Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

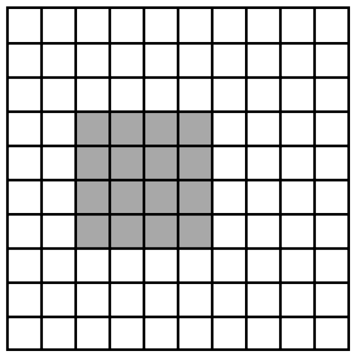
Lesson 22

- Array allocation and layout
- Intermediate code generation for array declaration and access
- Strings
- Variant and discriminated records
- Algebraic data types and classes as union types

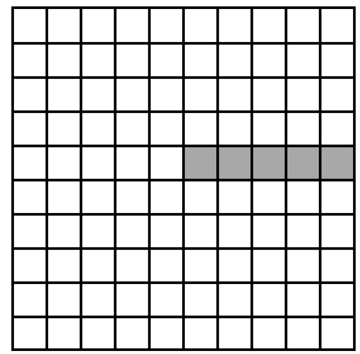
Array-level operations

- Assigment
  - Value or Reference Model
- Comparison for equality or lexicographic ordering (Ada)
- Arithmetic (pointwise) + specific *intrinsic* (built-in) operations in Fortran 90 (and APL)
  - Searching, transposition, reshaping…
- ***Slice*** or ***section***
  - Returns a sub-array by selecting sub-ranges of dimensions

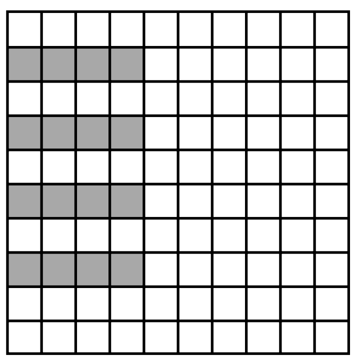
Slicing in Fortran 90



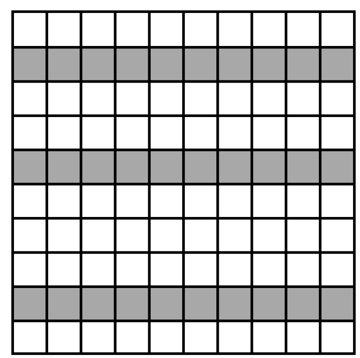
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



`matrix(:4, 2:8:2)`



`matrix(:, (/2, 5, 9/))`

Array allocation

- ***static array, global lifetime* —** If a static array can exist throughout the execution of the program, then the compiler can allocate space for it in *static global memory*
- ***static array, local lifetime* —** If a static array should not exist throughout the execution of the program, then space can be allocated *in the subroutine's stack frame* at run time.
- ***dynamic array, local lifetime* —** If the index range is known at runtime, the array can still be allocated *in the stack*, but in a variable size area
- ***fully dynamic* —** If the index range can be modified at runtime it has to be allocated *in the heap*

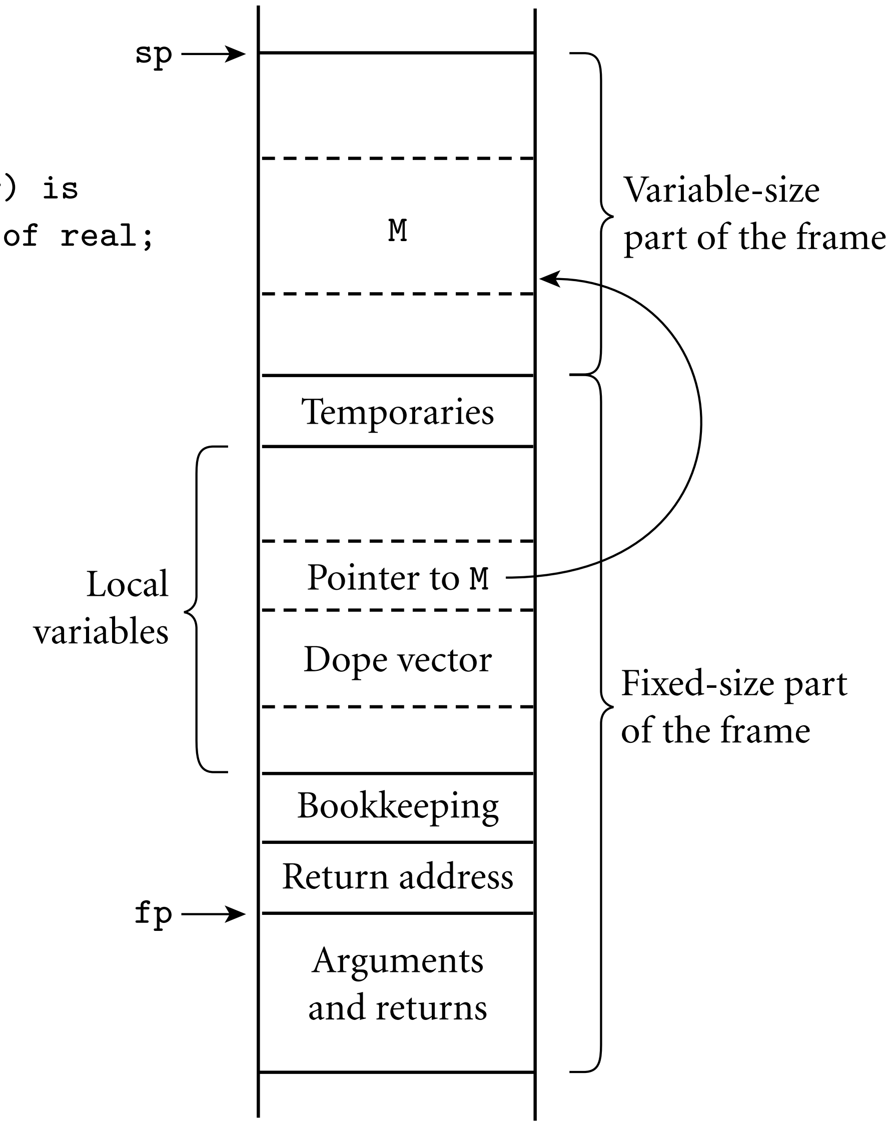
Dope vector: run-time data structure that keeps information about lower (and upper) limits of arrays ranges

  – Needed for checking bounds and computing addresses of elements

Allocation of dynamic arrays on stack

```
-- Ada:
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



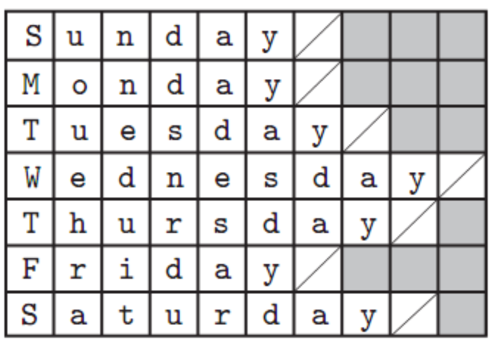
Arrays: memory layout

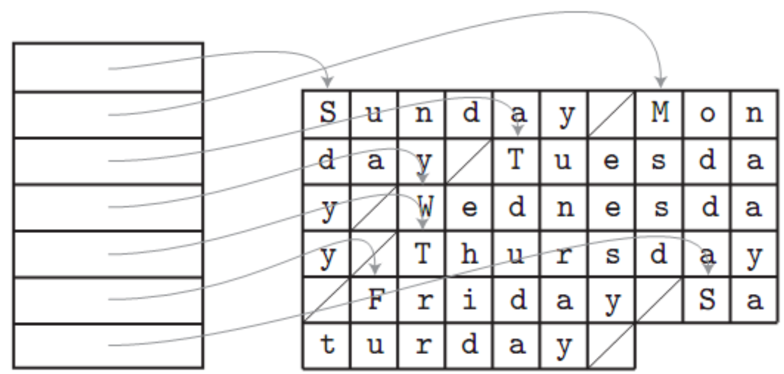
- Contiguous elements
  - column major - only in Fortran
  - row major
    - used by everybody else
- Row pointers
  - an option in C, the rule in Java
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

Arrays’ memory layout in C

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```





- Address computation varies a lot
- With contiguous allocation part of the computation can be done statically

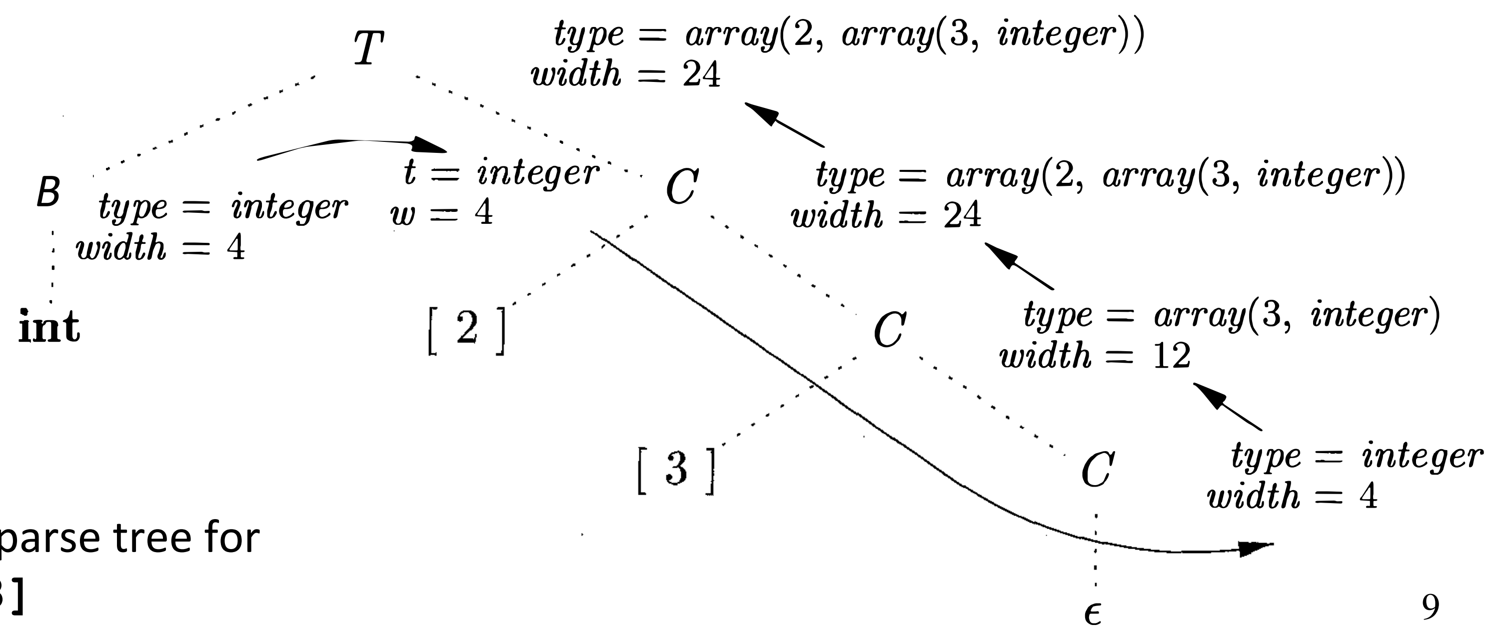
Compiling array declarations and addressing

- Translation scheme for associating with an array declaration a *type expression* and the *width* of its instances
- Computing the address of an array element: one- and multi-dimensional cases
- Generating three address code for addressing array elements

Declaration of Multidimensional Arrays: Syntax Directed Translation Scheme for type/width

Example: `int[2][3]`

$T \rightarrow$	B	{ t = $B.type$; w = $B.width$; }
	C	{ $T.type$ = $C.type$; $T.width$ = $C.width$ }
$B \rightarrow$	**int**	{ $B.type$ = 'integer'; $B.width$ = 4; }
$B \rightarrow$	**float**	{ $B.type$ = 'float'; $B.width$ = 8; }
$C \rightarrow$	ε	{ $C.type$ = t; $C.width$ = w; }
$C \rightarrow$	[**num**] C_1	{ $C.type$ = $array$(**num**.$value$, $C_1.type$); $C.width$ = **num**.$value$ * $C_1.width$; }



Annotated parse tree for
`int[2][3]`

Addressing Array Elements: One-Dimensional Arrays

- Assuming that elements are stored in adjacent cells:

```
A : array [10..20] of integer;
```

low (10), *high* (20), *Type's size* (w)

… `:= A[i]` $= base_{\mathtt{A}} + (i - low) * w$

- If *base, low* and *w* are known at compile time:

$= i * w + c$ ***where*** $c = base_{\mathtt{A}} - low * w$

Example with $low = 10$; $w = 4$

…

`t1 := c` **`//c =`** $base_{\mathtt{A}}$ - 10 * 4, can be stored in the symbol table

```
t2 := i * 4
t3 := t1[t2]
…  := t3
```

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;`

$$low_1 = 1, low_2 = 1,$$
$$n_1 = high_1 - low_1 + 1 = 2, \quad n_2 = 3,$$
$$w = 4 \text{ (element type size)}$$

$base_{\mathtt{A}}$ (as in C) Row-major	$base_{\mathtt{A}}$ Column-major (as in Fortran)
A[1][1]	**A[1][1]**
A[1][2]	**A[2][1]**
A[1][3]	**A[1][2]**
A[2][1]	**A[2][2]**
A[2][2]	**A[1][3]**
A[2][3]	**A[2][3]**

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;` (Row-major)

… **`:= A[i][j]`** $= base_{\mathtt{A}} + ((i - low_1) * n_2 + j - low_2) * w$

$= ((i * n_2) + j) * w + c$

where $c = base_{\mathtt{A}} - ((low_1 * n_2) + low_2) * w$

Example with $low_1 = 1$; $low_2 = 1$; $n_2 = 3$; $w = 4$

```
t1  := i * 3
t1  := t1 + j
t2  := c          // c = baseA - (1 * 3 + 1) * 4
t3  := t1 * 4
t4  := t2[t3]     // base t2, offset t3
…   := t4
```

Addressing Array Elements: Grammar

Grammar:

$S \rightarrow$ **id** = E ;
    | L = E ;
$E \rightarrow E$ + E
    | **id**
    | L
$L \rightarrow$ **id** [E]
    | L [E]

Synthesized attributes:

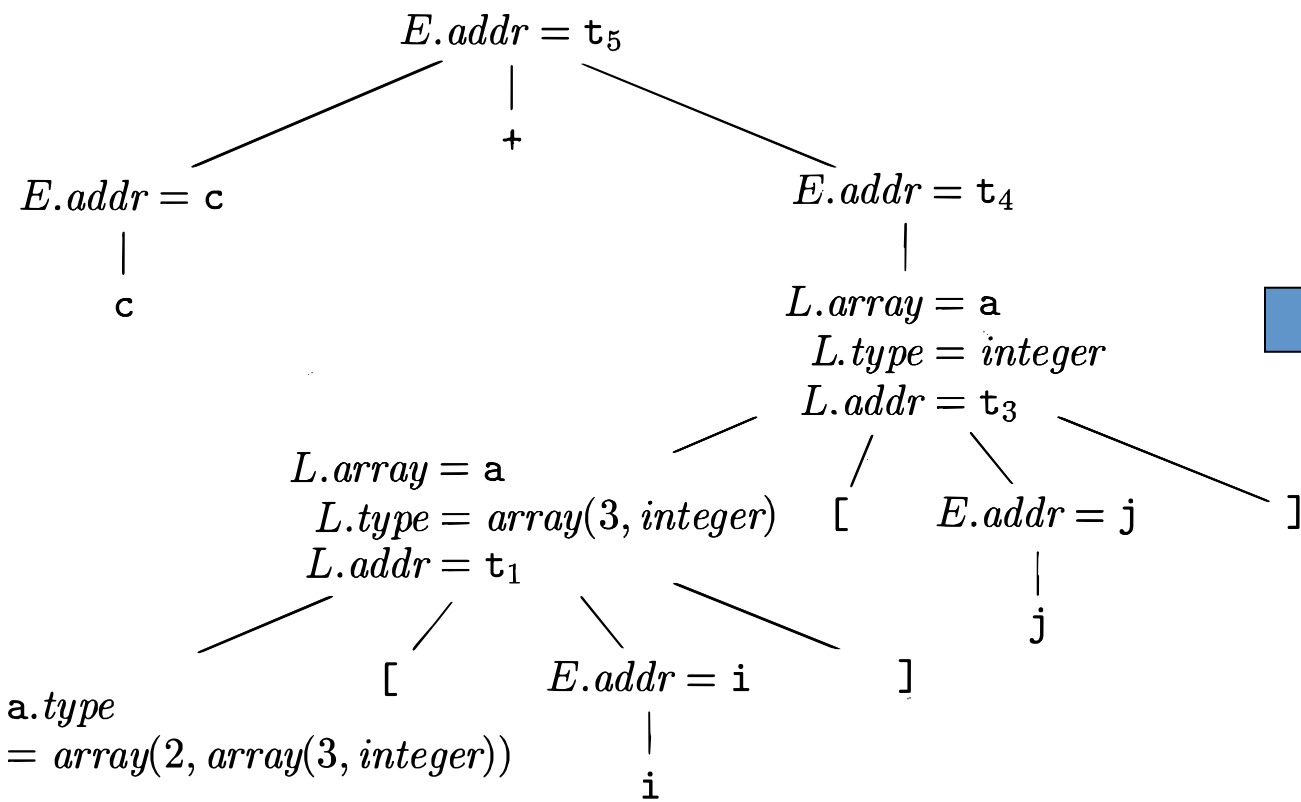
E.addr	name of temp holding value of *E*
L.addr	temporary to compute offset
L.array	pointer to symbol table entry for the array name
L.array.base	base address
L.array.type	type of the array, eg. *array(2, array(3,int))*
L.array.type.elem	type of array elements, eg. *array(3,int)*
L.type	type of the subarray generated by *L*
L.type.width	memory allocated for data of type *L.type*

- Nonterminal *L* generates an array name followed by a sequence of indexes, like
  `a[i][j][k]`
- *L* can appear both as left- and right-value

Addressing array elements: generating three address statements

$S \rightarrow$ **id** = E ;	{ *gen(top.get(***id***.lexeme)* '=' *E.addr);* }	// no array
| L = E ;	{ *gen(L.array.base* '[' *L.addr* ']' '=' *E.addr);* }	// address = base + offset
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp()*; *gen(E.addr* '=' E_1*.addr* '+' E_2*.addr);* }	// similarly for *, -, …
| **id**	{ *E.addr = top.get(***id***.lexeme);* }	
| L	{ *E.addr* = **new** *Temp();* *gen(E.addr* '=' *L.array.base* '[' *L.addr* ']'*);* }	// address = base + offset
$L \rightarrow$ **id** [E]	{ *L.array = top.get(***id***.lexeme);* *L.type = L.array.type.elem;* *L.addr* = **new** *Temp();* *gen(L.addr* '=' *E.addr* '*' *L.type.width);* }	// computes the offset
| L_1 [E]	{ *L.array=*L_1*.array;* *L.type =* L_1*.type.elem;* *t* = **new** *Temp();* *L.addr=* **new** *Temp();* *gen(t* '=' *E.addr* '*' *L.type.width);* *gen(L.addr* '=' L_1*.addr* '+' *t);* }	

Example - generating intermediate code for access to array: `c + a[i][j]`




```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Strings

- A **string** is a sequence of 0 or more characters.
- Usually ad-hoc syntax is supported
- Some PLs (ML, Python) treat strings as *primitive*.
- Haskell treats strings as *lists* of characters. Strings are thus equipped with general list operations (length, head selection, tail selection, concatenation, …).
- Ada treats strings as *arrays* of characters. Strings are thus equipped with general array operations (length, indexing, slicing, concatenation, …).
- Also in C strings are arrays of characters, but handled differently from other arrays
- Java treats strings as *objects*, of class `String`.

Disjoint Unions

- In a **disjoint union**, a value is chosen from one of several different types.
- Let ***S* + *T*** stand for a set of disjoint-union values, each of which consists of a **tag** together with a **variant** chosen from either type *S* or type *T*. The tag indicates the type of the variant:

  $$S + T = \{ \mathit{left}\ x \mid x \in S \} \cup \{ \mathit{right}\ y \mid y \in T \}$$

  – *left x* is a value with tag *left* and variant *x* chosen from *S*

  – *right x* is a value with tag *right* and variant *y* chosen from *T*.
- We write ***left S* + *right T*** (instead of *S* + *T*) when we want to make the tags explicit.

Disjoint Unions

- Basic operations on disjoint-union values in $S + T$:
  - **construction** of a disjoint-union value from its tag and variant
  - **tag test**, to see whether the variant is from S or T
  - **projection**, to recover the variant in S or in T
- **Algebraic data types** (Haskell), **discriminated records** (Ada), **unions** (C) and **objects** (Java) can be understood as disjoint unions.
- We can generalise to multiple variants: $S_1 + S_2 + \ldots + S_n$.

Variant records (unions)

- Origin: Fortran I *equivalence statement:* variables should share the same memory location
- C's *union* types
- Motivations:
  - Saving space
  - Need of different access to the same memory locations for system programming
  - Alternative configurations of a data type

Fortran I -- equivalence statement

```
integer i
real r
logical b
equivalence (i, r, b)
```

C -- union

```
union {
    int i;
    double d;
    _Bool b;
};
```

Variant records (unions) (2)

- In Ada, Pascal, unions are *discriminated* by a tag, called *discriminant*
- Integrated with records in Pascal/Ada, not in C

ADA – discriminated variant

tag

```
type Form is
     (pointy, circular, rectangular);
type Figure (f: Form := pointy) is record
        x, y: Float;
        case f is
           when pointy      => null;
           when circular    => r: Float;
           when rectangular => w, h: Float;
        end case;
     end record;
```

Using discriminated records in Ada

- Application code:

```
box: Figure :=
     (rectangular, 1.5, 2.0, 3.0, 4.0);
function area (fig: Figure) return Float
is
begin
  case fig.f is
    when pointy =>
      return 0.0;
    when circular =>
      return 3.1416 * fig.r**2;
    when rectangular =>
      return fig.w * fig.h;
  end case;
end;
```

discriminated-record construction

tag test

projection

(Lack of) Safety in variant records

- Only Ada has strict rules for assignment: tag and variant have to be changed *together*
- For *nondiscriminated unions* (Fortran, C) no runtime check: responsibility of the programmer
- In Pascal the tag field can be modified independently of the variant. Even worse: the tag field is optional.
- Unions not included recent OO laguages: replaced by *algebraic data types* or *classes + inheritance*

Haskell/ML algebraic data types

- Type declaration:

```
data Number = Exact Int | Inexact Float
```

Each Number value consists of a tag (**constructor**), together with either an Integer variant (if the tag is *Exact*) or a Float variant (if the tag is *Inexact*).

- Application code:

```
pi = Inexact 3.1416
rounded :: Number -> Integer
rounded num =
   case num of
     Exact i   -> i
     Inexact r -> round r
```

projection (by pattern matching)

Active patterns in F#

- With algebraic data types, the type definition determines uniquely the patterns
- *Active patterns*, can be used to “wrap” a data type, algebraic or not, providing a different perspective for use of pattern matching
- Essentially, active patterns define ad-hoc, unnamed union types

Active pattern definition

```
let (|Even|Odd|) n =
  if n % 2 = 0 then
    Even
  else
    Odd
```

Roughly equivalent to

```
type numKind =
  | Even
  | Odd

let get_choice n =
  if n % 2 = 0 then
    Even
  else
    Odd
```

Using active patterns

```
let testNum n =
  match n with
  | Even -> printfn "%i is even" n
  | Odd -> printfn "%i is odd" n;;§
```

Active Patterns defining Constructors with Parameters

```
/* Active pattern for Sequences */
let (|SeqNode|SeqEmpty|) s =
   if Seq.isEmpty s then SeqEmpty
   else SeqNode ((Seq.head s), Seq.skip 1 s)

/* SeqNode is a constructor with two parameters */

let perfectSquares = seq { for a in 1 .. 10 -> a * a }

let rec printSeq = function
   | SeqEmpty -> printfn "Done."
   | SeqNode(hd, tl) ->
      printf "%A " hd
      printSeq tl;;

> printSeq perfectSquares;;
1 4 9 16 25 36 49 64 81 100 Done.
```

Active Patterns in F# (2)

- Active Patterns
  - Can introduce union constructors with parameters

Java objects as unions

- Type declarations:

```
class Point {
  private float x, y;
  ...   // methods
}
class Circle extends Point {
  private float r;
  ...   // methods
}
class Rectangle extends Point {
  private float w, h;
  ...   // methods
}
```

inherits x and y from Point

inherits x and y from Point

Java objects as unions (2)

- Methods:

```
class Point {
   …
   public float area()
   { return 0.0; }
}
class Circle extends Point {
   …
   public float area()
   { return 3.1416 * r * r; }
}
class Rectangle extends Point {
   …
   public float area()
   { return w * h; }
}
```

overrides Point's area() method (Circle's area())

overrides Point's area() method (Rectangle's area())

Java objects as unions (3)

- Application code:

```
Rectangle box =
    new Rectangle(1.5, 2.0, 3.0,4.0);

float a1 = box.area();
Point it = …;
float a2 = it.area();
```

it can refer to a Point, Circle, or Rectangle object

calls the appropriate area() method

Assignment of composite values

- What happens when a composite value is assigned to a variable of the same type?
- **Value model**: all components of the composite value are copied into the corresponding components of the composite variable.
- **Reference model**: the composite variable is made to contain a reference to the composite value.
- **Note:** this makes no difference for basic or immutable types.

- C and Ada adopt value model
- Java adopts value model for primitive values, reference model for objects.
- Functional languages usually adopt the reference model

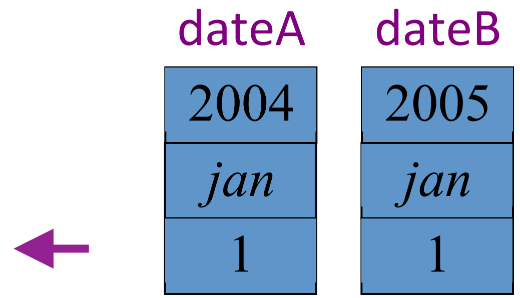
Example: Ada value model (1)

- Declarations:

```
type Date is
     record
        y: Year_Number;
        m: Month;
        d: Day_Number;
     end record;
dateA: Date := (2004, jan, 1);
dateB: Date;
```

- Effect of copy semantics:

```
dateB := dateA;
dateB.y := 2005;
```



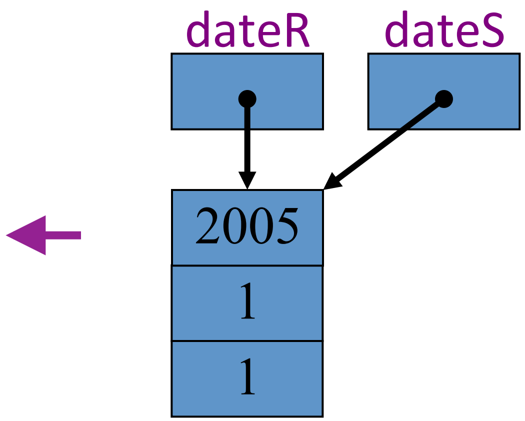
Example: Java reference model (1)

- Declarations:

```
class Date {
  int y, m, d;
  public Date (int y, int m, int d)
{ … }
}
Date dateR = new Date(2004, 1, 1);
Date dateS = new Date(2004, 12, 25);
```

- Effect of reference semantics:

```
dateS = dateR;
dateR.y = 2005;
```



Ada reference model with pointers (2)

- We can achieve the *effect* of reference model in Ada by using explicit ***pointers***:

```
type Date_Pointer is access Date;
Date_Pointer dateP = new Date;
Date_Pointer dateQ = new Date;
…
dateP.all := dateA;
dateQ := dateP;
```

Java value model with cloning (2)

- We can achieve the *effect* of copy semantics in Java by cloning:

```
Date dateR = new Date(2004, 4, 1);
dateT = dateR.clone();
```

Pointers

- Thus in a language adopting the *value model*, the *reference model* can be simulated with the use of pointers.
- A **pointer** (value) is a reference to a particular variable.
- A pointer's **referent** is the variable to which it refers.
- A **null pointer** is a special pointer value that has no referent.
- A pointer is essentially the address of its referent in the store, but it also has a *type*. The type of a pointer allows us to infer the type of its referent.
- Pointers mainly serve two purposes:
  - efficient (sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures, in conjunction with a heap storage manager

Dangling pointers

- A **dangling pointer** is a pointer to a variable that has been destroyed.
- Dangling pointers arise from the following situations:
  - where a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator
  - where a pointer to a local variable still exists at exit from the block in which the local variable was declared.
- A deallocator immediately destroys a heap variable. All existing pointers to that heap variable become dangling pointers.
- Thus deallocators are inherently unsafe.

Dangling pointers in languages

- C is highly unsafe:
  - After a heap variable is destroyed, pointers to it might still exist.
  - At exit from a block, pointers to its local variables might still exist (e.g., stored in global variables).
- Ada and Pascal are safer:
  - After a heap variable is destroyed, pointers to it might still exist.
  - But pointers to local variables may not be stored in global variables.
- Java is very safe:
  - It has no deallocator.
  - Pointers to local variables cannot be obtained.
- Functional languages are even safer:
  - they don't have pointers

Example: C dangling pointers

- Consider this C code:

```
struct Date {int y, m, d;};
struct Date *dateP, *dateQ;
dateP = (struct Date*)malloc(sizeof (struct Date));
dateP->y = 2004; dateP->m = 1; dateP->d = 1;
dateQ = dateP;
free(dateQ);

printf("%d", dateP->y);
dateP->y = 2005;
```

allocates a new heap variable

makes dateQ point to the same heap variable as dateP

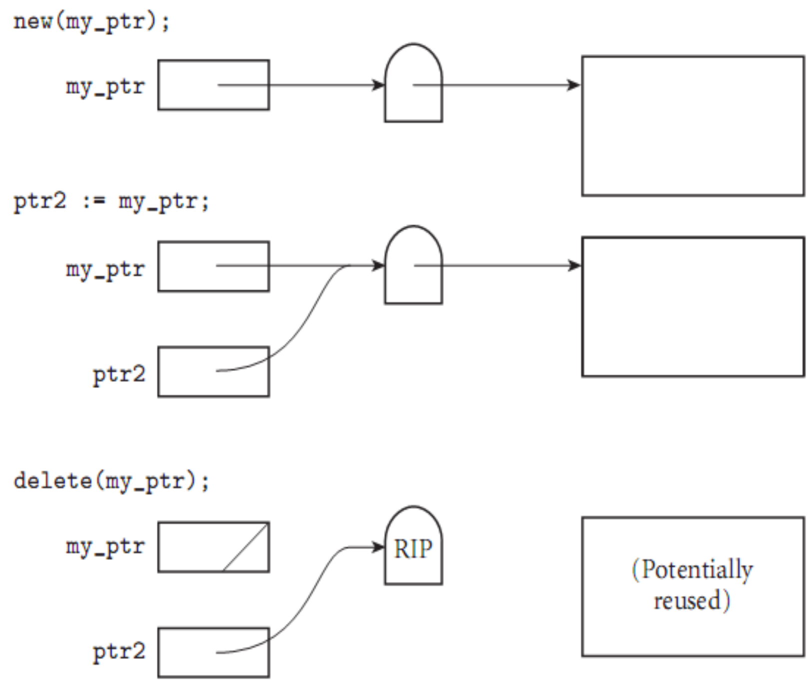
deallocates that heap variable (dateP and dateQ are now dangling pointers)

can fail

can fail

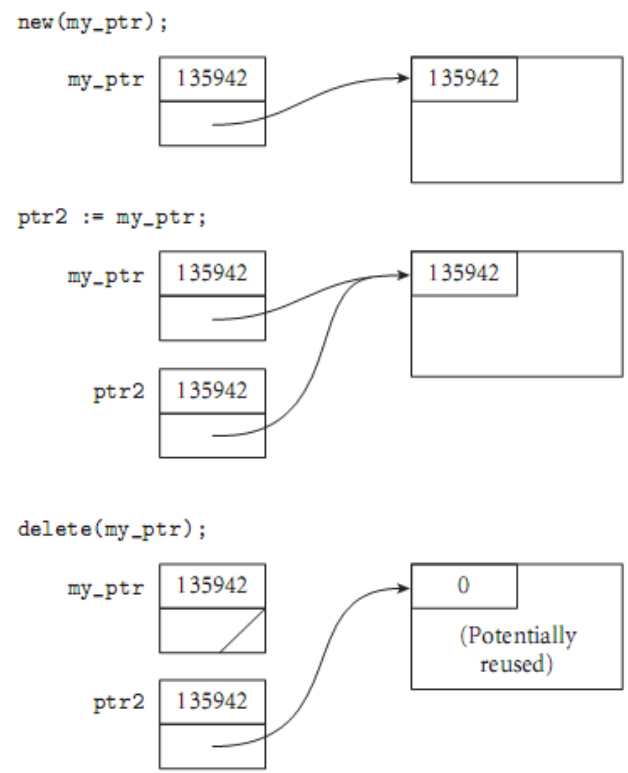
Techniques to avoid dangling pointers

- Tombstones
  - A pointer variable refers to a *tombstone* that in turn refers to an object
  - If the object is destroyed, the tombstone is marked as "expired"



Locks and Keys

- Heap objects are associated with an integer (lock) initialized when created.
- A valid pointer contains a key that matches the lock on the object in the heap.
- Every access checks that they match
- A dangling reference is unlikely to match.



Pointers and arrays in C

- In C, an array variable is a pointer to its first element

```
int *a == int a[]
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension, otherwise it allocates a pointer

`int **a, int *a[]` pointer to pointer to `int`

`int *a[n],` n-element array of row pointers

`int a[n][m],` 2-d array

- Pointer arithmetics: operations on pointers are scaled by the base type size. All these expressions denote the third element of **a**:

`a[2]` **`(a+2)[0]`** **`(a+1)[1]`** **`2[a]`** **`0[a+2]`**

C pointers and recursive types

- C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

```
int *a[n], n-element array of pointers to integer
int (*a)[n], pointer to n-element array of
  integers
```

- Compiler has to be able to tell the size of the things to which you point
  - So the following aren't valid:

```
int a[][]       bad
int (*a)[]      bad
```

Recursive types: Lists

- A **recursive type** is one defined in terms of itself, like lists and trees
- A **list** is a sequence of 0 or more component values.
- The **length** of a list is its number of components. The **empty list** has no components.
- A non-empty list consists of a **head** (its first component) and a **tail** (all but its first component).
- Typical constructor: **cons**: A x A-list -> A-list
- A list is **homogeneous** if all its components are of the same type. Otherwise it is **heterogeneous**.

List operations

- Typical list operations:
  - length
  - emptiness test
  - head selection
  - tail selection
  - concatenation
  - list comprehension

Example: Ada lists

- Type declarations for integer-lists:

```
type IntNode;
type IntList is access IntNode;
type IntNode is record
      head: Integer;
      tail: IntList;
   end record;
```

mutually recursive

- An IntList construction:

```
new IntNode'(2,
    new IntNode'(3,
        new IntNode'(5,
            new IntNode'(7, null)))
```

Example: Java lists

- Class declarations for generic lists:

```
class List<E> {
  public E head;
  public List<E> tail;            recursive
  public List<E> (E el, List<E> t) {
     head = h;  tail = t;
  }
}
```

- A list construction:

```
List<Integer> list =
    new List<Integer>(2,
        new List<Integer>(3,
            new List<integer>(5, null))));
```

Example: Haskell lists

- Haskell has built-in list types:
  - [1, 2, 3]   integer list containing 1, 2, 3
  - [Int] : type of lists of integers. Similarly [Char], [[Int]], [(Int,Char)]
  - 2:[4, 5] == [2, 4, 5]   **cons** is ":"
  - **head** [1, 2, 3] = 1   **tail** [1, 2, 3] = [2, 3]
  - Strings are lists of characters: "foo" == ['f','o','o'] : [Char]
  - **range** [1..10] == [1,2,3,4,5,6,7,8,9,10]
  - **range with step** [3,6..20] == [3,6,9,12,15,18]
  - **range with step** [7,6..1] == [7,6,5,4,3,2,1]
  - **infinite list** [1..] == [1, 2, 3, …]
  - **List comprehension**   [x*y | x <- [2,5,10], y <- [8,10,11]]
    == [16,20,22,40,50,55,80,100,110]