Lectures 4-5

Imperative languages

Iztok Savnik, FAMNIT

March, 2021.

Literature

- Textbooks:
  – John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 5 and 8)
  – Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 6 and 8)
- Many examples are from:
  – Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, English translation, O’REILLY, 2000

Imperative languages

- Functional programming
  - Program is a function
  - Outcome is done by the evaluation of the function
  - Does not matter how the result is produced
- Imperative programming
  - Program is sequence of instructions (states of the machine)
  - Outcome is build in the execution of instructions
  - Instruction changes the contents of main memory

Imperative programming

- Early imperative programming languages
  - Fortran, 1954; machine language
  - BASIC, 1960; Beginners' All-purpose Symbolic Instruction Code
  - Pascal, 1970; Algorithms + Data Structures = Programs
  - C, 1972; Constructs map efficiently to typical machine instructions
  - January 2021, C was ranked first in the TIOBE index

Example

Let us compute the greatest common divisor of two integers

OCaml

```
let rec gcd x y =
    if y = 0 then x
    else gcd y (x mod y);;
```

C++

```
int gcd(int x, int y) {
   x = abs(x); y=abs(y);
   while (y != 0) {
        int t = x % y;
        x = y;
        y = t;
    }
    return x;
}
```

Functional: values, recursion

Imperative: variables, loop, sequence

Imperative programming

- Machine has read-write memory for storing variables
- (Turing) machine consists of the states and instructions, which define computations and transitions between states
- Execution means the changes of states of the machine rather than the evaluation (reduction)
- The (way of) transitions are shaped (controlled) by the values of variables
- Execution of instruction can change the values of variables - side-effects

Structured control

- Structured and unstructured control flow
  - Unstructured: GOTO statements
  - Structured: syntactical constructs direct computation
- Structured programming, 1970
  - »Revolution« in software engineering
  - Top-down design (i.e., progressive refinement)
  - Modularization of code
  - Structured types (records, sets, pointers multidimensional arrays)
  - Descriptive variable and constant names
  - Extensive commenting conventions

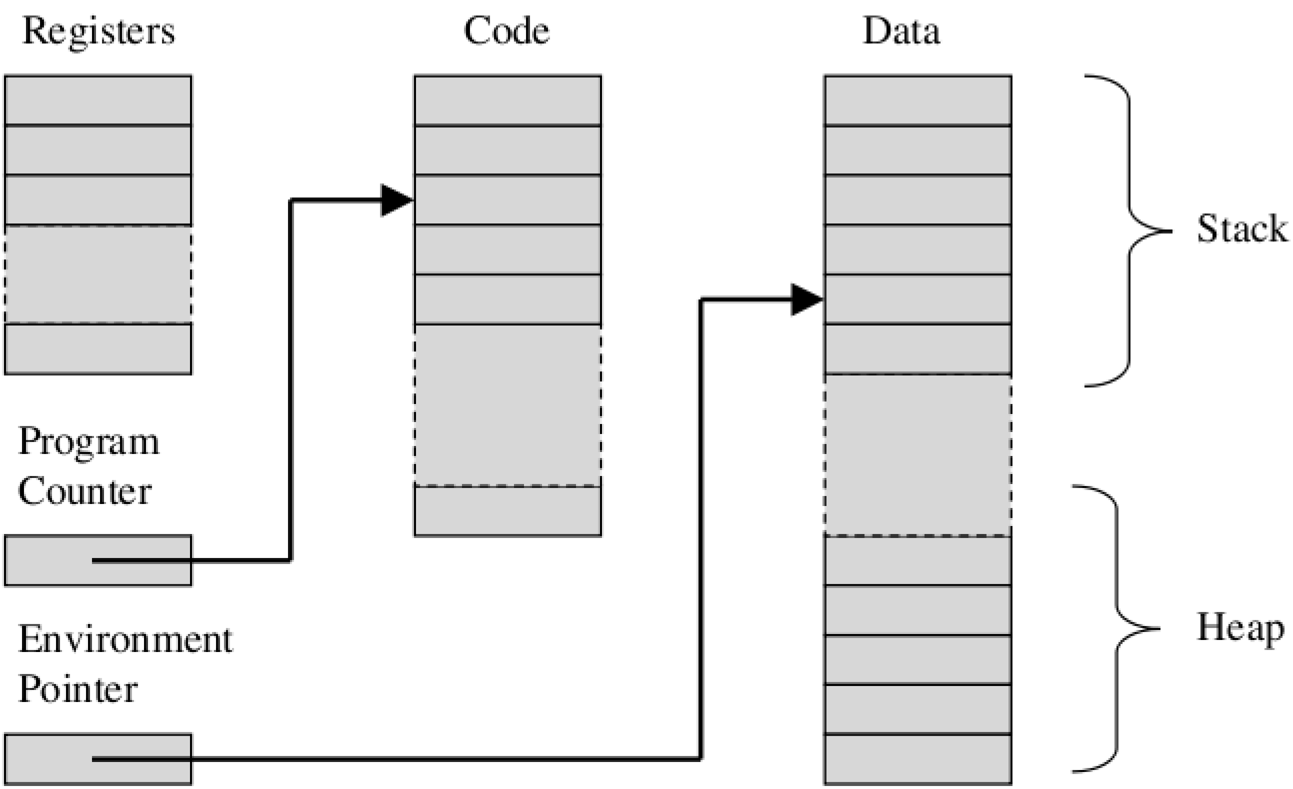
Structured control

- Most structured programming ideas were implemented in imperative languages
  - Pascal, C, Modula, Ada, Oberon, Java, ...
- Most of modern algorithms can be elegantly expressed in imperative languages

Concepts of imperative languages

- Read-write memory, variables
- Instructions and sequences of instructions
- Blocks
- Conditional statements
- Loops - conditional loops, iterations through ranges or through containers
- Procedures and functions

Program memory



Variables

- Memory of a program is organised into memory cells
- Any cell can be accessed via its address; an integer, usually in range 0 – ($2^{62} - 1$)
- Variable is a symbolic name for a memory space of given size
  - Variable can be accessed by using the name instead of the address of the memory cell
- Every program has symbol table, which stores the information about variables
  - Every record consists of:
    - Name (identifier), the address of the beginning of memory space, the size of allocated memory
  - Table changes during the execution of the program.

Operations with variables

- Program must allocate the memory space before the variable is used
  - The allocation can be either static or dynamic
- Program reads the contents of the memory in the moment we refer to the identifier (name)
- The contents of the memory referred by a variable is changed by assignment
- The variable is freed either on the end of execution or on demand
- Possible problems:
  - Read/write to unallocated memory, concurrent write, memory leak
  - We will study these problems in lecture on memory management

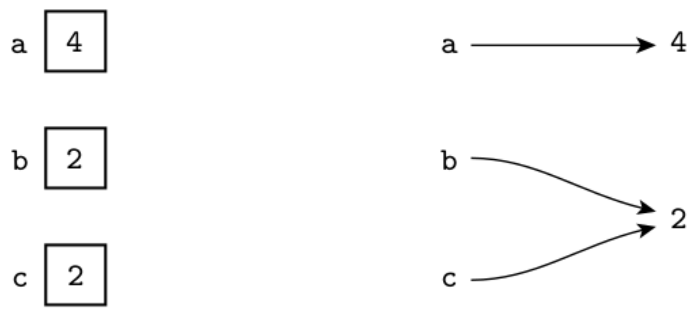
Models of variables

```
d = a;
a = b + c;
```

- Two models of variabes
  - Value model and reference model
- Value model of variables
  - Variable is a named container for a value
  - Location and value (see variable a)
    - l-value = left-hand side of assignment statements
    - r-value = expressions that denote values
    - both l-values and r-values can be complicated expressions
  - An expression can be either an l-value or an r-value, depending on the context
  - C, Pascal, Ada, etc.

Models of variables

- Reference model of variables



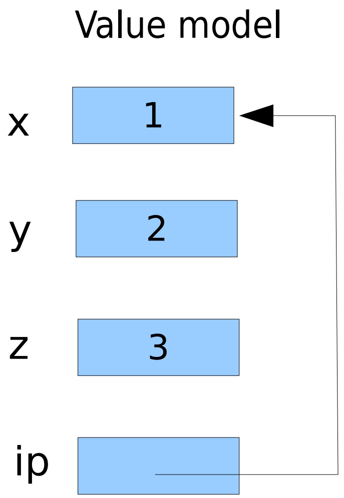

```
b := 2;
c := b;
a := b + c;
```

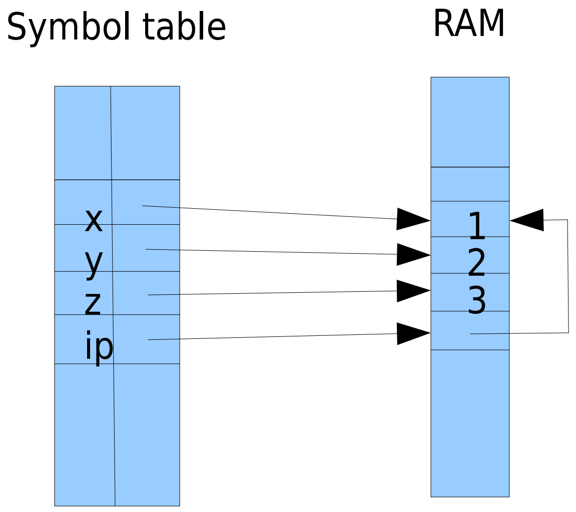
  – A variable is a named reference to a value
    - every variable is an l-value
    - variable in a context of an r-value must be dereferenced
    - dereference is automatic in most PL but not in ML
  – Reference model is not more expensive
    - Use multiple copies of immutable objects
  – Algol68, Clu, Lisp/Scheme, ML, Haskell, and Smalltalk

Variables in C

- Two important operators
  - Operator »&«: returns address of variable
  - Operator »*«: returns value of variable (from an address)

```
int x = 1, y = 2, z = 3;
int *ip;
ip = &x;
y = *ip;
*ip = 0;
*ip = *ip + 10;
y = *ip + 1;
z = *ip * 10;
```

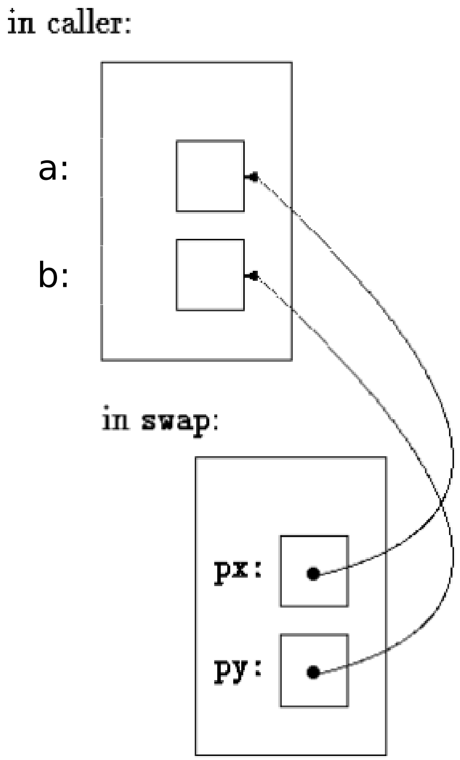




Two important operators

– Operator »&«: returns address of variable
– Operator »*«: returns value of a variable

```
swap(&a, &b);
...
void swap(int *px, int *py) {
   int temp;
   /* interchange *px and *py */
   temp = *px;
   *px = *py;
   *py = temp;
}
```



Variables in OCaml

```
type 'a ref = {mutable contents: 'a}
```

- Variables are implemented by using a reference type
- OCaml has weaker, but safer model of a variable
  - Reference is initialised on creation by the referenced value
  - Memory space is automatically allocated using the type of referenced value
  - Assignment is a special function with resulting type unit
  - Reading has the type of the referenced variable
- Drawbacks of the model
  - Functions cannot be referenced
  - We do not have full access to the program's memory – no pointers, no pointer arithmetics

Example

```
# let x = ref 2 ;;                    (* declaration and allocation
val x : int ref = {contents=2}
# !x;;                                (* reading, notice operator '!'
- : int = 2
# x ;;                                (* reading of the reference
- : int ref = {contents=2}
# x := 5;;                            (* assignment
- : unit = ()
# !x;;
- : int = 5
# x := !x * !x;;                      (* reading, operation, and assignment
- : unit = ()
# !x;;
- : int = 25
```

Sequential control

Sequential control is one of the basic principles of imperative programming languages

1. Sequences
2. Blocks
3. Condition statements
4. Loops

Sequence

- Sequence is foundamental abstraction used to describe algorithms
  - Instruction cycle of Von Neumann model

```
LOOP: execute *PC;     // execute instruction referenced by PC
      PC++;            // increment programm counter (PC) by 1
      goto LOOP;       // loop
```

- Sequence of instructions change the state of variables (memory)

```
int t = x % y;
x = y;
y = t;
```

Sequences in OCaml

```
let t = ref 0
let x = ref 42
let y = ref 28 in
    t:= x mod y; x:=!y; y:=!x ;;
```

- Syntax of OCaml sequences

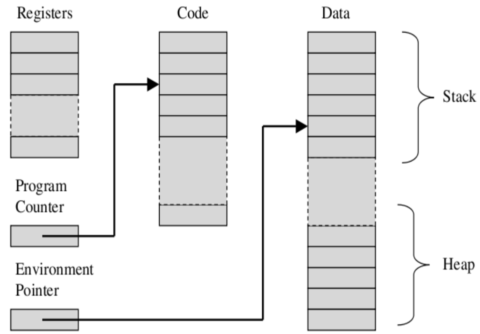
```
<expr1>; <expr2>; <expr3>          (* list of expressions *)
```

- Every expression in a OCaml sequence must be of type `unit`.

```
# print_int 1; 2; 3;;
Warning 10: this expression should have type unit.
1- : int = 3
```

```
# print_int 2; ignore 4; 6;;
2- : int = 6
```

Blocks




```
{int x = 2;
    { int y = 3;
      x = y+2;
    }
}
```

outer block

inner block

- Imperative languages are typicaly block-structured languages
- Block is a sequence of statements enclosed by some structural language form
  - Begin-end blocks, loop blocks, function body, etc.
- Each block is represented using activation record
  - Includes parameters and local variables
  - Includes memory location for return value
  - Includes control pointers to be detailed in next lectures
  - Control pointers are used to control computation
- Activation records are allocated on program stack

Conditional statements

JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry

- Machine languages use instructions for conditional jumps
  - Initial imperative approach
- OCaml syntax

```
if <cond_expr> then <expr_true> else <expr_false> ;;
```

- Conditional statements is concept shared between imperative and functional languages
  - Both branches must agree on type in Ocaml
  - Conditional statement in Ocaml has value

```
# (if 1 = 0 then 1 else 2) + 10;;
- : int = 12
```

Loops – while-do

- Repeat a block of commands while (or until) a condition is satisfied
  - Loop body changes the state of program
- Statement `while` in OCaml

```
while <cond_expr> do
   <sequence>
done
```

```
# let x = ref 42;;
val x : int ref = {contents = 42}
# let y = ref 28;;
val y : int ref = {contents = 28}
# let t = ref 0 in while !y != 0 do
   t := !x mod !y; x := !y; y := !t
   done;;
- : unit = ()
# !x;;
- : int = 14
```

Loops – `for` statement

- Statement `for` is classical construct of imperative prog. language
- Statement `for` in OCaml

```
for <sym> = <exp1> to <exp2> do
    <exp3>
done
for <sym> = <exp1> downto <exp2> do
    <exp3>
done
```

```
# for i=1 to 10 do
        print_int i;
        print_string " "
    done;
    print_newline ();;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do
        print_int i;
        print_string " "
    done;
    print_newline () ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

Loops – do-while statement

- Loop condition is at the end of loop block do-while syntax
- Not included in Ocaml!

```
do <sequence>
while <cond_expr>
```

- Statement **repeat-until**
- From Kernighan & Ritchie: The programming language C

```
/* itoa: convert n to characters in s */
void itoa(int n, char s[]) {
   int i, sign;
   if ((sign = n) < 0) /* record sign */
      n = -n;             /* make n positive */
   i = 0;
   do {     /* generate digits in reverse order */
       s[i++] = n % 10 + '0';     /* get next digit */
   } while ((n /= 10) > 0);       /* delete it */
   if (sign < 0)
       s[i++] = '-';
   s[i] = '\0';
   reverse(s);
}
```

Loop control

- Loop control in C programming language
  - Jumping out of loop – break
  - Jumping to loop condition – continue
  - Not included in OCaml

```
for (i = 0; i < n; i++)
    if (a[i] < 0)
        /* skip negative elements */
       continue;
    ...
        /* do positive elements */
```

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[]) {
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

- From Kernighan & Ritchie: The programming language C

Procedures and functions

- Abstraction is a process by which the programmer can associate a symbol or a pattern with a programming language construct
  - Control and data abstractions
- Subroutines are the principal mechanism for control abstraction
  - Part of program with well defined input and output is abstracted as subrutine, procedure, function
  - Subrutine performs operation on behalf of caller
  - Caller passes arguments to subrutine by using parameters
  - Subrutine that returns values is function
  - Subrutine that does not is called procedure

Procedures and functions

- Most subroutines are parameterized
- Procedure was first abstraction in Algol-family of programming languages
  - Formal and actual parameters of procedure

```
procedure Proc(First : Integer; Second: Character);
Proc(24,'h');
```

  - Actual parameters are mapped to formal parameters
- The most common parameter-passing modes
  - Some languages define a single set of rules that apply to all parameters (C, Java, Fortran, ML, and Lisp)
  - Others have more modes of parameter passing (Pascal, C++, Ada, ...)

Parameter passing

- Input and output of procedure is realized by means of parameter passing
  - Passing values
    - C (only cbv), Java, Ocaml, C++, Pascal, ...
  - Passing references
    - Pascal, C++, Fortran, ...

```
Procedure Square(Index : Integer;
                 Var Result : Integer);
 Begin
  Result := Index * Index;
 End
```

- Other parameter passing issues
  - Passing structured things
    - arrays, structures, objects
  - Missing and default parameters
  - Named parameters
  - Variable-length argument lists

Passing values

- The most commonly used method
  - Values of actual parameters are copied to formal parameters
  - Java uses only this method (arrays and structures are identified by references)
- Parameter is seen as local variable of procedure
  - It is initialized by the value of actual parameter

Java

```
int plus(int a, int b) {
  a += b;
  return a;
}
int f() {
  int x = 3; int y = 4;
  return plus(x, y);
}
```

Ocaml

```
# let plus (a,b) = let a = a + b in a;;
val plus : int * int -> int = <fun>
# let f =
    let x = 3
    and y = 4
    in plus (x,y);;
val f : int = 7
```

Passing references

- Reference to variable is passed to procedure
  - Code of procedure is changing passed variable
  - All changes are retained after the call
  - Passed variable and formal parameter are aliases
- Older method used in Fortran, Pascal, ...
- Best method for larger structures

```
Procedure plus(a: integer, var b: integer) {
Begin
   b = a+b;
End;
```

```
Procedure  p() {
Var x: integer;
Begin
   x = 3;
   plus(4, x);
End;
```

Pascal

C

```
void plus(int a, int *b) {
   *b += a;
}
void p() {
   int x = 3;
   plus(4, &x);
}
```

Variations on value and reference parameters

- C++ references
  - In C++ references are made explicit
  - C++ implements call-by-reference

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

- Call-by-sharing
  - Barbara Liskov, CLU (also Smalltalk)
  - Objects (identifiers) are references
  - No need to pass reference (to references)
  - Just pass reference

Variations on value and reference parameters

- Call-by-value/Result
  - Actual params are copied to formal params initially
  - Result is copied back to actual parameter before exit
- Read-only parameters
  - Modula-3 provided read-only params
  - Parameter values can not be changed
  - Read-only params are available also in C (const)
- Parameters in Ada
  - in, out, in out (also in Oracle PL/SQL)
  - Named parameters / position parameters

Variations on value and reference parameters

- Default values of parameters
  - Ada, Oracle PL/SQL
- Variable length argument lists
  - Programming language C, Perl, ...
  - No type-checking, no control, may be dangerous

Type declaration

- From the lecture on Fun. prog. languages
- Type is defined from simpler types using type constructors: *, |, list, array, ...
- Type definition in Ocaml
- Example:

```
type name = typedef ;;
type name_1 = typedef_1
and name_2 = typedef_2
...
and name_n = typedef_n ;;
```

```
# type int_pair = int*int;;
type int_pair = int * int
# let v:int_pair = (1,1);;
val v : int_pair = (1, 1)
```

Parametrized types

- Type declarations can include type variables
- Type variable is a variable that can stand for arbitrary type
- Types that include variables are called parametrized types or also polymorphic types
- Parametrized type in Ocaml:

```
# type ('a,'b) pair = 'a*'b;;
type ('a, 'b) pair = 'a * 'b
# let v:(char,int) pair = ('a',1);;
val v : (char, int) pair = ('a', 1)
```

```
type 'a name = typedef ;;
type ('a_1 . . . 'a_n ) name = typedef ;;
```

Records

- Record is a product with named components
- Record type definition in Ocaml

```
type name = { name_1 : t_1 ; . . . ; name_n : t_n } ;;
```

- Record construction

```
{ name_1 = expr_1 ; . . . ; name_n = expr_n } ;;
```

- Example in Ocaml:

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

In C:

```
struct complex {
  double rp;
  double ip;
};
```

Records

- Operations:
  - Accessing components
  - Pattern matching

```
expr.name
```

$\{\ name_i = p_i\ ;\ .\ .\ .\ ;\ name_j = p_j\ \}$

```
# let add_complex c1 c2 = {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-.y1*.y2; im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

Mutable records

- Record components can be defined mutable

```
type name = { ...; mutable name_i: t_i ; ... }
```

- Operation:
  - Component assignement

```
expr1.name <- expr2
```

- Example in Ocaml

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc=3; yc=0}
```

Example in Ocaml

```
# let moveto p dx dy =
    let () = p.xc <- p.xc +. dx
    in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

Pointers

- A pointer is a reference to an object in the memory
  - The pointer can be typed (Pascal,C,...) / untyped (Lisp)
- Pointer is more than memory address
  - A pointer is a high-level concept: a typed reference to an object.
  - An address is a low-level concept: the location of a word in memory.
  - Pointers are often implemented as addresses, but not always

Pointers

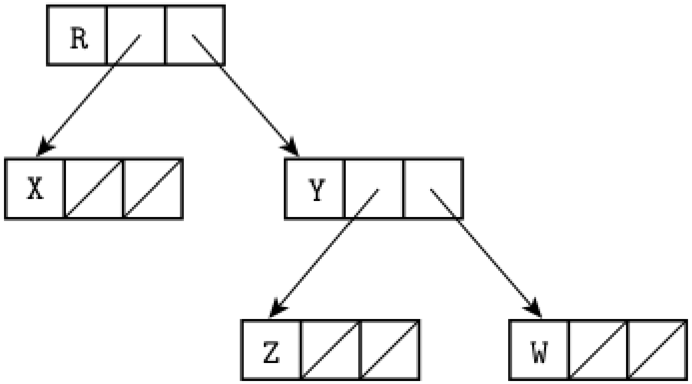
- Pointed object location
  - In some languages pointers are restricted to refer to objects on heap (Java, Pascal, Ada, Modula)
  - Other languages use pointers that can point to any location (C, C++, ...)
    - »address of« operator
- Operations on pointers
  - Allocation and deallocation of objects on the heap
  - Dereferencing a pointer to access an objects to which it points
  - Assignment of one pointer to another

Pointers

- Explicit pointer type
  - Programming languages C, C++, Pascal, … include a pointer type
    - Pointer is represented by an address
    - Explicit allocation and deallocation
  - ML references
    - Typed reference
    - Automatic allocation/deallocation
- Implicit pointers
  - Java, Scala, Ocaml,… object are represented by pointers
    - Structures of objects are implemented as references
  - ML uses reference model of variables
    - All data structures are represented as pointers (references)

Pointers and recursive types

- Recursive data structures in languages with explicit pointers
  - Value model of variables
  - Pascal, Ada, C, C++, ...
- Example in Pascal and C




```
type chr_tree_ptr = ^chr_tree;
     chr_tree = record
        left, right : chr_tree_ptr;
        val : char
     end;
```

```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};
```

```
new(my_ptr);
```

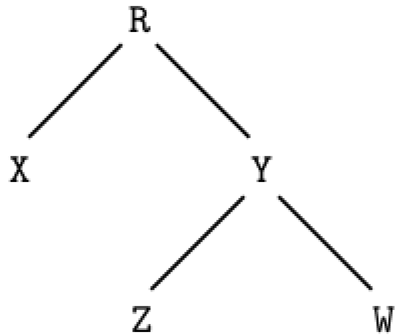
```
my_ptr = malloc(sizeof(struct chr_tree));
```

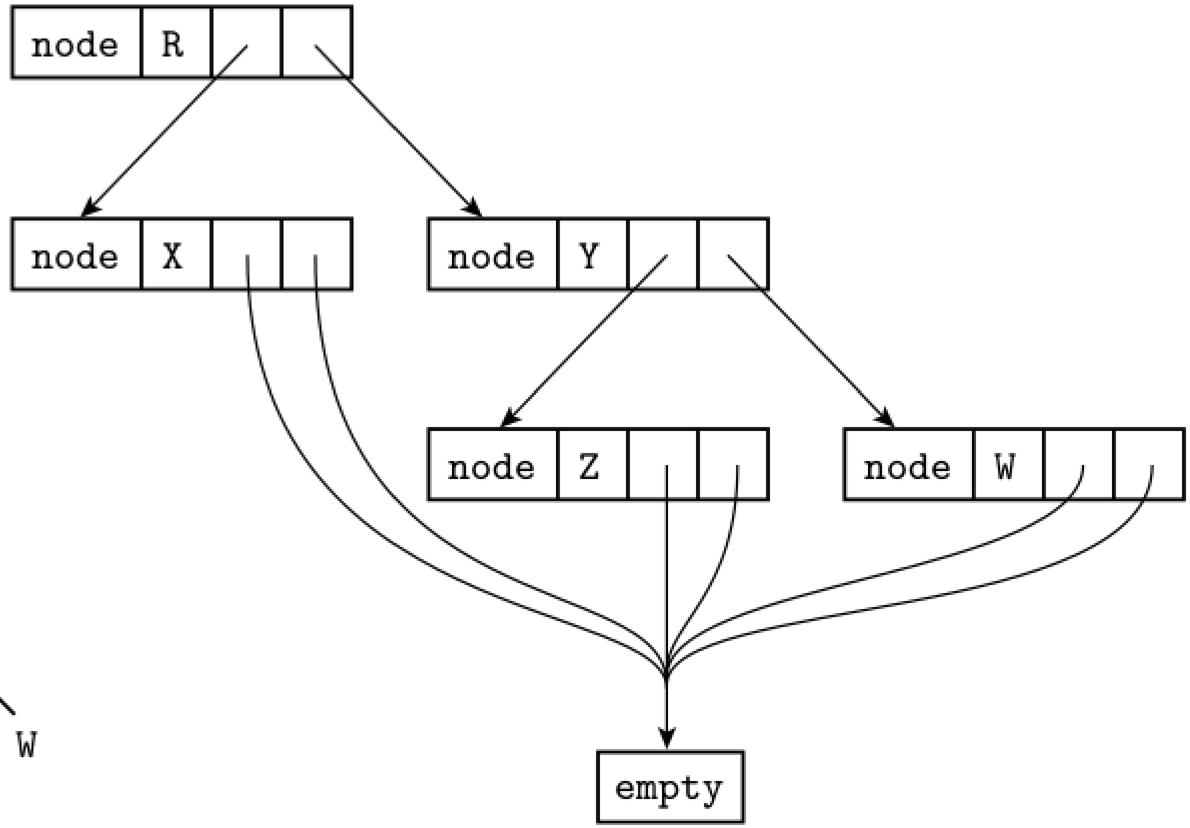
Recursive types in OCaml

- Recursive data structures include the pointers to structures of the same kind
- An example of a recursive data structure in Ocaml
  - ML has implicit pointers for records, arrays, ...
  - ML uses reference model of variables

```
# type ctree = Empty | Node of char * ctree * ctree;;
type ctree = Empty | Node of char * ctree * ctree
# Node(»R«,Node(»X«,Empty,Empty),
                Node(»Y«,Node(»Z«,Empty,Empty),
                            Node(»W«,Empty,Empty)))
```

Recursive types in OCaml





Example: Implementation of lists

```
# type 'a rnode = { mutable cont: 'a; mutable next: 'a rlist }
    and  'a rlist = Nil | Elm of 'a rnode;;
type 'a rnode = { mutable cont : 'a; mutable next : 'a rlist; }
and 'a rlist = Nil | Elm of 'a rnode
```

```
# let l1 = Elm {cont = 1; next = Elm {cont = 2; next = Nil}};;
val l1 : int rlist =  Elm {cont = 1; next = Elm {cont = 2; next = Nil}}
# let cons v l = Elm {cont=v; next=l};;
val cons : 'a -> 'a rlist -> 'a rlist = <fun>
```

```
# let ( ** ) v l = cons v l;;
val ( ** ) : 'a -> 'a rlist -> 'a rlist = <fun>
# let l2 = cons 3 (cons 4 Nil));;
val l2 : int rlist =  Elm {cont = 3; next = Elm {cont = 4; next = Nil}}
# let l3 = 5**6**Nil;;
val l3 : int rlist = Elm {cont = 5; next = Elm {cont = 6; next = Nil}}
```

Example: Implementation of lists

```
# exception EmptyList;;
exception EmptyList
# let head l = match l with Nil -> raise EmptyList | Elm r -> r.cont;;
val head : 'a rlist -> 'a = <fun>
# let tail l = match l with Nil -> raise EmptyList | Elm r -> r.next;;
val tail : 'a rlist -> 'a rlist = <fun>
# head l1;;
- : int = 1
# tail l1;;
- : int rlist = Elm {cont = 2; next = Nil}
```

```
# let rec length l = match l with Nil -> 0 | Elm {next=t} -> 1+length t;;
val length : 'a rlist -> int = <fun>
# length l1;;
- : int = 2
```

Example: Implementation of lists

```
# let rec append l1 l2 = match l1,l2 with
  Elm r1,_ -> Elm {cont=r1.cont; next=append r1.next l2}
| Nil,Elm r2 -> Elm {cont=r2.cont; next=append Nil r2.next}
| Nil,Nil -> Nil;;
val append : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append l1 l2;;
- : int rlist =
Elm {cont=1; next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist = Elm {cont=1;  next=Elm {cont=2; next=Nil }}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil }}
```

Example: Implementation of lists

```
# let rec append1 l1 l2 = match l1 with
  Nil -> l2
| Elm r when r.next=Nil -> r.next <- l2; l1
| Elm r -> ignore (append1 r.next l2); l1;;
val append1 : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append1 l1 l2;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil}}
```

Arrays

- Arrays are data structures holding the finite number of elements of certain data type
- In imperative languages an array is an important data structure
  - C, C++, Java, Fortran, Pascal, ...
  - Similar relationship as between functional PL and lists
- An array is by definition mutable, but its size is fixed

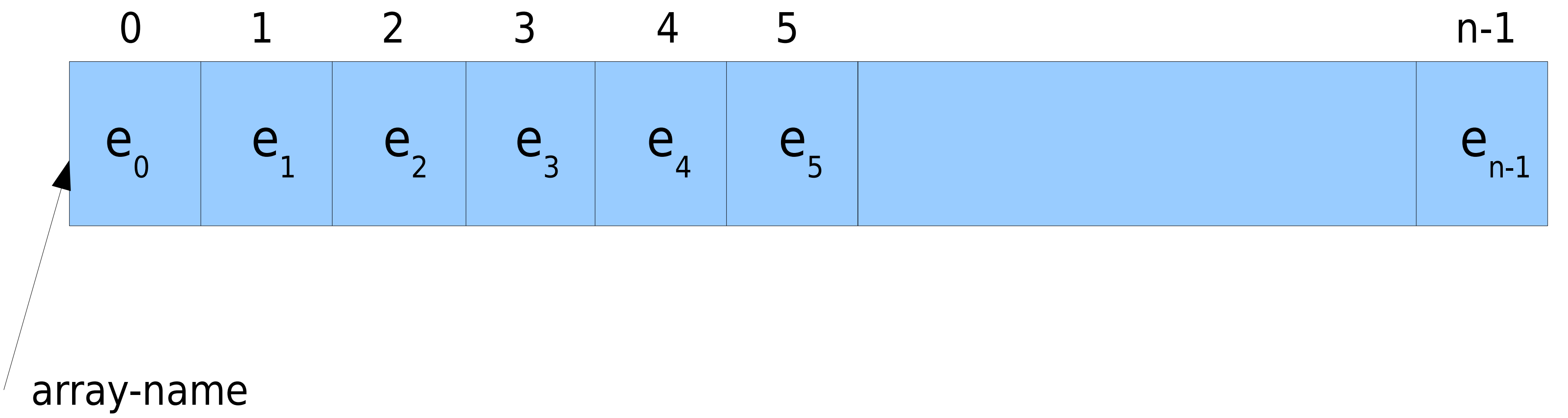
Arrays

- Accessing elements of array
  - Most languages append index delimited by a variant of parentheses to the array name (a(), a[], a{}, ...)
  - Indexes of arrays are usually of integer type but can be also of discrete type
- Declaration of an array
  - Indexes in most languages are defined by range
  - Index in C starts with 0

```
char[] upper;          /* Java */
char upper[];          /* alternative declaration */
upper = new char[26];
char upper[26];     /* C */
character, dimension (1:26) :: upper  /* Fortran */
character (26) upper    /* shorthand notation */
var upper : array ['a'..'z'] of char;   /* Pascal */
```

Implementation of array

- Array is represented in subsequent memory words
  - It is divided into n elements
  - Each element uses m memory words, depending on the size of element type
  - Elements are accessible via indexes



Arrays in OCaml

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

- Elements can be enumerated between [|...|]
- Arrays are integrated into Ocaml
  - (but not so profoundly as lists)
- Similarly to lists, there is a module Array that includes all necessary operations
- Create an array
- Access/update an array element
  - Accessing an element
  - Setting new value

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

$expr_1.(expr_2)$

$expr_1.(expr_2) <- expr_3$

Arrays in OCaml

- Example:
- Array index must not go accross the borders

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

- Checking that the index is not used outside borders is expensive
  - Some languages do not check this by default (C)

Functions on arrays

```
# let n = 10;
val n : int = 10
# let v = Array.create n 0;;
val v:int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
# for i=0 to (n-1) do v.(i)<-i done;;
- : unit = ()
# v;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

```
# let reverse v =
        let tmp=ref 0
        and n = Array.length(v)
        in for i=0 to (n/2-1) do
                tmp := v.(i);
                v.(i) <- v.(n-i-1);
                v.(n-i-1) <- (!tmp);
            done;;
- : unit = ()
# reverse(v);;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
```

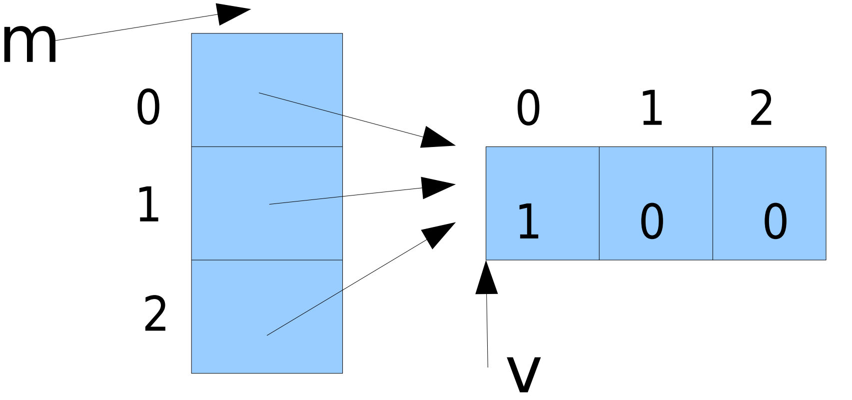
```
# let u = [|2;3|];;
val u : int array = [|2; 3|]
# let m = 2;;
val m : int = 2
# let subarray u v =
        let found = ref false
        and i = ref 0
        in while ((!i<=(n-m)) && not !found) do
                found := true;
                for j=0 to (m-1) do
                    if v.(!i+j) != u.(j) then
                        found := false
                done;
                i := !i+1
            done;
        !found;;
val subarray : 'a array -> 'a array -> bool = <fun>
# subarray u v;;
- : bool = true
```

Example: subarray()

```
# let prefix u v i =
    let found = ref true
    and m = Array.length(u)
    in for j=0 to (m-1) do
           if v.(i+j) != u.(j) then
              found := false
           done;
        !found;;
val prefix : ’a array -> ’a array -> int ->
                   bool = <fun>
# prefix u v 0;;
- : bool = false
# prefix u v 2;;
- : bool = true
```

```
# let subarray u v =
    let found = ref false
    and i = ref 0
    and m = Array.length(u)
    and n = Array.length(v)
    in while ((!i<=(n-m)) && not !found) do
            found := prefix u v !i;
            i  := !i+1
        done;
    !found;;
val subarray : 'a array -> 'a array ->
                        bool = <fun>
```

Implementation of arrays in OCaml




```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array =
          [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

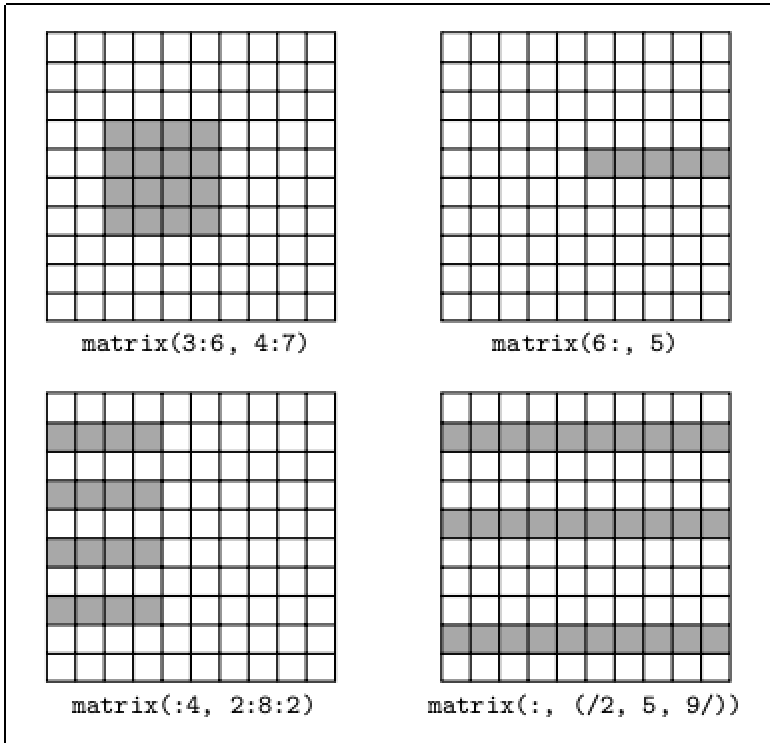
```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array =
   [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

Matrices

- Multidimensional arrays
  - Declaration
  - Arrays of arrays
    - C, Pascal, Modula, ...
  - Two-dimensional array
    - Ada, Fortran, ...
- Slices
  - A slice is a rectangular portion of an array.
  - R, Fortran, Python

```
/* C */
double mat[10][10];
/* Ocaml */
type 'a matrix = array array 'a;;
/* Modula-3 */
VAR mat : ARRAY [1..10] OF ARRAY [1..10] OF REAL;
/* Ada */
mat1 : array (1..10, 1..10) of real;
```



Matrices

- Matrix has n rows and k columns
- Implementation
  - One bock of memory (Fortran, Pascal, ...)
  - Each row in separate block (C, Java, Ocaml, ...)

```
# let m = Array.create_matrix 4 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
# for i=0 to 3 do m.(i).(i) <- 1; done;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0; 0|]; [|0; 1; 0; 0|]; [|0; 0; 1; 0|]; [|0; 0; 0; 1|]|]
# m.(1);;
- : int array = [|0; 1; 0; 0|]
```

Operations on matrices

```
# let add_mat a b =
    let r = Array.create_matrix n m 0.0 in
      for i = 0 to (n-1) do
        for j = 0 to (m-1) do
          r.(i).(j) <- a.(i).(j) +. b.(i).(j)
        done
      done ; r;;
val add_mat : float array array -> float array array -> float array array = <fun>
# a.(0).(0) <- 1.0; a.(1).(1) <- 2.0; a.(2).(2) <- 3.0;;
- : unit = ()
# b.(0).(2) <- 1.0; b.(1).(1) <- 2.0; b.(2).(0) <- 3.0;;
- : unit = ()
# add_mat a b;;
- : float array array = [|[|1.; 0.; 1.|]; [|0.; 4.; 0.|]; [|3.; 0.; 3.|]|]
```

Other types in imperative languages

- Sets
- Unions
- Dictionaries

Sets

- A set stores unique values, without any particular order
- Basic operations
  - Set ops: create, delete, add_element, delete_element
  - Boolean ops: membership, subset, equality, disjoint
  - Set algebra: union, difference, intersection
- Implementation
  - There are many different ways of implementing sets
    - Each with serious weaknesses for some purposes
    - For any specific purpose, it is not hard to implement set functionality using commonly available data structures

Sets

- Implementation
  - Lists, arrays
    - Unefficient
  - Bitstrings
    - Storage efficient representation
    - Operations are converted to processor instructions
  - Binary search trees (library: Ocaml, Haskell)
  - Hash tables
  - Dictionary representation of sets
    - Dictionary is a data structure storing key/value pairs
    - Elements of a set are the keys of a dictionary
    - See later the implementations of a dictionary

Sets

- Sets in programing languages
  - Libraries: C++, Java, .NET, Ruby, Ocaml, Swift, Erlang
  - Build-in: Javascript, Python, Pascal

Sets in Pascal

```
program SetDemo;

type
  TCharSet = set of Char;

var
  Ch: Char;
  MyCharSet: TCharSet;
begin
  MyCharSet := ['P','N','L'];
  if 'A' in MyCharSet then
    WriteLn ('Wrong: A in set MyCharSet')
  else
    WriteLn ('Right: A is not in set MyCharSet');
  Include (MyCharSet, 'A');  { A, L, N, P }
  Exclude (MyCharSet, 'N');  { A, L, P }
  MyCharSet := MyCharSet + ['B','C'];  { A, B, C, L, P }
  MyCharSet := MyCharSet - ['C','D'];  { A, B, L, P }
  WriteLn ('set MyCharSet contains:');
  for Ch in MyCharSet do
    WriteLn (Ch);
end.
```

Sets in Pascal

In the following description, S1 and S2 are variables of set type, s is of the base type of the set.

`S1 := S2`
: Assign a set to a set variable.

`S1 + S2`
: Union of sets.

`S1 - S2`
: Difference between two sets.

`S1 * S2`
: Intersection of two sets.

`S1 >< S2`
: Symmetric difference

`S1 = S2`
: Comparison between two sets. Returns boolean result. `True` if `S1` has the same elements as `S2`.

`S1 <> S2`
: Comparison between two sets. Returns boolean result. `True` if `S1` does not have the same elements as `S2`.

`S1 < S2`
`S2 > S1`
: Comparison between two sets. Returns boolean result. `True` if `S1` is a strict subset of `S2`.

`S1 <= S2`
`S2 >= S1`
: Comparison between two sets. Returns boolean result. `True` if `S1` is a subset of (or equal to) `S2`.

`s in S1`
: Set membership test between an element `s` and a set. Returns boolean result. `True` if `s` is an element of `S1`.

Unions

- Type constructed by union
  - Make a new type by taking the union of existing types
- Unions in Ocaml
  - Type definition
  - Construction of instance
  - Pattern matching
- Union in other languages
  - ML-family, Haskell
  - Tagged union: Pascal, Ada, Modula2
    - Also called: Variant records
  - Untagged union: C, C++

type name = ...
    | $Name_i$ ...
    | $Name_j$ of t_j ...
    | $Name_k$ of t_k * ...* t_l ... ;;

Unions in C

- Type that allows multiple different values to be stored in the same memory space

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Unions in C

```c
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

```
Memory size occupied by data : 20
```

Unions in C

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Variant records in Pascal

- Parts of records are variant
  - Type tag is used to differentiate among the variants
  - Records must include the largest variant

```
type paytype=(salaried, hourly);
var employee: record
          id: integer;
          dept: array [1...3] of char;
          age: integer;
          case payclass: paytype of
               salaried: (monthlyrate: real; stardate: integer);
               hourly (hourrate: real; reghours: integer; overtime: integer);
     end;
```

Dictionaries

- Alternative names
  - Associative array, map, symbol table
- A store of key/value pairs
  - Keys and values are of arbitrary type
  - Operations provided
    - Create, access, update, delete, to-list, keys, ...
- Programming languages
  - Initial implementations
    - TMG (1965, compiler-compiler), SETL (late 1960s), Snobol (1969)
  - Script languages
    - AWK, Rexx, Perl, PHP, Tcl, JavaScript, Python, Ruby, Go, Lua

Dictionaries

  - Other languages
    - C++, Java, Scala, Erlang, OCaml, Haskell
- Implementation of a dictionary
  - Hash tables
    - `O(1)`
  - Search trees
    - Binary search trees, B+-trees, ...
- Very popular and useful data structure
  - Any data structure can be represented

Python dictionary operation

- Creation
  - `D = {}, D = {’key1’:value1,’key2’:value2, … }`
  - `dict(name1=value1,name2=value2, …)`
  - From a list: of pairs, names, ...
- Access by a key
  - `D[’name’], D[’name1’][’name2’], ’name’ in D`
  - `D.get(key)`
- Update and delete a key
  - `D.update(D1)`
  - `del D[key], D.pop(key)`
- Readig keys, values and key/value pairs
  - `D.keys(), D.values(), D.items()`