

# 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.  
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  
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)
- Disjoint union (union (C), variant record (pascal), Data (haskell))
- Mapping (arrays, functions)
- Powerset (set datatype (Pascal))
- Recursive compositions (lists, trees, complex data structures)

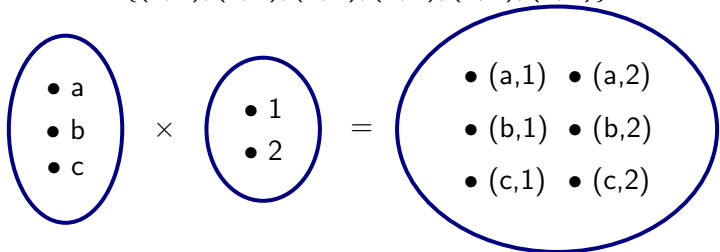
# Cartesian Product

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

■ Example:

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

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



■  $\#(S \times T) = \#S \cdot \#T$



- 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

- $S^n = \overbrace{S \times S \times S \times \dots \times S}^n$   
double<sup>4</sup> :

```
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.

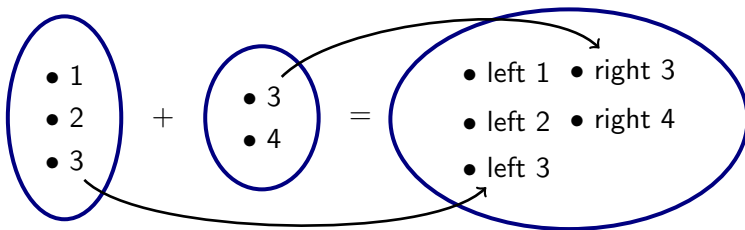
# Disjoint Union

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

■ Example:

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

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



$$\blacksquare \#(S + T) = \#S + \#T$$

■ C union's are disjoint union?

- **C:**  $\text{int} + \text{double}$ :

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

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

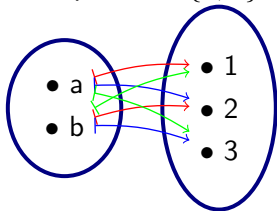
```
x.real=3.14; printf("%d\n",x.integer);
```

- **Haskel:**  $\text{Float} + \text{Int} + (\text{Int} \times \text{Int})$ :

```
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

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

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

# 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 `int (*)(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
- $\text{double} \mapsto \text{double} ?$

## Functions

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

- 
- Cartesian mappings:

```
double a[3][4];
double f(int m, int n);
```

- $\text{int} \times \text{int} \mapsto \text{double}$  and  $\text{int} \mapsto (\text{int} \mapsto \text{double})$

# Cartesian Mapping vs Nested mapping

## ■ Pascal arrays

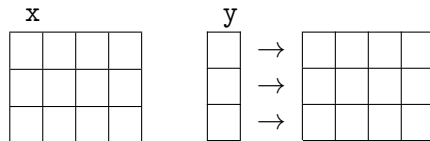
```
var
  x : array [1..3,1..4] of double;
  y : array [1..3] of array [1..4] of double;
...
x[1,3] := x[2,3]+1;      y[1,3] := y[2,3]+1;
```



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
```

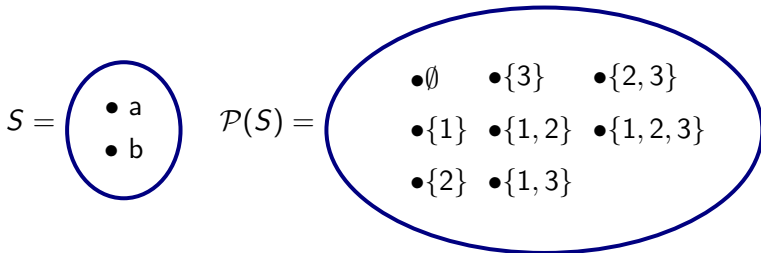
- `g 3` ✓  
`f 3` ✗

- 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
- 



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

- 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;
```

- 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.
- $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

```
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]) | []`
- `x = (1:(2:(3:[])))`
- Syntactic sugar:  
`[1,2,3] ≡ (1:(2:(3:[])))`  
`["ali"] ≡ ("ali":[])`

# 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.

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

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

- 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:

- 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.
- 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.
- 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.
- 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?
  - **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.
- $T = \{nil\} + A \times T$  ,  $T' = \{nil\} + A \times T'$   
 $T = \{nil\} + A \times T'$  ,  $T' = \{nil\} + A \times T$
- `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:
  - 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.
- 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`
- `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]
```

# 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

# Programming Languages

## Variables and Storage

Onur Tolga Şehitoğlu

Computer Engineering



# Outline

## 1 Storage

- Array Variables

## 2 Semantics of Assignment

## 3 Variable Lifetime

- Global Lifetime
- Local Lifetime
- Heap Variable Lifetime
- Dangling Reference and Garbage

- Persistent Variable Lifetime

## 4 Memory Management

## 5 Commands

- Assignment
- Procedure Call
- Block commands
- Conditional commands
- Iterative statements

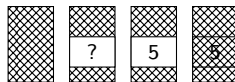
## 6 Summary

# Storage

- Functional language variables: math like, defined or solved. Remains same afterwards.
- Imperative language variables: variable has a state and value. It can be assigned to different values in same phrase.
- Two basic operations on a variable: [inspect](#) and [update](#).

Computer memory can be considered as a collection of **cells**.

- Cells are initially **unallocated**.
- Then, **allocated/undefined**.  
Ready to use but value unknown.
- Then, **storable**
- After the including block terminates, again **unallocated**



```
f();
void f() {
    int x;
    ...
    x=5;
    ...
    return;
}
```

# Total or Selective Update

- Composite variables can be inspected and updated in total or selectively

```
■  
struct Complex { double x,y; } a, b;  
...  
a=b;           // Total update  
a.x=b.y*a.x;   // Selective update
```

- Primitive variables: single cell  
Composite variables: nested cells

# Array Variables

Different approaches exist in implementation of array variables:

- 1 Static arrays
- 2 Dynamic arrays
- 3 Flexible arrays

# Static arrays

- Array size is fixed at compile time to a constant value or expression.
- C example:

```
#define MAXELS 100  
int a[10];  
double x[MAXELS*10][20];
```



# Dynamic arrays

- Array size is defined when variable is allocated. Remains constant afterwards.
- Example: C90/GCC (not in ANSI)

```
int f(int n) {  
    double a[n]; ...  
}
```

- Example: C++ with [templates](#)

```
template<class T> class Array {  
    T *content;  
public:  
    Array(int s) { content=new T[s]; }  
    ~Array()     { delete [] content; }  
};  
...  
Array<int>    a(10);  
              Array<double> b(n);
```

# Flexible arrays

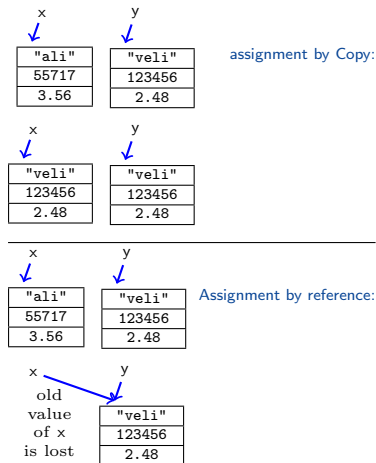
- Array size is completely variable. Arrays may expand or shrink at run time. Script languages like Perl, PHP, Python
- Perl example:

```
@a=(1,3,5);           # array size: 3
print $#a , "\n";     # output: 2 (0..2)
$a[10] = 12;          # array size 11 (intermediate elements undefined)
$a[20] = 4;            # array size 21
print $#a , "\n";     # output: 20 (0..20)
delete $a[20];        # last element erased, size is 11
print $#a , "\n";     # output: 10 (0..10)
```

- C++ and object orient languages allow overload of [] operator to make flexible arrays possible. STL (Standard Template Library) classes in C++ like `vector`, `map` are like such flexible array implementations.

# Semantic of assignment in composite variables

- Assignment by **Copy** vs **Reference**.
- **Copy**: All content is copied into the other variables storage. Two copies with same values in memory.
- **Reference**: Reference of variable is copied to other variable. Two variables share the same storage and values.



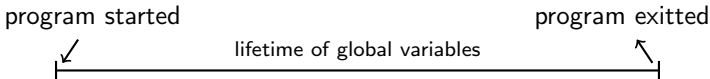
- Assignment semantics is defined by the language design
- C structures follows copy semantics. Arrays cannot be assigned. Pointers are used to implement reference semantics. C++ objects are similar.
- Java follows copy semantics for primitive types. All other types (objects) are reference semantics.
- Copy semantics is slower
- Reference semantics cause problems from storage sharing (all operations effect both variables). Deallocation of one makes the other invalid.
- Java provides copy semantic via a member function called `copy()`. Java garbage collector avoids invalid values (in case of deallocation)

# Variable Lifetime

- **Variable lifetime:** The period between allocation of a variable and deallocation of a variable.
- 4 kinds of variable lifetime.
  - 1 Global lifetime (while program is running)
  - 2 Local lifetime (while declaring block is active)
  - 3 Heap lifetime (arbitrary)
  - 4 Persistent lifetime (continues after program terminates)

# Global lifetime

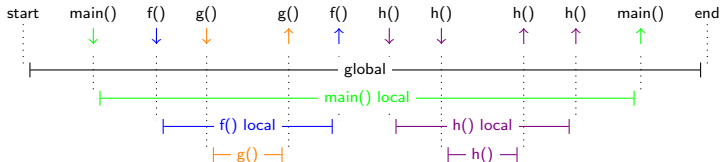
- Life of global variables start at program startup and finishes when program terminates.
- In C, all variables not defined inside of a function (including `main()`) are global variables and have global lifetime:



- What are static variables inside functions in C?

# Local lifetime

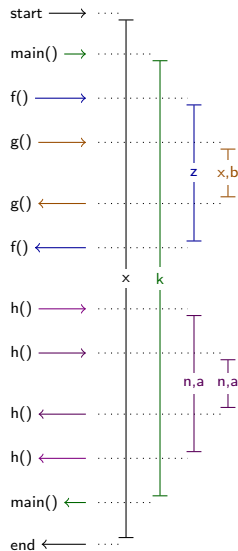
- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.
- Formal parameters are local variables.
- Multiple instances of same local variable may be alive at the same time in recursive functions.



```

double x;
int h(int n) {
    int a;
    if (n<1) return 1
    else return h(n-1);
}
void g() {
    int x;
    int b;
    ...
}
int f() {
    double z;
    ...
    g();
    ...
}
int main() {
    double k;
    f();
    ...
    h(1);
    ...;
    return 0;
}

```





# Heap Variable Lifetime

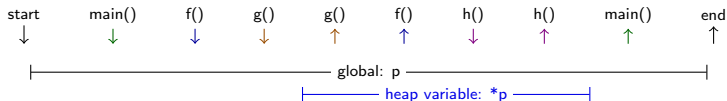
- **Heap variables:** Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.
- Heap variables are accessed via pointers. Some languages use references

```
double *p;  
p=malloc(sizeof(double));  
*p=3.4; ...  
  
free(p);
```
- **p and \*p are different variables** p has pointer type and usually a local or global lifetime, \*p is heap variable.
- heap variable lifetime can start or end at anytime.

```

double *p;
int h() { ...
}
void g() { ...
    p=malloc(sizeof(double));
}
int f() { ...
    g(); ...
}
int main() { ...
    f();    ...
    h();    ...;
    free(p); ...
}

```



# Dangling Reference

- **dangling reference**: trying to access a variable whose lifetime is ended and already deallocated.

```

char *p, *q;

p=malloc(10);
q=p;
...
free(q);
printf("%s",p);

char *f() {
    char a[]="ali";
    ....
    return a;
}
....
char *p;
p=f();
printf("%s",p);

```

- both p's are deallocated or ended lifetime variable, thus dangling reference
- sometimes operating system tolerates dangling references. Sometimes generates run-time errors like "protection fault", "segmentation fault" are generated.

# Garbage variables

- **garbage variables:** The variables with lifetime still continue but there is no way to access.

```
char *p, *q;  
...  
p=malloc(10);  
p=q;  
...  
  
void f() {  
    char *p;  
    p=malloc(10); ...  
    return  
}  
...  
f();
```

- When the pointer value is lost or lifetime of the pointer is over, heap variable is inaccessible. (\*p in examples)

# Garbage collection

- A solution to dangling reference and garbage problem:  
PL does management of heap variable deallocation automatically. This is called **garbage collection**. (Java, Lisp, ML, Haskell, most functional languages)
- no call like `free()` or `delete` exists.
- Language runtime needs to:
  - Keep a reference counter on each reference, initially 1.
  - Increment counter on each new assignment
  - Decrement counter at the end of the reference lifetime
  - Decrement counter at the overwritten/lost references
  - Do all these operations recursively on composite values.
  - When reference count gets 0, deallocate the heap variable

- Garbage collector deallocates heap variables having a reference count 0.
- Since it may delay execution of tasks, GC is not immediately done.
- GC usually works in a separate thread, in low priority, works when CPU is idle.
- Another but too restrictive solution to garbage: Reference cannot be assigned to a longer lifetime variable. local variable references cannot be assigned to global reference/pointer.

# Persistent variable lifetime

- Variables with lifetime continues after program terminates: file, database, web service object,...
- Stored in secondary storage or external process.
- Only a few experimental language has transparent persistence. Persistence achieved via IO instructions  
C files: `fopen()`, `fseek()`, `fread()`, `fwrite()`
- In object oriented languages; [serialization](#): Converting object into a binary image that can be written on disk or sent over the network.
- This way objects snapshot can be taken, saved, restored and object continue from where it remains.

# Memory Management

- Memory management of variables involves architecture, operating system, language runtime and the compiler.
- A typical OS divides memory in sections (segments):
  - Stack section: run time stack
  - Heap section: heap variables
  - Data section: global variables
  - Code section: executable instructions, read only.
- Global variables are fixed at compile time and they are put in data section.
- Heap variables are stored in the dynamic data structures in heap section. Heap section grows and shrinks as new variables are allocated and deallocated.
- Heap section is maintained by language runtime. For C, it is `libc`.



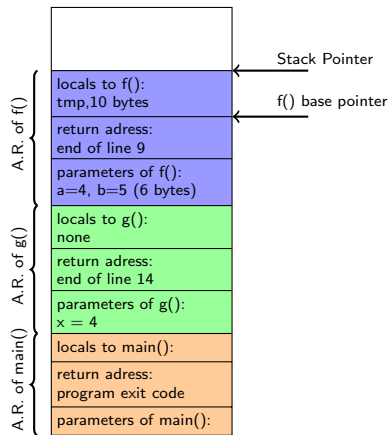
# Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use **run-time stack**. Each function call will introduce an **activation record** to store its local context. It is also called **stack frame**, **activation frame**.
- In a typical architecture, **activation record** contains:
  - Return address. Address of the next instruction after the caller.
  - Parameter values.
  - A reserved area for local variables.

```

1  int f(short a, int b) {
2      char tmp[10];
3      ...
4      return a+b;
5  }
6  int g(int x) {
7      int tmp, p;
8      ...
9      tmp = f(x, x+1);
10     ...
11     return tmp+p;
12 }
13 int main() {
14     return g(4);
15 }

```



# Function Call

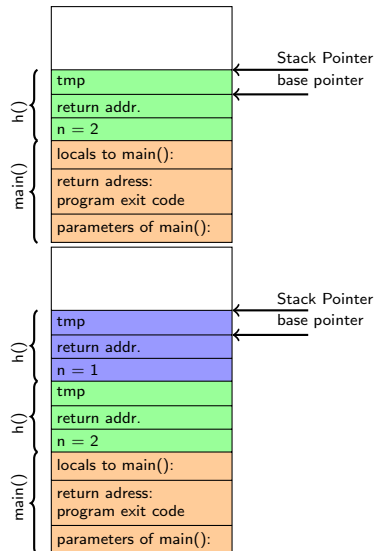
- Caller side:
  - 1 Push parameters
  - 2 Push return address and jump to function code start (usually a single CPU instruction like `callq`)
- Function entry:
  - 1 Set base pointer to current stack pointer
  - 2 Advance stack pointer to size of local variables
- Function body can access all local variables relative to base pointer
- Function return:
  - 1 Set stack pointer to base pointer
  - 2 Pop return address and jump to return address (single CPU instruction like `retq`)
- Caller side after return:
  - 1 Recover stack pointer (remove parameters on stack)
  - 2 Get and use return value if exists (typically from a register)

- All locals and parameters have the same offset from base pointer
- Recursive calls execute same instructions

```

1  int h(int n) {
2      int tmp;
3      if (n <= 1) return 0;
4      else {
5          tmp = h(n-1);
6          return n+tmp;
7      }
8  }
9  int main() {
10     printf("%d\n", h(2));
11     return 0;
12 }

```



- Order of values in the activation record may differ for different languages
- Registers are used for passing primitive value parameters instead of stack
- Garbage collecting languages keep references on stack with actual variables on heap
- Languages returning nested functions as first order values require more complicated mechanisms

```
def multiplier(a):  
    def f(x):  
        return a*x  
    return f  
  
twice = multiplier(2)  
# multiplier/a is not alive anymore but twice=f is using it  
print(twice(14))
```

# Commands

**Expression:** program segment with a value.

**Statement:** program segment without a value, but alters the state.

Input, output, variable assignment, iteration...

- 1 Assignment
- 2 Procedure call
- 3 Block commands
- 4 Conditional commands
- 5 Iterative commands

# Assignment

- C: “Var = Expr;”, Pascal “Var := Expr;”.
- Evaluates RHS expression and sets the value of the variable at RHS
- $x = x + 1$ . LHS  $x$  is a variable reference (l-value), RHS is the value
- **multiple assignment:**  $x=y=z=0$ ;
- **parallel assignment:** (Perl, PHP)  $(\$a, \$b) = (\$b, \$a)$ ;  
 $(\$name, \$surname, \$no) =$   
 $(\text{"Onur"}, \text{"Şehitoğlu"}, 55717)$ ;  
Assignment: “reference aggregate”  $\rightarrow$  “value aggregate”
- **assignment with operator:**  $x += 3$ ;  $x *= 2$ ;

# Procedure call

- **Procedure:** user defined commands. Pascal: `procedure`, C: `function` returning `void`
- `void funcname(param1 , param2 , ... , paramn)`
- Usage is similar to functions but call is in a statement position (on a separate line of program)



# Block commands

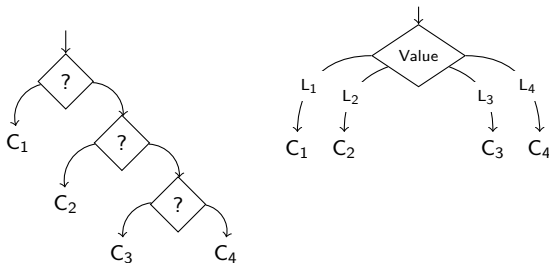
- Composition of a block from multiple statements
- **Sequential commands:**  $\{ C_1 ; C_2 ; \dots ; C_n \}$   
A command is executed, after it finishes the next command is executed,...
- Commands enclosed in a block behaves like single command: “if” blocks, loop bodies,...
- **Collateral commands:**  $\{ C_1, C_2, \dots, C_n \}$  (not C ',')!  
Commands can be executed in any order.
- The order of execution is non-deterministic. Compiler or optimizer can choose any order. If commands are independent, effectively deterministic:  
'y=3 , x=x+1 ;' vs 'x=3, x=x+1 ;'
- Can be executed in parallel.

- **Concurrent commands:** concurrent paradigm languages:  
 $\{ C_1 \mid C_2 \mid \dots \mid C_n \}$
- All commands start concurrently in parallel. Block finishes when the last active command finishes.
- Real parallelism in multi-core/multi-processor machines.
- Transparently handled by only a few languages. Thread libraries required in languages like Java, C, C++.

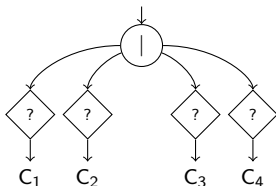
```
void producer(...) {....}  
void collectgarbage(...) {....}  
void consumer(...) {....}  
int main() {  
    ...  
    pthread_create(tid1, NULL, producer, NULL);  
    pthread_create(tid2, NULL, collectgarbage, NULL);  
    pthread_create(tid3, NULL, consumer, NULL);  
    ...  
}
```

# Conditional commands

- Commands to choose between alternative commands based on a condition
- in  $C$  : `if (cond) C1 else C2 ;`  
`switch (value) { case L1 : C1 ; case L2 : C2 ; ... }`
- if commands can be nested for multi-conditioned selection.
- switch like commands chooses statements based on a value



- **non-deterministic conditionals:** conditions are evaluated in collaterally and commands are executed if condition holds.
- **hyphotetically:**  
 if ( $cond_1$ )  $C_1$  or if ( $cond_2$ )  $C_2$  or if ( $cond_3$ )  $C_3$  ;  
  
 switch ( $val$ ) {  
     case  $L_1$ :  $C_1$  | case  $L_2$ :  $C_2$  | case  $L_3$ :  $C_3$  }
- Tests can run concurrently. First test evaluating to **True** wins. Others discarded.

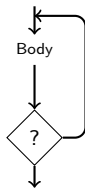
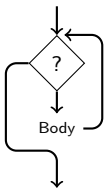


# Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.

- Loop classification: minimum number of iteration: 0 or 1.

C: while (...) { ... }      C: do {...} while (...);



- Definite vs indefinite loops
- **Indefinite iteration:** Number of iterations of the loop is not known until loop finishes
- C loops are indefinite iteration loops.
- **Definite iteration:** Number of iterations is fixed when loop started.
- Pascal for loop is a definite iteration loop.  
for  $i := k$  to  $m$  do begin .... end; has  $(m - k + 1)$  iterations.  
Pascal forbids update of the loop index variable.
- List and set based iterations: PHP, Perl, Python, Shell

```
$colors=array('yellow','blue','green','red','white');  
foreach ($colors as $i) {  
    print $i," is a color","\n";  
}
```

# Summary

- Variables with storage
- Variable update
- Lifetime: global, local, heap, persistent
- Commands

# Programming Language Concepts/Binding and Scope

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



# Outline

## 1 Binding

## 2 Environment

## 3 Block Structure

- Monolithic block structure
- Flat block structure
- Nested block structure

## 4 Hiding

## 5 Static vs Dynamic

### Scope/Binding

- Static binding
- Dynamic binding

## 6 Binding Process

## 7 Declarations

- Definitions and Declarations
- Sequential Declarations
- Collateral Declarations
- Recursive declarations
- Recursive Collateral Declarations
- Block Expressions
- Block Commands
- Block Declarations

## 8 Summary

# Binding

- Most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called **identifiers**.
- definition of an identifier  $\leftrightarrow$  used position of an identifier.  
Formally: binding occurrence  $\leftrightarrow$  applied occurrence.
- Identifiers are declared once, used  $n$  times.
- Language should map which corresponds to which.
- **Binding**: Finding the corresponding binding occurrence (definition/declaration) for an applied occurrence (usage) of an identifier.

for binding:

- 1 **Scope of identifiers** should be known. What is the block structure? Which blocks the identifier is available.
- 2 What will happen if we use same identifier name again  
 “C forbids reuse of same identifier name in the same scope.  
 Same name can be used in different nested blocks. The identifier inside **hides** the outside identifier”.

```
double f,y;
int f() {  × error!
    ...
}
double y; × error!
```

```
double y;
int f() {
    double f;  ✓ OK
    int y ;    ✓ OK.
}
```

# Environment

- **Environment:** The set of binding occurrences that are accessible at a point in the program.
- **Example:**

```
struct Person { ... } x;
int f(int a) {
    double y;
    int x;
    ... ①
}
int main() {
    double a;
    ... ②
}
```

$O(①) = \{\text{struct Person} \mapsto \text{type},$   
 $x \mapsto \text{int}, f \mapsto \text{func}, a \mapsto \text{int},$   
 $y \mapsto \text{double}\}$

$O(②) = \{\text{struct Person} \mapsto \text{type},$   
 $x \mapsto \text{struct Person}, f \mapsto \text{func},$   
 $a \mapsto \text{double}, \text{main} \mapsto \text{func}\}$

# Block Structure

- Program blocks define the scope of the identifiers declared inside. (boundary of the definition validity) For variables, they also define the lifetime.
- Languages may have different block structures:

**C** function definitions and command blocks (`{ ... }`) define local scopes. Also each source code define a block.

**Java** Class definitions, class member function definitions, block commands define local scopes. Nested function definitions and namespaces possible.

**Haskell** '`let definitions in expression`' defines a block expression. Also '`expression where definitions`' defines a block expression. (the definitions have a local scope and not accessible outside of the expression)

- Block structure of the language is defined by the organization of the blocks.

# Monolithic block structure

- Whole program is a block. All identifiers have global scope starting from the definition.
- **Cobol** is a monolithic block structure language.

```
int x;  
int y;  
...  
...
```

- In a long program with many identifiers, they share the same scope and they need to be distinct.

# Flat block structure

- Program contains the global scope and only a single level local scope of function definitions. No further nesting is possible.
- **Fortran** and partially **C** has flat block structure.

```
int x;  
int y;  
int f()  
{  
  int a;  
  double b;  
  ...  
}  
int g()  
{  
  int a;  
  double b;  
  ...  
}  
....
```

# Nested block structure

- Multiple blocks with nested local scopes can be defined.
- **Pascal** and **Java** have nested block structure.



- C block commands can be nested.
- GCC extensions (**Not C99 standard!**) to C allow nested function definitions.



# Hiding

- Identifiers defined in the inner local block hides the outer block identifiers with the same name during their scope. They cannot be accessed within the inner block.

```
int x,y;
int f(double x) {
    ...           // parameter x hides global x in f()
}
int g(double a) {
    int y;         // local y hides global y in g()
    double f;      // local f hides global f() in g()
    ...
}
int main() {
    int y;         // local y hides global y in main()
}
```

# Static vs Dynamic Scope/Binding

The binding and scope resolution is done at compile time or run time? Two options:

- 1 Static binding, static scope
  - 2 Dynamic binding, dynamic scope
- First defines scope and binding based on the lexical structure of the program and binding is done at compile time.
  - Second activates the definitions in a block during the execution of the block. The environment changes dynamically at run time as functions are called and returned.

# Static binding

- Programs shape is significant. Environment is based on the position in the source (lexical scope)
- Most languages apply static binding (C, Haskell, Pascal, Java, ...)

```

int x=1,y=2;
int f(int y) {
    y=x+y;                /* x global, y local */
    return x+y;
}
int g(int a) {
    int x=3;              /* x local, y global */
    y=x+x+a;      x=x+y;   y=f(x);
    return x;
}
int main() {
    int y=0;      int a=10;      /* x global y local */
    x=a+y;      y=x+a;      a=f(a);      a=g(a);
    return 0;
}

```

```
int x=1 y=2;
```

# Dynamic binding

- Functions called update their declarations on the environment at **run-time**. Delete them on return. Current stack of activated blocks is significant in binding.
- Lisp and some script languages apply dynamic binding.

```

1  int x=1,y=2;
2  int f(int y) {
3      y=x+y;
4      return x+y;
5  }
6  int g(int a) {
7      int x=3;
8      y=x+x+a;  x=x+y;
9      y=f(x);
10     return x;
11 }
12 int main() {
13     int y=0;  int a=10;
14     x=a+y;    y=x+a;
15     a=f(a);   a=g(a);
16     return 0;
17 }

```

	Trace	Environment (without functions)
	initial	{x:GL, y:GL }
12	call main	{x:GL, y:main, a:main }
15	call f(10)	{x:GL, y:f , a:main }
4	return f : 30	back to environment before f
15	in main	{x:GL, y:main, a:main }
15	call g(30)	{x:g, y:main, a:g }
9	call f(39)	{x:g, y:f, a:g }
4	return f : 117	back to environment before f
9	in g	{x:g, y:main, a:g }
10	return g : 39	back to environment before g
15	in main	{x:GL, y:main, a:main }
16	return main	x:GL=10, y:GL=2, y:main=117, a:main=39

- It gets more complicated if nested functions are allowed:

```
let x = 5
  y = 10
  func x = x * x + other y
  other x = x + x
in func x + let other x = x - 1
  y = 2
  in func x
```

# Binding Process

- Language processor keeps track of current environment in a data structure called **Symbol Table** or **Identifier Table**
- Symbol table maps identifier strings to their type and binding.
- Each new block introduces its declarations/bindings to the symbol table and on exit, they are cleared.
- Usually implemented as a Hash Table.
- For static binding, Symbol Table is a compile time data structure and maintained during different stages of compilation.
- For dynamic binding, symbol table is maintained at run time.

# Declarations

- Definitions vs Declarations
- Sequential declarations
- Collateral declarations
- Recursive declarations
- Collateral recursive declarations
- Block commands
- Block expressions

# Definitions and Declarations

- **Definition:** Creating a new name for an existing binding.
- **Declaration:** Creating a completely new binding.
- in C: `struct Person` is a declaration. `typedef struct Person persontype` is a definition.
- in C++: `double x` is a declaration. `double &y=x;` is a definition.
- creating a new entity or not. Usually the distinction is not clear and used interchangeably.



# Sequential Declarations

- $D_1 ; D_2 ; \dots ; D_n$
- Each declaration is available starting with the next line.  $D_1$  can be used in  $D_2$  and afterwards,  $D_2$  can be used in  $D_3$  and afterwards,...
- Declared identifier is not available in preceding declarations.
- Most programming languages provide only such declarations.

# Collateral Declarations

- `Start; D1 and D2 and ... and Tn ; End`
- Each declaration is evaluated in the environment preceding the declaration group. Declared identifiers are available only after all finish.  $D_1, \dots, D_n$  uses in the environment of `Start`. They are available in the environment of `End`.
- ML allows collateral declarations additionally.

# Recursive declarations

- Declaration:Name = Body
- The body of the declaration can access the declared identifier. Declaration is available in the body of itself.
- C functions and type declarations are recursive. Variable definitions are usually not recursive. ML allows programmer to choose among recursive and non-recursive function definitions.

# Recursive Collateral Declarations

- All declarations can access the others regardless of their order.
- All Haskell declarations are recursive collateral (including variables)
- All declarations are mutually recursive.
- ML allows programmer to do such definitions.
- C++ class members are like this.
- in C a similar functionality can be achieved by prototype definitions.

# Block Expressions

- Allows an expression to be evaluated in a special local environment. Declarations done in the block is not available outside.
- in Haskell: `let D1; D2; ... ; Dn in Expression or Expression where D1; D2; ... ; Dn`

```
x=5
t=let xsquare=x*x
    factorial n = if n<2 then 1 else n*factorial (n-1)
    xfact = factorial x
  in (xsquare+1)*xfact/(xfact*xsquare+2)
```

- Hiding works in block expressions as expected:

```
x=5 ; y=6 ; z = 3
t=let x=1
  in let y=2
    in x+y+z
{-- t is 1+2+3 here. local x and y hides the ones above --}
```

- GCC (only GCC) block expressions has the last expression in block as the value:

```
double min ;
...
min = ({ double tmp;
        if (b < a) then {
            tmp = a;  a = b ; b = tmp;
        }
        a; // this is the value of the block
    });
```

# Block Commands

- Similar to block expressions, declarations done inside a block command is available only during the block. Statements inside work in this environment. The declarations lost outside of the block.



```
int x=3, i=2;
x += i;
while (x>i) {
    int i=0;
    ...
    i++;
}
/* i is 2 again */
```

# Block Declarations

- A declaration is made in a local environment of declarations. Local declarations are not made available to the outer environment.
- in Haskell:  $D_{exp}$  where  $D_1; D_2; \dots ; D_n$   
Only  $D_{exp}$  is added to environment. Body of  $D_{exp}$  has all local declarations available in its environment.

```
fifthpower x = (forthpowerx) * x where  
    squarex = x*x  
    forthpowerx = squarex*squarex
```



# Summary

- Binding, scope, environment
- Block structure
- Hiding
- Static vs Dynamic binding
- Declarations
- Sequential, recursive, collateral
- Expression, command and declaration blocks

# Programming Language Concepts

## Abstraction

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



# Outline

## 1 Abstraction

- Function and Procedure Abstractions
- Selector Abstraction
- Generic Abstraction
- Iterator Abstraction
- Iterator Abstraction

## 2 Abstraction Principle

## 3 Parameters

## 4 Parameter Passing Mechanisms

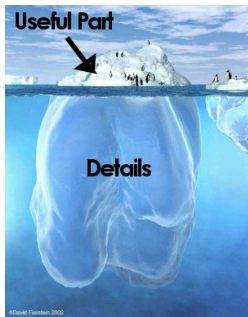
- Copy Mechanisms
- Binding Mechanisms
- Pass by Name

## 5 Evaluation Order

## 6 Infinite Values

## 7 Correspondence Principle

# Abstraction



- Iceberg: Details at the bottom, useful part at the top of the ocean. Animals do not care about the bottom.
  - User: "how do I use it?", Developer: "How do I make it work?"
  - User: "what does it do?", Developer: "How does it do that?"
- 
- **Abstraction:** Make a program or design reusable by enclosing it in a body, hiding the details, and defining a mechanism to access it.
  - Separating the usage and implementation of program segments.
  - Vital large scale programming.

- Abstraction is possible in any discipline involving design:
- radio tuner. Adjustment knob on a radio is an abstraction over the tuner element, frequency selection.
- An ATM is an abstraction over complicated set of bank transaction operations.
- Programming languages can be considered as abstraction over machine language.
- ...

# Purpose

- Details are confusing
- Details may contain more error
- Repeating same details increase complexity and errors
- Abstraction philosophy: **Declare once, use many times!**
- **Code reusability** is the ultimate goal.
- Parameterization improves power of abstraction

# Function and procedure abstractions

- The computation of an expression is the detail (algorithm, variables, etc.)
- Function call is the usage of the detail
- Functions are **abstractions over expressions**
- void functions of C or procedure declarations of some languages
- No value but contains executable statements as detail.
- Procedures are **abstractions over commands**
- Other type of abstractions possible?

# Selector abstraction

- arrays: `int a[10][20]; a[i]=a[i]+1;`
- `[..]` operator selects elements of an array.
- User defined selectors on user defined structures?
- Example: Selector on a linked list:

```
struct List {
    int data;
    List *next;
    int & operator[](int el) {
        int i; List *p = this;
        for (i = 1 ; i < el ; i++)
            p = p->next;          /* take the next element */
        return p->data;
    };
    ...
};
List h;
...
h[1] = h[2] + 1;
```

- C++ allows overloading of `[]` operator for classes.



- Python `__setitem__ (k,v)` implements l-value, `__getitem__ (k)` r-value selector.

```
class BSTree:
    def __init__(self):
        self.node = None
    def __getitem__(self, key):
        if self.node == None:
            raise KeyError
        elif key < self.node[0]: return self.left[key]
        elif key > self.node[0]: return self.right[key]
        else: return self.node[1]
    def __setitem__(self, key, val):
        if self.node == None:
            self.node = (key, val)
            self.left = BSTree() # empty tree
            self.right = BSTree() # empty tree
        elif key < self.node[0]: self.left[key] = val
        elif key > self.node[0]: self.right[key] = val
        else: self.node = (key, val)

a = BSTree()
a["hello"] = 4
a["world"] = a["hello"] + 5
```

```

class BST {
    struct Node { string key; double val;
                 Node *left, *right; } *node;
public:
    BST() { node = NULL; };
    double & operator[] (const string &k) {
        Node **parent = NULL, *p = node, *newnode;
        while (p != NULL) {
            if (k < p->key) {
                parent = &p->left; p = p->left;
            } else if (k > p->key) {
                parent = &p->right; p = p->right;
            } else return p->val;
        }
        newnode = new Node;
        newnode->left = newnode->right = NULL;
        newnode->key = k;
        if (parent == NULL) node = newnode;
        else *parent = newnode;
        return newnode->val;
    }
};

BST a;
a["carrot"] = 3; a["onion"] = 4;
a["patato"] = a["onion"] + 2;

```

# Generic abstraction

- Same declaration pattern applied to different data types.
- **Abstraction over declaration.** A function or class declaration can be adapted to different types or values by using type or value parameters.

```
template <class T>
  class List {
      T content;
      List *next;
  public: List() { next=NULL };
      void add(T el) { ... };
      T get(int n) { ...};
  };

template <class U>
  void swap(U &a, U &b) { U tmp; tmp=a; a=b; b=tmp; }
...
List<int> a; List<double> b; List<Person> c;
int t,x; double v,y; Person z,w;
swap(t,x); swap(v,y); swap(z,w);
```

# Iterator abstraction

- Iteration over a user defined data structure. [Ruby](#) example:

```
class Tree
  def initialize(v)
    @value = v ; @left = nil ; @right = nil
  end
  def traverse
    @left.traverse {|v| yield v} if @left != nil
    yield @value           # block argument replaces
    @right.traverse {|v| yield v} if @right != nil
  end
end

a=Tree.new(3) ; l=[]
a.traverse { |node|      # yield param
               print node # yield body
               l << node  # yield body
            }
```

# Iterator abstraction

- Iteration over a user defined data structure. **Python** generator example:

```
class BSTree(object):
    def __init__(self):
        self.val = ()
    def inorder(self):
        if self.val == ():
            return
        else:
            for i in self.left.inorder():
                yield i
            yield self.val
            for i in self.right.inorder():
                yield i

v = BSTree()
...
for v in v.inorder():
    print v
```

# C++ iterators

- C++ Standard Template Library containers support **iterators**
- `begin()` and `end()` methods return iterators to start and end of the data structure
- Iterators can be dereferenced as `*iter` or `iter->member`.
- '+' operation on an iterator skips to the next value.

- ```
for (ittype it = a.begin(); it != a.end(); ++it) {
    // use *it or it->member it->method() in body
}
```

- C++11 added:

```
for (valtype & i : a ) {
    // use directly i as l-value or r-value.
}
```

This syntax is equivalent to:

```
for (ittype it = a.begin() ; it != a.end(); it++) {
    valtype & i = *it;
    // use directly i as l-value or r-value
}
```

# C++ iterators

```
template<class T> class List {
    struct Node { T val; Node *next;} *list;
public: List() { list = nullptr;}
    void insert(const T& v) { Node *newnode = new Node;
        newnode->next = list; newnode->val = v; list = newnode;}
    class Iterator {
        Node *pos;
    public: Iterator(Node *p) { pos = p;}
        T & operator*() { return pos->val; }
        void operator++() { pos = pos->next; }
        bool operator!=(const Iterator &it) { return pos != it.pos; }
    };
    Iterator begin() { Iterator it = Iterator(list); return it; }
    Iterator end() { Iterator it = Iterator(nullptr); return it; }
};

...
List<int> a;
// C++11 syntax below
for (int & i : a ) { i *= 2; cout << i << '\n'; }
for (const char * s : { "ankara", "istanbul", "izmir" }) {
    cout << s ; }
```

# Abstraction Principle

- If any programming language entity involves computation, it is possible to define an abstraction over it

| <b>Entity</b> | → | <b>Abstraction</b> |
|---------------|---|--------------------|
| Expression    | → | Function           |
| Command       | → | Procedure          |
| Selector      | → | Selector function  |
| Declaration   | → | Generic            |
| Command Block | → | Iterator           |



# Parameters

- Many purpose and behaviors in order to take advantage of “declare once use many times”.
- **Declaration part:** `abstraction_name(Fp1, Fp2, ..., Fpn)`  
**Use part:** `abstraction_name(Ap1, Ap2, ..., Apn)`
- Formal parameters: identifiers or constructors of identifiers (patterns in functional languages)
- Actual parameters: expression or identifier based on the type of the abstraction and parameter
- **Question:** How actual and formal parameters relate/communicate?
- Programming language design should answer
- **Parameter passing mechanisms**

# Parameter Passing Mechanisms

Programming language may support one or more mechanisms. 3 basic methods:

- 1 Copy mechanisms (assignment based)
- 2 Binding mechanisms
- 3 Pass by name (substitution based)

# Copy Mechanisms

- Function and procedure abstractions, assignment between actual and formal parameter:
  - 1 Copy In:  
On function call:  $Fp_i \leftarrow Ap_i$
  - 2 Copy Out:  
On function return:  $Ap_i \leftarrow Fp_i$
  - 3 Copy In-Out:  
On function call:  $Fp_i \leftarrow Ap_i$ , and  
On function return:  $Ap_i \leftarrow Fp_i$
- C only allows copy-in mechanism. This mechanism is also called as **Pass by value**.

```
int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;
    b=a/2;
}
int main() {
    f(x,y);
    printf("x:%d, y:%d\n", x, y);
    return 0;
}
```

### Copy In:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 1        | 2        |
| 4        |          | 2        | 1        |

x:4, y:2

### Copy Out:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 0        | 0        |
| 1        | 0        | 1        | 0        |
| 1        |          |          |          |

x:1, y:0

### Copy In-Out:

| <u>x</u> | <u>y</u> | <u>a</u> | <u>b</u> |
|----------|----------|----------|----------|
| 1        | 2        | 1        | 2        |
| 4        | 1        | 2        | 1        |
| 2        |          |          |          |

x:2, y:1

# Binding Mechanisms

- Based on binding of the formal parameter variable/identifier to actual parameter value/identifier.
- Only one entity (value, variable, type) exists with more than one names.
  - 1 **Constant binding:** Formal parameter is constant during the function. The value is bound to actual parameter expression value.  
Functional languages including Haskell uses this mechanism.
  - 2 **Variable binding:** Formal parameter variable is bound to the actual parameter variable. Same memory area is shared by two variable references.  
Also known as **pass by reference**
- The other type and entities (function, type, etc) are passed with similar mechanisms.

```

int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;
    b=a/2;
}
int main() {
    f(x,y);
    printf("x:%d, y:%d\n", x, y);
    return 0;
}

```

Variable binding:

| f():a /   | f():b / |
|-----------|---------|
| x         | y       |
| <hr/>     | <hr/>   |
| 1         | 2       |
| 4         | 2       |
| 5         |         |
| x: 5, y:2 |         |

# Pass by name

- Actual parameter syntax replaces each occurrence of the formal parameter in the function body, then the function body evaluated.
- C macros works with a similar mechanism (by pre-processor)
- Mostly useful in theoretical analysis of PL's. Also known as **Normal order evaluation**
- Example (Haskell-like)

```
f x y = if (x<12) then x*x+y*y+x
      else x+x*x
```

Evaluation:  $f\ (3*12+7)\ (24+16*3) \mapsto \text{if } ((3*12+7)<12) \text{ then } (3*12+7)*(3*12+7)+(24+16*3)*(24+16*3)+(3*12+7) \text{ else } (3*12+7)+(3*12+7)*(3*12+7) \xrightarrow{*} \text{if } (43<12) \text{ then } \dots \mapsto \text{if } (\text{false}) \text{ then } \dots \mapsto (3*12+7)+(3*12+7)*(3*12+7) \xrightarrow{*} (3*12+7)+43*(3*12+7) \mapsto \dots \mapsto 1892$  (12 steps)

# Evaluation Order

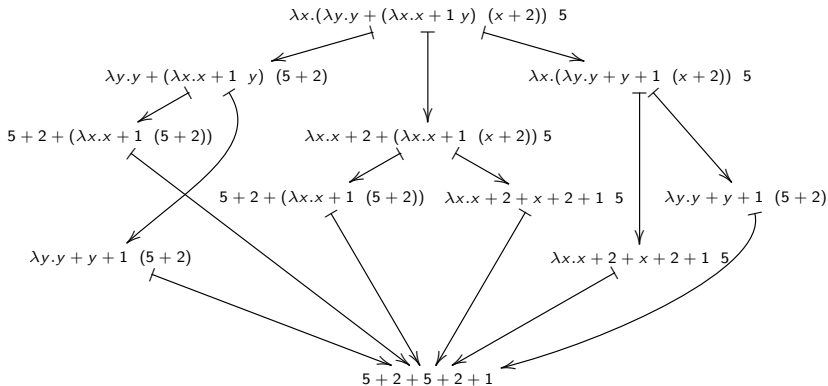
- **Normal order evaluation** is mathematically natural order of evaluation.
- Most of the PL's apply **eager evaluation**: Actual parameters are evaluated first, then passed.

$f(3*12+7)(24+16*3) \mapsto f(36+7)(24+16*3) \xrightarrow{*} f\ 43\ 72 \mapsto \text{if } (43 < 12)$   
 $\text{then } 43*43+72*72+43 \text{ else } 43+43*43 \mapsto \text{if } (\text{false}) \text{ then } \dots \mapsto$   
 $43+43*43 \xrightarrow{*} 1892$  (8 steps)

- Consider “ $g \ x \ y = \text{if } x > 10 \text{ then } y \text{ else } x$ ” for  $g \ 2 \ (4/0)$
- Side effects are repeated in NOE.
- **Church–Rosser Property**: If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing eager and normal-order evaluation), then all of these evaluation orders yield the same result.



In  $\lambda$ -calculus, all orders reduce the same normal form.



- Haskell implements **Lazy Evaluation** order.
- Eager evaluation is faster than normal order evaluation but violates Church-Rosser Property. Lazy evaluation is as fast as eager evaluation but computes same results with normal order evaluation (unless there is a side effect)
- Lazy evaluation expands the expression as normal order evaluation however once it evaluates the formal parameter value other evaluations use previously found value:

```
f (3*12+7) (24+16*3)  $\mapsto$  if (x:(3*12+7)<12) then
x:(3*12+7)*x:(3*12+7)+y:(24+16*3)*y:(24+16*3)+x:(3*12+7) else
x:(3*12+7)+x:(3*12+7)*x:(3*12+7)  $\xrightarrow{*}$  if (x:43<12) then
x:43*x:43+y:(24+16*3)*y:(24+16*3)+x:43 else x:43+x:43*x:43  $\mapsto$  if
(false) then ...  $\mapsto$  x:43+x:43*x:43  $\mapsto$  x:43+1849  $\mapsto$  1892 (7 steps)
```

# Lazy Evaluation

- Parameters are passed by name but compiler keeps evaluation state of them. Parameter value is store once it is evaluated. Further evaluations use that.
- Python implementation. First delay evaluation of expressions. Convert to functions:  
 $\text{exp} \rightarrow \text{lambda} : \text{exp}$   
 $\eta$  expansion. Function version is also called **thunk**.
- Inside function, call these functions to evaluate the expression.

```
def E(thunk):
    if not hasattr(thunk, "stored"):
        thunk.stored = thunk()      # evaluate and store
    return thunk.stored             # use stored value

def f(x,y):
    if E(x) < 10:                    # call E() on all evaluations
        return E(x)*E(x)+E(y)
    else:
        return E(x)*E(x)+E(x)

f(lambda : 3*32+4, lambda: 4/0)    # call by converting to function
```

# Infinite Values

- Delayed evaluation in normal order or lazy evaluation enables working on infinite values:

```
take _ [] = []
take n (a:r) | n == 0 = []
               | otherwise = a : take (n-1) r
```

```
x = (1:2:x)
```

```
take 3 x  $\mapsto$  take 3 (1:2:x)  $\mapsto$  1:take (3-1) (2:x)  $\mapsto$ 
1:2:take (2-1) x  $\mapsto$  1:2:take 1 (1:2:x)  $\mapsto$  1:2:1:take (1-1) (2:x)  $\mapsto$ 
1:2:1:[]
```

- Programmers can take advantage of this. Construct an infinitely value, take as many as program needs. For example expand  $\pi$  in an infinite value, stop when desired resolution achieved.

# Correspondence Principle

## ■ Correspondence Principle:

For each form of declaration there exists a corresponding parameter mechanism.

## ■ C:

|                             |                   |                                    |
|-----------------------------|-------------------|------------------------------------|
| <code>int a=p;</code>       | $\leftrightarrow$ | <code>void f(int a) {</code>       |
| <code>const int a=p;</code> | $\leftrightarrow$ | <code>void f(const int a) {</code> |

## ■ Pascal:

|                              |                   |                                               |
|------------------------------|-------------------|-----------------------------------------------|
| <code>var a: integer;</code> | $\leftrightarrow$ | <code>procedure f(a:integer) begin</code>     |
| <code>const a:5;</code>      | $\leftrightarrow$ | <code>??? {</code>                            |
| <code>???</code>             | $\leftrightarrow$ | <code>procedure f(var a:integer) begin</code> |

## ■ C++:

|                             |                   |                                    |
|-----------------------------|-------------------|------------------------------------|
| <code>int a=p;</code>       | $\leftrightarrow$ | <code>void f(int a) {</code>       |
| <code>const int a=p;</code> | $\leftrightarrow$ | <code>void f(const int a) {</code> |
| <code>int &amp;a=p;</code>  | $\leftrightarrow$ | <code>void f(int &amp;a) {</code>  |

# Programming Language Concepts

## Encapsulation

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



# Outline

- 1 Encapsulation
- 2 Packages
- 3 Hiding
- 4 Abstract Data Types
- 5 Class and Object
  - Object
  - Class
- 6 Closure

# Encapsulation

Managing the complexity → Re-usable code and abstraction.

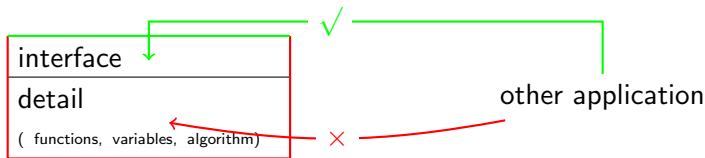
Example:

|               |                                                                                                |
|---------------|------------------------------------------------------------------------------------------------|
| 50 lines      | no abstraction is essential, all in main()                                                     |
| 500 lines     | function/procedure abstraction sufficient                                                      |
| 5,000 lines   | function groups forming modules, modules are combined to form the application                  |
| 500,000 lines | heavy abstraction and modularization, all parts designed for reuse (libraries, components etc) |



# Modularization and Encapsulation

- Building an independent and self complete set of function and variable declarations ([Packaging](#))
- Restricting access to this set only via a set of interface function and variables. ([Hiding and Encapsulation](#))



# Advantages of Encapsulation

- High volume details reduced to interface definitions ([Ease of development/maintenance](#))
- Many different applications use the same module via the same interface ( [Code re-usability](#))
- Lego like development of code with building blocks ([Ease of development/maintenance](#))
- Even details change, applications do not change (as long as interface is kept same) ([Ease of development/maintenance](#))
- Module can be used in following projects ([Code re-usability](#))

- A group of declarations put into a single body.
- C has indirect way of packaging per source file. Python defines modules per source file.
- C++

```
namespace Trig {
    const double pi=3.14159265358979;
    double sin(double x) { ... }
    double cos(double x) { ... }
    double tan(double x) { ... }
    double atan(double x) { ... }
    ...
};
```

- `Trig::sin(Trig::pi/2+x)+Trig::cos(x)`
- C++: (`::`) Scope operator.
- Identifier overlap is avoided. `List::insert(...)` and `Tree::insert(...)` no name collisions.

# Hiding

- A group of functions and variables hidden inside. The others are interface. Abstraction inside of a package:

```
double taylorseries(double);
double sin(double x);
double pi=3.14159265358979;
double randomseed;
double cos(double x);
double errorcorrect(double x);
```

```
{-- only sin, pi and cos are accessible --}
module Trig(sin,pi,cos) where
  taylorseries x = ...
  sin x = ...
  pi=3.14159265358979
  randomseed= ...
  cos x = ...
  errorcorrect x = ...
```

# Abstract data types

- Internals of the datatype is hidden and only interface functions provide the access.
- Example: rational numbers:  $3/4$  ,  $2/5$ ,  $19/3$ 

```
data Rational = Rat (Integer,Integer)
x = Rat (3,4)
add (Rat(a,b)) (Rat(c,d)) = Rat (a*d+b*c,b*d)
```

  - 1 Invalid value? `Rat (3,0)`
  - 2 Multiple representations of the same value?  
`Rat (2,4) = Rat (1,2) = Rat(3,6)`
- Solution: avoid arbitrary values by the user.

Main purpose of abstract data types is to use them transparently (as if they were built-in) without losing **data integrity**.

```
module Rational(Rational, rat, add, subtract, multiply, divide) where
  data Rational = Rat (Integer, Integer)
  rat (x,y) = simplify (Rat(x,y))
  add (Rat(a,b)) (Rat(c,d)) = rat (a*d+b*c, b*d)
  subtract (Rat(a,b)) (Rat(c,d)) = rat (a*d-b*c, b*d)
  multiply (Rat(a,b)) (Rat(c,d)) = rat (a*c, b*d)
  divide (Rat(a,b)) (Rat(c,d)) = rat (a*d, b*c)
  gcd x y = if (x==0) then y
             else if (y==0) then x
             else if (x<y) then gcd x (y-x)
             else gcd y (x-y)
  simplify (Rat(x,y)) = if y==0 then error "invalid value"
                        else let a=gcd x y
                              in Rat(div x a, div y a)
```

Initial value? We need **constructor** function/values. (remember we don't have the data definition)

rat (x,y) instead of Rat (x,y)

# Object

- Packages containing hidden variables and access is restricted to interface functions.
- Variables with state
- Data integrity and abstraction provided by the interface functions.
- Entities in software can be modelled in terms of functions (server, customer record, document content, etc). Object oriented design.
- Example (invalid syntax! imaginary C++)

```
namespace Counter {  
private:    int counter=0;  
public:     int get() { return counter;}  
public:     void increment() { counter++; }  
};  
Counter::get()           Counter::increment()
```

# Class

- The set of same typed objects form a **class**
- An object is an **instance** of the class that it belongs to (a counter type instead of a single counter)
- Classes have similar purposes to abstract data types
- Some languages allows both objects and classes
- C++ class declaration (valid syntax):

```
class Counter {  
private:    int counter;  
public:    Counter() { counter=0; }  
           int get() { return counter;}  
           void increment() { counter++; }  
} men, vehicles;  
men.increment(); vehicles.increment();  
men.get(); vehicles.get();
```



## Abstract data type

interface (constructor, functions)

detail (**data type definition**, auxiliary functions)

## Object

interface (constructor, functions)

detail (**variables**, auxiliary functions)

### Purpose

- preserving data integrity,
- abstraction,
- re-usable codes.

# Closure

- **Closure** is an abstraction method using the saved environment state in a scope.
- When a function returns a local object or function as its result and language keeps the environment state along with the returned value, it becomes a **closure**

```
def newid():
    c = 0 # this is the hidden variable in the environment
    def incget():
        nonlocal c #python 3, binds c above
        c += 1
        return c
    return incget

>>> a = newid()
>>> b = newid()
>>> a()
1
>>> b()
1
>>> b()
2
```

- Local variables of closures stay alive after call, as long as returned value is alive.
- **closures** can be used for generating new functions as in higher order functions:

```
def mult(a):
    def nested(b):
        return a*b
    return nested    # a different behaviour for each a value
twice = mult(2)
tentimes = mult(10)
a=twice(4)+tentimes(50)
```

- Also can be used for prototyping objects. Javascript example:

```
function counter() {
    var c = 0    // this is jailed in local environment, hidden
    var newObj = {} // create a new empty object
    newObj.incr = function () { c++; }
    newObj.get = function () { return c; }
    return newObj
}
a = counter()
b = counter()
a.incr()
a.get()
b.get()
```

# C++ 2011 Closures

- C++ 2011 implements closures in lambda expressions by adding a set of captured variables within `[]`. This copy or get reference of auto variables in the environment in an object.
- However C++ closures do not extend lifetime of captured variables. After exit, the behaviour is undetermined.
- `[a,&b] (int x) { return a+x+b; }` captures `a` and `b` from current environment, `a` is by copy, `b` by reference.

```
std::function<int(int)> multiply(int a) {
    return [&] (int b) { return a*b; }; // capture by value
};
std::function<int()> cid() {
    int c = 0;
    return [=] () mutable { return ++c; }; // capture by copy
};
int main() {
    std::function<int(int)> twice = multiply(2);
    std::function<int(int)> three = multiply(3);

    cout << twice(12) << '\n' << three(34) << endl;

    auto c1 = cid();
    auto c2 = cid();

    cout << c1() << '\n' << c2() << endl;
    c1(); c1(); c1();
    cout << c1() << '\n' << c2() << endl;
    return 0;
}
```

# Further Reusability

- Class relations. Extending one class definition to create more specific class definitions.
- Classes containing other classes
- Classes derived from other classes: [inheritance](#)
- Abstract classes and [interfaces](#)
- Polymorphism
- [Design patterns](#): standard object oriented designs applicable to a family of similar software problems. Not included in this course.

# Programming Language Concepts

## Type Systems

Onur Tolga Şehitoğlu

Computer Engineering



# Outline

## 1 Type Systems

## 2 Polymorphism

### ■ Inclusion Polymorphism

### ■ Parametric Polymorphism

## 3 Overloading

## 4 Coercion

## 5 Type Inference

# Type Systems

Design choices for types:

- **monomorphic** vs **polymorphic** type system.
- **overloading** allowed?
- **coercion**(auto type conversion) applied, how?
- **type relations and subtypes** exist?



# Polymorphism

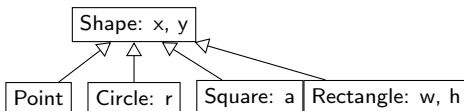
- **Monomorphic** types: Each value has a single specific type. Functions operate on a single type. C and most languages are monomorphic.
- **Polymorphism**: A type system allowing different data types handled in a uniform interface:
  - 1 **Ad-hoc polymorphism**: Also called overloading. Functions that can be applied to different types and behave differently.
  - 2 **Inclusion polymorphism**: Polymorphism based on subtyping relation. Function applies to a type and all subtypes of the type (class and all subclasses).
  - 3 **Parametric polymorphism**: Functions that are general and can operate identically on different types

# Subtyping

- C types:  
 $\text{char} \subseteq \text{short} \subseteq \text{int} \subseteq \text{long}$
- Need to define arithmetic operators on them separately?
- Consider all strings, alphanumeric strings, all strings from small letters, all strings from decimal digits.  
Need to define special concatenation on those types?
- $f : T \rightarrow V$  ,  $U \subseteq T \Rightarrow f : U \rightarrow V$
- Most languages have arithmetic operators operating on different precisions of numerical values.

# Inheritance

- `struct Point { int x, y; };`  
`struct Circle { int x, y, r; };`  
`struct Square { int x, y, a; };`  
`struct Rectangle { int x, y, w, h; };`
- `void move (Point p, int nx, int ny) {`  
`p.x=nx; p.y=ny;}`
- Moving a circle or any other shape is too different?



Haskell extensible records (only works for Hugs and in 98 mode!!):

```
import Hugs.Trex;  -- Only in -98 mode!!!

type Shape = Rec (x::Int, y::Int)
type Circle = Rec (x::Int, y::Int, r::Int)
type Square = Rec (x::Int, y::Int, w::Int)
type Rectangle = Rec (x::Int, y::Int, w::Int, h::Int)

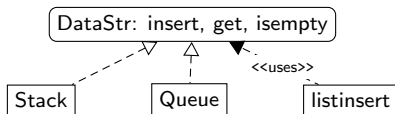
move (x=_,y=_ | rest) b c = (x=b,y=c | rest)

(a::Shape)=(x=12,y=24)
(b::Circle)=(x=12,y=24,r=10)
(c::Square)=(x=12,y=24,w=4)
(d::Rectangle)=(x=12,y=24,w=10,h=5)

Main> move b 4 5
(r = 10, x = 4, y = 5)
Main> move c 4 5
(w = 4, x = 4, y = 5)
Main> move d 4 5
(h = 5, w = 10, x = 4, y = 5)
```

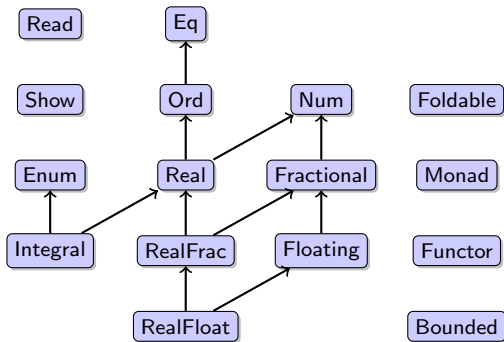
# Haskell Classes

- Subtyping hierarchy based on classes
- An instance implements interface functions of the class
- Functions operating on classes (using interface functions) can be defined



- Called **interface** in OO programming

# Haskell Default Class Hierarchy



`Eq` : `(==)`, `(/=)`

`Ord`: `(<=)`, `compare`

`Num`: `(+)`, `(-)`, `(*)`, `negate`, `abs`,  
`signum`, `fromInteger`

`Real`: `toRational`

`Fractional`: `(/)`, `recip`, `fromRational`

`RealFrac`: `truncate`, `round`, `ceiling`,...

`Enum`: `succ`, `pred`, `toEnum fromEnum`,...

`Integral`: `quot`, `rem`, `div`, `mod`,  
`toInteger`, `divMod`, `quotRem`

`Floating`: `pi`, `exp`, `log`, `sin`, `(**)`,  
`cos`, `asin`, `sinh`, ...

`RealFloat`: `floatRadix`, `exponent`,  
`isNaN`, `isIEEE`, `decodeFloat`,...

`Show`: `show`

```

class DataStr a where
  insert :: (a v) -> v -> (a v)
  get :: (a v) -> Maybe (v, (a v))
  isempty :: (a v) -> Bool

instance DataStr Stack where
  insert x v = push v x
  get x = pop x
  isempty Empty = True
  isempty _ = False

instance DataStr Queue where
  insert x v = enqueue v x
  get x = dequeue x
  isempty EmptyQ = True
  isempty _ = False

insertlist :: DataStr a => (a v) -> [v] -> (a v)
insertlist x [] = x
insertlist x (el:rest) = insertlist (insert x el) rest

data Stack a = Empty | St [a] deriving Show
data Queue a = EmptyQ | Qu [a] deriving Show

```

# Parametric Polymorphism

- **Polymorphic** types: A value can have multiple types.  
Functions operate on multiple types **uniformly**
- **identity** `x = x` function. type:  $\alpha \rightarrow \alpha$   
`identity 4 : 4`, `identity "ali" : "ali"` , `identity (5,"abc") : (5,"abc")`  
 $int \rightarrow int$ ,  $String \rightarrow String$ ,  $int \times String \rightarrow int \times String$
- **compose** `f g x = f (g x)` function  
 type:  $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$   
`compose square double 3 : 36`,  
 $(int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \rightarrow int$ .  
`compose listsum reverse [1,2,3,4] : 10`  
 $([int] \rightarrow int) \rightarrow ([int] \rightarrow [int]) \rightarrow [int] \rightarrow int$



- `filter f [] = []`  
`filter f (x:r) = if (f x) then x:(filter f r) else (filter r)`  
 $(\alpha \rightarrow Bool) \rightarrow [\alpha] \rightarrow [\alpha]$   
`filter ((<) 3) [1,2,3,4,5,6] : [4,5,6]`  
 $(int \rightarrow Bool) \rightarrow [int] \rightarrow [int]$   
`filter identity [True, False, True, False] :`  
`[True,True]`  
 $(Bool \rightarrow Bool) \rightarrow [Bool] \rightarrow [Bool]$
- Operations are same, types are different.
- Types with type variables: [polytypes](#)
- Most functional languages are polymorphic
- Object oriented languages provide polymorphism through inheritance, run time binding and generics

# Polymorphism in C++ and Java

- Inheritance provides subtyping polymorphism
- C++ **virtual** methods, and all methods in Java implements **late binding** to improve polymorphism through inheritance.
- Generic abstractions, C++ **templates** and Java **generics** provide polymorphic classes and functions.

```
template <typename T>
void sort(T arr[], int n) {
    // ... your favorite sort algorithm here
}
```

```
class Test { //Java requires functions be in a class
void <T> sort(T[] arr) {
    // ... your favorite sort algorithm here
}
```

- C++ **templates** use compile time binding. Java **generics** binds at run time.

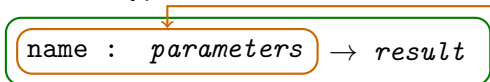
# Overloading

- **Overloading**: Using same identifier for multiple places in same scope
- Example: Two different functions, two distinct types, same name.
- Polymorphic function: one function that can process multiple types.
- C++ allows overloading of functions and operators.

```
typedef struct Comp { double x, y; } Complex;
double mult(double a, double b) { return a*b; }
Complex mult(Complex s, Complex u) {
    Complex t;
    t.x = s.x*u.x - s.y*u.y;
    t.y = s.x*u.y + s.y*u.x;
    return t;
}
Complex a,b; double x,y; ... ; a=mult(a,b) ; x=mult(y,2.1);
```

- Binding is more complicated. not only according to name but **according to name and type**

- Function type:



`name : parameters → result`

- **Context dependent overloading:** \_\_\_\_\_

Overloading based on function name, parameter type and return type.

- **Context independent overloading :** \_\_\_\_\_

Overloading based on function name and parameter type. No return type!

# Context dependent overloading

## ■ Which

type does the expression calling the function expects (context) ?

```
int f(double a) { .... ① }
int f(int a) { .... ② }
double f(int a) { .... ③ }
double x,y;
int a,b;
```

- `a=f(x);` ① (x double)
- `a=f(a);` ② (a int, assign int)
- `x=f(a);` ③ (a int, assign double)
- `x=2.4+f(a);` ③ (a int, mult double)
- `a=f(f(x));` ②(①) ( x double, f(x):int, assign int)
- `a=f(f(a));` ②(②) or ①(③) ???
- Problem gets more complicated. (even forget about coercion)

# Context independent overloading

- Context dependent overloading is more expensive.
- Complex and confusing. Useful as much?
- Most overloading languages are context independent.
- Context independent overloading forbids ② and ③ functions defined together.
- “name: parameters” part should be unique in “name: parameters → result”, in the same scope
- Overloading is not much useful. So languages avoid it.

## Use carefully:

Overloading is useful only for functions doing same operations. Two functions with different purposes should not be given same names. Confuses programmer and causes errors

- Is variable overloading possible? What about same name for two types?

# Coercion

- Making implicit type conversion for ease of programming.

```
double x;      int k;
x = k+4.2;     /* x = (double) k + 4.2 */
k = x+3.45;    /* k=(int) (x+3.45); */
k = x+2;       /* k=x+(double)2; */
k = x+k-2;     /* k=(int)(x+ (double)k - (double)2) ; */
```

- C makes *int*  $\leftrightarrow$  *double* coercions and pointer coercions (with warning)
- Are other type of coercions are possible? (like  $A * \rightarrow A$ ,  $A \rightarrow A *$ ). Useful?
- May cause programming errors:  $x=k=3.25$  :  $x$  becomes 3.0
- Coercion + Overloading: too complex.
- Most newer languages quit coercion completely ([Strict type checking](#))

# Type Inference

- Type system may force user to declare all types (C and most compiled imperative languages), or
- Language processor infers types. How?
- Each expression position provide information (put a constraint) on type inference:
  - Equality  $e = x, x :: \alpha, y :: \beta \Rightarrow \alpha \equiv \beta$
  - Expressions  $e = a + f\ x, + :: Num \rightarrow Num \rightarrow Num \Rightarrow a :: Num, f :: \alpha \rightarrow Num, e :: Num$
  - Function application  $e = f\ x \Rightarrow e :: \beta, x :: \alpha, f :: (\alpha \rightarrow \beta)$
  - Type constructors  $f\ (x : r) = t \Rightarrow x :: \alpha, t :: \beta, f :: ([\alpha] \rightarrow \beta)$
- Inference of all values start from the most general type (i.e: any type  $\alpha$ )
- Type inference finds the **most general type** satisfying the constraints.



# Inferring Type from Initializers

- C++11 `auto` type specifier gets type from initializer or return expression.
- C++11 `decltype(varexp)` gets type same as the variables declared type

```
auto f(int a) {
    return a/3.0; // double, function becomes double
}
struct P { double x, y;} *pptr;

decltype(pptr->x) xval; // double since pptr->x is double

auto v = (P)({ 2.0, 4.0}); // initializer is P typed
auto t = f(3); // f(3) returns double so t is double
```

- GCC has `typeof(expr)`, some other dialects have `__typeof__ (expr)` macro having a similar mechanism in C.

# Summary

- Monomorphic vs Polymorphic types
- Subtyping
- Inheritance
- Overloading
- Parametric polymorphism
- Coercion
- Type Inference

# CENG242 - Haskell Recitation \*

Deniz Sayın

2020  
February

## 1 Introduction to Haskell

Features:

- **Purely functional**, every function is a single expression with no side effects, its result depends entirely on its arguments. Values are immutable. Also, functions are first-class members; i.e. functions can be taken as arguments or returned by other functions.
- **Lazy**, values are not evaluated until forced (e.g. by printing). This results in an increase in expressivity, as we can work with types such as infinite lists, or infeasibly large data structures.
- **Statically typed**, the types of every declaration are known at compile time.
- **Type inference**, when no types are specified, the compiler performs bidirectional unification to fully infer the types of declarations.

**Note:** The standard extension for Haskell source files is `.hs`

## 2 GHC: Glasgow Haskell Compiler

### 2.1 Introduction

- GHC is a state-of-the-art & open source compiler and interactive environment for Haskell.
- GHC is the compiler and GHCi is the interactive environment.
- The interactive environment functions similarly to the shell (as well as python's interpreter interface, which you already know about). Bindings can be declared, and expressions can be evaluated and printed.
- Installation instructions can be found at <https://www.haskell.org/ghc/download>. Simplest method to just get ghc: `sudo apt install ghc`

---

\*This material is an updated summary composed from [haskell.org](http://haskell.org) and [learnyouahaskell.com](http://learnyouahaskell.com), and my own understanding

- When launched, the environment automatically loads the `Prelude` module, which contains definitions of standard basic data types, typeclasses and functions. Get familiar with it! Check out the documentation at <https://hackage.haskell.org/package/base-4.12.0.0/doc/text/Prelude.html> once you're comfortable with Haskell.
- GHC/GHCi will be used when grading the code you write during homeworks/lab exams.

## 2.2 GHCi Basics

- Run with `ghci`. By default, the prompt shows the names of loaded modules:

```
shell-prompt$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

- Can be used for basic arithmetic, like many other environments. `^` is for exponentiation with integer exponents, while `**` is for floating point exponentiation. Note that operators are implemented as functions with infix notation, thus negative values need to be paranthesised when used in expressions:

```
Prelude> 3 * 5 + 7 * 8
71
Prelude> 3 * 2^5
96
Prelude> 3.0 ** 0.5
1.7320508075688772
Prelude> -7
-7
Prelude> 3 + (-7)
-4
Prelude> 3 + -7

<interactive>:2:1: error:
  Precedence parsing error
    cannot mix `+' [infixl 6] and prefix `-' [infixl 6]
in the same infix expression
```

- Comparisons are similar to other languages, except for inequality:
  - `>`, `>=`: greater than, greater than or equal to
  - `<`, `<=`: less than, less than or equal to
  - `==`, `/=`: equal, not equal
- And and or are represented by `&&` and `||` like in C/C++/Java. `not` replaces the `!` operator:

```
Prelude> 3 > 5
False
Prelude> 3 > 5 || 3 < 5
True
Prelude> 1 < 2 && 2 > 5
False
Prelude> not (8 /= 9)
False
```

- Contains commands related to the `ghci` environment (not the Haskell language itself), prefixed with the colon character `:`. Some examples:
  - `:?` shows help
  - `:type` or `:t` shows the type of an expression
  - `:load` or `:l` loads functions from a Haskell source file
  - `:reload` or `:r` reloads a previously loaded source file
  - `:info` or `:i` shows information about a name
  - `:set` sets environment options. e.g. `:set +s` will show the time elapsed and number of bytes allocated while evaluating an expression.
  - `:sprint` shows how much of a value has been evaluated
  - `:q` quits `ghci`
- Bindings can be made directly in the interactive environment <sup>1</sup>.

```
Prelude> a = 3
Prelude> b = 7
Prelude> diff = a - b
Prelude> diff
-4
Prelude> diff + b
3
```

- Bindings can also be loaded from a source file with the `:load` command. Running `ghci` with a source file as a command line argument will load it automatically.

```
source.hs
x = 3.0 -- also, here is single line comment!
y = 8.7
```

---

<sup>1</sup>This is a new feature in GHC 8. Previously, introducing a binding directly like `'x = 3'` would result in a parse error, and the `let` keyword had to be used as in `'let x = 3'`. This has to do with `ghci` essentially acting like an IO `do` block, which is information outside the scope of CENG242. New versions of `ghci` act as if `let` is present before the binding even when it is not. See [here](#) for more details.

```
shell-prompt$ ghci source.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main                ( source.hs, interpreted )
Ok, modules loaded: Main.
*Main> x
3.0
*Main> x + y
11.7
```

**Note:**

Haskell is case-sensitive! Value names (including functions) **must** start with an underscore or a lowercase letter. Conversely, type names and value constructors (to be covered soon) **must** start with an uppercase letter. Anything else is an error.

### 3 Basic Data Types

- **Bool**: Boolean values, either **True** or **False**.
- **Char**: Characters with unicode values. e.g. `'a'`, `'?'`. Note that unicode is an extension of ASCII, thus characters which are already in ASCII retain their ASCII values (e.g. `'a'` is 97).
- **Int**: Bounded signed integer values, at least 30 bits. Usually 64 bits.
- **Word**: Unsigned integer values, same size as **Int**.
- **Integer**: Arbitrary-precision integers.
- **Float**: Single precision floating point numbers.
- **Double**: Double precision floating point numbers.
- **Rational**: Arbitrary-precision rational numbers, represented with two **Integer** values, one for the numerator and the other for the denominator.

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t True
True :: Bool
```

More basic types can be found in the **Prelude**'s documentation.

Note:

Note that unlike `Char` and `Bool`, numeric values defined in `ghci` will have a polymorphic type (more on this later). You can force them to have a concrete type using a type specification with the `::` keyword:

```
Prelude> x = 3
Prelude> :t x
x :: Num t => t
Prelude> x = 3 :: Int
Prelude> :t x
x :: Int
Prelude> x
3
Prelude> x = 3 :: Float
Prelude> x
3.0
```

### 3.1 Defining Functions

Now that we know about basic types, we can move on to defining functions with them. A simplified syntax for defining functions <sup>2</sup> would be as follows:

`<function-name> <param1-name> {<param2-name> ...} = <result-expression>`

Here are some example functions along with how to call them in `ghci`. Note that unlike popular languages you might have used so far (i.e. C/C++/Java/Python), function definition/application does not require parentheses in Haskell and has the highest precedence:  
<sup>3</sup>

funcs.hs

```
f x = x^2 + 3*x + 7           -- polynomial
g x y = x * y + x             -- multivariate polynomial
sumOfThree a b c = a + b + c  -- sums its three arguments
isA c = c == 'a' || c == 'A'  -- checks whether the argument is an a
```

<sup>2</sup>This syntax disregards the existence of pattern matching (detailed in 4.1)

<sup>3</sup>We will define our functions in separate source files, but they can also be defined on the fly in `ghci` like any other binding.

```

[1 of 1] Compiling Main                ( funcs.hs, interpreted )
Ok, modules loaded: Main.
*Main> f 0
7
*Main> f(3)
25
*Main> f ((3))
25
*Main> f 0 + 10 -- parsed as (f 0) + 10 due to high precedence
17
*Main> g 3 5
18
*Main> sumOfThree 1 2 3
6
*Main> isA 'c'
False
*Main> isA 'A'
True
*Main> :t isA
isA :: Char -> Bool

```

As we have said before, Haskell is a statically typed language with type inference. Since we did not provide types for the functions we defined in `funcs.hs`, GHC inferred the most generic type possible for the functions. For example, the type of the `isA` function turns out to be `Char -> Bool`. This means that the function takes a `Char` and returns a `Bool`. The type of the other functions is more complicated as they can operate on any numeric type.<sup>4</sup> For functions that take multiple arguments, we add more arrows. For example, `Char -> Char -> Int -> Bool` would be the type of a function taking two `Chars` and an `Int` and returning a `Bool`. Why more arrows rather than something that clearly separates the arguments and the result, such as `{Char, Char, Int} -> Bool`? This is due to a feature of Haskell called *currying*, which we cover in 5.1.

We can set types of functions once again with the `::` operator. This is seen as good practice because it clarifies the parameters and the function's purpose. The source file `typed-funcs.hs` given below shows the previous definitions with provided types. Note that a type that cannot be resolved by the compiler will cause compilation errors. For example, setting the type of `isA` to be `Bool -> Bool` will cause an error because the function will try to compare the input `Bool` value with the `Char` values `'a'` and `'A'`, which is not defined.

---

<sup>4</sup>In this case, this means any type which is an instance of the `Num` typeclass. For example, the type of `f` would be `(Num a) => a -> a`. The compiler infers this type due to the use of arithmetic operators in `f`.



typed-funcs.hs

```
f :: Integer -> Integer
f x = x^2 + 3*x + 7

g :: Double -> Double -> Double
g x y = x * y + x

sumOfThree :: Int -> Int -> Int -> Int
sumOfThree a b c = a + b + c

isA :: Char -> Bool
isA c = c == 'a' || c == 'A'
```

Another important property of functions in Haskell is that they can be applied 'partially', without applying all of their arguments, which results in a new function equivalent to the previous function with some leading arguments already set. This is best shown with an example over the previously defined `sumOfThree` function:

```
[1 of 1] Compiling Main                ( typed-funcs.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t sumOfThree
sumOfThree :: Int -> Int -> Int -> Int
*Main> sumOfTwo = sumOfThree 0 -- sumOfTwo x y = sumOfThree 0 x y
*Main> :t sumOfTwo
sumOfTwo :: Int -> Int -> Int
*Main> sumOfTwo 5 6
11
*Main> addEight = sumOfTwo 8 -- addEight x = sumOfTwo 8 x = sumOfThree 0 8 x
*Main> :t addEight
addEight :: Int -> Int
*Main> addEight 15
23
```

The operators we have used so far (+, \*, <, && etc.) are also functions. Since these functions are defined with symbols, they can be used directly with infix notation. They have to be put between parentheses to be used as a prefix. Similarly, any function we define with alphanumeric characters can be used with prefix notation directly, but needs to be put between backticks (` , ASCII 96) to be used as an infix:

```
Prelude> 7*2 + 3*5
```

```
29
```

**Note:** `Prelude> (+) ((*) 7 2) ((*) 3 5)`

```
29
```

```
Prelude> p x y = sqrt (x - y)
```

```
Prelude> p 3 1
```

```
1.4142135623730951
```

```
Prelude> 3 `p` 1
```

```
1.4142135623730951
```

It is possible to modify the precedence and associativity of your own functions. See Fixity Declarations.

## 3.2 Lists

### 3.2.1 Introduction

The primary sequential data structure in Haskell is the list, which is essentially a singly-linked list. The empty list is denoted as []. Note that a list is not a type but a type constructor. This means that a value cannot have the type of 'list', but the type 'list of something'. Actual types could be a list of booleans [Bool], a list of lists of integers [[Int]] etc. This also shows that lists are homogeneous, i.e. the elements of a list all share the same type. Lists can be defined using commas (just like in Python), e.g. [3, 42, 24, 59].

**Note:** The `String` type is defined as [Char] by default.

### 3.2.2 A Few Basic Functions

The most basic operation on a list is appending to the beginning, with the (:) function (called cons, which is the name of the same function in Lisp derivatives): <sup>5</sup>.

---

<sup>5</sup>The definition of lists using commas is actually just syntactic sugar for multiple applications of the (:) operator. e.g. [4, 8, 15, 17] stands for 4:8:15:17:[]

```

Prelude> 1:[2, 3]
[1,2,3]
Prelude> 4:5:6:7:[]
[4,5,6,7]
Prelude> [0]:[[1, 2], [3, 4, 5]]
[[0],[1,2],[3,4,5]]
Prelude> 'c':"eng"
"ceng"

```

You can decompose a list into the first element and the rest using the **head** and **tail** functions. You can also decompose it into the initial elements and the final element using **init** and **last**, and the **length** function can be used to find the length of a list. Note that the whole list has to be traversed for these final three, while the first two are constant time operations.

```

Prelude> myList = [17, 21, 3, 99, 44]
Prelude> head myList
17
Prelude> tail myList
[21,3,99,44]
Prelude> init myList
[17,21,3,99]
Prelude> last myList
44
Prelude> length myList
5

```

The **take** and **drop** functions can be used to make a new list from the first  $k$  elements of a list, or a list excluding the first  $k$  elements. **elem** checks whether a value is inside a list.

```

Prelude> take 4 "ceng242"
"ceng"
Prelude> drop 4 "ceng242"
"242"
Prelude> elem 3 [1, 5, 0, 3, 7]
True

```

Lists can be concatenated with the **(++)** function and indexed with the **(!!)** function. Once again, note that concatenating two lists requires the first list to be traversed and that indexing the  $k$ th element also requires traversing the first  $k$  elements. If you find yourself making use of these a lot when writing low-level list processing functions, you are probably using the wrong approach.

```
Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> ['b', 'o'] ++ ['o']
"boo"
Prelude> [7, 8, 15, 23, 38, 61] !! 0
7
Prelude> [7, 8, 15, 23, 38, 61] !! 3
23
Prelude> [7, 8, 15, 23, 38, 61] !! 100
*** Exception: Prelude.!!: index too large
```

Infinite lists can be constructed using `repeat`, `cycle`. `repeat` repeats the same value forever, `cycle` repeats a list forever . While infinite lists allow for nice abstractions, remember that when working with them in practice you have to cut them off at some point.

[illegible]

### 3.2.3 Ranges

Some data types <sup>6</sup> can be used to define ranges on lists. The syntax: [**<start>**{, **<next>**}..**<end>**] is for defining ranges with parts enclosed in curly braces denoting optional parts. The step size (increment) is assumed to be 1 if left undefined. Observe that the end of the range is also optional, which means that we can define infinite ranges!

```
Prelude> [3..9]
[3,4,5,6,7,8,9]
Prelude> [3,5..16]
[3,5,7,9,11,13,15]
Prelude> [3,5..17]
[3,5,7,9,11,13,15,17]
Prelude> [10,9..(-1)]
[10,9,8,7,6,5,4,3,2,1,0,-1]
Prelude> [10..(-1)]
[]
Prelude> take 5 [1..]
[1,2,3,4,5]
```

---

<sup>6</sup>Types which are instances of the `Enum` typeclass

### 3.2.4 List Comprehensions

Haskell also includes list comprehensions (just like Python, and previous functional languages which influenced both Haskell and Python). The syntax is as follows:

```
[<expr> | <name1> <- <list1> {, <name2> <- <list2> ...} {, <filter-expr>}]
```

Essentially, values taken from one or more lists (multiple lists correspond to the cartesian product, like nested loops) are used to build a new list using the expression provided at the start of the list comprehension. Optionally, a boolean expression can be included after all the lists that will filter out certain values.

Some examples in `ghci` to help with understanding the syntax and workings:

```
Prelude> [x^2 | x <- [0..10]]
[0,1,4,9,16,25,36,49,64,81,100]
Prelude> [x^2 | x <- [0..10], x > 5]
[36,49,64,81,100]
Prelude> [x^2 | x <- [0..10], odd x]
[1,9,25,49,81]
Prelude> [[x, y] | x <- [0,1,2], y <- [3, 4, 5]]
[[0,3],[0,4],[0,5],[1,3],[1,4],[1,5],[2,3],[2,4],[2,5]]
Prelude> [[x, y] | x <- [0,1,2], y <- [3, 4, 5], x * y < 5]
[[0,3],[0,4],[0,5],[1,3],[1,4]]
Prelude> [x | x <- [0,1,2], y <- [1..5]]
[0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2]
```

### 3.3 Tuples

Another important composite data type in Haskell is the tuple. Unlike lists, tuples can be composed of multiple types and have a fixed length. Tuples are defined using parentheses like in Python: (`<item1>`, `<item2>` {, `<item3>` ...}). The type of a tuple is determined by the types of its elements:

```
Prelude> myTuple = ("hello", True)
Prelude> :t myTuple
myTuple :: ([Char], Bool)
Prelude> mySecondTuple = ("a", [False, True], 'b', True, "cccc")
Prelude> :t mySecondTuple
mySecondTuple :: ([Char], [Bool], Char, Bool, [Char])
```

As you can see from the examples, tuples can have as many elements as you like, and have types depending on the types of the arguments. Tuples have a fixed length, unlike lists. This means that you cannot add more values to the front/back, concatenate them etc. It is of course possible to have lists of tuples, however the tuples have to have the same type since lists are homogeneous. For example, you could have a list produced from a string which holds how many of each characters are present in the string, its type could be `[(Char, Int)]`. This list will only hold tuples of that type.

Two basic functions that work on two element tuples (pairs) are `fst` and `snd`. They simply return the first and second element of the pair, respectively.

Before moving on, make sure you distinguish between lists and tuples:

- **Lists** are homogeneous and of arbitrary length. Useful for collecting data of the same type and operating on it.
- **Tuples** can be heterogeneous and have fixed length. Useful for gluing related data together in a lightweight manner.

**Note:**

Haskell includes a zero element tuple, `()`, which is its own type. However, there is no single element tuple by default.

### 3.4 Type Variables

Functions do not necessarily have to work on a concrete type. As an example, consider the `head` function which returns the first element of a list. What could the type of this function be? This would actually depend on the type of the list. If it only worked on lists of integers, `[Int]`, it would be `[Int] -> Int`. If this was the case for a list of characters, it would be `[Char] -> Char`. However, the function actually works on lists of all types. It takes a list of some type and returns a value of the same type; the actual operation is the same no matter the type of the value that is in the list.

For cases like the `head` function, Haskell allows us to work with polymorphic functions that work on many different types. When stating the type of these functions, we use **type variables**, which stand in for any type. A type variable can be any identifier starting with a lowercase letter (since actual types start with uppercase letters). Usually, single characters like `a`, `b`, `c` or `t` are used. Two example definitions that use type variables are given below in the `typevars.hs` file. More examples with some functions we introduced previously follow. Make sure you also experiment on your own later on to gain a better understanding!

`typevars.hs`

```
identity :: someType -> someType
identity x = x -- returns its argument

-- we let GHC infer the type of this function
itsgonnabeq x = 'q' -- returns 'q' no matter the argument
```

```
*Main> :t identity
identity :: someType -> someType
*Main> :t itsgonnabeq
itsgonnabeq :: t -> Char
*Main> :t head
head :: [a] -> a
*Main> :t tail
tail :: [a] -> [a]
*Main> :t fst
fst :: (a, b) -> a
*Main> :t take
take :: Int -> [a] -> [a]
```

## 4 Syntactic Constructs in Haskell

### 4.1 Pattern Matching

A very useful feature of Haskell when defining functions is the ability to make different definitions for different patterns. This leads to elegant code that appears to be quite readable. Pattern matching can be applied to anything. Integers, characters, lists, tuples, and even your own data types later on. Here's an example:

match.hs

```
commentOnGrade :: String -> String
commentOnGrade "AA" = "You rock, friend!"
commentOnGrade "BA" = "Ah, must have been bad luck!"
commentOnGrade other = "You missed the Haskell recitation, didn't you?"
```

When calling the `commentOnGrade` function, the argument will be checked against the patterns starting from the top and continuing towards the bottom. In our case, the string "AA" will match the first definition, while "BA" will match the second definition. Anything else will be captured and bound to the name `other` by the third definition, since there is no pattern defined there.

If the patterns do not cover everything, which would be the case if we removed the third definition, anything that does not match the provided patterns will cause an error due to "non-exhaustive patterns in function". Also keep in mind that the matching is done in order. If we took the third definition and put it before the first one, it would match any pattern and the definitions made for the "AA" and "BA" patterns would be unreachable.

Let's also have an example for integers by defining our own factorial function:

fact.hs

```
fact :: Integer -> Integer
fact 0 = 1 -- base case, should be the first!
fact n = n * fact (n - 1) -- recursive case
```

And an example for tuples that returns the third element of a triplet along with the whole triplet. We also introduce the `_` syntax, which captures a value but does not bind it to a name (similar to Python, where `_` still binds but is used for unused variables by convention), which you can use for values that you don't care about. Also, you can use the `@` character to bind a whole pattern to a name:

trd.hs

```
trd :: (a, b, c) -> ((a, b, c), c)
trd triplet@(_, _, z) = (triplet, z) -- don't care about the first two
```

```
*Main> trd ('a', 1, True)
(('a',1,True),True)
*Main> trd ("first", "second", "third")
(("first","second","third"),"third")
```

For lists, we can pattern match with the cons (:) function to separate the list into its head and its tail as a pattern. Note that a list needs to have at least one element to match that pattern. As an example, we can write our own function to join a list of strings using another string:

join.hs

```
join :: String -> [String] -> String
join _ [] = "" -- empty list case, an empty string
join _ [str] = str -- single string case, only that string
join joinStr (firstStr:rest) = firstStr ++ joinStr ++ join joinStr rest
```

```
*Main> join " " ["in", "the", "beginning"]
"in the beginning"
*Main> join "-" ["not much"]
"not much"
*Main> join "->" ["follow", "the", "arrows!"]
"follow->the->arrows!"
```

## 4.2 if-then-else

We've been putting this one off long enough! Like most other languages, Haskell also has an if-else construct. Unlike in imperative languages however, the else part is mandatory because the construct is an expression; it must have a value whether the predicate holds or not. As an example, we can rewrite the factorial function with it and add a fancy one which causes an error when a negative value is input:

ifact.hs

```
ifact :: Integer -> Integer
ifact n = if n == 0 then 1 else n * ifact (n - 1) -- one line

ifactChecked :: Integer -> Integer
ifactChecked n = if n < 0 -- broken over multiple lines
                  then error "factorial of a negative value"
                  else ifact n
```



```

*Main> ifactChecked 0
1
*Main> ifactChecked 5
120
*Main> ifactChecked (-10)
*** Exception: factorial of a negative value
CallStack (from HasCallStack):
  error, called at ifact.hs:6:26 in main:Main

```

### 4.3 Guards

With pattern matching, we learned to match on structures of values, by putting in literals (0, "AA" etc.) or constructors as in the case of lists ([], (x:xs) etc.). With guards, we instead check whether values themselves satisfy certain constraints or not. This works just like a chain of if-then-else constructs, but looks a lot nicer. Here's an example for classifying ranges of values with strings:

```

classify.hs

classify :: Double -> String
classify x
  | x < 1e-10 = "That's tiny! Just like a pebble."
  | x < 1e-5  = "Almost negligible."
  | x < 1e5   = "A pretty normal value, if I've ever seen one."
  | x < 1e10  = "Not bad, some pretty good stuff you've got there."
  | otherwise = "BigValue (TM)"

```

The predicates are checked one after another starting from the top, just like in pattern matching. The only constraint to satisfy is that predicates have to `Bool` values of course. The final case is marked with an `otherwise` <sup>7</sup> Hitting a missing case will cause a "non-exhaustive patterns in function" error, as with pattern matching.

### 4.4 let-in

Sometimes we are faced with the need to compute a value once and reuse it multiple times, or chain lots of function calls together. Being able to name such intermediate values helps in terms of both readability, especially when faced with lots of/long function calls, and performance because function calls are not repeated <sup>8</sup>. The construct which lets us do this in Haskell is the `let-in` construct. Here's an example:

<sup>7</sup>Tidbit: `otherwise` is simply defined as `otherwise = True`

<sup>8</sup>This can easily be optimized away by a compiler, especially with the side-effectless functions of Haskell, but the readability argument alone is enough.

```
firstNLast.hs
```

```
firstNLast :: String -> String
firstNLast fullName = let nameList = words fullName -- splits on spaces
                        firstName = head nameList
                        lastName = last nameList
                        in [head firstName, '.', head lastName, '.']
```

```
*Main> firstNLast "Haskell Curry"
"H.C."
*Main> firstNLast "Marcus Ulpius Traianus"
"M.T."
```

## 4.5 where

We introduced the **let-in** expression which allows us to name and reuse values before an expression. Now, we introduce the **where** construct which does the same thing, but after an expression is finished. Just like a **let-in** expression, it has access to all the bindings of its enclosing scope. Here's the same example we gave for **let-in**, but using **where** instead:

```
firstNLastwhere.hs
```

```
firstNLast :: String -> String
firstNLast fullName = [head firstName, '.', head lastName, '.']
    where nameList = words fullName -- splits on spaces
          firstName = head nameList
          lastName = last nameList
```

Remember that **where** is not an expression like **let-in**, but a different block, thus it can span across guards. However, it cannot span across pattern matches.

**Note:**

Remember that bindings are not restricted to values. It is perfectly valid to define functions (even including different pattern matches!) in **let-in** **where**.

## 4.6 case-of

Pattern matching is quite nice, isn't it? It's almost too bad that we can only use the top-to-bottom match style in function definitions. Well, fear not! The **case-of** construct is the expression version of pattern matching <sup>9</sup>, which you can use to pattern match anywhere. Here's an example that determines how 'lucky' a list of integers is, we also mix in a **let** to cram in even more expressions:

---

<sup>9</sup> *Ackchyually*, pattern matching is syntactic sugar for a **case-of** expression

luck.hs

```
luck :: [Int] -> Int
luck [] = 0
luck (x:xs) = let luckIncrement = case x of 7 -> 1    -- lucky!
   13 -> -1   -- unlucky :(
   _ -> 0    -- who cares?
            in luckIncrement + luck xs
```

## 5 Higher Order Functions

### 5.1 Currying

We were previously left wondering why functions with multiple arguments had type signatures like  $X \rightarrow Y \rightarrow Z \rightarrow T$ . Now, it is time to demystify this by explaining the technique called *Currying*, which is also present in Haskell. It is a method by which functions taking multiple arguments are transformed into multiple function applications, each taking a single argument. The gist of it is that functions in Haskell only ever take a single parameter and return a single value. The  $\rightarrow$  operator is right-associative, so the previous type signature actually corresponds to this:  $X \rightarrow (Y \rightarrow (Z \rightarrow T))$ . What does this mean exactly? Let us consider the application of a simple function with two arguments, and see how currying works:

- Let us define the function `f`:  
`f :: Int -> Int -> Int`  
`f x y = x * x + y`
- Now, as an example, let's evaluate `f 3 5`. This actually corresponds to two function applications in the following way: `(f 3) 5`.
- First, `f` is applied on 3, and returns a function of type `Int -> Int`, which internally is something like this: `f3 y = 3 * 3 + y`.
- Then, the resulting function is applied to the second argument 5, which results in the `Int` expression `3 * 3 + 5`.

It is easy to see how this extends to cases with multiple arguments. In the end, this is equivalent to having functions with multiple arguments, so we will still call functions with multiple arguments as such. This mechanism is what allows seamless partial function application in Haskell.

### 5.2 Higher Order Functions over Lists

The `Prelude` and `Data.List` modules contain some higher order functions that work on lists that we have not introduced yet, but are very useful. The most important pair are arguably the `map` and `filter` functions.

The `map` function simply takes a function and applies it to every element of a list. Let's see how it works in practice:

```

Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> map (/5) [1, 3, 5, 18, 24, 111]
[0.2,0.6,1.0,3.6,4.8,22.2]
Prelude> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]
Prelude> map (++ "!!!") ["hello", "why", "am I", "so excited"]
["hello!!!","why!!!","am I!!!","so excited!!!"]

```

The next important function is **filter**. It takes a predicate (a function returning **Bool**) and list, and returns a new list only including elements of the list for which the predicate is **True**. Here are some examples:

```

Prelude> :t filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude> filter (>7) [1, 4, 19, 2, 8, 13, 0, 22]
[19,8,13,22]
Prelude> filter odd [0..15]
[1,3,5,7,9,11,13,15]
Prelude> filter not [False, False, True, False]
[False,False,False]

```

You may think that the same functionality can be achieved with list comprehensions. That is correct. However, when working with a single function, using **map** and **filter** is much more readable. e.g. **filter f xs** would be `[x | x <- xs, f x]` using a list comprehension. In some cases, a list comprehension might be more readable; the choice is up to you.

These two functions will be your bread and butter when working with lists. Their compositions also work quite nicely. For example, if we had a nested list (two level, like `[[Int]]`) and wanted to double each element without changing the structure, we could simply nest a **map** call in the following manner: `map (map (*2)) myNestedList`.

Before moving on let us define them on our own to gain insight and see how simple they are:

```

mapfilt.hs

map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs

filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs) = let rest = filter' p xs
                  in if p x
                     then x : rest
                     else rest

```

Another great function for working with lists is **zipWith**. This function takes a function and two lists, and combines elements of the two lists using the provided function:

```

ghci> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
ghci> zipWith (+) [1, 10, 100] [2, 3, 8]
[3,13,108]
ghci> zipWith (+) [1, 10, 100, 10, 1] [2, 44, 57, 11] -- short cutoff
[3,54,157,21]
ghci> zipWith (:) [1, 2, 3] [[5, 7], [6, 9, 10], [3..7]]
[[1,5,7],[2,6,9,10],[3,3,4,5,6,7]]
ghci> zipWith (++) ["haskell", "so", "fun"] [" is", " much", "", right?""]
["haskell is","so much","fun, right?"]

```

There is also `takeWhile` and `dropWhile`, which is like `take` and `drop`, but instead of taking or dropping a certain number of elements, they take or drop until the provided predicate is no longer satisfied.

```

ghci> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
ghci> takeWhile (<= "Birkan") ["Ahmet", "Ali", "Beril", "Mehmet"]
["Ahmet","Ali","Beril"]
ghci> takeWhile (<10) [1..]
[1,2,3,4,5,6,7,8,9]
ghci> dropWhile (<10) [1..30]
[10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]

```

Finally, there is `iterate`, which is another function to create infinite lists by starting from a value and repeatedly applying a function:

```

ghci> :t iterate
iterate :: (a -> a) -> a -> [a]
ghci> take 10 (iterate (*2) 3)
[3,6,12,24,48,96,192,384,768,1536]

```

### 5.3 Lambdas

Lambdas are anonymous functions; essentially they are nameless functions that can be defined locally, an expression that can define a function without binding it to a name. Their definition has the following syntax: `\<parameters> -> <body>`. As an example, let us define a function to multiply three values using a lambda, with both explicit and implicit currying:

```

multthree.hs

multthree :: Int -> Int -> Int -- implicit currying
multthree = \x y z -> x * y * z

multthree' :: Int -> Int -> Int -- explicit currying
multthree' = \x -> (\y -> (\z -> x * y * z))

```

These are very useful for working with higher order functions, for using functions that you cannot get with partial application without binding them to a name. You can see two examples

below, in the first one we zip two lists with a non-trivial two argument function and in the second one we extract the third element from a list of three tuples:

```
ghci> zipWith (\x y -> x^3 + x^2*y + x*y^2) [1, 2, 3] [-1, -2, -3]
[1,8,27]
ghci> map (\(_, _, z) -> z) [('a', 'b', 'c'), ('x', 'y', 'z'), ('p', 'q', 'r')]
"czr"
```

## 5.4 Composition

Chaining the application of functions is another one of the things you may wish to do often in functional programming. We could define it on our own like this <sup>10</sup>, and use it as infix for readability:

```
ghci> comp f g = \x -> f (g x)
ghci> ((+5) `comp` (*2)) 10
25
ghci> ((+5) `comp` (*2) `comp` (/10)) 10
7.0
ghci> (\x -> ((x / 10) * 2) + 5) 10 -- same, but with a lambda
7.0
```

As you can see, this is useful for rendering successive function applications more readable. The `comp` function we defined is still a bit clunky with the backticks and all, so thankfully there is a built-in function for this which is the dot (`.`). Let's have one more example with it, where we attempt to get lists having an odd number of elements less than five from a list of lists:

```
ghci> filter (\xs -> odd (length (filter (<5) xs))) [[0..5], [1..5], [10..100]]
[[0,1,2,3,4,5]]
ghci> filter (odd . length . filter (<5)) [[0..5], [1..5], [10..100]]
[[0,1,2,3,4,5]]
```

## 5.5 Application

The final useful built-in we introduce in this section is the `($)` operator, which applies its first argument to the second. This seems redundant because application happens by default, e.g. the `f x` expression will apply `f` to `x`. However, since the application operator has a very low precedence and is right associative it becomes useful for avoiding lots of nested parentheses, which makes your code more readable *and* more writable.

To illustrate, let us slightly modify the example lambda function we had above. We will try to determine if the number of elements less than seven in a given list after all its values have been doubled is not odd <sup>11</sup>. Here we go:

<sup>10</sup>This definition is not entirely valid as infixing is left-associative by default while function composition is right-associative

<sup>11</sup>Not being odd is of course being even, but hey... What could you do if I, as the programmer, simply thought that 'not odd' is more readable?

```
ghci> xs = [-1..10]
ghci> not (odd (length (filter (<7) (map (*2) xs)))) -- is this Lisp?
False
ghci> not $ odd $ length $ filter (<7) $ map (*2) xs -- pleasant 8)
False
```

It is also possible to use this operator to force application where it will not happen on its own:

```
ghci> zipWith ($) [(+1), (*7), (/40), (\x -> x*x - x)] [1, 2, 3, 4]
[2.0,14.0,7.5e-2,12.0]
```

## 6 Typeclasses

Typeclasses are another important concept in Haskell. They are somewhat similar to Java's interfaces. Just as a class can implement an interface, types can be instances of typeclasses. A typeclass is simply a set of functions. If a type is an instance of a typeclass, that means those functions declared by the typeclass have been implemented for that type and will work on it. A great example is the Prelude's `Num` typeclass which brings together numeric types. Let's get some information about it from `ghci`:

```
Prelude> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in 'GHC.Num'
instance Num Word -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int -- Defined in 'GHC.Num'
instance Num Float -- Defined in 'GHC.Float'
instance Num Double -- Defined in 'GHC.Float'
```

That's quite a lot of information! Here's an approximate Haskell to English translation of the information:

- If a type `a` is in the `Num` typeclass...
- ... it implements `(+)`, `(-)` and `(*)`, which take two values of type `a` and return another value of type `a`...
- ... it implements `negate`, `abs` and `signum`, which take a value of type `a` and also return a value of type `a`...
- ... and it implements `fromInteger`, which takes an `Integer` value and returns an `a`.
- The next line which starts with `MINIMAL` is a pragma, i.e. a compiler directive for GHC,

and not directly related to Haskell.<sup>12</sup>

- The final lines show that the basic types `Word`, `Integer`, `Int`, `Float` and `Double` are all instances of the `Num` typeclass.

In the end, this simply means that we expect a numeric type to implement addition, subtraction and multiplication; as well as negation, absolute value and signum operations. We also expect a function to transform `Integers` to that value. Of course, these definitions do not have to make sense as long as the type constraints are satisfied, we could define a type `WeirdInt` and implement addition as exponentiation. But, when used sensibly, typeclasses are a powerful mechanism for gathering similar types together.

Let's introduce some of the already defined basic typeclasses that you will come across quite often:

- **Eq**: values of types belonging to this class can be checked for equality. Instances implement the `==` and `/=` functions. Most standard types (except for functions) are instances of this typeclass.
- **Ord**: this typeclass is for types whose values can be ordered. This means that they support comparison with the standard comparison functions, and their values relate to each other as less than, equal or greater than.
- **Show**: members of the typeclass can be represented with strings. Most standard types are an instances of this typeclass, just like `Eq`. `ghci` uses functions of this typeclass when printing evaluation results. The backbone of this typeclass is `show`, which converts values to their string representation:

```
Prelude> show 77
"77"
Prelude> show [1..5]
"[1,2,3,4,5]"
Prelude> show 'a'
"'a'"
Prelude> show True
"True"
Prelude> show head

<interactive>:24:1: error:
    • No instance for (Show ([a0] -> a0)) arising from a use of 'show'
      (maybe you haven't applied a function to enough arguments?)
    • In the expression: show head
      In an equation for 'it': it = show head
```

- **Read**: is like the opposite of **Show**. It is for types that can be parsed from their string representations. The cornerstone of this typeclass is the `read` function, which reads a value from a string. Unlike `show`, `read` cannot deduce its own type when no context is given since its input is always a string:

---

<sup>12</sup>Actually, it simply shows the minimal definition necessary when instantiating the typeclass. Since there is an `|` between `negate` and `(-)`, we understand that we can get away with only defining one of those. The compiler will define the other one based on the definition of the one we provided. For example, it could define `(-) x y` as `(+) x (negate y)`.



```

Prelude> read "3.14"           -- no type provided, chaos and confusion
*** Exception: Prelude.read: no parse
Prelude> read "3.14" :: Float -- type provided, cool
3.14
Prelude> read "3.14" :: Int   -- type provided, but not valid
*** Exception: Prelude.read: no parse
Prelude> read "3.14" + 7.0    -- type deduced, result is the arg of (+7.0)
10.14

```

- **Num**: for numeric data types, we have already introduced this one.

It is possible to define polymorphic functions that only work on types belonging to a certain typeclass. This is called a **class constraint** and is provided with `=>`, a thick arrow. We can observe this in several standard functions:

```

Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
Prelude> :t read
read :: Read a => String -> a

```

One type can be an instance of multiple typeclasses. For example, see the information about `Bool`:

Note:

```

Prelude> :i Bool
data Bool = False | True -- Defined in 'GHC.Types'
instance Bounded Bool -- Defined in 'GHC.Enum'
instance Enum Bool -- Defined in 'GHC.Enum'
instance Eq Bool -- Defined in 'GHC.Classes'
instance Ord Bool -- Defined in 'GHC.Classes'
instance Read Bool -- Defined in 'GHC.Read'
instance Show Bool -- Defined in 'GHC.Show'
Similarly, multiple class constraints can exist in a function type, such as
myFunction :: (Eq a, Read a, Num b) => b -> a.

```

## 7 Custom Data Types & Typeclasses

### 7.1 data

The `data` keyword is used to define algebraic data types with the following syntax:

```
data <typename> {<type-vars>} = <value-constr1> { | <value-constr2> } { | ... }
```

Value constructors can be nullary or have arguments. Here are some examples:

data.hs

```
-- Enumerated (sum) types
data Bool' = T | F
data Color = Red | Green | Blue
-- A product type, like tuples. The type can
-- have the same name as the value constructor.
data Point = Point Double Double
-- A combination of both sum & prod. types
data Pet = Dog String Int | Cat String Int | Butterfly String
-- Some example values:
bestColor = Green
origin = Point 0.0 0.0
yourPets = [Cat "Boncuk" 3, Cat "Prenses" 7, Butterfly "Pırpır"]
```

The way to define functions that work with algebraic data types is through pattern matching:

mario.hs

```
data Character = Mario | Luigi | Yoshi

describe :: Character -> String
describe Mario = "It's-a me, Mario!"
describe Luigi = "The ultimate in second choice."
describe Yoshi = "Open the door, get on the floor..."
```

So far, the left hand side of our declarations have been concrete types. It is also possible for data declarations to declare type constructors rather than types. Just like value constructors take some values and return a new value, type constructors take some types and return a new type. In fact, you are already familiar with some type constructors. One of them is the list. `[]` is a type constructor (with special syntax). Given a concrete type, such as `Int`, it will produce a new concrete type `[Int]`. Let us define our own, which is essentially a box that may or may not contain a value<sup>13</sup>

schrodinger.hs

```
data SchrödingersBox a = Full a | Empty

checkBox :: Show a => SchrödingersBox a -> String
checkBox (Full sth) = "Aha, I found a " ++ show sth ++ "!"
checkBox Empty = "Oh, the box is empty..."
```

---

<sup>13</sup>Equivalent to the `Maybe` type constructor from the `Prelude`

```
ghci> checkBox (Full 3)
"Aha, I found a 3!"
ghci> checkBox (Full False)
"Aha, I found a False!"
ghci> checkBox Empty
"Oh, the box is empty..."
```

We can also define recursive types, like a binary tree:

```
tree.hs

data Tree a = Empty | Leaf a | Node a (Tree a) (Tree a)

sizeOf :: Tree a -> Int
sizeOf Empty = 0
sizeOf (Leaf _) = 1
sizeOf (Node _ left right) = 1 + sizeOf left + sizeOf right
```

## 7.2 deriving

The `deriving` keyword can be used to make types instances of existing typeclasses using default behavior (i.e. checking equality of fields when deriving `Eq`, directly printing value constructor names when deriving `Show` etc.):

```
ghci> data Külüstür = Murat | Serçe | R12
ghci> Murat

<interactive>:2:1: error:
    • No instance for (Show Külüstür) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
ghci> Murat == Serçe

<interactive>:3:1: error:
    • No instance for (Eq Külüstür) arising from a use of ‘==’
    • In the expression: Murat == Serçe
      In an equation for ‘it’: it = Murat == Serçe
ghci> data Külüstür = Murat | Serçe | R12 deriving (Eq, Show)
ghci> Murat
Murat
ghci> Murat == Serçe
False
```

The details of how this functionality is implemented depends on the compiler; only basic typeclasses can be derived from, and making your own typeclasses derivable from is non-standard.

## 7.3 type and newtype

The `type` declaration can be used to create type synonyms, just like C’s `typedef`:

typedekl.hs

```
type ValPair = (Double, Double)
type String' = [Char] -- exactly how String is defined
type AssocList a b = [(a, b)] -- with type variables
```

There also exists a special declaration that can be used for types that only have a single value constructor taking a single argument (i.e. useful for making a new type by wrapping an already existing one), which is the **newtype** declaration. There are some differences between using **newtype** and **data** in such cases, but these are outside our scope<sup>14</sup>:

newtype.hs

```
newtype Point = Point (Double, Double)
```

## 7.4 Record Syntax

You may have noticed that the value constructors can become a little confusing due to the lack of names of their fields. For example, how can you even guess the parameters of this class?

address.hs

```
data Address = Info String String String String String String String -- ??
```

It is possible to alleviate this problem by writing your own getters for each field, or creating lots of type synonyms; however this is clearly not very practical. This is why Haskell allows you to create getter functions for the fields on the go, with the added benefit that those become visible as field names when deriving from **Show**. Here's the redefinition of the **Address** type:

recaddr.hs

```
data Address = Info { -- fields are clarified with record syntax
    name :: String,
    addressLine :: String,
    city :: String,
    county :: String,
    zipCode :: String,
    email :: String,
    mobilePhone :: String
} deriving Show
```

<sup>14</sup>Short story: **data** is lazy while **newtype** is strict. Long story: feel free to do your own research!

```

ghci> address = Info "Çınar Akçalı" "Etlik Mah. Kıvrımlı Cad. Kanarya Apt. No:75
D:3" "Ankara" "Çankaya" "06010" "cinarakcaliceng242@gmail.com" "571 242 02 20"
ghci> address
Info {name = "\199\305nar Ak\231al\305", addressLine = "Etlik Mah. K\305vr\305ml
\305 Cad. Kanarya Apt. No:75 D:3", city = "Ankara", county = "\199ankaya", zipCo
de = "06010", email = "cinarakcaliceng242@gmail.com", mobilePhone = "571 242 02
20"}
ghci> email address
"cinarakcaliceng242@gmail.com"
ghci> mobilePhone address
"571 242 02 20"

```

## 7.5 instance

The `instance` keyword is used for making your new type an instance of an existing typeclass. For instance, let us make a new array type over a list and make it an instance of the `Num` typeclass:

```

valarray.hs

data Valarray a = Array [a] deriving Show -- diff. names to confuse less

-- could be defined more easily with common helpers
instance Num a => Num (Valarray a) where
    (+) (Array v1) (Array v2) = Array $ zipWith (+) v1 v2
    (-) (Array v1) (Array v2) = Array $ zipWith (-) v1 v2
    (*) (Array v1) (Array v2) = Array $ zipWith (*) v1 v2
    negate (Array v) = Array $ map negate v
    abs (Array v) = Array $ map abs v
    signum (Array v) = Array $ map signum v
    fromInteger x = Array [fromInteger x]

```

```

ghci> a1 = Array [1, 2, 3, 4, 5]
ghci> a2 = Array [5, 4, 3, 2, 1]
ghci> a1 + a2
Array [6,6,6,6,6]
ghci> a1 * a2
Array [5,8,9,8,5]
ghci> negate a1
Array [-1,-2,-3,-4,-5]
ghci> fromInteger 5 :: Valarray Double
Array [5.0]

```

Note that `instance` needs to be used with a concrete type. In our case, we made the declaration suitable for all `Num a`, but we could also make it work only for a specific concrete type, such as `instance Num (Valarray Double) where`.

## 7.6 class

The `class` keyword can be used to define your own typeclasses:

polygon.hs

```
class Polygon a where
  points :: a -> [(Double, Double)]
  circumference :: a -> Double
  area :: a -> Double
```

## 8 Modules

Modules are collections of related functions, types and typeclasses. The standard library contains various useful modules, and a large amount of third party libraries also exist. A great tool for browsing through their documentation is Hackage. For the standard libraries, see this.

### 8.1 Importing Modules

There are various different ways of importing modules:

modules.hs

```
-- import everything in the module
-- call like any other, e.g. sort [1, 5, 2]
import Data.List

-- import only certain functions
import Data.List (sort, intercalate)

-- import everything except some functions
import Data.List hiding (sort)

-- import with the module name as prefix
-- e.g. Data.List.sort [1, 5, 2]
import qualified Data.List

-- import with whatever name you like
-- e.g. L.sort [1, 5, 2]
import qualified Data.List as L

-- can add (sort), hiding etc. in front
import qualified Data.List as L (sort, intercalate)
```

## 8.2 Creating Your Own Modules

To make your own module, you simply add a `module (...) where` declaration <sup>15</sup> to the beginning of your module file where you list the functions you wish to export between the parentheses:

```
Vec3.hs

module Vec3 (
    Vec3,
    -- Vec3(..) exports every value constructor when there are multiple
    mag,
    dot,
    cross
) where

data Vec3 = Vec3 Double Double Double

mag :: Vec3 -> Double
mag (Vec3 x y z) = sqrt $ x^2 + y^2 + z^2

dot :: Vec3 -> Vec3 -> Double
dot (Vec3 x1 y1 z1) (Vec3 x2 y2 z2) = x1*x2 + y1*y2 + z1*z2

cross :: Vec3 -> Vec3 -> Vec3
cross (Vec3 x1 y1 z1) (Vec3 x2 y2 z2) =
    let cx = y1*z2 - y2*z1
        cy = z1*x2 - z2*x1
        cz = x1*y2 - x2*y1
    in Vec3 cx cy cz
```

You can load your module in `ghci` using `:load`. Other source files in the same directory can also import your module, as long as the file name and module name are the same.

---

<sup>15</sup>The parentheses can be omitted, which will mean that every top-level definition in the file will be exported.