Today's Topics

Previously

- Modelling variables, scopes and visibility using the Run Stack and Display with (LL,ON) addressing

Today

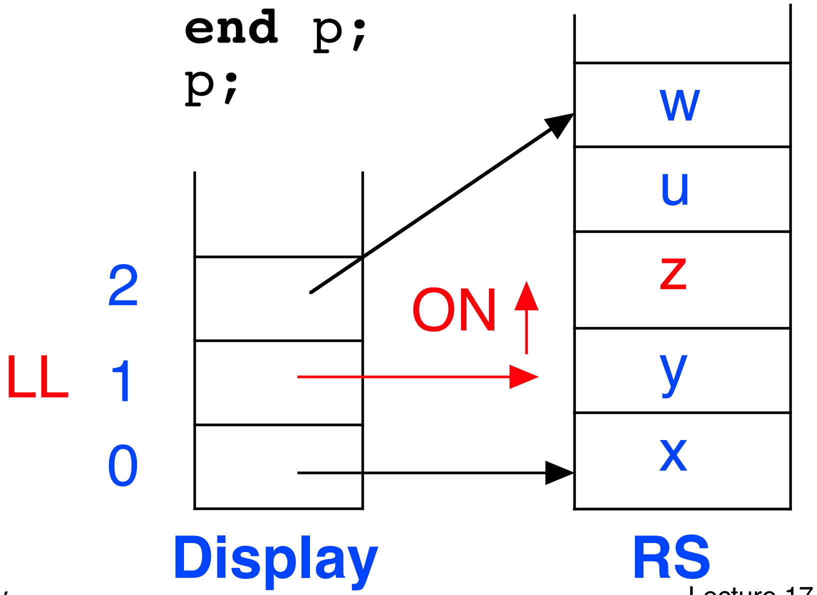
- Modelling storage layout of arrays and records (including classes)

Recall: (LL,ON) Address Calculation

- At run time, the address of a variable in the Run Stack is computed from its (LL,ON)
- RS [Display [LL] + ON] is the storage for variable (LL,ON)
- Display [LL] is the *base* of the storage for the scope, and ON is the *displacement* of the particular variable within that storage
- But what happens if a variable is non-scalar (e.g. array)?

  => Needs more than one RS slot!

```
0        var x: integer;
         procedure p;
   1        var y: integer; 1,0
                z: integer; 1,1
            procedure q;
      2        var w: integer;
              z := 1;
            end q;
            procedure r;
      2       var u : integer;
              q;
            end r;
            r;
         end p;
         p;
```



Storage Layout

- When variables are scalar (primitive, unstructured) values, they take only one element of the Run Stack (of course, in practice **char** and **integer** scalars may be different sizes, but both still fit in a "word")
- Non-scalar types such as *arrays* and *records* (structs, classes) will take more than one element of the stack
- And we get even more complexity due to the fact that elements of an array may be records, and elements of records may be arrays

```
var r:
   record
      size: integer
      data: array [1..10] of
            record
               x: integer
               y: integer
            end record
   end record
```

- Variables of these types require a *contiguous segment* of the RS

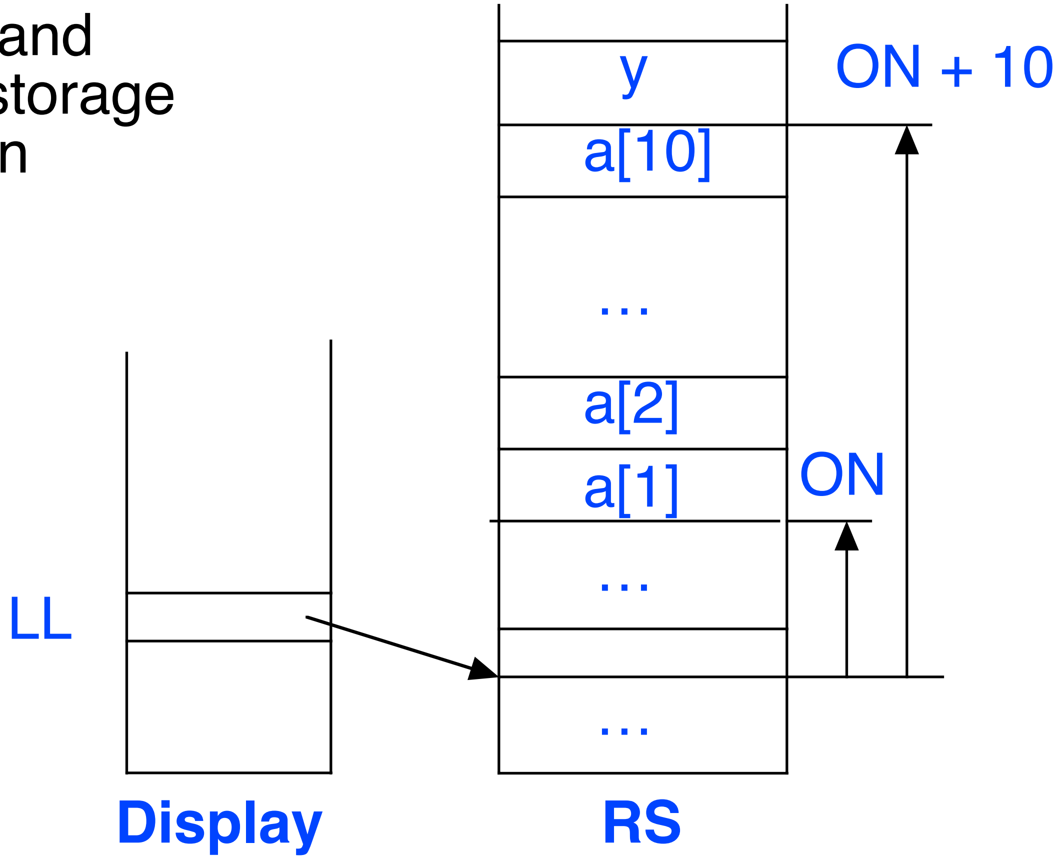
Storage Layout - Arrays of Scalars

- An array of scalar values is laid out as a sequence of *N* contiguous scalars in a row, where *N* is the number of elements in the array

```
var a: array [1..10] of integer;
var y: integer;
```

- If array *a* is at lexical level *LL* and order number *ON*, then its storage layout would look like this on the Run Stack

- The order number of the following variable *y* is shifted by the number of elements in the array



Storage Layout - Arrays of Scalars

- Given this representation, the (LL,ON) addresses of *y* and *a* are:

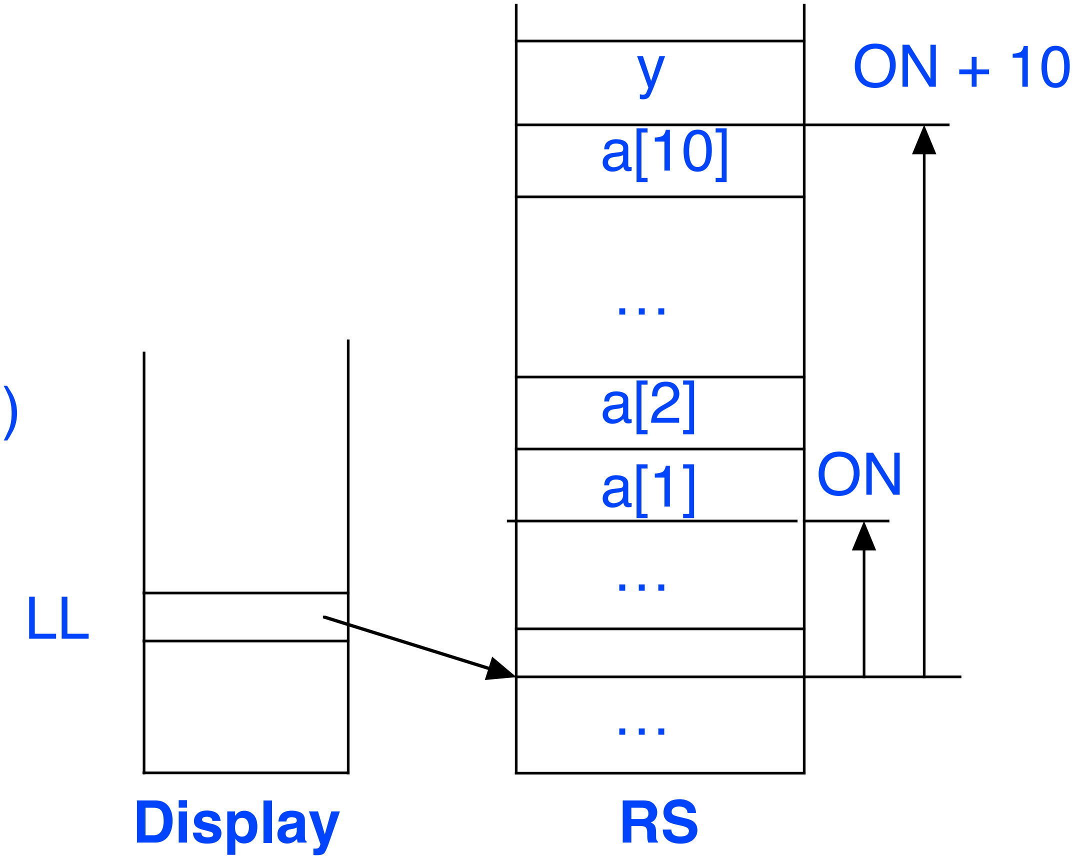
a	(LL,ON)
y	(LL,ON + 10)

- The (LL,ON) addresses of the elements of *a* are:

a[1]	(LL,ON)
a[2]	(LL,ON + 1)
a[3]	(LL,ON + 2)
. . .	
a[10]	(LL,ON + 9)

- Or in general:

a[i]	(LL,ON + i - 1)



Storage Layout - Arrays of Scalars

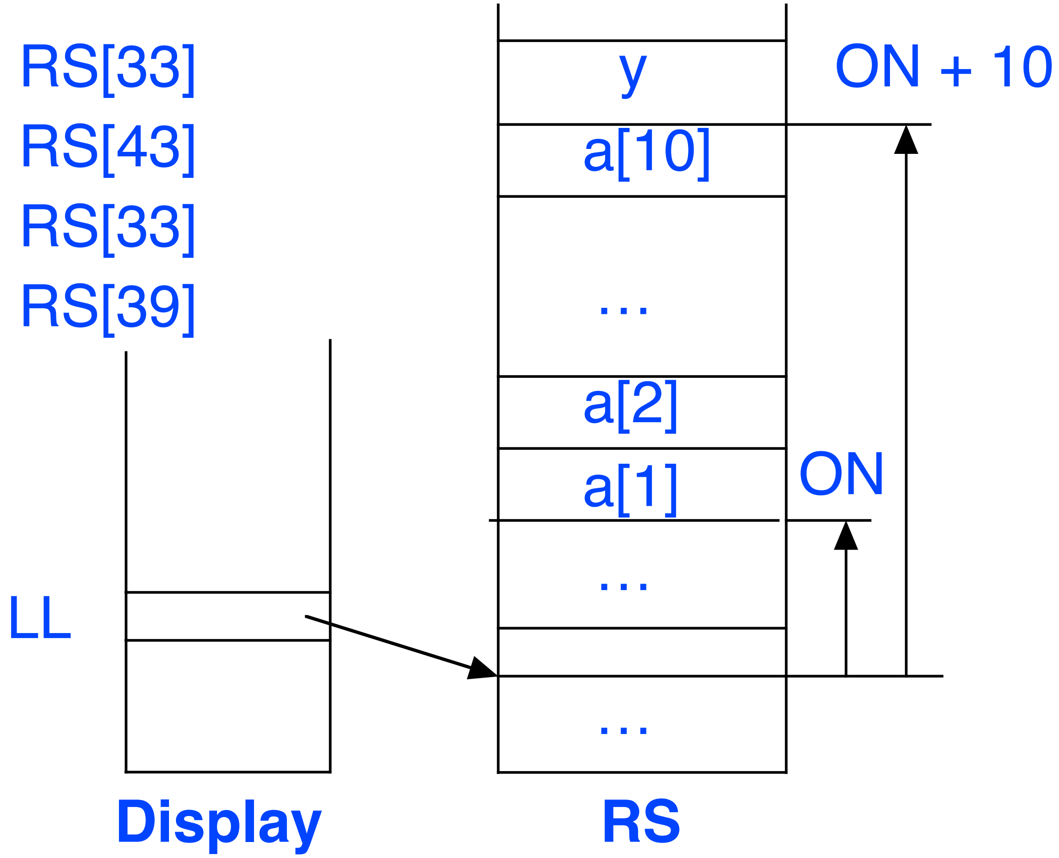
- For example, suppose LL = 1, Display[1] = 22, and ON = 11 then:

$$ON_y = ON_a + 10 = 21$$

$$ON_{a[i]} = ON_a + i - 1$$

$$ON_{a[3]} = ON_a + 3 - 1$$

a	RS[22 + 11]	RS[33]
y	RS[22 + 11 + 10]	RS[43]
a[1]	RS[22 + 11 + 1 - 1]	RS[33]
a[7]	RS[22 + 11 + 7 - 1]	RS[39]



Storage Layout - General 1D Arrays of Scalars

- The general form of a one dimensional array of scalars is:

```
var a: array [lower..upper] of integer;
```

- The *size* of this array (number of elements) is (upper - lower + 1)
  - this is the number of slots in the RS required for it

- Given: address(a) = (LL,ON)

  then: address(a[i]) = address(a) + i - lower
  = (LL, ON + i - lower)

- So a reference to element a[i] can be implemented in our abstract machine as:

```
pushaddress   (LLa,ONa)        i.e.,  pushaddress a
push          (LLi,ONi)        i.e.,  push i
push          lower            lower bound is a constant
subtract                       i - lower
add                            address(a) + i - lower
evaluate                       fetch value of element
```

Variable Addressing - New Instructions

- *evaluate* is a new abstract machine instruction that pops the top value *v* from the Expression Stack and pushes the value of the location in the Run Stack that it is the address of, i.e., *RunStack[v]*

```
push  x         pushaddress   x
                evaluate
```

- Once we have the *pushaddress* instruction, the old *pop* instruction that took a variable as operand is no longer needed - a new and more general instruction called *assign* is used instead

```
x := 1      push  1      pushaddress   x
            pop   x      push  1
                         assign
```

- *assign* pops the top two values from the ES, a *value* (the top element) and an *address* (second from top), and assigns the *value* to *RunStack[address]*

Variable Addressing - New Instructions

- *assign* and *evaluate* allow us to treat computed addresses of array elements and record fields as variables

```
a[i] := 1      pushaddress a     addr(a)
               push i
               push lower
               subtract          i - lower
               add               addr(a) + i - lower
               push 1
               assign
```

- The *assign* instruction is more general than the *pop* instruction so we remove the *pop* instruction from our model and use *pushaddress*, *evaluate* and *assign* from now on
- Note that *evaluate* and *assign* are also needed for reference parameters - otherwise we could not get or set their values

Multi-Dimensional Arrays of Scalars

- If an array has more than dimension (e.g., a matrix), then we can think of it as an *array* of *arrays*
- Example: the matrix

```
var a: array [1..5, 2..10] of integer;
```

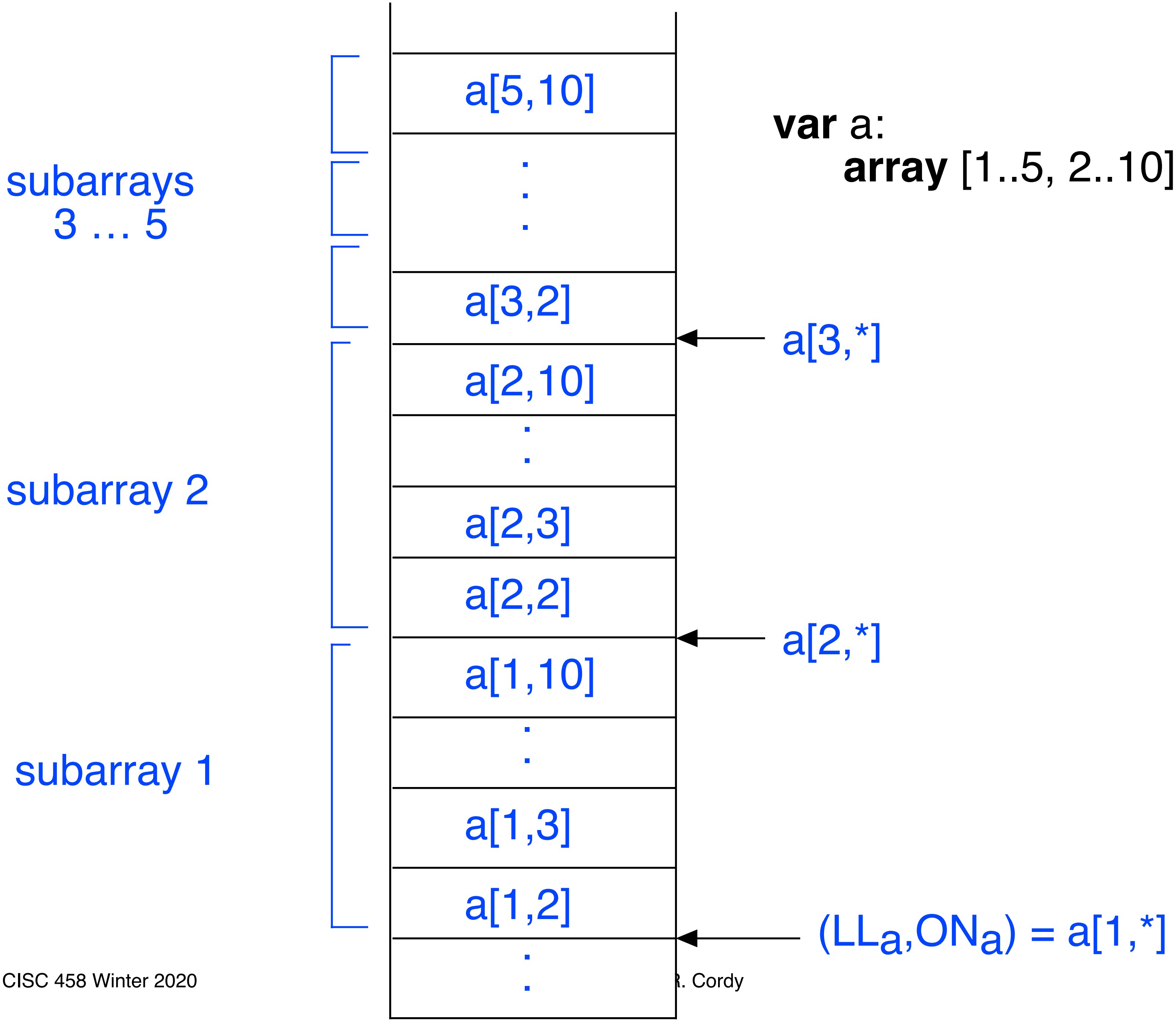
is equivalent (in memory storage) to

```
var a: array [1..5] of array [2..10] of integer;
```

- The element (sub)arrays are simply laid out one after the other in the RS

Example: 2-Dimensional Array (Matrix) of Scalars

var a:
array [1..5, 2..10] **of** integer;



a[5,10]

:

a[3,2]

a[2,10]

:

a[2,3]

a[2,2]

a[1,10]

:

a[1,3]

a[1,2]

:

subarrays
3 … 5

subarray 2

subarray 1

a[3,*]

a[2,*]

(LL_a, ON_a) = a[1,*]

Example: 2-Dimensional Array (Matrix) of Scalars

```
var a: array[1..5, 2..10] of integer;
```

- The address of element `a[i,j]` is calculated as

```
address(a[i,j]) = address(a[i,*]) + j - 2
                = address(a) + (i-1) * 9 + j - 2
```

where: 2 is the lower bound of each inner subarray
9 is the size of the inner subarray (elements 2 through 10)
1 is the lower bound of the outer array

- In general, for

$$\texttt{var a: array[} l_1 \texttt{ .. } u_1\texttt{, } l_2 \texttt{.. } u_2 \texttt{] of integer;}$$

$$\begin{aligned}\texttt{address(a[i,j])} &= \texttt{address(a[i,*])} + j - l_2 \\ &= \texttt{address(a)} + (i-l_1) * (u_2-l_2+1) \\ &\qquad + j - l_2\end{aligned}$$

- $(u_2 - l_2+1)$ is the size of the inner array (i.e. the size of each of the subarray elements of the outer array) - normally a constant, except when we have a dynamically sized array in languages that allow them

Multi-Dimensional Arrays of Scalars

- The completely general case of a multi-dimensional array is:

```
var a:
    array [l1..u1, l2..u2, …, ln .. un] of integer
```

- In this case:

$$\text{address}(a[i_1, i_2, \ldots i_n]) = \text{address}(a) + \sum_{k=1}^{n} (i_k - l_k) * s_k$$

where s_k is the size of the subarray elements of the subarray

$$a[i_1, \ldots, i_k, *, \ldots *]$$

- In Pascal and C this is a compile time constant defined as :

$$s_k = s_{k+1} * (u_{k+1} - l_{k+1} + 1)$$
$$s_n = 1$$

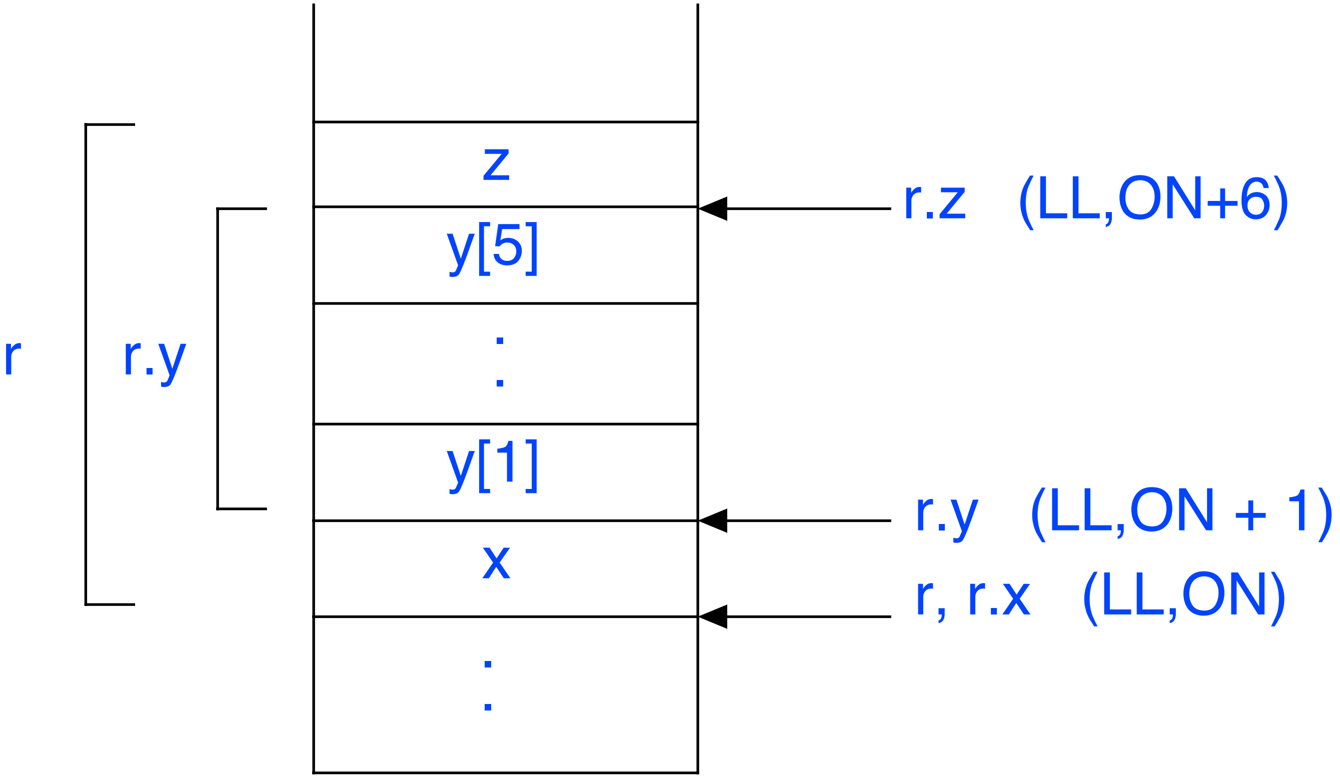
which can be calculated at compile time as:

$$s_k = \prod_{j=k}^{n-1} u_{j+1} - l_{j+1} + 1$$

Records

- Records (or structs in C) are similar to classes, but without methods
- Records are allocated in consecutive locations for the each field

```
var r:
    record
        x: integer;
        y: array [1..5] of integer;
        z: real;
    end record
```



Record Fields

- The address of a field is computed as an offset from the beginning of the record (i.e., the ON within the record scope)

```
address(r.z)  =  address(r) + offset(z)
              =  address(r) + 6
```

```
i := r.z    pushaddress (LLi,ONi)    address of i
            pushaddress (LLr,ONr)    address of r
            pushliteral 6            offset of z
            add                      address of r.z
            evaluate                 value of r.z
            assign
```

- Sometimes the address of the record is known at compile time, as in our example - other times it is not, for example,

  Arrays of records – record address depends on the subscript

  Dynamic records (objects) - record address depends on pointer

- When the address of the record *is* known, the *add* and *pushliteral* can be optimized out, to give: pushaddress (LL_r,ON_r+6)

Arrays as Record Fields

```
var r:
   record
      x: integer;
      y: array [1..5] of integer;
      z: real;
   end record

i := r.y[j]

address(r.y[j])  =  address(r.y) + j - 1
                 =  address(r) + offset(y) + j - 1
                 =  address(r) + 1 + j - 1
                 =  address(r) + j
```

Arrays as Record Fields

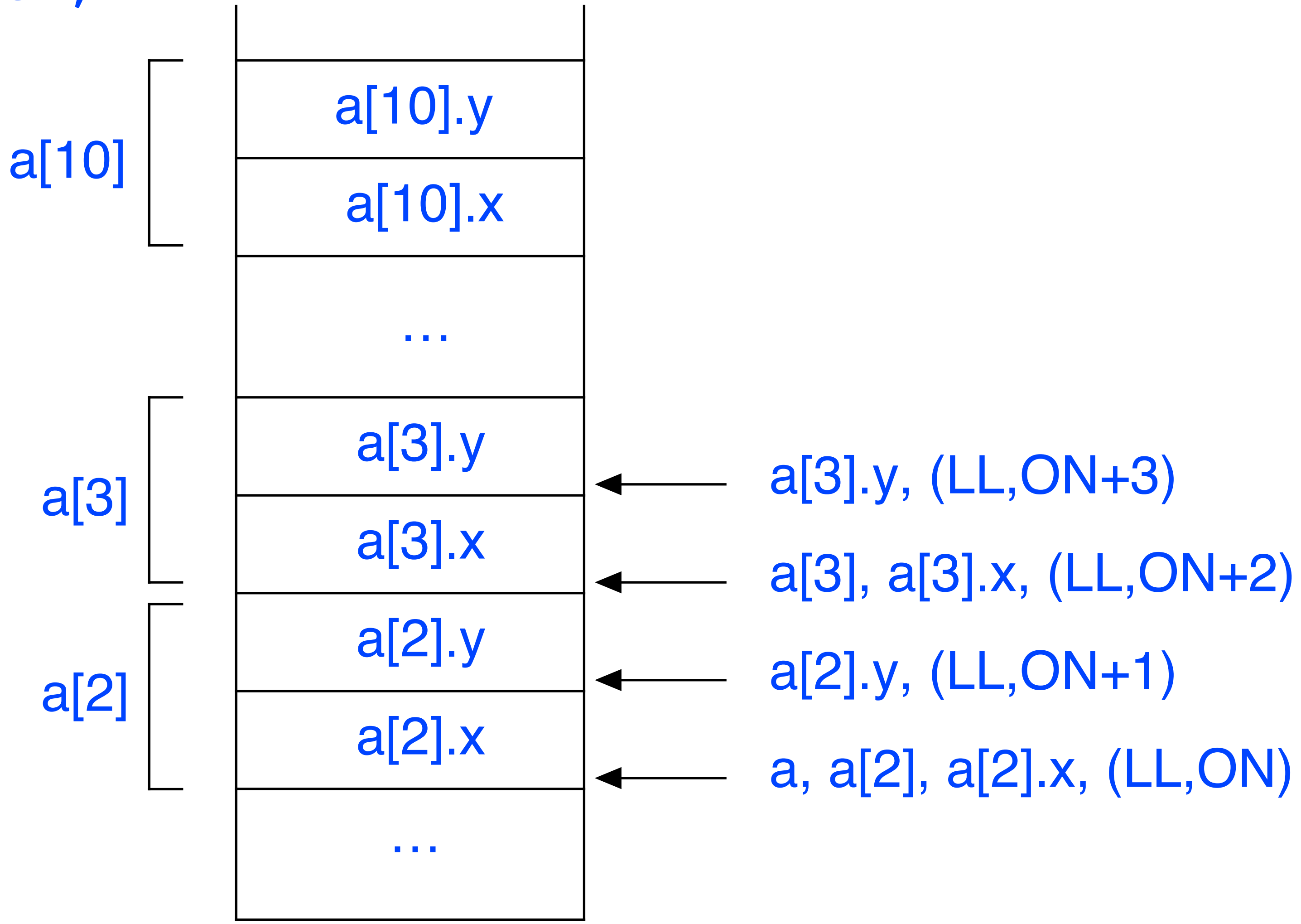
```
var r:
   record
      x: integer;
      y: array [1..5] of integer;
      z: real;
   end record

i := r.y[j]

pushaddress   i          address of i
pushaddress   r          address of r
pushliteral   1          offset of y in r
add                      address of r.y
pushaddress   j
evaluate                 value of j
pushliteral   1          lower bound of y
subtract                 i - lower
add                      address of r.y[j]
evaluate                 value of r.y[j]
assign                   i := r.y[j]
```

Arrays of Records

```
var a: array [2..10] of
   record
      x: integer;
      y: integer;
   end record
```



Arrays of Records

- Indexing of arrays of records must be *scaled* by the size of the records in the array

```
var a: array [2..10] of
   record
      x: integer;
      y: integer;
   end record


address(a[i].y)  = address(a[i]) + offset(y)
                 = address(a) + (i-2) * recsize
                    + offset(y)
                 = address(a) + (i - 2) * 2 + 1
```

Arrays of Records

- In general, the array subscripting formulas do not change, so the formula still holds:

$$\text{address}(a[i_1, i_2, \ldots i_n]) = \text{address}(a) + \sum_{k=1}^{n} (i_k - l_k) * s_k$$

except that

$$s_k = s_{k+1} * (u_{k+1} - l_{k+1} + 1)$$
$$s_n = \text{recsize}$$

(instead of 1)

or more compactly,

$$s_k = \left[\prod_{j=k}^{n-1} u_{j+1} - l_{j+1} + 1 \right] * \text{recsize}$$

Summary

Arrays and Records

- The storage model for records and arrays treats each like a consecutive sequence of variables for each of the elements or fields
- Elements are addressed by computing the sum of the *base* address of the record or array and the *offset* of the element or field within it
- Can often optimize by doing constant computations at compile time instead of using abstract machine instructions to compute them at run time

Next

- Begin Semantic Analysis