardinality mean? How many bits required to store the datatype?

User Defined Primitive Types

- **enumerated types**

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

- **ranges** (Pascal and Ada)

```
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

i. can be array indices, switch/case labels

ii. can be used as for loop variable (some languages like pascal)

Composite Datatypes

User defined types with composition of one or more other datatypes. Depending on composition type:

- Cartesian Product (struct, tuples, records)

Composite Datatypes

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))

Composite Datatypes

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)

Composite Datatypes

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))

Composite Datatypes

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)

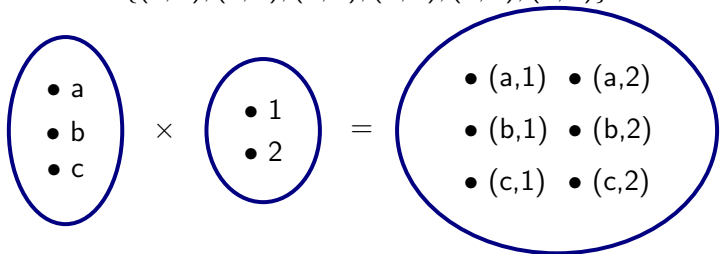
Cartesian Product

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

■ Example:

$$S = \{a, b, c\} \quad T = \{1, 2\}$$

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



■ $\#(S \times T) =$

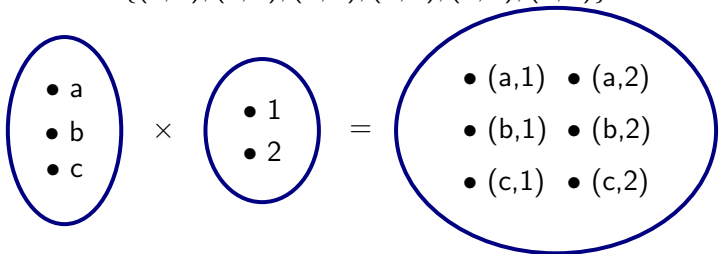
Cartesian Product

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

■ Example:

$$S = \{a, b, c\} \quad T = \{1, 2\}$$

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



■ $\#(S \times T) = \#S \cdot \#T$

- C struct, Pascal record, functional languages **tuple**
- **in C:** $\text{string} \times \text{int}$

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

- **in Haskell:** $\text{string} \times \text{int}$

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

- **in Python:** $\text{string} \times \text{int}$

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

■ Multiple Cartesian products:

C: $\text{string} \times \text{int} \times \{\text{MALE}, \text{FEMALE}\}$

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

Haskell: $\text{string} \times \text{int} \times \text{float} \times \text{string}$

```
x = ("Osman_Hamdi", 23141, 3.98, "Yazar")
```

Python: $\text{str} \times \text{int} \times \text{float} \times \text{str}$

```
x = ("Osman_Hamdi", 23141, 3.98, "Yazar")
```

Homogeneous Cartesian Products

$$S^n = \overbrace{S \times S \times S \times \dots \times S}^n$$

double⁴ :

```
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.
- C **void**. Means no value. Error on evaluation.
- Python: `()` . **None** used for no value.

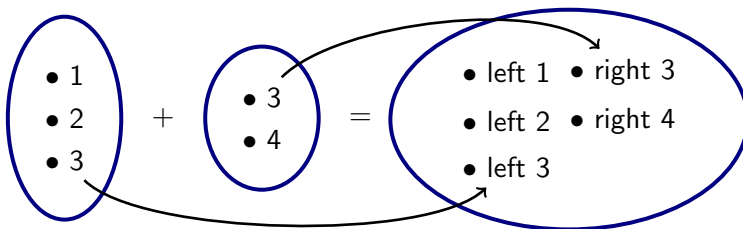
Disjoint Union

■ $S + T = \{\text{left } x \mid x \in S\} \cup \{\text{right } x \mid x \in T\}$

■ Example:

$$S = \{1, 2, 3\} \quad T = \{3, 4\}$$

$$S + T = \{\text{left } 1, \text{left } 2, \text{left } 3, \text{right } 3, \text{right } 4\}$$



■ $\#(S + T) =$

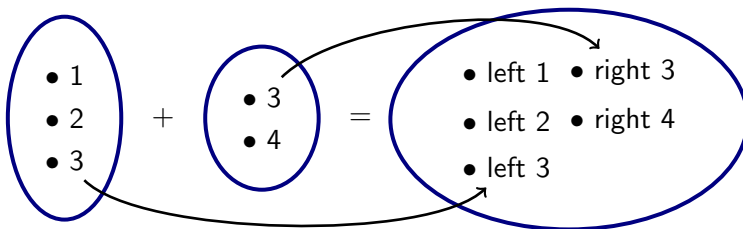
Disjoint Union

■ $S + T = \{\text{left } x \mid x \in S\} \cup \{\text{right } x \mid x \in T\}$

■ Example:

$$S = \{1, 2, 3\} \quad T = \{3, 4\}$$

$$S + T = \{\text{left } 1, \text{left } 2, \text{left } 3, \text{right } 3, \text{right } 4\}$$



■ $\#(S + T) = \#S + \#T$

■ C union's are disjoint union?

■ C: int + double:

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

■ C union's are not safe! Same storage is shared. Valid field is unknown:

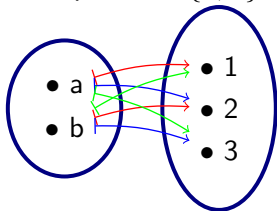
```
x.real=3.14; printf("%d\n",x.integer);
```

■ Haskell: Float + Int + (Int × Int):

```
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: $S = \{a, b\}$ $T = \{1, 2, 3\}$



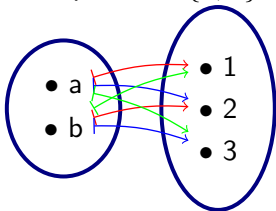
Each color is a value in the mapping. Other 6 values are not drawn

$$S \mapsto T = \{\{a \mapsto 1, b \mapsto 1\}, \{a \mapsto 1, b \mapsto 2\}, \{a \mapsto 1, b \mapsto 3\}, \\ \{a \mapsto 2, b \mapsto 1\}, \{a \mapsto 2, b \mapsto 2\}, \{a \mapsto 2, b \mapsto 3\}, \\ \{a \mapsto 3, b \mapsto 1\}, \{a \mapsto 3, b \mapsto 2\}, \{a \mapsto 3, b \mapsto 3\}\}$$

- $\#(S \mapsto T) =$

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: $S = \{a, b\}$ $T = \{1, 2, 3\}$



Each color is a value in the mapping. Other 6 values are not drawn

$$S \mapsto T = \{\{a \mapsto 1, b \mapsto 1\}, \{a \mapsto 1, b \mapsto 2\}, \{a \mapsto 1, b \mapsto 3\}, \\ \{a \mapsto 2, b \mapsto 1\}, \{a \mapsto 2, b \mapsto 2\}, \{a \mapsto 2, b \mapsto 3\}, \\ \{a \mapsto 3, b \mapsto 1\}, \{a \mapsto 3, b \mapsto 2\}, \{a \mapsto 3, b \mapsto 3\}\}$$

- $\#(S \mapsto T) = \#T^{\#S}$

Arrays

- `double a[3]={1.2,2.4,-2.1};`
 $a \in (\{0, 1, 2\} \mapsto \text{double})$
 $a = (0 \mapsto 1.2, 1 \mapsto 2.4, 2 \mapsto -2.1)$
- Arrays define a mapping from an integer range (or DOPT) to any other type
- **C:** $T \ x[N] \Rightarrow x \in (\{0, 1, \dots, N-1\} \mapsto T)$
- 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:

```
int f(int a) {  
    if (a%2 == 0) return 0;  
    else return 1;  
}
```

- $f : \text{int} \mapsto \{0, 1\}$

regardless of the function body: $f : \text{int} \mapsto \text{int}$

- Haskell:

```
f a = if mod a 2 == 0 then 0 else 1
```

- in C, f expression is a pointer type $\text{int} (*) (\text{int})$

in Haskell it is a mapping: $\text{int} \mapsto \text{int}$

Array and Function Difference

Arrays:

- Values stored in memory
- Restricted: only integer domain
- `double` \rightarrow `double` ?

Functions

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

Cartesian Mappings

- Cartesian mappings:
`double a[3][4];`
`double f(int m, int n);`
- Cartesian mapping versus mappings of mappings:
`int × int → double` and `int → (int → double)`
- For cartesian mapping, you need two or more values to get the value of the mapping
- In mapping of mappings, you get a new mapping as you supply a value.

Cartesian Mapping vs Nested mapping

■ Pascal arrays

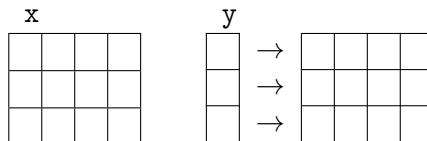
```
var
  x : array [1..3,1..4] of double;
  y : array [1..3] of array [1..4] of double;
...
x[1,3] := x[2,3]+1;      y[1,3] := y[2,3]+1;
```



Row operations:

y[1] := y[2] ; ✓

x[1] := x[2] ; ✗



■ Haskell functions:

```
f (x,y) = x+y  
g x y = x+y  
...  
f (3+2)  
g 3 2
```

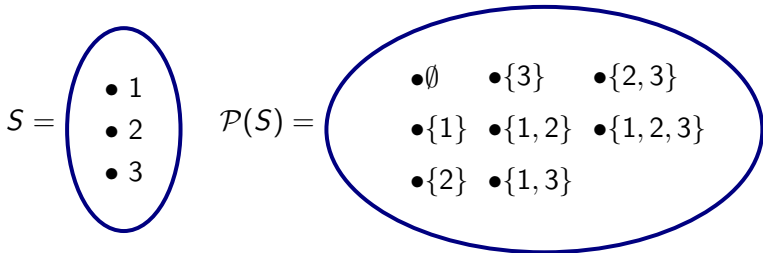
■ `g 3` ✓
`f 3` ✗

■ Reuse the old definition to define a new function:

```
increment = g 1  
increment 1  
2
```

Powerset

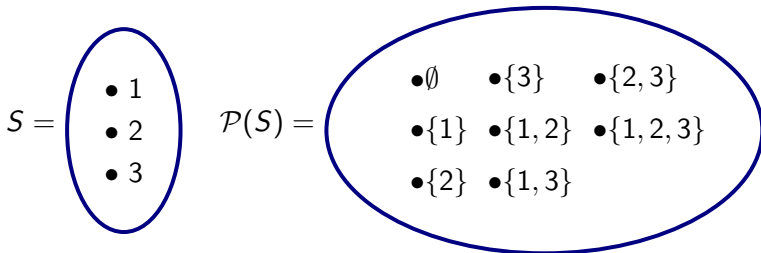
- $\mathcal{P}(S) = \{T \mid T \subseteq S\}$
- The set of all subsets
-



- $\#\mathcal{P}(S) =$

Powerset

- $\mathcal{P}(S) = \{T \mid T \subseteq S\}$
- The set of all subsets
-



- $\#\mathcal{P}(S) = 2^{\#S}$

- Set datatype is restricted and special datatype. Only exists in **Pascal** and special set languages like **SetL**
- set operations (Pascal)

```

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++** and **Python** implemented as class.

Recursive Types

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

Lists

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

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

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

- C lists: pointer based. Not actual recursion.

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

- C lists: pointer based. Not actual recursion.

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

- Haskell lists.

```
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.

- Polymorphic lists: a single definition defines lists of many types.
- $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

```
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] --}
```

- Polymorphic lists: a single definition defines lists of many types.
- $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

```
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] --}
```

- $Left(1, Left("ali", Left(15.23, Empty))) \in List\ \alpha$?

- Polymorphic lists: a single definition defines lists of many types.
- $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

```
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] --}
```

- $Left(1, Left("ali", Left(15.23, Empty))) \in List\ \alpha$?

- Polymorphic lists: a single definition defines lists of many types.
- $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

```
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] --}
```

- $Left(1, Left("ali", Left(15.23, Empty))) \in List\ \alpha$? No.
Most languages only permits homogeneous lists.

Haskell Lists

- binary operator “:” for list construction:
data [alpha] = (alpha : [alpha]) | []

Haskell Lists

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

Haskell Lists

- binary operator ":" for list construction:
data [alpha] = (alpha : [alpha]) | []
- `x = (1:(2:(3:[])))`
- Syntactic sugar:
`[1,2,3] ≡ (1:(2:(3:[])))`
`["ali"] ≡ ("ali":[])`

General Recursive Types

■ $T = \dots T \dots$

General Recursive Types

- $T = \dots T \dots$
- Formula requires a minimal solution to be representable:
 $S = \text{Int} \times S$
Is it possible to write a single value? No minimum solution here!

General Recursive Types

- $T = \dots T \dots$
- Formula requires a minimal solution to be representable:
 $S = \text{Int} \times S$
Is it possible to write a single value? No minimum solution here!
- List example:
 $x = \text{Left}(1, \text{Left}(2, x))$
 $x \in S?$

General Recursive Types

- $T = \dots T \dots$
- Formula requires a minimal solution to be representable:
 $S = \text{Int} \times S$
Is it possible to write a single value? No minimum solution here!
- List example:
 $x = \text{Left}(1, \text{Left}(2, x))$
 $x \in S?$

General Recursive Types

- $T = \dots T \dots$
- Formula requires a minimal solution to be representable:
 $S = \text{Int} \times S$
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 $[1, 2, 1, 2, 1, 2, \dots]$ value?
- Some languages like Haskell lets user define such values. All iterations go infinite. Useful in some domains though.

General Recursive Types

- $T = \dots T \dots$
- Formula requires a minimal solution to be representable:
 $S = \text{Int} \times S$
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 $[1, 2, 1, 2, 1, 2, \dots]$ 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.

$$\blacksquare \text{ Tree } \alpha = \text{empty} + \text{node } \alpha \times \text{Tree} \alpha \times \text{Tree} \alpha$$

$$\begin{aligned} \text{Tree } \alpha = & \{ \text{empty} \} \cup \{ \text{node}(x, \text{empty}, \text{empty}) \mid x \in \alpha \} \cup \\ & \{ \text{node}(x, \text{node}(y, \text{empty}, \text{empty}), \text{empty}) \mid x, y \in \alpha \} \cup \\ & \{ \text{node}(x, \text{empty}, \text{node}(y, \text{empty}, \text{empty})) \mid x, y \in \alpha \} \cup \\ & \{ \text{node}(x, \text{node}(y, \text{empty}, \text{empty}), \text{node}(z, \text{empty}, \text{empty})) \mid x, y, z \in \alpha \} \cup \dots \end{aligned}$$

■ $Tree\ \alpha = empty + node\ \alpha \times Tree\alpha \times Tree\alpha$

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

■ C++ (pointers and template definition)

```
template<class Alpha>
struct Tree {
    Alpha x;
    Tree *left, *right;
} root;
```

■ $Tree\ \alpha = empty + node\ \alpha \times Tree\alpha \times Tree\alpha$

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

■ C++ (pointers and template definition)

```
template<class Alpha>
struct Tree {
    Alpha x;
    Tree *left, *right;
} root;
```

■ 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:

Strings

- Language design choice:
 - 1 Primitive type (ML, Python):
Language keeps an internal table of strings

Strings

- Language design choice:
 - 1 Primitive type (ML, Python):
Language keeps an internal table of strings
 - 2 Character array (C, Pascal, ...)

Strings

- Language design choice:
 - 1 Primitive type (ML, Python):
Language keeps an internal table of strings
 - 2 Character array (C, Pascal, ...)
 - 3 Character list (Haskell, Prolog, Lisp)

Strings

- Language design choice:
 - 1 Primitive type (ML, Python):
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.

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.

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;
```

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:

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 → static type checking

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 → static type checking
 - 2 Run time → dynamic type checking

Static Type Checking

- Compile time type information is used to do type checking.

Static 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.

Static 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

Static 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:

Static 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,...)

Static 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...)

Static 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.

Dynamic Type Checking

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

Dynamic Type Checking

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

```
def whichmonth(inp):  
    if isinstance(inp, int):  
        return inp  
    elif isinstance(inp, str):  
        if inp == "January":  
            return 1  
        elif inp == "February":  
            return 2  
        ....  
        elif inp == "December":  
            return 12  
    ...  
inp = input()      /* user input at run time? */  
month=whichmonth(inp)
```

- Run time decision based on users 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

- $S \stackrel{?}{=} T$ How to decide?

Type Equality

- $S \stackrel{?}{=} T$ How to decide?
 - **Name Equivalence**: Types should be defined at the same exact place.

Type Equality

- $S \stackrel{?}{=} T$ 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).

Type Equality

- $S \stackrel{?}{=} T$ 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**.

Type Equality

- $S \stackrel{?}{=} T$ 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:

```
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

$S \equiv T$ 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 $S = A + B$, $T = A' + B'$, and $(A \equiv A' \text{ and } B \equiv B')$ or $(A \equiv B' \text{ and } B \equiv A')$,
- 4 if $S = A \mapsto B$, $T = A' \mapsto B'$, $A \equiv A'$ 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.

- Harder to implement structural equality. Especially recursive cases.
- $T = \{nil\} + A \times T$, $T' = \{nil\} + A \times T'$

- Harder to implement structural equality. Especially recursive cases.
- $T = \{nil\} + A \times T$, $T' = \{nil\} + A \times T'$

- Harder to implement structural equality. Especially recursive cases.
- $T = \{nil\} + A \times T$, $T' = \{nil\} + A \times T'$
 $T = \{nil\} + A \times T'$, $T' = \{nil\} + A \times T$
- `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.

- Harder to implement structural equality. Especially recursive cases.
- $T = \{nil\} + A \times T$, $T' = \{nil\} + A \times T'$
 $T = \{nil\} + A \times T'$, $T' = \{nil\} + A \times T$
- `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.

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


Type Completeness

- First order values:

Type Completeness

- First order values:
 - Assignment

Type Completeness

- First order values:
 - Assignment
 - Function parameter

Type Completeness

- First order values:
 - Assignment
 - Function parameter
 - Take part in compositions

Type Completeness

- First order values:
 - Assignment
 - Function parameter
 - Take part in compositions
 - Return value from a function

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)

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 .

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

- 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:

	Primitive	Array	Struct	Func.
Assignment	✓	×	✓	×
Function parameter	✓	×	✓	×
Function return	✓	×	✓	×
In compositions	✓	✓	✓	×

Haskell Types:

	Primitive	Array	Struct	Func.
Variable definition	✓	✓	✓	✓
Function parameter	✓	✓	✓	✓
Function return	✓	✓	✓	✓
In compositions	✓	✓	✓	✓

Pascal Types:

	Primitive	Array	Struct.	Func.
Assignment	✓	✓	✓	×
Function parameter	✓	✓	✓	×
Function return	✓	×	×	×
In compositions	✓	✓	✓	×

Expressions

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.

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

Aggregates

- Used to construct composite values without any declaration/definition. Haskell:

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

- Python:

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

■ Ansi C has aggregates

only at the definition. There is no aggregates in the statements!

```
struct Person { char name[20], int no };
struct Person p = {"Ali_Cin", 332314};
double arr[3][2] = {{0,1}, {1.2,4}, {12, 1.4}};
p={"Veli_Cin",123412}; × /* not possible in ANSI C!*/
```

■ C99 Compound literals allow array and structure aggregates

```
int (*arr)[2];
arr = {{0, 1}, {1.2,4}, {12, 1.4}}; ✓
p = (struct person) {"Veli_Cin",123412}; ✓ /* C99 */
```

■ C++11 has function aggregates (lambda)

```
void sort(int a[], int n, (*f)(int,int)) {
    ...
}
auto f = [](int a) { return a+1;} ;
...
sort(arr, n, [](int a, int b) { return a-b;});
n = f(n)
```

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

- $F(Gp_1, Gp_2, \dots, Gp_n)$
- 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.

Conditional Expressions

- Evaluate to different values based on a condition.
- Haskell: `if condition then exp1 else exp2` .
`case value of p1 -> exp1 ; p2 -> exp2 ...`

Conditional Expressions

- Evaluate to different values based on a condition.
- Haskell: *if condition then exp1 else exp2* .
 case value of p1 -> exp1 ; p2 -> exp2 ...
- C: *(condition)?exp1:exp2 ;*

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

Conditional Expressions

- Evaluate to different values based on a condition.
- Haskell: *if condition then exp1 else exp2* .
 case value of p1 -> exp1 ; p2 -> exp2 ...
- C: *(condition)?exp1:exp2 ;*

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

- Python: *exp1 if condition else exp2*

Conditional Expressions

- Evaluate to different values based on a condition.
- Haskell: *if condition then exp1 else exp2* .
 $\text{case value of } p1 \rightarrow exp1 ; p2 \rightarrow exp2 \dots$
- C: *(condition)?exp1:exp2 ;*

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

- Python: *exp1 if condition else exp2*
- *if .. else* in C is **not** conditional expression but conditional statement. No value when evaluated!

■ 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:

$\{ \text{expr} \mid \text{var} \in \text{list}, \text{condition} \}$

- Haskell:

```
x = [1,2,3,4,5,6,7,8,9,10,11,12]
y = [ a*2 | a <- x ]                {-- [2,4,6,8,...24 ] --}
z = [ a | a <- x, mod a 3 == 1 ]    {-- [1,4,7,10] --}
```

- Python:

```
x = [1,2,3,4,5,6,7,8,9,10,11,12]
y = [ a*2 for a in x ]              # [2,4,6,8,...24 ]
z = [ a for a in x if a % 3 == 1 ]  # [1,4,7,10]
z = { a:a*a for a in x }           # {1:1,4:16,7:49,10:100}
```

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.

- ML has similar block expression syntax.
- This allows arbitrary computation for evaluation of the expression.

Summary

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