

The infinite recursion is eliminated by using explicit *pointers*:

```

type Expr = record
  case kind: ExpKind of
    Lit: (value: integer)
    Add: (op1,op2: ↑Expr)
    Sub: (op1,op2: ↑Expr)
  end
end

```

Now the record size is *finite*, because the size of ↑Expr is independent of the size of Expr.

A matching evaluation function:

```

function eval( exp : Expr ) : integer
begin
  case exp.kind of
    Lit: eval := exp.value;
    Add: eval := eval( exp.op1 ↑ ) + eval( exp.op2 ↑ );
    Sub: eval := eval( exp.op1 ↑ ) - eval( exp.op2 ↑ );
  end
end

```

The C example:

```

struct tree
  { int label; tree* left; tree* right; };

```

Allowing pointers to be null enables trees to be finite.

Sequence data types

Pascal Arrays

type constructor:

```

array ⟨index-type⟩ of ⟨element-type⟩
  where
  index-type
    ∈ (enumerations ∪ subranges ∪ { char }
  examples:

```

```

array [0..99] of char
array [ 'a'.. 'z' ] of integer
array (north, east, south, west) of real

```

☞ An array's index-type is part of the array's type. Hence

```

array [0..9] of real
array [1..10] of real
array [-3..3] of real

```

are *different types*, and cannot be used interchangeably as procedure arguments.

value construction:

1. allocate storage (*at compile time*)
2. assign values to individual elements

selector:

```

⟨array-name⟩ [ ' ⟨index-value⟩ ' ]

```

Indexing includes bounds checks (which can be turned off).

Modula-2 arrays

Modula-2's array types resemble Pascal's *except* for procedures' parameters.

Modula-2 allows *ordinary* array parameters

```

PROCEDURE P( VAR x: ARRAY [1..12] OF INTEGER );

```

In any call of P, the argument type is

```

ARRAY [1..12] OF INTEGER.

```

But Modula-2 also allows *open* array parameters:

```

PROCEDURE Q( VAR y : ARRAY OF INTEGER );

```

In any call of Q, the argument type is

```

ARRAY [i..j] OF INTEGER

```

where *i* and *j* can be *any integers*.

Within Q, y's index range is 0..HIGH(y), *regardless* of the corresponding argument's declaration.

```

VAR a: ARRAY [1..12] OF INTEGER ,
    ... b: ARRAY [10..99] OF INTEGER;

```

Q(a); (* within Q, y's index range is 0..11 *)

Q(b); (* within Q, y's index range is 0..89 *)

C/C++: Arrays

type constructor:

```

⟨element-type⟩ [ ' ' ]

```

example:

```

double[]

```

variable declaration:

```

⟨element-type⟩ ⟨name⟩ [ ' ⟨integer-constant⟩ ' ]

```

example:

```

double x[12]

```

Pascal equivalent

```

x : array [0..11] of real

```

☞ Array indexing begins at 0 (an array's size is its smallest invalid index).

☞ Array size is a **compile-time constant**.

value constructor:

```

double a[7] = { 4,7,1,9,5,2,5 }

```

selector: ⟨array-name⟩ [' ⟨index-value⟩ ']

☞ Indexing includes **no bounds check**.

☞ An array's size is *not* part of the array's type. Hence

```

double y[3];    double z[300];

```

are the same type and can be used interchangeably as procedure arguments.

A procedure, however, cannot discover an array argument's size (which must therefore be declared as a separate argument).

Java arrays

type constructor:
 $\langle \text{element-type} \rangle ' [' '] '$

example:
`double[]`

variable declaration:

$\langle \text{element-type} \rangle ' [' '] ' \langle \text{name} \rangle$

value constructor:

`new <element-type> ' [' <integer-expression> '] '`

example:

`double[] x = new double[size_of_x];`

☞ Array size is a **run-time value**.

selector: $\langle \text{array-name} \rangle ' [' \langle \text{index-value} \rangle '] '$

☞ Indexing includes **bounds check**
 (IndexOutOfBoundsException)

Haskell: **lists**

type constructor: `[]`

if `a` is a type, then `[a]` is the type *list of a*.

value constructors:

- list *enumeration*: `[3,4,2,1]`
- list *construction*:
 if `x::a` and `xs::[a]`, then `x:xs ::[a]`

value selectors:

- patterns
 if `xs` is a non-empty list then
`y:ys = xs`
 ★ binds `y` to the *first element* of `xs`
 ★ binds `ys` to the *rest* of `xs`.
- indexing: `xs!!n`
`> (!!) :: [t] -> Int -> t`
`> (b:bs)!!0 = b`
`> (b:bs)!!(n+1) = bs!!n`

Comparing arrays and lists:

	<i>size</i>	<i>access time</i>	<i>insert/delete time</i>
Arrays	fixed (-)	constant (+)	linear (-)
Lists	variable (+)	linear (-)	constant (+)

Array indexing

The key feature of arrays is $O(1)$ -time access to any array element.

Hence the time to compute the l -value of $A[k]$ must be independent of k .

Suppose A is declared by

$A : \mathbf{array} [low..high] \mathbf{of} T$

An array is implemented by a block of *contiguous* memory locations. For array A , let these locations' addresses be

$m_0, m_1, \dots, m_{(high-low+1)*size(T)-1}$

Then $A[k]$'s address is computed by

$m_0 + (k - low) * size(T)$

In a statically typed language, $size(T)$ is a compile-time constant; assuming that m_0 is available, the cost of computing $A[k]$ is clearly independent of k .

Computing $A[k]$'s address with *bounds check*:

if $low \leq k \leq high$
then $m_0 + (k - low) * size(T)$
else error

The check adds a constant to the access time.
 Is it worth the cost?

Strings

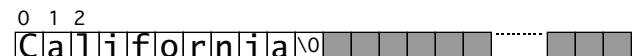
In Pascal, C, and C++, a (character) string is a sequence of characters stored in an array—but it is not the same as an array of characters.

A variable of type *string* needs to be able to store strings of varying lengths, but in these languages array variables' sizes are fixed at compile time.

Pascal solution: `str255` = array of 256 characters; first character is a one-byte integer specifying the string's length, i.e., how many array elements are valid.

0 1 2 255


C solution: `string` = char array of arbitrary length; string's last character is followed by `'\0'`.

0 1 2


C++ libraries (STL and others), Java, and Haskell provide much more convenient character-string types:

- In C++/STL and Java, a string's properties include its length, and no termination character is necessary
- in Haskell, a string is just a list of characters, manipulable by all polymorphic list functions—`length`, `(!!)`, `map`, `filter`, ...

What makes this possible is that in these languages variables' sizes are not fixed at compile time.