

CEng 242, Programming Language Concepts

Introduction

Onur Tolga Şehitoğlu

Computer Engineering



Introduction

- Church-Turing hypothesis say all programming languages and computational devices have the same power regarding computability.
- If you can define a computation in one of the universal programming languages, you can define the same computation in any other universal programming language.
- Why do we have so many programming languages?
- Is it desirable to know as much distinct P.L. as possible?

Course Objectives

- *Not* to teach specific programming languages. Haskell, C++ and Prolog languages are tools of this course.
- Studying the common concepts of programming languages.
- Studying the different paradigms and approaches.
- What is the measure of **quality** in a programming language?
- Construct a basis for other topics like compiler design, software engineering, object oriented design, human computer interaction...

Related Areas

- Human computer interaction
- Operating systems
- Computer Architecture
- Databases and information retrieval
- Software engineering

Programming Languages vs Natural Languages

- Formal vs Informal
- Strict rules of well-formedness vs Error tolerant
- Restricted vs Unrestricted domain

What makes a language programming language?

Is any formally defined language a programming language?
(HTML?)

- **Universal:** All computation problems should be expressible.
condition+(loop and/or recursion)
- **Natural:** All features required for the application domain.
Fortran: numerical computation, COBOL: file processing,
LISP tree and list operations.
- **Implementable:** It is possible to write a compiler or
interpreter working on a computer.
Mathematics, natural language?
- **Efficient:** Works with acceptable amount of CPU and
memory.

“Hello world” in different languages:

<https://helloworldcollection.github.io/>

Paradigms

Paradigm: Theoretical or model frame. A model forming a basis for all similar approaches.

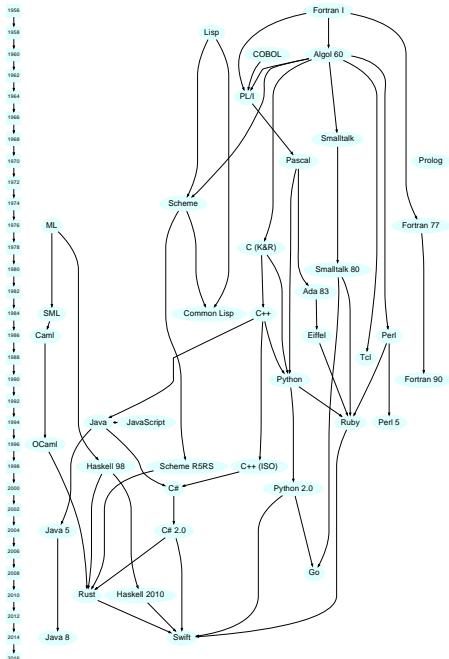
- **Imperative** Fortran, Cobol, Algol, Pascal, C, C++, Java, Basic.
- **Functional** Lisp, Scheme, ML, Haskell, Miranda
- **Object Oriented** Smalltalk, C++, Object Pascal, Eiffel, Java, Csharp.
- **Concurrent** Ada, Occam, Par-C, Pict, Oz
- **Logic** Prolog, Icon
- **Constraint Programming** CLP, CHR
- **Mixed** Parallel Prolog, Oz (A functional, logic, object oriented, concurrent language: <http://www.mozart-oz.org>)

Syntax and Semantics

- **Syntax** Form. How language is structured, how it is expressed.
- Syntax is represented by Context Free Grammars expressed in BNF (Bacus Naur Form) notation (See CEng 280, Formal Languages and Abstract Machines)
- **Semantics** What does a program mean? How it works.

Language Processors

- Compilers (gcc, javac, f77)
- Interpreters (scheme, hugs, sml, bash)
- Beautifiers, pretty printers (a2ps, ident)
- Syntax directed editors (vim, anjuta, eclipse, visual studio)
- Validators (vgrind, lint)
- Verifiers



Taken from:
<http://rigaux.org/language-study/>

O'reilly poster:
http://archive.oreilly.com/pub/a/oreilly/news/languageposter_0504.html
 download link

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)

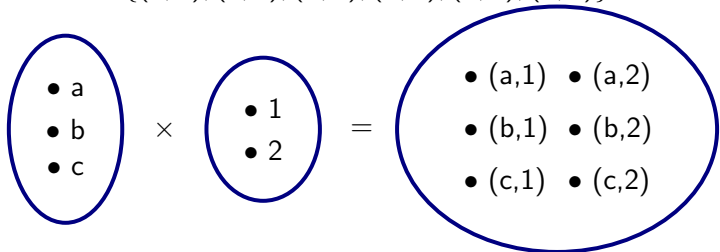
Cartesian Product

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

■ 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

- $S^n = \overbrace{S \times S \times S \times \dots \times S}^n$
double⁴ :

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

- $S^0 = \{()\}$ is 0-tuple.
- **not** empty set. A set with a single value.
- terminating value (nil) for functional language lists.
- C **void**. Means no value. Error on evaluation.
- Python: `()` . **None** used for no value.

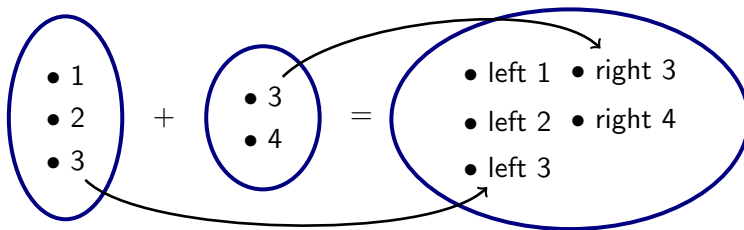
Disjoint Union

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

■ Example:

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

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



■ $\#(S + T) = \#S + \#T$

■ 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:

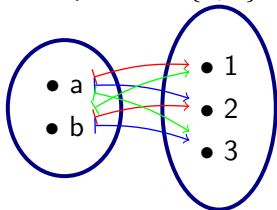
```
x.real=3.14; printf("%d\n",x.integer);
```

- **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
- $\text{double} \mapsto \text{double} ?$

-
- Cartesian mappings:

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

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

Functions

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

Cartesian Mapping vs Nested mapping

■ Pascal arrays

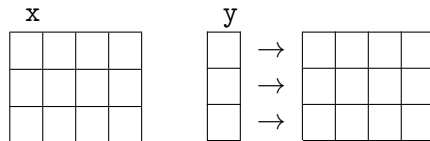
```
var
  x : array [1..3,1..4] of double;
  y : array [1..3] of array [1..4] of double;
...
x[1,3] := x[2,3]+1;      y[1,3] := y[2,3]+1;
```



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
```

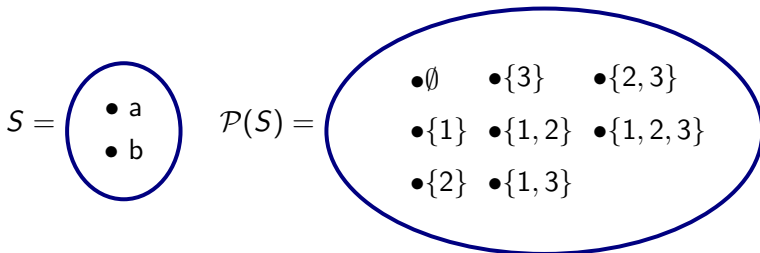
- `g 3` ✓
`f 3` ✗

- 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

- $S \stackrel{?}{\equiv} T$ How to decide?
 - **Name Equivalence**: Types should be defined at the same exact place.
 - **Structural Equivalence**: Types should have same value set. (mathematical set equality).
- Most languages use **name equivalence**.
- C example:

```
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.

- $[\textit{expr} \mid \textit{variable} \leftarrow \textit{list} , \textit{condition}]$

- Similar to set notation in math:
 $\{ \textit{expr} \mid \textit{var} \in \textit{list}, \textit{condition} \}$

- 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 Language Concepts

Higher Order Functions

Onur Tolga Şehitoğlu

Computer Engineering



Outline

- 1 Lambda Calculus
- 2 Introduction
- 3 Functions
 - Curry
 - Map
 - Filter
 - Reduce
 - Fold Left
 - Iterate
 - Value Iteration (for)
- 4 Higher Order Functions in C
- 5 Some examples
 - Fibonacci
 - Sorting
 - List Reverse

Lambda Calculus

- 1930's by Alonso Church and Stephen Cole Kleene
- Mathematical foundation for computability and recursion
- Simplest functional paradigm language
- $\lambda var.expr$
defines an anonymous function. Also called **lambda abstraction**
- $expr$ can be any expression with other lambda abstractions and applications. Applications are one at a time.
- $(\lambda x.\lambda y.x + y) 3 4$

- In ' $\lambda var.expr$ ' all free occurrences of var is bound by the λvar .
- Free variables of expression $FV(expr)$
 - $FV(name) = \{name\}$ if $name$ is a variable
 - $FV(\lambda name.expr) = FV(expr) - \{name\}$
 - $FV(M N) = FV(M) \cup FV(N)$
- **α conversion**: expressions with all bound names changed to another name are equivalent:

$$\lambda f.f\ x \equiv_{\alpha} \lambda y.y\ x \equiv_{\alpha} \lambda z.z\ x$$

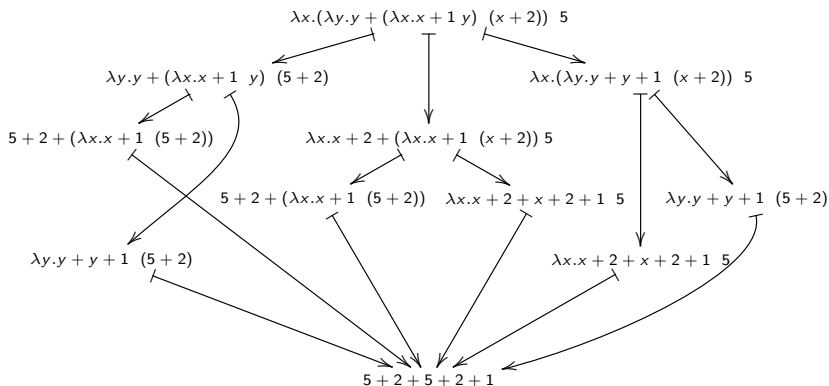
$$\lambda x.x + (\lambda x.x + y) \equiv_{\alpha} \lambda t.t + (\lambda x.x + y) \equiv_{\alpha} \lambda t.t + (\lambda u.u + y)$$

$$\lambda x.x + (\lambda x.x + y) \not\equiv_{\alpha} \lambda x.x + (\lambda x.x + t)$$

β Reduction

- Basic computation step, function application in λ -calculus
- Based on substitution. All bound occurrences of λ variable parameter is substituted by the actual parameter
- $(\lambda x.M)N \mapsto_{\beta} M[x/N]$ (all x 's once bound by lambda are substituted with N).
- $(\lambda x.(\lambda y.y + (\lambda x.x + 1) y)(x + 2)) 5$
- If no further β reduction is possible, it is called a normal form.
- There can be different reduction strategies but should reduce to same normal form. (Church Rosser property)

All possible reductions of a λ -expression. All reduce to the same normal form.



Introduction

- Mathematics:

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

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

- in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

$$u \ 3 = (3+3)*(3+3) = 36$$

- $\text{compose}: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

- “compose” function is a function getting two functions as parameters and returning a new function.
- Functions getting one or more functions as parameters are called [Higher Order Functions](#).
- Many operations on functional languages are repetition of a basic task on data structures.
- Functions are first order values \rightarrow new general purpose functions that uses other functions are possible.

Functions/Curry

- Cartesian form vs curried form:

$$\alpha \times \beta \rightarrow \gamma \text{ vs } \alpha \rightarrow \beta \rightarrow \gamma$$

- Curry function gets a binary function in cartesian form and converts it to curried form.

```
curry f x y = f(x,y)
add (x,y) = x+y
increment = curry add 1
--
increment 5
6
```

- curry: $(\alpha \times \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$
- Haskell library includes it as `curry`.

Functions/Map

```

square x = x*x
day no = case no of 1 -> "mon" ; 2 -> "tue" ; 3 -> "wed";
                4 -> "thu" ; 5 -> "fri" ; 6 -> "sat" ; 7 -> "sun"
map func [] = []
map func (el:rest) = (func el):(map func rest)
-----
map square [1,3,4,6]
[1,9,6,36]
map day [1,3,4,6]
["mon","wed","thu","sat"]

```

- $\text{map}:(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$
- Gets a function and a list. Applies the function to all elements and returns a new list of results.
- Haskell library includes it as `map`.

Functions/Filter

```

iseven x = if mod x 2 == 0 then True else False
isgreater x = x>5

filter func [] = []
filter func (el:rest) = if func el then
                        el:(filter func rest)
                        else (filter func rest)

----
filter iseven [1,2,3,4,5,6,7]
[2,4,6]
filter isgreater [1,2,3,4,5,6,7]
[6,7]

```

- $\text{filter}:(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$
- Gets a boolean function and a list. Returns a list with only members evaluated to True by the boolean function.
- Haskell library includes it as `filter`.

Functions/Reduce (Fold Right)

```

sum x y = x+y
product x y = x*y

reduce func s [] = s
reduce func s (el:rest) = func el (reduce func s rest)
-----
reduce sum 0 [1,2,3,4]
10                                // 1+2+3+4+0
reduce product 1 [1,2,3,4]
24                                // 1*2*3*4*1

```

- $\text{reduce}:(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
- Gets a binary function, a list and a seed element. Applies function to all elements right to left with a single value.
 $\text{reduce } f \ s \ [a_1, a_2, \dots, a_n] = f \ a_1 \ (f \ a_2 \ (\dots \ (f \ a_n \ s)))$
- Haskell library includes it as `foldr`.

- Sum of a numbers in a list:
`listsum = reduce sum 0`
- Product of a numbers in a list:
`listproduct = reduce product 1`
- Sum of squares of a list:
`squaresum x = reduce sum 0 (map square x)`

Functions/Fold Left

```

subtract x y = x - y
foldl func s [] = s
foldl func s (el:rest) =
    foldl func (func s el) rest
----
reduce subtract 0 [1,2,3,4]
-2                                // 1-(2-(3-(4-0)))
foldl subtract 0 [1,2,3,4]
-10                               // (((0-1)-2)-3)-4)

```

- $\text{foldl}:(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$
- Reduce operation, left associative.:

$$\text{reduce } f \ s \ [a_1, a_2, \dots, a_n] = f \ (f \ (f \ \dots (f \ s \ a_1) \ a_2 \ \dots)) \ a_n$$
- Haskell library includes it as `foldl`.

Functions/Iterate

```
twice x = 2*x

iterate func s 0 = s
iterate func s n = func (iterate func s (n-1))
----
iterate twice 1 4
16                                // twice (twice ( twice (twice 1))
iterate square 3 3
6561                             // square (square (square 3))
```

- $\text{iterate}:(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{int} \rightarrow \alpha$
- Applies same function for given number of times, starting with the initial seed value. $\text{iterate } f \ s \ n = f^n \ s = \underbrace{f(f(f \dots (f \ s)))}_n$

Functions/Value Iteration (for)

```

for func s m n =
    if m>n then s
    else for func (func s m) (m+1) n

----
for sum 0 1 4
10          // sum (sum (sum (sum 0 1) 2) 3) 4
for product 1 1 4
24          // product (product (product (product 1 1) 2) 3) 4

```

- $\text{for}:(\alpha \rightarrow \text{int} \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{int} \rightarrow \text{int} \rightarrow \alpha$
- Applies a binary integer function to a range of integers in order.

$$\text{for } f \text{ s } m \text{ n} = f(f(f(f(f \text{ s } m) (m+1)) (m+2)) \dots) n$$

- multiply (with summation):
`multiply x = iterate (sum x) x`
- integer power operation (Haskell '^'):
`power x = iterate (product x) x`
- sum of values in range 1 to n:
`seriesum = for sum 0 1`
- Factorial operation:
`factorial = for product 1 1`

Higher Order Functions in C

C allows similar definitions based on function pointers. Example: `bsearch()` and `qsort()` functions in C library.

```
typedef struct Person { char name[30]; int no;} person;
int cmpnmba(void *a, void *b) {
    person *ka=(person *)a; person *kb=(person *)b;
    return ka->no - kb->no;
}
int cmpnames(void *a, void *b) {
    person *ka=(person *)a; person *kb=(person *)b;
    return strcmp(ka->name,kb->name,30);
}
int main() {
    int i;
    person list []={{ "veli",4},{ "ali",12},{ "ayse",8},
                    { "osman",6},{ "fatma",1},{ "mehmet",3}};
    qsort(list,6,sizeof(person),cmpnmba);
    ...
    qsort(list,6,sizeof(person),cmpnames);
    ...
}
```

Fibonacci

Fibonacci series: 1 1 2 3 5 8 13 21 ..

$fib(0) = 1$; $fib(1) = 1$; $fib(n) = fib(n-1) + fib(n-2)$

```
fib n = let f (x,y) = (y,x+y)
        (a,b) = iterate f (0,1) n
        in b

----
fib 5      // f(f(f(f(0,1))))
8          //(0,1)->(1,1)->(1,2)->(2,3)->(3,5)->(5,8)
```

Sorting

Quicksort:

- 1** First element of the list is x and rest is xs
- 2** select smaller elements of xs from x , sort them and put before x .
- 3** select greater elements of xs from x , sort them and put after x .

```
notfunc f x y = not (f x y)

sort _ [] = []
sort func (x:xs) = (sort func (filter (func x) xs)) ++
                  (x: (sort func (filter ((notfunc func) x) xs)))
---
sort (>) [5,3,7,8,9,3,2,6,1]
[1,2,3,3,5,6,7,8,9]
sort (<) [5,3,7,8,9,3,2,6,1]
[9,8,7,6,5,3,3,2,1]
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
    reverse2 ' [] x = x
    reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000]      // slow
reverse2 [1..10000]      // fast
```


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

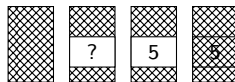
6 Summary

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

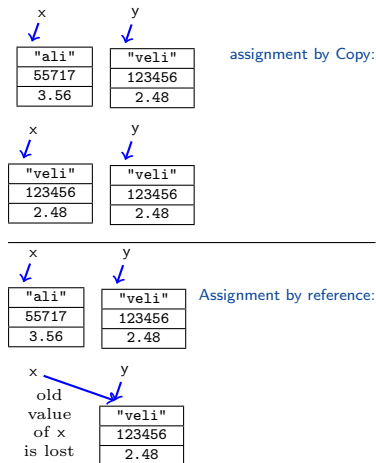
- Array size is completely variable. Arrays may expand or shrink at run time. Script languages like Perl, PHP, Python
- Perl example:

```
@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.



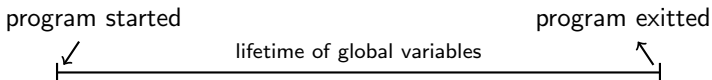
- Assignment semantics is defined by the language design
- C structures follows copy semantics. Arrays cannot be assigned. Pointers are used to implement reference semantics. C++ objects are similar.
- Java follows copy semantics for primitive types. All other types (objects) are reference semantics.
- 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

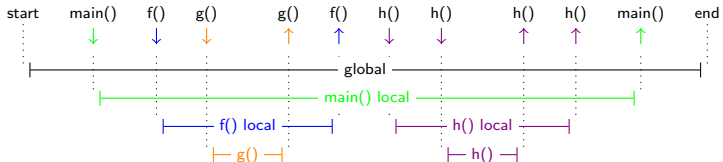
- Life of global variables start at program startup and finishes when program terminates.
- In C, all variables not defined inside of a function (including `main()`) are global variables and have global lifetime:



- What are static variables inside functions in C?

Local lifetime

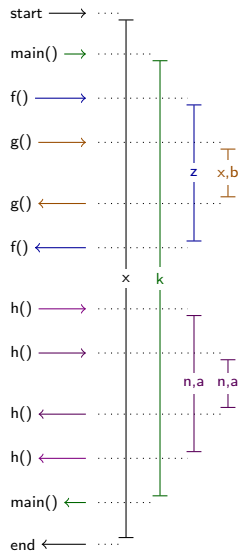
- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.
- Formal parameters are local variables.
- Multiple instances of same local variable may be alive at the same time in recursive functions.



```

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

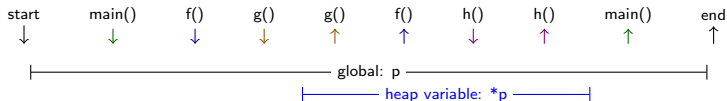
- **Heap variables:** Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.
- Heap variables are accessed via pointers. Some languages use references

```
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`.

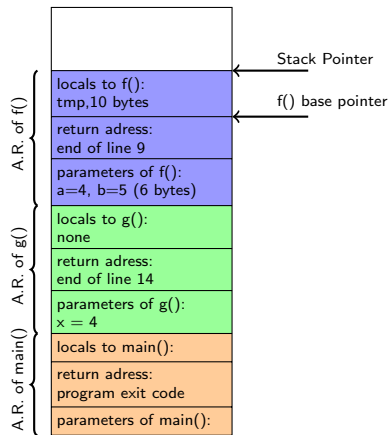
Local Variables

- 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

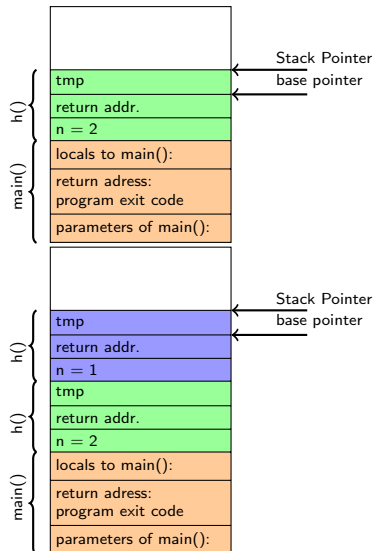
- Caller side:
 - 1 Push parameters
 - 2 Push return address and jump to function code start (usually a single CPU instruction like `callq`)
- Function entry:
 - 1 Set base pointer to current stack pointer
 - 2 Advance stack pointer to size of local variables
- Function body can access all local variables 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

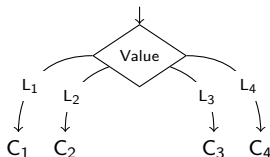
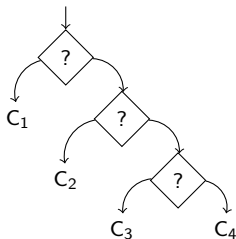
- Composition of a block from multiple statements
- **Sequential commands:** $\{ C_1 ; C_2 ; \dots ; C_n \}$
A command is executed, after it finishes the next command is executed,...
- Commands enclosed in a block behaves like single command: “if” blocks, loop bodies,...
- **Collateral commands:** $\{ C_1, C_2, \dots, C_n \}$ (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);  
    ...  
}
```

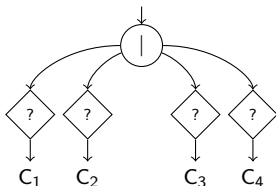
Conditional commands

- Commands to choose between alternative commands based on a condition
- in C : `if (cond) C_1 else C_2 ;`
`switch (value) { case L_1 : C_1 ; case L_2 : C_2 ; ... }`
- 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.

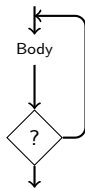
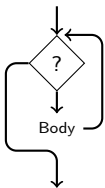


Iterative statements

- 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

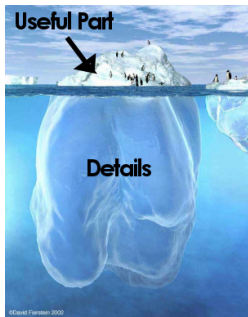
- Copy Mechanisms
- Binding Mechanisms
- Pass by Name

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)

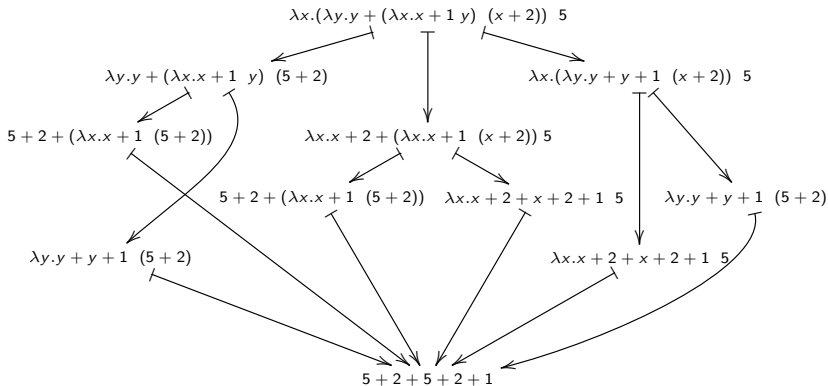
# Evaluation Order

- **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

## 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

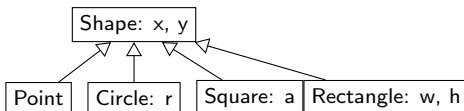


# Subtyping

- 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)

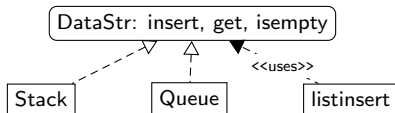
move (x=_,y=_ | rest) b c = (x=b,y=c | rest)

(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)
```

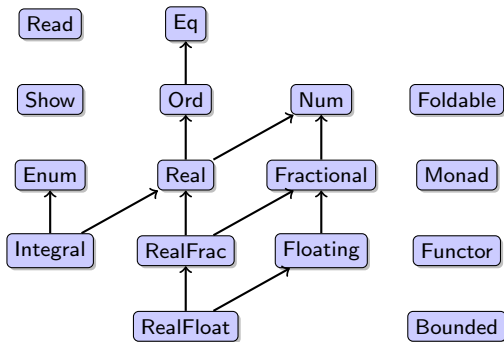
# Haskell Classes

- 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);
```



# 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

# 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

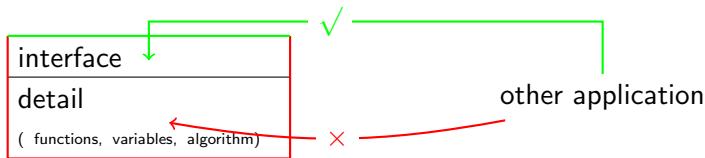
Managing the complexity → Re-usable code and abstraction.

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 Languages

## Control Flow

Onur Tolga Şehitoğlu

Computer Engineering



# Outline

1 Control Flow

2 Jumps

3 Escapes

4 Exceptions

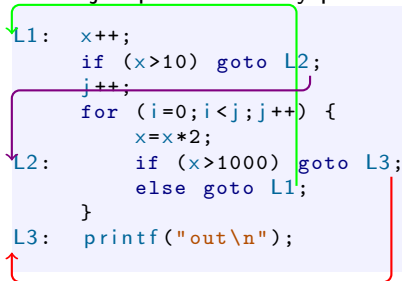
5 Co-routines

# Control Flow

- Usual control flow: a command followed by the other.  
Executed in sequence. [single entrance - single exit](#)
- Commands to change control flow and transfer execution to another point: [sequencers](#)
  - Jumps
  - Escapes
  - Exceptions

# Jumps

- Jumps transfer control to a point in the code. The destination is marked with **labels**
- When jumps to arbitrary positions are possible...:



```
L1: x++;
 if (x>10) goto L2;
 j++;
 for (i=0; i<j; j++) {
 x=x*2;
L2: if (x>1000) goto L3;
 else goto L1;
 }
L3: printf("out\n");
```

The diagram illustrates the flow of control in the provided code. A green arrow starts at the beginning of the code and points to the `L1` label. A purple arrow starts at the `goto L2` statement and points to the `L2` label. A red arrow starts at the `goto L3` statement and points to the `L3` label. The code is enclosed in a light blue box.

- Called **spaghetti coding**

- Unrestricted jumps  $\Rightarrow$  spaghetti coding.
- GCC extension allows storing labels in variables. Would you like to debug that code? 😊
- Further problems. Which jumps have problems?:

```

L1:
 goto L2; ①

 for (i=0; i<10; i++) {
 int x=t;
L2:
 goto L1; ②
 ...
 goto L2: ③
 }

```

- Lifetime and values of local variables? Values of index variables?
- C: Labels are local to enclosing block. No jumps allowed into the block. Newer languages avoid jumps.
- Single entrance multiple exit is still desirable.  $\rightarrow$  escapes

# Escapes

- Restricted jumps to out of textually enclosing block(s)
- Depending on which enclosing block to jump out of:
  - loop: `break` sequencer.
  - loops: `exit` sequencer.
  - function: `return` sequencer.
  - program: `halt` sequencer.

- **break sequencer** in C, C++, Java terminates the innermost enclosing loop block.
- **continue** in C, C++ stays in the same block but ends current iteration.
- **exit sequencer** in Ada or labeled break in Java can terminate multiple levels of blocks by specifying labels. Java code:

```
L1: for (i=0;i<10;i++) {
 for (j=i;j<i;j++) {
 if (...) break;
 else if (...) continue;
 else if (...) break L1;
 else if (...) continue L1;
 s+=i*j;
 }
}
```



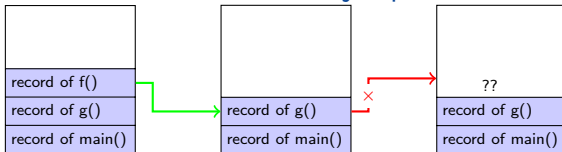
- **return sequencer** exist in most languages for terminating the innermost function block.
- **halt sequencer** either provided by operating system or PL terminates the program.
- Consider jump inside of a block or jump out of a block for the function case:

```
int f(int n) {
 int a;

L1: if (n<0) goto L2; ①
 else if (n=1) return 1;
 else return f(n-1)*n;
}

int main() {
 ...
 f(12);
L2:
 goto L1: ②
}
```

- Jump out of a function block, jump inside of a function block
- Jumps update current instruction pointer. But what about environment, activation record (run-time stack)?
- Possible only for one direction if stack position can be recovered. Called **non-local jumps**



- Are non-local jumps useful? One of the uses: unexpected error occurring inside of many levels of recursion. Jump to the outer-most related caller function. **Exceptions**

# Exceptions

- Controlled jumps out of multiple levels of function calls to an outer control point (handler or catch)
- C does not have exceptions but non-local jumps possible via `setjmp()`, `longjmp()` library calls.
- C++ and Java: `try {...} catch(...) {...}`
- Each try-catch block introduces a non-local jump point. try block is executed and whenever a `throw expr` command is called in any functions called (even indirectly) inside try block execution jumps to the `catch()` part.
- try-catch blocks can be nested. Execution jumps to closes catch block with a matching type in the parameters with the thrown expression.

- Conventional error handling. Propagate errors with return values.

```

...
int searchopen(char *f) { ...
 /* if search fails error occurs here*/
 return -5;
...}
int openparse(char *f) { ...
 if ((r = searchopen(f)) < 0)
 return r;
 else ...
}
int main() { ...
 if ((rv=openparse("file.txt")) < 0) {
 /*handle error here */
 }
 ...
}

```

## ■ Error handling with try-catch. (based on run-time stack)

```

...
enum Exception { NOTFOUND, ..., PERMS};
void searchopen(char *f) { ...
 /* if open fails error occurs here */
 throw PERMS;
...}
void openparse(char *f) { ...
 searchopen(f);
 ...
}
int main() { ...
 try {...
 openparse("file.txt");
 ...
 } catch(Exception e) {
 /*handle error here */
 }
 ...
}

```

Nested exceptions are handled based on types. C++:

```
int main() {... try { C1; f() ; C2 } catch (double a) {...}}
```

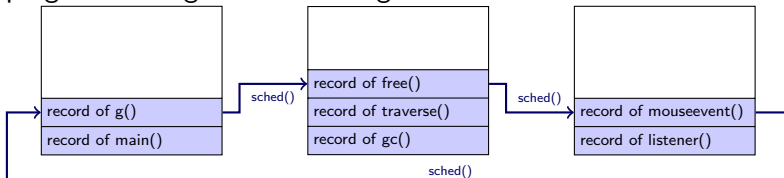
```
void f() {...; try {...; g() ; ... } catch (int a) {...} }
```

```
void g() {...; throw 4; ... ; throw 1.5; ...}
```

In case no handlers found a run time error generated. Program halts.

# Co-routines

- Sequential flow: local jumps, subroutine calls, exceptions
- Concurrent flow: multiple contexes (stack and instruction pointer). Execution switches between them.
- Multiple uses: [callbacks](#), [generators](#) (iterators), [threads](#), [fibers](#), [asynchronous](#), [event based](#), or [concurrent](#) programming
- Non-local jumps to different environments guided coordinated programs or a global scheduling mechanism:



# Programming Language Concepts

## Logic Programming Paradigm

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği





# Outline

- 1 Introduction
- 2 Prolog basics
- 3 Prolog Terms
- 4 Unification
- 5 Backtracking
- 6 List Processing
- 7 Arithmetical Operations
- 8 List Examples
- 9 Cut

# Logic Programming Paradigm

- Based on logic and [declarative programming](#)
- 60's and early 70's
- Prolog (**P**rogramming in **l**ogic, 1972) is the most well known representative of the paradigm.
- Prolog is based on [Horn clauses](#) and [SLD resolution](#)
- Mostly developed in [fifth generation computer systems project](#)
- Specially designed for theorem proof and artificial intelligence but allows general purpose computation.
- Some other languages in paradigm: ALF, Frill, Gödel, Mercury, Oz, Ciao,  $\lambda$ Prolog, datalog, and CLP languages

# Constraint Logic Programming

- Clause: disjunction of universally quantified literals,

$$\forall(L_1 \vee L_2 \vee \dots \vee L_n)$$

- A logic program clause is a clause with exactly one positive literal

$$\begin{aligned}\forall(A \vee \neg A_1 \vee \neg A_2 \dots \vee \neg A_n) &\equiv \\ \forall(A \Leftarrow A_1 \wedge A_2 \dots \wedge A_n)\end{aligned}$$

- A goal clause: no positive literal

$$\forall(\neg A_1 \vee \neg A_2 \dots \vee \neg A_n)$$

- Proof by refutation, try to unsatisfy the clauses with a goal clause  $G$ . Find  $\exists(G)$ .
- Linear resolution for definite programs with constraints and selected atom.

# What does Prolog look like?

```
father(ahmet, ayse).
father(hasan, ahmet).
mother(fatma, ayse).
mother(hatice, fatma).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

- CLP on first order terms. (Horn clauses).
- **Unification**. Bidirectional.
- **Backtracking**. Proof search based on trial of all matching clauses.

# Prolog Terms

- Every valid phrase in prolog is a **Term** and instead of strict type checking, **unification** is used. A term can be one of the following:
  - Atoms:
    - 1 Strings with starting with a small letter and followed by any letter, digit and `_`. `[a-z][a-zA-Z_0-9]*`
    - 2 Strings consisting of only punctuation symbols as:  
`[+~*/\^<>=~:~.?@#$!&]+`
    - 3 `[]`, `{}`, `;`, `!` are only treated as atoms in these forms (alone and only spaces in between).
    - 4 Any string enclosed in single or back quotes. Quotes are not part of the atom.
  - Numbers
    - 1 Any integer `[0-9]+`
    - 2 Any floating point value `[0-9]+.[0-9]+`
    - 3 Any scientific notation value `[0-9]+.[0-9]+e[0-9]+`

## ■ Variables:

- 1 Strings with starting with a capital letter or `_` and consist of `[_A-Z][a-zA-Z_0-9]*`
- 2 `_` alone is the universal match symbol. Not variable

## ■ Structures:

- starts with an `atom` head. No number, no variable
- has one or more arguments enclosed in paranthesis, separated by comma
- no space between structure head and paranthesis.
- arguments can be any valid prolog term, including other structures.

| Term                | Atom | Num. | Var. | Struct. | not a term |
|---------------------|------|------|------|---------|------------|
| hELLO               | ✓    |      |      |         |            |
| Hello               |      |      | ✓    |         |            |
| _abc                |      |      | ✓    |         |            |
| 'A_and_B'           | ✓    |      |      |         |            |
| "hello"             |      |      |      | ✓       |            |
| :->>                | ✓    |      |      |         |            |
| :-P                 |      |      |      |         | ✓          |
| --                  |      |      | ✓    |         |            |
| 1e1                 |      | ✓    |      |         |            |
| 1.0e1               |      | ✓    |      |         |            |
| 0.2                 |      | ✓    |      |         |            |
| .2                  |      |      |      |         | ✓          |
| 3.                  |      |      |      |         | ✓          |
| 0000123             |      | ✓    |      |         |            |
| x(4)                |      |      |      | ✓       |            |
| ++(a,b)             |      |      |      | ✓       |            |
| R(3)                |      |      |      |         | ✓          |
| 2(4)                |      |      |      |         | ✓          |
| a(a(a,a(a(a(a)))))) |      |      | ✓    |         |            |
| a(X,Y,.(X),2,3)     |      |      |      | ✓       |            |



# Syntax Elements

- A Prolog program consists of **clauses** or **predicates**.
- **Unit clauses** are structure or atom terms followed by a dot.  
`father(ayse, ahmet).`
- Unit clauses are considered as facts, no implication.
- Non-unit clauses consists of a **head clause** and a **body**  
`grand(X, Y) :- parent(X,Z), parent(Z,X).`
- In order to prove head clause, body should be proven.
- Body consist of structures seperated by comma, semi-colon and optionally combined with parantheses.  
`uncle(X,Y) :- (brother(X,Z), father(Z,Y)) ; (brother(X,Z), mother(Z,Y)).`
- Comma stands for conjunction ( $\wedge$ ), semicolon stands for disjunction ( $\vee$ ).
- Structures in the body are **goal clauses** to be proven.

# Prolog Lists

- `[1,2,3]` is parsed and interpreted as `.(1,.(2,.(3,[])))`
- `[Head | Tail]` form is interpreted as `.(Head , Tail)`
- `[]` denotes empty list
- `[1,2,3 | R]` is interpreted as `.(1, .(2, .(3, R)))`
- strings in double quotes like `"abc"` are interpreted as list of ASCII numbers as `[97, 98, 99]`.
- As Prolog structures can contain arbitrary terms, lists are heterogeneous as `[1, 2.1, a(b,c), [a,b,c], "hello"]` is a valid list.

# Unification

- In functional languages, caller arguments are pattern-matched against the function definition. This operation is also called **unification**. All constructors and values in caller are matched against the patterns in the definition. The variables in definition are **instantiated** with the values in the caller.
- Unification in Prolog is bi-directional. Both the defining clause and goal clause have variables instantiated.

`same(X,X).`

goal: `same(ali,Y).`

`X = ali, Y = X, Y = ali`

- Result of a unification can result in some variable instantiations as:

`X = ali, Y = ali`  $\Rightarrow$  `true`

# Unification of Terms

unification of  $x$  and  $y$  is successfull,  $x = y \Leftrightarrow$

- 1  $x$  is atom or number and  $y$  is the same atom or number
- 2  $x$ , and  $y$  are structures with same arity  $n$ ,  
 $x = h_x(x_1, x_2, \dots, x_n)$ ,  $y = h_y(y_1, y_2, \dots, y_n)$  and  
 $h_x = h_y$  and  $\forall x_i = y_i$ ,  $i = 1, \dots, n$ . Head and all coressponding  
 elements are unified.
- 3 If  $x$  is a variable and  $x = y$  is compatible with the current set  
 of instantiations, unification is successfull with  $x = y$  is added  
 to current set of instantiations.
- 4 Otherwise, unification fails.

|                                            |                                                            |
|--------------------------------------------|------------------------------------------------------------|
| <code>a = b</code>                         | false                                                      |
| <code>'abc' = abc</code>                   | true                                                       |
| <code>X = 12</code>                        | true $\Leftarrow$ X=12                                     |
| <code>a(1,X) = a(Y,2)</code>               | true $\Leftarrow$ X=2, Y=1                                 |
| <code>a(1,X) = a(Y,Y)</code>               | true $\Leftarrow$ X=Y=1                                    |
| <code>a(1,X,X) = a(Y,Y,2)</code>           | false (Y=1, X=Y, X=2)                                      |
| <code>a(1,_) = a(1)</code>                 | false (different arities)                                  |
| <code>X = a(X)</code>                      | true $\Leftarrow$ X = a(X) (cannot display but succesfull) |
| <code>a(c(X,d),c(a,Y),X) = a(Y,Z,t)</code> | true $\Leftarrow$ X = t, Y = c(t,d) , Z = c(a,c(t,d))      |
| <code>a(c(X,d),c(a,Y)) = a(Y,X)</code>     | true $\Leftarrow$ X = c(a,Y), Y = c(X,d)                   |

# Prolog Program

- A Prolog program can be written by putting all alternatives as a separate head clause with same name and arity.
- You define **relations** instead of functions returning a value.
- For example, not a membership test function but a **member relation**.
- Membership relation for a list can be defined verbally as:  
 “*x* is member of list *lst* if either *x* is the first element of the *lst* or it is the member of the remaining list”
- Each alternative is another **member(X,LST)** clause.  

```
member(X, [First | Remaining]) :- X = First.
```

```
member(X, [First | Remaining]) :- member(X, Remaining).
```
- Shorter form:  

```
member(X, [X | _]).
```

```
member(X, [_ | Remaining]) :- member(X, Remaining).
```

# Prolog Interpreter

- [Gnu Prolog](#) or [Sicstus Prolog](#) are free alternatives.
- entering '[\[filename\].](#)' in interpreter loads the clauses from `filename.pl`.
- '?- ' prompt asks user to enter goal clauses like:  
?- [member\(b, \[a,b,c\]\).](#)
- Prolog checks if this goal can be proven with the current program and replies [yes](#), [no](#)
- If there are alternatives, it prompts [true ?](#) and asks for continuation. pressing enter will terminate, [;](#) will try other alternatives.

```

~$ swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
...
Please visit http://www.swi-prolog.org for details.

?- [testmember]. % load testmember.pl
true.

?- member(b,[a,b,c]).
true % hit enter
?- member(d,[a,b,c]).
false.

?- member(b,[a,b,c]).
true ; % hit ; , try alternatives
false. % no other alternatives true
?- member(b,[a,b,b]).
true ; % hit ; , try alternatives
true ; % one more alternative, no other
false.

?- member(X,[a,b,c]). % ask who is member of [a,b,c]?
X = a ;
X = b ;
X = c ;
false.

```

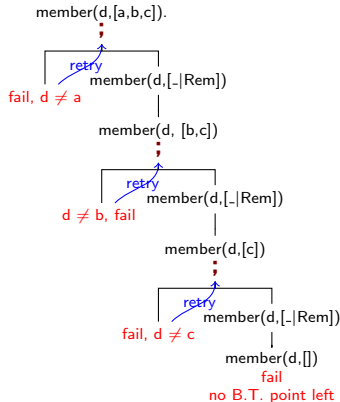
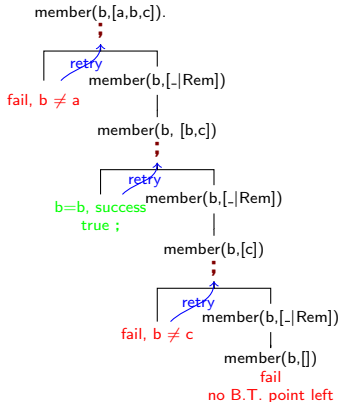


# Backtracking

- Backtracking is the search procedure of Prolog and makes it a universal programming language.
- Each alternative head clause that can be unified with goal clause is a backtracking point.
- similarly each operand of ';' is a backtracking point.
- Prolog saves the current state in backtracking points. On failure, tries the next backtracking branch.
- On success, if user hits ';' in prompt, it resumes search from the next backtracking point.

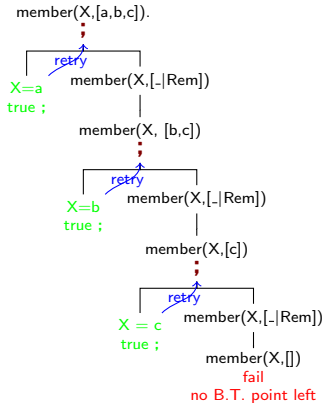
`member(X, [X | _]).`

`member(X, [_ | Rem]) :- member(X, Rem).`



```
member(X, [X | _]).
```

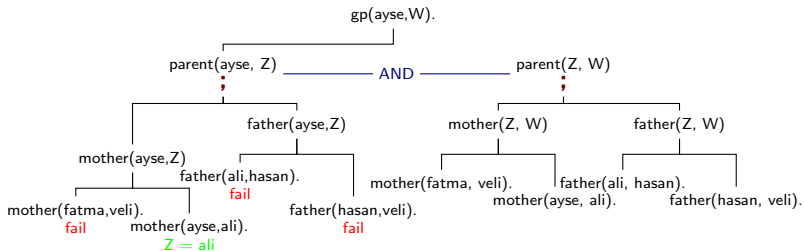
```
member(X, [_ | Rem]) :- member(X, Rem).
```



```

mother(fatma,veli). father (ali , hasan). parent(X,Y) :- mother(X,Y).
mother(ayse, ali). father (hasan, veli). parent(X,Y) :- father(X,Y).
gp(X,Y) :- parent(X,Z), parent(Z,Y).

```

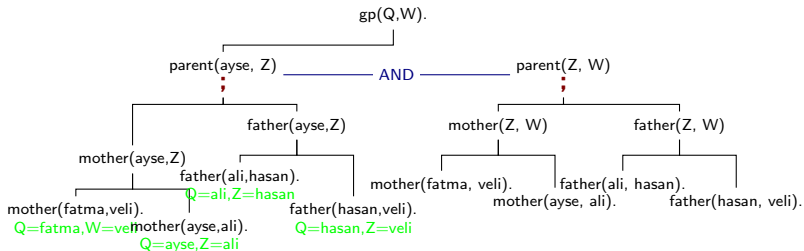


Only solution from left branch is  $Z=ali$ , applied to right branch. `father ( ali , hasan)` matches. Result is  $W = hasan$ .

```

mother(fatma,veli). father (ali , hasan). parent(X,Y) :- mother(X,Y).
mother(ayse, ali). father (hasan , veli). parent(X,Y) :- father(X,Y).
gp(X,Y) :- parent(X,Z), parent(Z,Y).

```



For each solution in left parent branch it backtracks and test solution from right parent branch, keeping the instantiated variables.  $Z=ali$  and  $Z=hasan$  returns success. Results are:  $Q=ayse, W=hasan$  and  $Q=ali, W=veli$

# List Processing

## ■ Appending lists.

```
append([], LST, LST).
append([H | Rem], LST, [H | Res]) :- append(Rem, LST, Res).
```

## ■ `append(X,Y,[a,b,c,d])` works as well.

## ■ Reverse:

```
reverse([], []). % reverse of empty list
reverse([H|Rem],Rev) :- reverse(Rem, RR), append(RR, [H], Rev).
```

## ■ Efficient reverse:

```
reverse2([], L, L). % no element left, result is stack
reverse2([H | Rem],P, L) :- reverse2(Rem, [H | P], L). % insert
reverse(LST, LSTREV) :- reverse2(LST, [], LSTREV).
```

# Arithmetical Operations

- $X = 3 * 5$  is equivalent to  $X = *(3,5)$  and does not make any calculation.
- A special operator 'is' evaluates the expressions:  
 $X \text{ is } 3 * 5$  will instantiate  $X$  to 15.
- `is` requires right handside to be fully instantiated (no variables without a value) and evaluates it, the resulting number is unified with LHS.
- ' $2+X \text{ is } 5$ ' is equivalent to unification of  $+(2,X)$  to 5, which fails.
- Comparison operators also evaluate their both operands which should be fully instantiated:  
 $< , > , = < , > = , =: = , = \backslash =$
- Some of the arithmetic operators (in evaluation context):  
 $+, +, *, / , //$  (int.div.) `mod` .
- Also mathematical functions can be used:  
`sin , cos , ... , exp , log , log10 , abs , round , ceil , ...`

# Build-in Predicates

- Testing term type:

`var(T)`, `nonvar(T)`, `atom(T)`, `number(T)`, `atomic(T)`, `ground(T)`.

- Equivalence which does not cause instantiation:

`X == X` strict, `X \== Y` strict not eq., `X \= Y` not unifiable.

- Bidirectional list to term conversion:

`X =.. [+ ,b ,c] → X = +(b ,c)`

`f(a ,b ,c) =.. X → X = [f ,a ,b ,c]`

`X =.. [t] → X = t`

- List predicates:

`member/2`, `length/2`, `append/3`, `select/3`, `union/3`, `reverse/2`

- Displaying all clauses with given name and arity:

`listing(father/2)` `listing(reverse/_)`.

- Find all solutions in a list:

`findall(X, father(ali,X), L)`, `setof(X, father(ali,X), L)`.



# Functional to Logical

- A function can be converted into a relation by adding a result argument. Result can be propagated from recursive calls in this argument.
- Haskell:

```
length [] = 0
length (_,r) = (length r) + 1
```

- Prolog:

```
length([], Res) :- Res is 0.
length([_|R], Res) :- length(R, RLen), Res is RLen + 1.
```

# Examples: List

- Relations may work different ways. Give all partitions of a list:

`append(P1, P2, [1,2,3]).`

`P1=[],P2=[1,2,3]; P1=[1],P2=[2,3]; P1=[1,2],P2=[3]; P1=[1,2,3],P2=[].`

- Selecting/removing element from a list1 results in list2:

```
select(E, [E|R],R).
select(E, [H|R], [H|R2]) :- select(E, R, R2).
```

`select(3,[1,2,3], L)`

`L=[1,2]`

- For all elements of list1, give remaining list as well:

`select(A,[1,2,3], L)`

`A=1,L=[2,3]; A=2,L=[1,3]; A=3,L=[1,2]`

- Inserting element to all positions of list1 results in list2:

`select(a,[1,2,3], L)`

`L=[a,1,2,3]; L=[1,a,2,3]; L=[1,2,a,3]; L=[1,2,3,a]`

## ■ Subset:

```
subset([], []).
subset(Rsub, [_|R]) :- subset(Rsub, R).
subset([H|Rsub], [H|R]) :- subset(Rsub, R).
```

## ■ Permutations:

```
% insert H to all positions in the remainder permutations
perm([], []).
% get first element H, get perms of rest
% insert H in every position of rest perm
perm([H|R], HINS) :- perm(R, RP), select(H, HINS, RP).
```

## ■ N combinations:

```
combin(_, [], 0). % 0 combination is empty
% all N combin of remain. is also in comb.
combin([_|R], Res, N) :- N > 0, combin(R, Res, N).
% N-1 combin of remain. add H
combin([H|R], [H|Res], N) :- N > 0, M is N-1,
 combin(R, Res, M).
```

## ■ N permutations:

```
permut(_, [], 0). % 0 permutation is empty
% for all elements H of L, permute remaining.
permut(L, [H|RP], N) :- N > 0, M is N-1,
 select(H, L, Rem), permut(Rem, RP, M).
```

- $\backslash+( P )$  or `not(P)` is **negation as failure** operator. Successful only if the argument clause fails (cannot be proven).
- Set intersection:

```
inter([],_,[]).
inter([H|R],S, [H|Res]) :- member(H,S), inter(R,S,Res).
inter([H|R],S, Res) :- not(member(H,S)), inter(R,S,Res).
```

- Set union:

```
union([],S,S).
union([H|R],S, Res) :- member(H,S), union(R,S,Res).
union([H|R],S, [H|Res]) :- not(member(H,S)), union(R,S,Res).
```

# Cut

- `f(x) = if x > 10 then 5 else if x > 5 then 3 else 1`

```
f(X, Y) :- X > 10, Y = 5.
```

```
f(X, Y) :- X =< 10, X > 5, Y = 3.
```

```
f(X, Y) :- X =< 5, Y = 1.
```

- Each clause test for interval but only one clause can be true.
- **Cut** symbol, '!' prunes search tree and change behaviour.

```
f(X, Y) :- X > 10, !, Y = 5.
```

```
f(X, Y) :- X > 5, !, Y = 3.
```

```
f(X, Y) :- Y = 1.
```

- Cut is always successfull with side effect of deleting all backtracking points from head clause so far. Only current solution is kept.
- Rewrite set intersection with a **cut**:

```
inter([], _, []).
```

```
inter([H|R], S, [H|Res]) :- member(H, S), !, inter(R, S, Res).
```

```
inter([H|R], S, Res) :- inter(R, S, Res).
```

- $\neg(P)$ , not operator can be implemented by a cut.

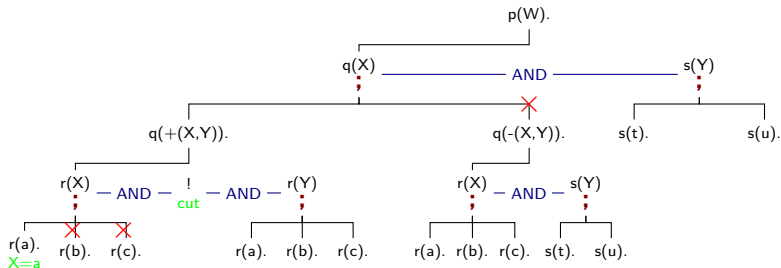
```
not(P) :- P , !, fail.
not(P).
```

- This is called **negation as failure** semantics, not **logical negation**. In **logical negation** you may expect `not(member(X,[a,b,c]))` to instantiate  $X$  to complement set of  $[a,b,c]$ . However it simply fails.
- When a **cut** does not change the program semantics, set of values returned, it is called a **green cut**.

$p(* (X,Y)) \text{ :- } q(X), s(Y).$        $r(a).$        $s(t).$

$q(+ (X,Y)) \text{ :- } r(X), !, r(Y).$        $r(b).$        $s(u).$

$q(- (X,Y)) \text{ :- } r(X), s(Y).$        $r(c).$



- When cut is hit, current solution is kept and all backtracking points from head clause to cut are pruned. Backtracking points introduced later still produces alternatives.
- $r(Y)$  produces 3 alternatives,  $s(Y)$  at top right produces 2. Prolog finds 6 solutions in total.
- Without a cut:  $3 \times 3 = 9$  from  $q(+ (X,Y))$ ,  $3 \times 2 = 6$  from  $q(- (X,Y))$  give 15 solutions for  $q(X)$ . 15 times 2 solutions for  $s(Y)$  gives 30 solutions.



The following program generates 90 alternatives. Putting **cut** in marked positions one at a time changes this behaviour.

```
p(+(X,Y,Z)) :- ○ q(X) ,○ r(Y),○ s(Z) ○. % 15*3*2 = 90

% 15 for q(X)
q(-(X,Y)) :-○ r(X),○ r(Y)○ . % 3 * 3 = 9
q(-(X,Y)) :-○ r(X),○ s(Y)○ . % 3 * 2 = 6

r(a). % 3 from r(X)
r(b).
r(c).

s(t). % 2 from s(X)
s(u).
```

Number of solutions per cut will be:

90, 6, 2, 1,  
54, 18, 6,  
90, 66, 60

# Example: Binary Search Tree

- we can represent a tree as a Prolog structure. Each node contains a key value pair in `p(k,v)`.

```
insert(e, K,V, tree(p(K,V),e,e)).
```

```
insert(tree(p(K,_),L,R), K, V, tree(p(K,V),L,R)) :- !.
```

```
insert(tree(p(H,HV),L,R), K, V, tree(p(H,HV),LRes,R)) :-
 K < H,!, insert(L, K, V, LRes).
```

```
insert(tree(p(H,HV),L,R), K, V, tree(p(H,HV),L,RRes)) :-
 insert(R, K, V, RRes).
```

```
insertlist(R,[],R).
```

```
insertlist(R,[K,V|L], RR) :- insert(R,K,V,RTemp),
 insertlist(RTemp,L,RR).
```

```
search(tree(p(K,V),_,_), K, V) :- ! .
```

```
search(tree(p(H,_),L,_), K, V) :- K < H, !, search(L,K,V).
```

```
search(tree(p(_,_),_,R), K, V) :- search(R,K,V).
```

```
?- insertlist(e,[4,veli,1,ali,6,hasan,2,ayse,5,fatma],R).
```

```
R = tree(p(4, veli), tree(p(1, ali), e, tree(p(2, ayse),e,e)),
 tree(p(6, hasan), tree(p(5, fatma),e,e),e))
```

# N-Queens

- Place  $N$  queens on a  $N$  by  $N$  board with no queen is threatening others.

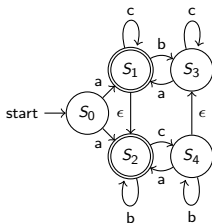
```
% give a list [N,N-1,...,0]
fill(N,[N|R]) :- M is N-1, M > 0, !, fill(M,R).
fill(L, [L]).

% get N queens, N columns and place them with state []
% queen id is assume to be the row
queen(N,R) :- fill(N,Queens), fill(1,N,Positions),
 place(Queens,Positions,[],R).

% get first queen, pick a position, check it is compatible with
% existing state, place rest with new state
place([],[],L,L).
place([Q|QRest],PList,In,RResult) :- select(P,PList,PRest),
 compatible(Q,P,In), place(QRest,PRest,[Q-P|In],RResult).

% compatibility, test not same column and diagonal.
compatible(_,_,[]).
compatible(Q,P,[QT-PT|R]) :- P \= PT, QDel is abs(Q-QT),
 PDel is abs(P-PT), QDel \= PDel, compatible(Q,P,R).
```

# Example: NDFA



- Defined by a 5-tuple  $(Q, \Sigma, \Delta, q_0, F)$ :  
 $Q$  set of states,  $\Sigma$  input symbols,  
 $\Delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$  set of transitions,  
 $q_0 \in Q$  start state,  $F \subseteq Q$  final states.
- In prolog, we can define all those relations:
  - 1 define all transitions as `trans(s0, a, s1)`.
  - 2 define all empty transitions as `empty(s1,s2)`.
  - 3 define all accepting states as `final(s1)`.
  - 4 define starting state as `start(s0)`.
- NDFA parser using backtracking power of Prolog is easy:

```

parse(State, []) :- final(State).
parse(State, [H|R]) :- trans(State, New, H), parse(State, R).
parse(State, Inp) :- empty(State, New), parse(State, Inp).
parse(Inp) :- start(S), parse(S, Inp).

```

# Example: Numbers game

- Given list of numbers, i.e.  $[1, 3, 5, 9, 30]$ , find arithmetic operations to calculate the given number, i.e.  $331 = (9-3+5)*30+1$
- `findit (numlist, number, expression)`
- Pick two numbers from list, pick an operator, compose an expression and put it back on the remaining list and try with new list. When the expression evaluates to the number. it is a success.

```
% if top element evaluates to number, it is the result
findit([A|_],V,A) :- V is A.
% pick two values, pick one operand, check if valid,
% compose an expression, test with new expression
findit(L,V,T) :- select(A,L,AR), select(B,AR,Rest),
 member(O,[+,-,/,*]), check(O,A,B), E =.. [O,A,B],
 findit([E|Rest],V,T).

check(+,_,_).
check(*,_,_).
check(-,A,B) :- A>B.
check(/,A,B) :- 0 is A mod B.
```

Implementation creates duplicate results because of symmetry in '+' and '\*'. In symmetric operators use combination, `taketwo` which will get `a`, `b` once, not `b,a`.

```
taketwo(A, B, [A|R] , Rem) :- select(B, R, Rem).
taketwo(A, B, [H|R], [H|Rem]) :- taketwo(A, B, R, Rem).

% if top element evaluates to number, it is the result
findit([A|_],V,A) :- V is A.
% asymmetric ops.
findit(L,V,T) :- select(A,L,AR), select(B,AR,Rest),
 member(O,[-,/,]), check(O,A,B), E =.. [O,A,B],
 findit([E|Rest],V,T).
% symmetric ops.
findit(L,V,T) :- taketwo(A,B,L,Rest),
 member(O,[+,*]), E =.. [O,A,B],
 findit([E|Rest],V,T).

check(-,A,B) :- A>B.
check(/,A,B) :- 0 is A mod B.
```

# Example: Symbolic Differentiation

- Given an expression containing the variable, find the derivative of the expression with respect to that variable.
- `diff(exp, var, diffexp).`
- `diff(3*x*x+2x+1, x, D)` results in `D = 6*x+2..`

```
diff(A, A, 1) :- !. % x -> 1
diff(A, _, 0) :- number(A). % c -> 0
diff(-A, X, -C) :- diff(A, X, C). % -f(x) = - f(x)
diff(A+B, C, D+E) :- diff(A, C, D), diff(B, C, E).
diff(A-B, C, D-E) :- diff(A, C, D), diff(B, C, E).
diff(A*B, C, A*D) :- number(A), diff(B, C, D), !.
diff(A*B, C, A*D+B*E) :- diff(A, C, E), diff(B, C, D).
diff(A/B, C, D) :- diff(A*B^(-1), C, D).
diff(A^B, C, B*A^B1*D) :- number(B), diff(A, C, D), B1 is B -
diff(log(A), B, C*A^ -1) :- diff(A, B, C).

?- diff(1/((x^2+1)*(x-1)), x, R).
R = 1* (-1* ((x^2+1)* (x-1)) ^ (-1-1)*
 ((x^2+1)* (1-0)+ (x-1)* (2*x^ (2-1)*1+0))).
```

A simplifier is needed.

```

constant(X) :- number(X),!.
constant(A):- A =.. [_ ,T1,T2] , constant(T1),constant(T2).
constant(A) :- A =.. [_ ,T1] , constant(T1).
s(X,Y) :- constant(X),!,Y is X.
s(A*1,Y) :- simp(A,Y),!.
s(1*A,Y) :- simp(A,Y),!.
s(-1*A,-Y) :- simp(A,Y),!.
s(A+0,Y) :- simp(A,Y),!.
s(0+A,Y) :- simp(A,Y),!.
s(A-0,Y) :- simp(A,Y),!.
s(0-A,-Y) :- simp(A,Y),!.
s(A^1,AE) :- simp(A,AE),!.
s(1^_,1) :- !.
s(_^0,1) :- !.
s(X,X).

simp(A*B,R) :- simp(A,AE),simp(B,BE),s(AE*BE,R),!.
simp(A+B,R) :- simp(A,AE),simp(B,BE),s(AE+BE,R),!.
simp(A-B,R) :- simp(A,AE),simp(B,BE),s(AE-BE,R),!.
simp(A^B,R) :- simp(A,AE),simp(B,BE),s(AE^BE,R),!.
simp(X,X).
derivative(X,R) :- diff(X,Z), simp(Z,R).

?- derivative(1/((x^2+1)*(x-1)), x, R).
R = - ((x^2+1)*(x-1))^-2* (x^2+1+ (x-1)* (2*x)).

```



# Programming Language Concepts

## Object Oriented Prog: Objects

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



# Outline

## 1 Object Oriented Programming

## 2 Constructors/Destructors

- Constructors
- Heap Objects
- Destructors
- Constructor Calls

- const Keyword
- Copy Constructor
- Pass by reference
- Move Constructor

## 3 Operator Overloading

- Friends

## 4 Implementation of Objects

# Object Oriented Programming

- Abstraction
- Encapsulation
- Hiding
- Inheritance

# Encapsulation/Scope

- Objects consist of:
  - attributes (member variables)
  - methods (member functions)

encapsulated in a package scope

- attributes: state of objects
- methods: behaviour of objects
- alternative terminology: messages  
call a method  $\equiv$  send message to an object
- A class is the family for similar objects.
- An object is an **instance** of a class.

| Person      |
|-------------|
| name        |
| surname     |
| no          |
| + getname() |
| + setno()   |

```

class Person {
 char name[40], surname[40];
 int no;
public:
 const char * getname() { return name;}
 void setno(int);
} obj ;

void Person::setno(int a) {
 no=a;
}

```

- C++ allows definitions inside the class or outside by **scope operator '::'**
- Environment is recursive collateral.
- `obj.getname()`; calls the method in the context of object `obj`.
- **this** keyword denotes pointer to current object in member functions. (`self()` in some other languages)

# Hiding

- Interface vs detail. Details are hidden, only interface members are exported outside.
- C++ uses `private:`, `protected:`, and `public:` labels to mark hiding.
- only members following a `public:` label are visible outside (the object for example). Member functions can access all members regardless of their labels.
- `obj.setno(4)` is legal, `obj.no` is not.
- Hiding depends on scope and it is lexical. In C++ pointer conversions can violate hiding.
- By convention all member variables should be private, some member functions can be private, only some of member functions are public.
- `protected` keyword is useful with inheritance.

# Abstraction

- An object is an abstraction over the programming entity defined by the model in the design.
- Model: customer, bank, registration, course, advisor, mail, chatroom,...
- Class should provide:
  - Transparent behaviour for the objects, access via interface functions.
  - Data integrity. Objects should be valid through their lifetimes.
- Data integrity at the beginning of lifetime provided by constructors (+destructors in C++)

# Constructors

- Special member functions called when lifetime of the object starts **just after storage of members are ready**
- Automatically called. No explicit calls.
- no return value, name is same with the class
- can be overloaded

```
class Person {
 char *name[40], *surname[40];
 int no;
public:
 Person(const char *n, const char *s) {
 strcpy(name,n) ; strcpy(surname,s) ; no=0;
 }
 Person() { name[0]=0; surname[0]=0; no=0;}
} obj ;
```



## ■ Constructors can be overloaded

| Definition              | Constructor                        |
|-------------------------|------------------------------------|
| Person a ;              | Person()                           |
| Person a("ali","veli"); | Person(const char *, const char *) |
| Number a=3;             | Number(int)                        |
| Number a(3);            | Number(int)                        |
| Number b=a;             | Number(Number &a)                  |
| Number a[2]={0,1}       | Number(int)                        |

- If no constructor implemented, empty constructor (do nothing) assumed
- If at least one constructor exists, variables should match at least one of them, no empty constructor assumed
- Constructors are called by the language when lifetime started:
  - 1 start of program for global objects
  - 2 entrance to function for local objects
  - 3 when heap objects are created (with `new`)

# Heap Objects

- `new` and `delete` operators instead of `malloc()` and `free()`.  
Why?
- `Person *p=new Person("ali","veli");`  
`delete p;`
- Array allocation/deallocation:  
`Person *p=new Person[100];`  
`delete [] p;`

# Destructors

- When storage (members) of an object allocated dynamically
- Lifetime is over : garbage
- We need calls to collect heap variables within the object
- Java solution: garbage collector does the job. We need nothing
- C++: **destructors**: member functions called when lifetime is over.
- A class only have one destructor with exact type and name: `~ClassName()`. Called:
  - 1 end of program for global objects
  - 2 return from function for local objects
  - 3 when heap objects are deallocated (with `delete`)

# Constructor Calls

- Calling a constructor as a member function is not allowed:

`Person p ; p.Person("john");` is a compiler error

- Constructor definitions can call each other with special syntax:

`Person():Person("john_doe") { ... } (C++11 only)`

- Explicit constructor calls create a temporary object:

`Person p; p = Person("john");` is equivalent to:

`Person p; { Person tmp("john"); p = tmp; }`

note that lifetime of `tmp` is over at the end of line

- In definitions, no intermediate object created:

`Person p = Person("john");`

- A constructor also works as a type conversion operator:

`p = (Person) "john";` is equivalent to `p = Person("john");`, creates a temporary object

- Also type conversion works implicitly. C++ calls constructors when type conversion is required:

`p = "john";` is equivalent to the call above.

# const Keyword

- C++ does strict type checking on constant restriction on `const`
- `const char *p` VS `char *const q`
  - 1 `p[3]='a';` ✗
  - 2 `q[3]='a';` ✓
  - 3 `p++;` ✓
  - 4 `q++;` ✗
- `const char * const p`
- `f(const ClassName &a)` makes the parameter object constant during the function scope
- `const ClassName &f()` makes the returned object reference constant in expression containing the function call
- What's beside assignment? `constant member functions`

# Constant Member Functions

```

■ void f(const Rational &a) { ...; a.clear(3);...;a.out();}
 void Rational::clear() { a=b=0;}

```

What is wrong above?

- ```
void Rational::out() const {...; a=b=0; }
```


 const keyword preceding the function body makes member function a constant function.
- Constant functions cannot update member variables, only can inspect them
`a=b=0 in out() is invalid above`
- If an object is constant, only constant member functions can be called.
`a.clear(3); is invalid above`
- Type system of C++ prohibits those → Syntax error.

Copy Constructor

- Destructor does not solve all problems with objects with heap members:
 - Semantics of assignment
 - Semantics of parameter passing
 - Semantics of return value
 - Initialization
- Default behaviour of C++ is **copy member values byte by byte**.
- Java assigns/passes by reference. No copying.
- C++ Solution: implement your own semantic by **Copy constructor** and overloading assignment operator.
- Assignment operator destroys an existing object and replaces with the data from new one, copy constructor copies data into an empty object.

Copy Constructor

- Type is: `ClassName(const ClassName &)`
- Called when:
 - Object passed by value: `void add(ClassName a) {...}`
 - Object initialized by object: `ClassName a,b=a;`
 - Object returned as a value `ClassName getVal() {...}`
- Last one is a little tricky.
- Default behaviour exists even if it other constructors exist.


```

class List {
    struct Node { int x; Node *next} *head;
public: List() { head=NULL;}
    List(const List &); // Copy constructor
    ~List();
};

void passbyvalue(List a) {
    ...
}

List returnasvalue(List &a) {
    List b = a;
    ...
    return a;
}

...
passbyvalue(c);
...
d=returnasvalue(c)
...

```

Diagram illustrating the use of the Copy Constructor:

- A green arrow points from the parameter `a` in the `passbyvalue` function to the `List` parameter in the `returnasvalue` function, labeled "Copy Constructor".
- A green arrow points from the variable `a` in the `returnasvalue` function to the variable `b` in the same function, labeled "Copy Constructor, explicit".
- A green arrow points from the variable `c` in the `passbyvalue` function to the variable `c` in the `returnasvalue` function, labeled "Copy Constructor".

- Pass by value of objects are constructed by the copy constructor
- Return an object as a value creates a temporary object in place of return and uses it:

```
d=returnasvalue(c); ≡ {List tmp=returnasvalue(c); d=tmp; }
```

- Temporary objects are created at such expressions and deallocated at the end of the line (at ';'), destructors are called regularly.
- Explicit call to a constructor also creates such a temporary object.

```
g=Person("ali","veli");
```

- C++ compilers avoid copy constructor calls when possible, called **copy elision**.

```
List f() { List t;...; return t;} ... ; d=f(); ...
```

If possible, compiler binds local object and returned temporary object same storage → No constructor call.

Pass by reference

- `Person &a` denotes a reference type and implements **pass by reference**. No new object is created for parameter.
- Can be used in declaration to create an alias:
`Person & q = p;` (subject to lifetime of `p`)
`const Person &t = Person("john");` (temporary read only object with a longer lifetime)
- Returning reference type is also possible:
`int & Person::get() { ... }`
`p.get()++;`
- `const` references follow the semantics of `const`.
- Temp. objects, **r-values** cannot be passed by non-const references:
`void print(Person &p) { ... }`
`print ((Person) "marry");` is an error.
- **r-values** can be passed by constant references:
`void print(const Person &p) { ... }`
`print ((Person) "marry");` is ok.

Efficiency of Parameter Passing

- Pass by reference is efficient but modification of actual parameter is not always desired
- `const` references avoid modification of actual parameters and may get `r-values` however parameter object cannot be modified.
- `Copy constructor` and pass by value solves modifiable parameter object and `r-value` problem.
- Copying is expensive and `r-values`, temporary objects have a very short lifetime.
- One solution is stealing the resources of an object instead of copying.
- An `r-value` has a short lifetime, so stealing its resources would not harm the integrity.
- C++11 defined `rvalue references`, `move constructor` and `assignment move operator` to solve this problem.

Move Constructor

- C++11 introduced the following:
 - 1 rvalue reference: `Person &&p`
 - 2 move constructor: `Person(Person &&p)`
 - 3 assignment move: `operator=(Person &&p)`
 - 4 `T &&std::move(T &)`, a reference converter.
- rvalue references are created for temporary objects and by making explicit `std::move()` calls.
- move constructor and all functions getting rvalue references (including assignment) gets resources directly from parameter object and leaves the parameter in a valid but nullified state.
- move constructor does not allocate and copy values. Just get references and pointers. Therefore they are more efficient.
- After call, parameter will not contain its previous values but a minimum valid state.

- **move constructor** is bound to return as a value cases (copy cons. called if it does not exist)
- Similarly passing temporary objects to assignment or rvalue parameters will be more efficient.
- Programmers may convert lvalue references to rvalue references explicitly by calling `std::move()` if the object does not need its content afterwards.
- Move constructor is subject to **copy elision**. Compiler avoids calling it if possible.
- Destructor is called for the moved rvalue when its lifetime is over. So it should be left in a state without dangling references, double free problems etc.

```

class LList {
    struct Node { int a; Node *next; } *head;
public:
    LList() { head = nullptr; }
    LList(const LList &l) { // copy constructor
        Node **prev = &head;
        for (Node *p = l.head; p != nullptr; p = p->next) {
            Node *q = new Node;
            q->a = p->a; (*prev) = q; prev = &(q->next);
        }
        *prev = nullptr;
    }
    LList(LList &&l) { // move constructor
        head = l.head; l.head = nullptr; // steal and nullify
    }
    ~LList() { /* a decent llist destructor here */
};

LList series(int n) { // assume no copy elision!
    LList t;
    for (int i = n; i >= 0 ; i--)
        t.push(i);
    return t;
}

...
series(10).out(); // picks MOVE (or none if copy elision)

```

```

LList LList::operator=(LList &&l) {    // move assignment
    for (Node *p = head; p != nullptr;) {
        Node *q = p;                // deallocate current list
        p = p->next; delete q;
    }
    head = l.head; l.head = nullptr; // steal and nullify
}

...
// first move constructor (if no elision) then assignment
// above is called. But it is the same linked list
// which is local to series() is assigned to c
c = series(10);

```


Operator Overloading

- Not an essential feature of object oriented programming but improves readability in some cases.
- Especially usefull in implementing selector abstraction, algebra based applications.
- Do not use it when the operator is not intuitive for the context (class and the operation).
- C++ allows overloading of existing operators with same arity and precedence and only if at least one class type involves in the operator
- Operator can be implemented as a member function (first parameter is the class) or as an external function (which has at least one parameter being a class)

- All C++ operators except '.', '?:', '::', '.*' and '->*'
- For unary operators:
 - ① `void ClassName::operator++();`
 - ② `void operator++(ClassName &a);`
- For binary operators:
 - ① `void ClassName::operator&&(int a);`
 - ② `void operator&&(int a,ClassName &b);`
- First versions are member functions, can exist private members. Only operand in unary case, LHS in binary case is the current object
- Second versions are outside of the definition. You need `friend` declaration if they need to access private members.

```

Rational & Rational::operator+(Rational &b) {...}
Rational & Rational::operator+(int n) {...}
Rational & Rational::operator<(Rational &b) {...}
Rational & Rational::operator!() {...}
Rational & Rational::operator++() {...}
Rational & Rational::operator++(int nouse) {...}
Rational & Rational::operator double() {...}

void Hash::operator=(Hash &a) {...}
double Hash::operator[](int a) {...}
double Hash::operator[](const char a[]) {...}
Hash & Hash::operator()(const char a[]) {...}

double Pointer::operator*() {...}
void * Pointer::new(size_t size) {...}
void * Pointer::delete(void *p, size_t size) {...}

Rational a,b,c; Hash h,j; Pointer p,*q;
a+b;           a+3;           if (a<b) ... ;           !a;
++a;           a++;           x=(double)a;

h=j;           x=h[3];       x=h["ali"];           i=h("a-b");

x=*p;           q=new Pointer;           delete q;

```

```

int operator+(int a, Rational &b) {...}
Rational & operator++(Rational &b) {...}
ostream & operator<<(ostream &os, Rational &a) {...}
istream & operator>>(istream &os, Rational &a) {...}

void operator+=(Hash &a, Rational b) {...}

Rational a,b; Hash h,j;

i=i+a;
++a;
cout << a; cout << 3 << a << b ;
cin >> b;
h+=a;

```

Friends

- When an external function or class needs to access private members, **friend** declaration is used to grant access.

```
class Rational {
    friend class Hash;
    friend ostream & operator<<(ostream &,const Rational &);
    int a,b;
public: ...
};
class Hash {
    ...
    void operator+=(Rational &a) { .. a.a; .. a.b; ...}
};
ostream & operator<<(ostream &os,const Rational &a) {
    os << a.a << "//" << a.b << '\n';
    return os;
}
```

Implementation of Objects

class Person

char name[40]	40*sizeof(char)
int id	sizeof(int)
char * getname()	sizeof(char *(*)())
void print()	sizeof(void (*)(*))

- What is size of object? Size of member variables + sizeof member function pointers?
- No! Each object does not have to store the function information.
Its storage is same with the structure without any member functions.
- Function membership handled by the type system:
Person::getname() instead of getname()

- How functions get object context (which object they refer to?)?
- `Person::getname(Person *this)` instead of no parameters
- `Person a; a.getname();`
converted to `Person::getname(&a);` internally
- All member references inside member function are converted to:
`char *getname() {.. id=5; ... ; strlen(name);...} →`
`char *Person::getname(Person *this) {`
`.. this->id=5; ... ; strlen(this->name);...}`

Programming Language Concepts

Object Oriented Prog: Polymorphism

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

1 Polymorphism

- Abstract Classes
- Interfaces
- Implementation of virtual members

2 Generic Abstraction

- Templates (C++)
- Generics (Java)

3 Class Members

Polymorphism

- Inheritance \rightarrow inclusion polymorphism
- Binding is still **static**, at compile time
- Pointers of derived classes are converted to superclass types

```
class A { int x;  
public: void get() { cout << 'A::get()';}  
};  
class B : public A { int y;  
public: void get() { cout << 'B::get()';}  
}  
...  
A a, *p;  
B b;  
p=&a; p->get();  
p=&b; p->get();
```

Late Binding

■ Delaying binding possible

```
class A { int x;  
public: virtual void get() { cout << 'A::get()';}  
};  
class B : public A { int y;  
public: void get() { cout << 'B::get()';}  
}  
...  
A a, *p;  
B b;  
p=&a; p->get();  
p=&b; p->get();
```

■ binding of **virtual** member functions done at run time.

Abstract Classes

- `void f() = 0 ;` makes the function an **abstract member**
- A class with at least one abstract member is an **abstract class**.
- Abstract classes cannot be instantiated
- A derived class remains abstract unless all abstract members are implemented somewhere in derivation chain.
- Java **interfaces**: abstract classes with only abstract member functions and constants.

- binding of `move()` is static but the `draw()`'s inside are still late.

```
class Shape { int x,y;
public: virtual void draw() = 0;
        void move(int a, b) {
            setbgcolor(); draw();
            x=a; y=b; setfgcolor(); draw();
        }
};

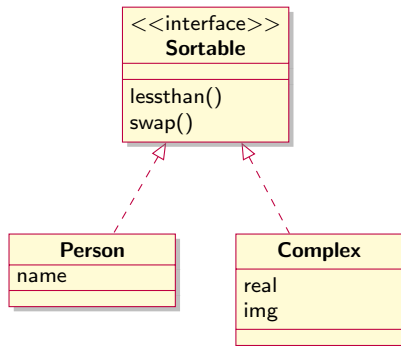
class Circle : public Shape { int r;
public: void draw() { /* draw circle here */ }
}

class Rectangle : public Shape { int w,h;
public: void draw() { /* draw rectangle here */ }
}

...
Circle a(...); Rectangle b(...);
a.move(2,4); b.move(3,4);
```

Interfaces

- Java does not have multiple inheritance but a class can **implement** multiple interfaces
- Functions working on interfaces provide polymorphism for the classes implementing them
- **Person** and **Complex** implements the interface **Sortable** so that `sort(...)` can work uniformly on both

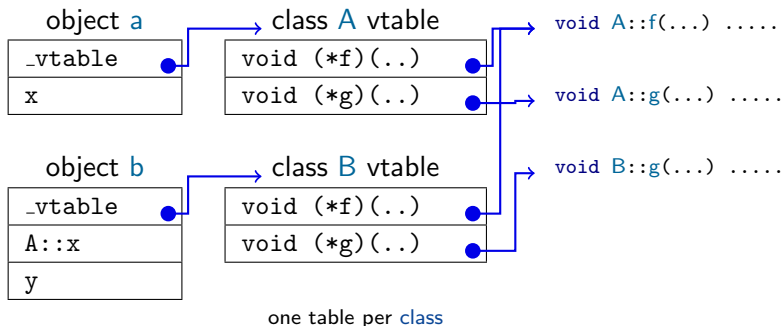


```
sort (Sortable a[],int n);
```

Implementation of virtual members

- For each class, a table for virtual member functions are kept globally (array of function pointers)
- Each **object** contains a pointer to its virtual function table
- Size of an object is : (size of member variables + pointer to virtual mem

```
class A { int x;  
public: virtual void f(...) {...}  
        virtual void g(...) {...}  
} a;  
class B : public A { int y;  
public: virtual void g(...) {...}  
} b;
```



one pointer

per object

Assuming p points to an object of A or B, $p \rightarrow g(\dots)$; call is mapped by the compiler as:

```
*((p->_vtable)[1])(...);
```

(assume 0 is the offset of f , 1 is the offset of g)

Generic Abstraction

- Abstraction over a declaration
- Polymorphism can be defined in terms of generic abstractions
- C++ templates
- Java generic classes

Templates (C++)

- Template metaprogramming approach:
All template definitions are expanded as they are **instantiated**
- Macro-like operation. Parameters can be an type or value.
- each **distinct** usage like `vector<Person> a` creates a new instance of the template class `vector`.
- All declaration body is expanded as an overloaded version.
- Functions can be declared with templates too. Each distinct typed call is a new instance, a new overload
- Very efficient but compiled code gets larger as different instances used
- Parametric polymorphism provied at compile time. Source code required.

Generics (Java)

- Restricts parameters to be classes. Primitive types and values does not work.
- Only one copy of the class and class functions exists.
- Type checking and verification done at compile time. Polymorphic code compiled in the binary.
- In Java: All object values are references, all member functions are virtual by default.
- Member functions of the parameter class are bound at run-time providing parametric polymorphism.

Class Members

- Members shared by objects of the same class. Only one copy per class.
- Assume you need a counter for each created object

```
int counter=0;

class A { int x;
public: A(int a) { x=a; counter++;}
      ~A() { counter--;}
      int getcount() { return counter;}
};
```

- What is wrong with this code?

- **static** keywords make a member a class member

```
class A { int x;
        static int counter;
public: A(int a) { x=a; counter++;}
        ~A() { counter--;}
        int getcount() { return counter;}
};
int A::counter=0; // this is required to define the storage
                 // it is scope of A
```

- Now the counter is safe. Arbitrary values cannot be assigned.
- Why do you need an object to call `getcount()`?

- Member functions can be class members too.

```
class A { int x;  
        static int counter;  
public: A(int a) { x=a; counter++;}  
        ~A() { counter--;}  
        static int getcount() { return counter;}  
};  
int A::counter=0;
```

- Class members can be accessed with scope operator:
`A::getcount();`
- No object required. What if `getcount()` tries to access an object? You don't have one!
- Class member functions can only access other class members.
- Objects can access class members.

Programming Language Concepts

Object Oriented Prog: Relations

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

1 Class Relations

2 Aggregate

3 Composition

- Integrity of Contained Objects

4 Generalization/Inheritance

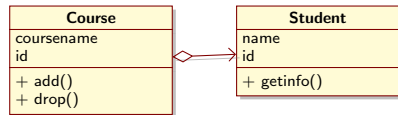
- Integrity of Superclass
- Member Hiding

5 Multiple Inheritance

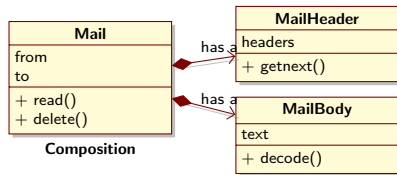
- Virtual base class

Class Relations

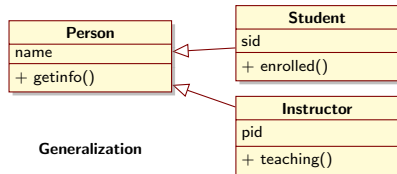
- In Object Oriented paradigm objects interact in order to solve a problem.
- Basic class relations:
 - Aggragate (“has a”)
 - Composition (“has a”)
 - Generalization (inheritance, “is a”)
- Other associations/relations exist.
- When two classes have such a relation, one **depends** on the other.



Aggregation



Composition



Generalization

Aggregate

- Class A can have 0 or more instance of class B
- Lifetime of class B objects are independent of class A
- Catalog relationship. In terms of [references](#).
- Members of class B are regular objects in scope of A
they are not in scope of A. So private members ... ?

```
class Course {  
    char name[40];  
    int no;  
    List students;  
public:  
    void register(Student &a) {  
        student.insert(&a);  
    };  
} ceng242 ;
```

```
void Student {  
    char name[30];  
    int no;  
public:  
    void add(Course &c) {  
        c.register(*this);  
    }  
};
```

Composition

- Class A can have 0 or more instance of class B
- Lifetimes of class B objects depend on the class A object
- Class B objects are destroyed when A is destroyed.
- Members of class B are regular objects in scope of A
they are not in scope of A as in aggregate.

```
class FrameBox {
    Shape frame;
    String text;
    double coordx, coordy;
public:
    Framebox(Frame &f,
              String &t) {
        ...}
    void draw() {
        frame.draw(); text.draw();
    }
} ceng242 ;
```

```
class Shape {
    enum Type {Circle, Rect} type;
    double sizex, sizey;
public:
    void draw();
};

class String {
    ...
};
```

Framebox

Shape frame	Type type	sizeof(int)
	double coordx	sizeof(double)
	double coordy	sizeof(double)
String text
double coordx	sizeof(double)	
double coordy	sizeof(double)	

- Container class vs. contained classes
- Composition nests storage of contained classes into container class.
- frame and text are regular object variables in member functions of Framebox
- Integrity of contained objects?

```

class Student {
    char name[40];
    int id;
public:
    Student() { name[0]=0; id=0;}
    void setnameid(const char *s,int i);
    ...
};

class StudentArr {
    Student *content;
public:
    StudentArr(int size) {
        content=new Student[size];
    }
    ~StudentArr() { delete [] content;}
    Student &operator[](int i) {
        return content[i];
    }
}
...
StudentArr a(10);
a[5].setnameid("onur",55717);

```

Integrity of Contained Objects

```
class A {
    int x;
public:
    A(int a) { x=a; }
};
class B {
    int y;
public:
    B(int a) { y=a; }
};
```

```
class C {
    int c;
    A a;
    B b;
public:
    C(int x,y,z):a(x),b(y) {
        c=z; /*can refer a,b */
    }
    ~C() { /*can refer a,b */ }
};
```

- When constructors called? Tip: Container class constructor may refer to the contained objects.
- When destructors called? Tip: Container class destructor may refer to the contained objects.

- Constructors of contained objects called just before the body of container constructor executed.
- Destructors of contained objects called just after the container destructor called.
- Container constructor can pass arguments to member object constructors.

```
ACons(int x):c(x),b(x+2),a(x+1) {...}
```

- The list of comma separated initializers between the column and opening brace is called **member initializer list**. It defines how members are initialized in constructor syntax.
- The order of member initializer list is irrelevant. Member object constructors are called in **declaration order**. For definition:

```
class ACons { int a, b, c; ...}
```

call order will be `a(x+1)`; `b(x+2)`; `c(x)`, then body of the constructor is executed.

- **friend** declaration can be used if the objects need to access others private member.

Generalization/Inheritance

- Class **Circle** is a **Shape** but has extra features.
- It has all members of **Shape** plus specific ones.
- **Circle** extends **Shape**
- **Shape** is super class of **Circle**
- **Shape** is more general, **Circle** has more information

```
class Shape {  
    double x,y;  
public:  
    Shape(double a, double b);  
    void draw();  
};  
  
class Circle: public Shape {  
    double radius;
```

```
public:  
    void draw();  
};  
  
class Square: public Shape {  
    double width;  
public:  
    void draw();  
};
```


Circle

Shape double x	sizeof(double)
double y	sizeof(double)
double radius	sizeof(double)

- There is an inherent Shape object in each Circle object.
- $\text{Env}(\text{Circle}) = \text{Env}(\text{Shape}) \cup \text{Members specific to Circle}$
- All members are inherited. They are in the scope of the subclass.
- How about their accessibility, protection?
- Two new thing: protected label, derivation label
- A subclass can access protected members of the upper classes.
- derivation label is a filter defining how members of superclass interpreted when used through subclass (object of subclass or further derivations from subclass)

```

class A {
private:    int a;
protected: int b;
public:    int c;
    void Amember() { ① }
} Aobj ;
class B: DLABEL A { // DLABEL=public/protected/private
    void Bmember() { ② };
} Bobj;
... Aobj.③ ;
... Bobj.④ ;
class C: public B {
    void Cmember() { ⑤ } };

```

	①	②	③		④			⑤		
				DLABEL	a	b	c	a	b	c
a	✓	×	×	private	×	×	×	×	×	×
b	✓	✓	×	protected	×	×	×	×	✓	✓
c	✓	✓	✓	public	×	×	✓	×	✓	✓

- DLABEL is only significant outside of the derived class
- protection is minimum of original label and DLABEL

- The inherent superclass object should have a valid value.
- Constructors/Destructors should be called

```
class A {  
    int x;  
public:  
    A(int a) { x=a;}  
    ~A() { ... }  
};  
class B : public A {  
    int y;  
public:  
    B(int a):A(a) { y=a;}  
    ~B() { ... }  
};
```

- Base class constructors are called just before the class constructor
Base class destructor is called just after the class destructor
- Similar to contained objects, **member initializer list** can contain base class initializers as well. The order of execution will be:
 - 1 base classes in order of appearance in declaration
 - 2 member objects in order of appearance.

Member Hiding

- members of the subclass hides member of the superclass with same name
- but superclass member still exists
- Scope operator can be used to access the member

```
class A {  
protected:  
    int x;  
public:  
    int get() {return x};  
} Aobj;  
class B : public A {  
    int x;  
public:  
    int get() {return x+A::x}  
} Bobj;  
...  
cout << Bobj.get() << Bobj.A::get() ;
```

Multiple Inheritance

- Can a class be derived from two superclasses?
- Land vehicle+Water vehicle → Hovercraft
- Student+Instructor → A lecturer still having PhD
- A class hierarchy for vehicle types, a class hierarchy for engines:
A boat with diesel engine, a car with electrical engine or hybrid engine
- Multiple inheritance is necessary in some rare cases. C++ provides it, Java avoids it and uses [Interfaces](#) for essential functionality similar to multiple inheritance.

```

class Shape {
    int x,y;
public:
    Shape(int a, int b) { x=a; y=b;}
    ~Shape() { ... }
};

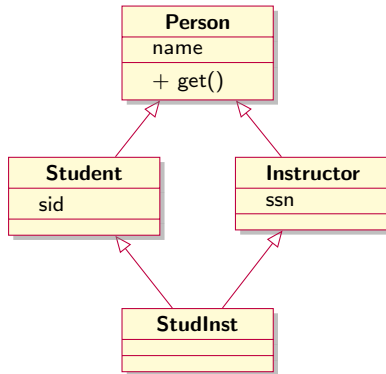
enum LineStyle {None, Solid, Dashed, Dotted, Double}
enum FillStyle {None, Full, Half, Pattern}
class ShapeAttr {
    LineStyle ls; double lw; FillStyle fill;
public:
    ShapeAttr(LineStyle a, double b, FillStyle c) {
        ls=a;lw=b;fill=c;}
    ~ShapeAttr() { ... }
};

class Circle: public Shape, public ShapeAttr {
    int radius;
public:
    Circle(int a, int b, int c, LineStyle d,
           double e, FillStyle f):Shape(a,b),ShapeAttr(d,e,f) {
        radius=c;
    }
}

```

Diamond Problem

- Multiple inheritance may cause same super class duplicated in the resulting class
- Causes ambiguity.
`StudInst` contains two `Person`'s `get()` call refers to which one?
What's the `name` ?
- Ambiguity can be solved by scope operator:
`Student::name` VS `Instructor::name`
- But a person with two names?
Do we need that redundancy?
NO!



Virtual base class

■ `virtual` keyword used

in inheritance gets only a single copy of base class in subclasses.

```
class Person {
    char name[40];
public: Person(char *s) {...}
};
class Student: virtual Person {
    int id;
public: Student(char *s, int i):Person(s) {...}
};
class Instructor: virtual Person {
    int ssn;
public: Instructor(char *s, int i):Person(s) {...}
};
class StudInst:public Student, public Instructor {
public: StudInst(char *s, int a, int b)
        :Person(s),Student(s,a),Instructor(s,b) {...}
};
```


- **virtual** keyword is for subclasses
- It is an overloaded keyword. We also have virtual member functions which is completely different.
- Multiple inheritance is not essential feature in OOP.
- There are ways to live without it. Assume two hierarchies with M and N classes. First is under **Vehicle**, second is **Engine**
- **Bridge pattern** Put a **Engine*** member in **Vehicle**
- **Nested classes** Create all $M \times N$ possibilities derived from **Vehicle**
- Such cases are rare and primary advantage of inheritance is **Polymorphism**

Programming Language Concepts

Syntax and Parsing

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

1 Describing Syntax

- Introduction
- Backus-Naur Form and CFGs
- Context Free Grammar
- Ambiguous Grammars
- Associativity
- An Assignment Grammar

2 Parsing

- if-then-else ambiguity
- Compilation
- Lexical Analysis
- Parsing
- Top-down Parsing
- Recursive Descent Parser
- LL Parsers
- Bottom-up Parsing

Introduction

- **Syntax**: the form and structure of a program.
- **Semantics**: meaning of a program
- Language definitions are used by:
 - Programmers
 - Implementors of the language processors
 - Language designers

Definitions

- A **sentence** is a string of characters over some alphabet
- A **language** is a set of sentences
- A **lexeme** is the lowest level syntactic unit of the language (i.e. `++`, `int`, `total`)
- A **token** is a category of lexemes (i.e. *identifier*)

Definitions

- **syntax recognition**: read input strings of the language and verify the input belonging to the language
- **syntax generation**: generate sentences of the language (i.e. from a given data structure)
- Compilers and interpreters recognize syntax and convert it into machine understandable form.

Backus-Naur Form and CFGs

- CFG's introduced by Noam Chomsky (mid 1950s)
- Programming languages are usually in **context free language** class
- BNF introduced by John Bakus and modified by Peter Naur for describing Algol language
- BNF is equivalent to CFGs. It is a **meta-language** that describes other languages
- Extended BNF improves readability of BNF

A Grammar Rule

$\langle \text{while_stmt} \rangle \rightarrow \text{while } (\langle \text{logic_expr} \rangle) \langle \text{stmt} \rangle$

- LHS is a non-terminal denoting an intermediate phrase
- LHS can be defined (rewritten) as the RHS sequence which can contain terminals (lexems and tokens) of the language and other non-terminals
- Non-terminals are denoted as strings enclosed in angle brackets.
- $::=$ may be used in BNF notation instead of the arrow
- $|$ is used to combine multiple rules with same LHS in a single rule

$$\begin{array}{lcl} \langle \text{lgc_cons} \rangle ::= \text{true} & \equiv & \langle \text{lgc_cons} \rangle ::= \text{true} \mid \text{false} \\ \langle \text{lgc_cons} \rangle ::= \text{false} & & \end{array}$$

Context Free Grammar

- A **grammar** G is defined as $G = (N, \Sigma, R, S)$:
 - N , finite set of non terminals
 - Σ , finite set of terminals
 - R is a set of grammar rules. A relation from N to $(N \cup \Sigma)^*$.
 - $S \in N$ the start symbol
- Application of a rule maps one sentential form into the other by replacing a non-terminal element in sentential form with its right handside sequence in the rule, $u \mapsto v$.
- Language of a grammar $L(G) = \{w \mid w \in \Sigma^*, S \xrightarrow{*} w\}$

- Recursive or list like structures can be represented using recursion

$$\langle \text{expr_list} \rangle \rightarrow \langle \text{expr} \rangle , \langle \text{expr_list} \rangle$$

$$\langle \text{btree} \rangle \rightarrow \langle \text{head} \rangle (\langle \text{btree} \rangle , \langle \text{btree} \rangle)$$

- A **derivation** starts with a starting non-terminal and rules are applied repeatedly to end with a sentence containing only terminal symbols.
- **leftmost derivation**: always leftmost non-terminal is chosen for replacement
- **rightmost derivation**: always rightmost non-terminal is chosen for replacement
- Same sentence can be derived using leftmost, rightmost, or other derivations.

Sample Grammar

$$\begin{aligned}\langle \text{stmt} \rangle &\rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle \\ \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \langle \text{id} \rangle \\ \langle \text{op} \rangle &\rightarrow + \mid * \\ \langle \text{id} \rangle &\rightarrow a \mid b \mid c\end{aligned}$$

- Leftmost derivation of $a = a * b$:

$$\begin{aligned}\langle \text{stmt} \rangle &\mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle \mapsto a = \langle \text{expr} \rangle \\ &\mapsto a = \langle \text{id} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mapsto a = a \langle \text{op} \rangle \langle \text{expr} \rangle \\ &\mapsto a = a * \langle \text{expr} \rangle \mapsto a = a * \langle \text{id} \rangle \mapsto a = a * b\end{aligned}$$

- Rightmost derivation of $a = a * b$:

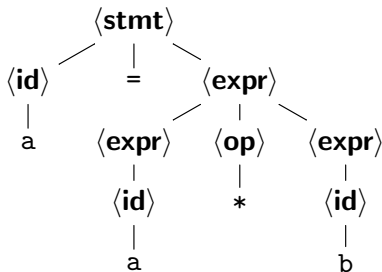
$$\begin{aligned}\langle \text{stmt} \rangle &\mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle \mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\ &\mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{id} \rangle \mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle \langle \text{op} \rangle b \\ &\mapsto \langle \text{id} \rangle = \langle \text{expr} \rangle * b \mapsto \langle \text{id} \rangle = \langle \text{id} \rangle * b \\ &\mapsto \langle \text{id} \rangle = a * b \mapsto a = a * b\end{aligned}$$

Parse Tree

- Steps of a derivation gives the structure of the sentence. This structure can be represented as a tree.
- All non-terminals used in derivation are **intermediate nodes**. Each grammar rule replaces the non-terminal node with its children. Root node is the start symbol.
- Terminal nodes are the **leaf nodes**.
- preorder traversal of leaf nodes gives the resulting sentence.
- leftmost and rightmost derivations can be retrieved by traversal of the tree.

Parse Tree Example

a = a * b

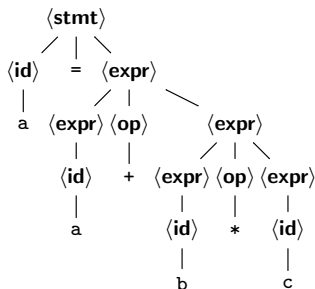


Parse Tree Generation

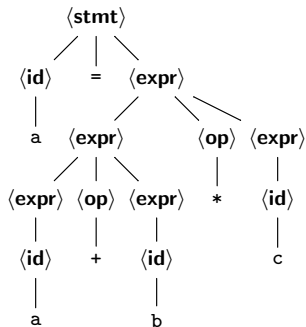
- A parse tree gives the structure of the program so semantics of the program is related to this structure.
- For example local scopes, evaluation order of expressions etc.
- During compilation, parse trees might be required for code generation, semantic analysis and optimization phases.
- After a parse tree generated, it can be traversed to do various tasks of compilation.
- The processing of parse tree takes too long, so creation of parse trees is usually avoided.
- Approaches like [syntax directed translation](#) combines parsing with code generation, semantic analysis etc..

Ambiguous Grammars

- Consider $a = a + b * c$ in our grammar:



VS



- Both can be derived by the grammar!

- A grammar is called **ambiguous** if same sentence can be derived by following different set of rules, thus resulting in a different parse tree
- If structure changes semantic meaning of the program, ambiguity is a serious problem.
- Even if not, which one is the result?
- i.e. Precedence of operators affects the value of the expression.
- Programming languages enforces precedence rules to resolve ambiguity.
- Solution:
 - 1 design grammar not to be ambiguous, or
 - 2 during parsing, choose rules to generate the correct parse tree

Precedence and Grammar

- Operators with different precedence levels should be treated differently
- Higher precedence operations should be deep in the parse tree → their rules should be applied later.
- Lower precedence operations should be closer to root → applied earlier in derivation.
- For each precedence level, define a non-terminal
- One rewritten on the other based on the precedence lower to higher

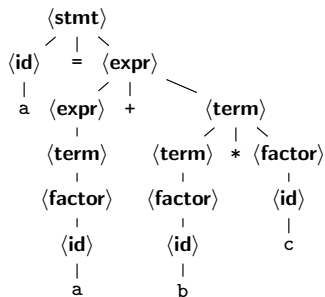
Rewritten Grammar

$$\langle \text{stmt} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$$

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$

$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$$

$$\langle \text{factor} \rangle \rightarrow \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$$

$$\langle \text{id} \rangle \rightarrow a \mid b \mid c$$


- `⟨term⟩` and `⟨expr⟩` has different precedence.
- Once inside of `⟨term⟩`, there is no way to derive `+`
- Only one parse possible

Associativity

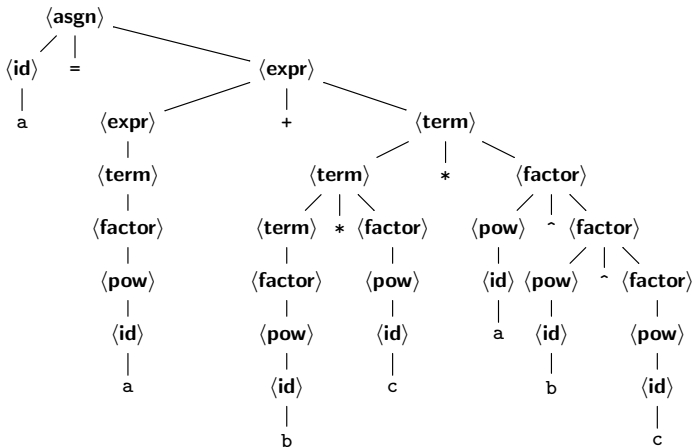
- Associativity of operators is another issue
 $a - b - c \equiv (a - b) - c \quad \text{or} \quad a - (b - c)$
- Recursion of grammar defines how tree is constructed for operators in the same level.
- If left recursive, later operators in the sentence will be closer to root, if right recursive earlier operators will be closer to root
- **left recursion** implies left associativity, **right recursion** implies right associativity.
- Consider $a + b + c$ in these grammars:

$$\begin{array}{l} \langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{id} \rangle \mid \langle \text{id} \rangle \\ \langle \text{id} \rangle \rightarrow a \mid b \mid c \end{array} \quad \text{VS} \quad \begin{array}{l} \langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle \mid \langle \text{id} \rangle \\ \langle \text{id} \rangle \rightarrow a \mid b \mid c \end{array}$$

Sample Grammar

$$\begin{aligned}
 \langle \text{asgn} \rangle &\rightarrow \langle \text{id} \rangle = \langle \text{asgn} \rangle \mid \langle \text{id} \rangle = \langle \text{expr} \rangle \\
 \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle \\
 \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\
 \langle \text{factor} \rangle &\rightarrow \langle \text{pow} \rangle ^ \langle \text{factor} \rangle \mid \langle \text{pow} \rangle \\
 \langle \text{pow} \rangle &\rightarrow \langle \text{id} \rangle \mid (\langle \text{expr} \rangle) \\
 \langle \text{id} \rangle &\rightarrow a \mid b \mid c
 \end{aligned}$$

- $\langle \text{asgn} \rangle$ is right recursive like right associative C assignments.
- $\langle \text{expr} \rangle$ and $\langle \text{term} \rangle$ are left recursive, $*$ and $+$ left associative
- $\langle \text{factor} \rangle$ is right recursive for power operation $^$ to be right associative.
- precedence order is $(...) \prec ^ \prec * \prec + \prec =$

$$a = a + b * c * a \wedge b \wedge c$$


if-then-else ambiguity

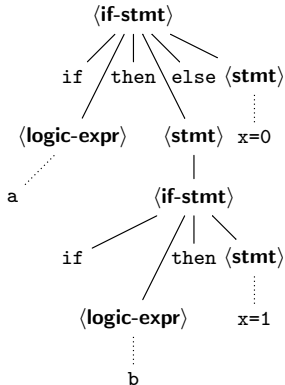
- Following grammar is ambiguous:

$$\langle \text{stmt} \rangle \rightarrow \langle \text{if-stmt} \rangle$$

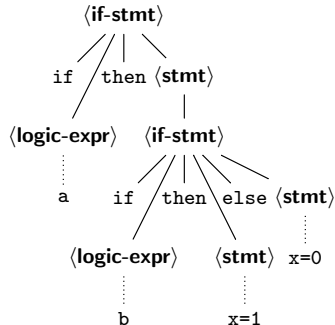
$$\langle \text{if-stmt} \rangle \rightarrow \text{if } \langle \text{logic-expr} \rangle \text{ then } \langle \text{stmt} \rangle \mid$$

$$\text{if } \langle \text{logic-expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$$

- Consider if a then if b then x=1 else x=0:



VS



Solution

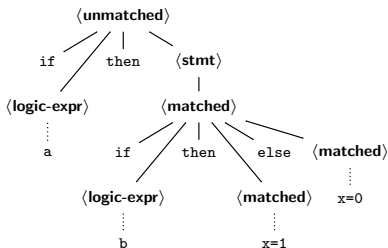
- Distinguish categories of statements. if's matched with else and unmatched:

$$\langle \text{stmt} \rangle \rightarrow \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle$$

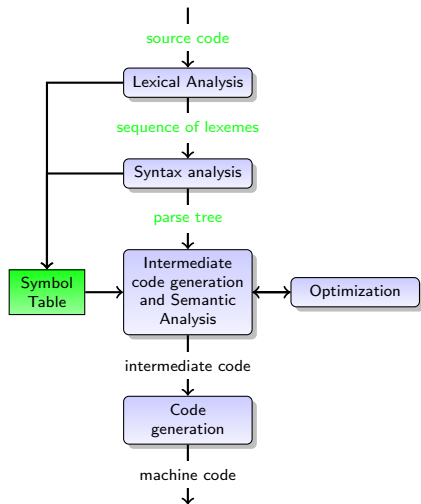
$$\langle \text{matched} \rangle \rightarrow \text{if } \langle \text{logic-expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle \mid \langle \text{other-stmt} \rangle$$

$$\langle \text{unmatched} \rangle \rightarrow \text{if } \langle \text{logic-expr} \rangle \text{ then } \langle \text{stmt} \rangle \mid \text{if } \langle \text{logic-expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{unmatched} \rangle$$

- if a then if b then x=1 else x=0:



Compilation



```

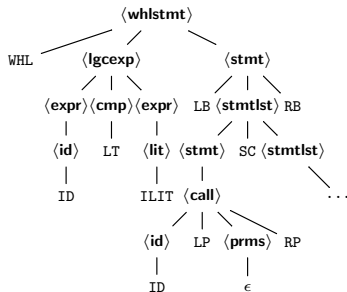
while (counter < 12341) {
    f() ;
    counter += 12;
}

```

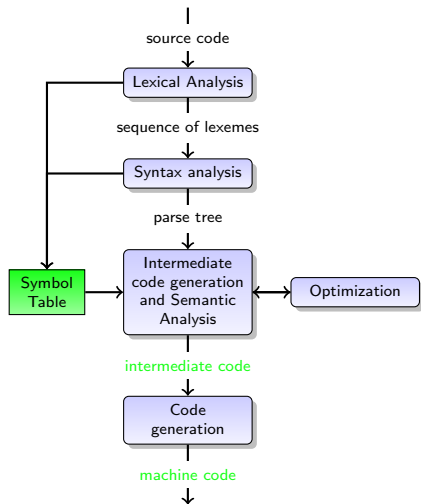
```

WHL LP ID LT ILIT RP LB
    ID LP RP SC
    ID PLEQ ILIT SC
RB

```



Compilation



```

.L3:      movl    $0, %eax
          call    f
          addl    $12, -4(%rbp)

.L2:      cmpl    $12340, -4(%rbp)
          jle     .L3
          leave   ret

001f 0001 0006 0000
0034 0021 0000 0000
0004 0000 001b 0009
0000 0000 02f4 0008
0008 0001 0004 0008
0025 0001 0003 0000
0058 0000 0000 0000
0004 0000 002b 0008
  
```

Lexical Analysis

- **input**: sequence of characters, source code.
- **output**: sequence of lexemes
- Worst case complexity of parsing is $\mathcal{O}(n^3)$. Depending on algorithm type, recursion type and number of grammar rules, this might change. n is the length of the string.
- Regular language processing complexity is $\mathcal{O}(n)$. Grammars can be defined in terms of lexemes.
- # of chars vs # of lexemes?
- Lexical analysis convert character sequences into lexemes.
Identifiers registered on symbol table

Parsing

- **input**: sequence of lexemes (output of lexical analysis) or characters.
- **output**: parse tree, intermediate code, translated code, or sometimes only if document is valid or not.
- Two main classes of parser:
 - Top down parsing
 - Bottom up parsing

Top-down Parsing

- Start from the starting non-terminal, apply grammar rules to reach the input sentence

$$\begin{aligned}
 \langle assign \rangle &\mapsto a = \langle expr \rangle \mapsto a = \langle expr \rangle + \langle term \rangle \mapsto \\
 &a = \langle term \rangle + \langle term \rangle \mapsto a = \langle fact \rangle + \langle term \rangle \mapsto \\
 &a = a + \langle term \rangle \mapsto a = a + \langle term \rangle * \langle fact \rangle \mapsto \\
 &a = a + \langle fact \rangle * \langle fact \rangle \mapsto a = a + b * \langle fact \rangle \mapsto \\
 &a = a + b * a
 \end{aligned}$$

- Simplest form gives leftmost derivation of a grammar processing input from left to right.
- Left recursion in grammar is a problem. Elimination of left recursion needed.
- **Deterministic parsing:** Look at input symbols to choose next rule to apply.
- **recursive descent parsers, LL family parsers** are top-down parsers

Recursive Descent Parser

```

typedef enum {ident, number, lparen, rparen, times,
             slash, plus, minus} Symbol;
int accept(Symbol s) { if (sym == s) { next(); return 1; }
                      return 0;
}
void factor(void) {
    if (accept(ident)) ;
    else if (accept(number)) ;
    else if (accept(lparen)) { expression(); expect(rparen);}
    else { error("factor: syntax error at ", currsym); next(); }
}
void term(void) {
    factor();
    while (accept(times) || accept(slash))
        factor();
}
void expression(void) {
    term();
    while (accept(plus) || accept(minus))
        term();
}

```

- Each non-terminal realized as a parsing function
- Parsing functions calls the right handside functions in sequence
- Rule choices are based on the current input symbol. accept checks a terminal and consumes if matches.
- Cannot handle direct or indirect left recursion. A function has to call itself before anything else.
- Hand coded, not flexible.

LL Parsers

- First L is 'left to right input processing', second is 'leftmost derivation'
- Checks next N input symbols to decide on which rule to apply: $LL(N)$ parsing.
- For example $LL(1)$ checks the next input symbol only.
- $LL(N)$ parsing table: A table for $V \times \Sigma^N \mapsto R$
- for expanding a nonterminal $NT \in V$, looking at this table and the next N input symbols, $LL(N)$ parser chooses the grammar rule $r \in R$ to apply in the next step.

- Grammar and lookup table for a LL(1) parser:

1 $S \rightarrow E$

2 $S \rightarrow -E$

3 $E \rightarrow N + E$

4 $E \rightarrow (E)$

5 $N \rightarrow a$

6 $N \rightarrow b$

	a	b	-	(
S	1	1	2	1
E	3	3		4
N	5	6		

- What if we add $E \rightarrow N$ to grammar?
- You need an LL(2) grammar. What if N is recursive?
see LL(*) parser

Bottom-up Parsing

- Start from input sentence and merge parts of sentential form matching RHS of a rule into LHS at each step. Try to reach the starting non-terminal. reach the input sentence

$$\begin{aligned}
 a = a + b * a &\mapsto a = \langle fact \rangle + b * a \mapsto a = \langle term \rangle + b * a \mapsto \\
 a = \langle expr \rangle + b * a &\mapsto a = \langle expr \rangle + \langle fact \rangle * a \mapsto \\
 a = \langle expr \rangle + \langle term \rangle * a &\mapsto a = \langle expr \rangle + \langle term \rangle * \langle fact \rangle \mapsto \\
 a = \langle expr \rangle + \langle term \rangle &\mapsto a = \langle expr \rangle \mapsto \langle assign \rangle
 \end{aligned}$$

- Simplest form gives rightmost derivation of a grammar (in reverse) processing input from left to right.
- Shift-reduce parsers are bottom-up:
 - **shift**: take a symbol from input and push to stack.
 - **reduce**: match and pop a RHS from stack and reduce into LHS.

Shift-Reduce Parser in Prolog

```

% Grammar is E-> E-T/E+T/T  T -> a/b
rule(e,[e,-,t]).
rule(e,[e,+,t]).
rule(e,[t]).
rule(t,[a]).
rule(t,[b]).

parse([], [S]) :- S = e .  % starting symbol alone in the stack
% reduce: find RHS of a rule on stack, reduce it to LHS
parse(Input, Stack) :- match(LHS, Stack, Remainder),
                        parse(Input, [LHS|Remainder]).

% shift: nonterminals are removed from input added on stack
parse([H|Input], Stack) :- member(X, [a,b,-,+]),
                           parse(Input, [H|Stack]).

% check if RSH of a rule is a prefix of Stack (reversed).
match(LHS, List, L) :- rule(LHS, RHS),  reverse(RHS, NRHS),
                      prefix(NRHS, List, L).

```

- Shift reduce parser tries all non-deterministic shift combinations to get all parses.
- For deterministic parsing states based on input lookahead or precedence required
- Deterministic bottom up parsers: LALR, SLR(1).