- Binding of Names / ID's (variables)
  - binding time
    - Variables (memory binding)
      - L values and R values
      - Aliasing: 2 variables w/ same L value
      - memory binding: • Static or dynamic
        (↓ before program runs, ↓ while program runs)
        - Static memory allocation
          - need to know mem. reqs. in advance
          - but it's fast
          - can't deallocate
        - Dynamic memory allocation
          - puts stuff in heap
          - slower, but more flexible
    - type binding
      - static type binding (static typing) → explicit typing (like Java) or type inference
      - dynamic type binding (dynamic typing) → typing delayed until program execution. variables allowed to change type on the fly, like in JavaScript and Perl

- x := 8
- y := x − 1
- z := 35 + y

L

'x'	3
'y'	0
'z'	n

RAM (R)

0	7
1	V_1
2	V_2
3	8
⋮	⋮
n	42
⋮	⋮

- x := 4
- y := &x
- *y := 42
- (x = 42)

& (ampersand) → aliasing

* 4 RAM sections: text, data, stack, heap

Type Binding

- Static typing
- Dynamic typing
- Use of lexical features to indicate typing

Scoping

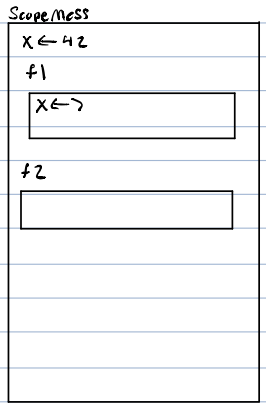
- Any feature used to indicate execution context: an assignment / binding of names to values

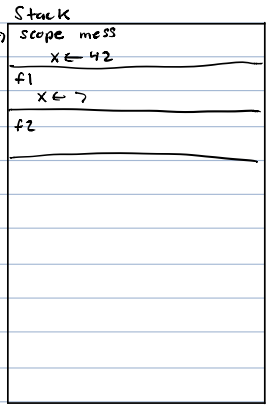
```
func addValue(x) {
    local val := 42
    return x + 42
}
```

- Static (lexical) scoping
  - Define if nested scopes are allowed (nested functions)
- Dynamic Scoping: hard and messy

```
begin prog ScopeMess:
  var x := 42

  func f1():
    var x := 7
    f2()
  end func          => Static scoping = prints 42 ->
  func f2():        => dynamic " = prints 7
    print x
  end func
  run:
    f1()
end prog
```





Globals

- Visible in all scopes
- Rules for shadowing

Syntactic features to support scopes

- Braces, indentation, or other delimiters to define blocks

Static: before program runs

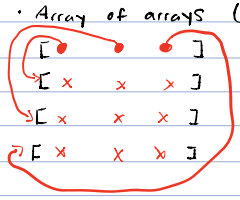
dynamic: while program runs

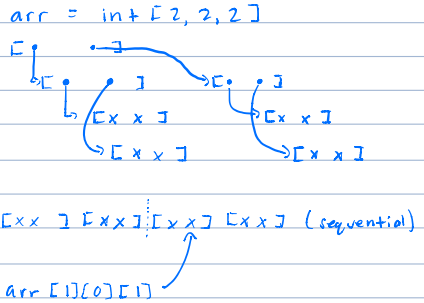
Data Types

- Data types are data classifications used to discriminate data, both from a structural and semantic point of view
- Inform programmer/system on how data is to be manipulated and what operations can be performed on it

- Primitive Data Types (Java has 8)
  - Integers, usually stored as a 32 bit binary # (Java int = 4 bytes = 32 bits)
  - Reals, usually represental/stored as a IEEE 32 bit floating point binary #
  - Characters, represented as either ASCII (1 byte) or Unicode (2 bytes) binary codes
  - Boolean, usually stored as a whole byte (using only one bit, because bits are not addressable)

- Arrays
  - Most basic data structure
  - Values stored sequentially and are indexed
  - Design Issues
    - Indexing
      - type of indices
      - Are they 0-indexed?
      - allocated statically or dynamically?
      - can store multiple types?
    - Multidimensional arrays
      - how are these implemented, if allowed?
        - Array of arrays (can implement jagged arrays)





        - Store sequentially (faster than arrays of arrays)

          arr → [x x x] [x xx] [x x x]

          arr[2][2]
    - Jagged Arrays
    - Range checking: leave it to programmer or build int array (Java)

Strings

Design Issues:

- Allocation: Dynamic vs. static
  - Immutable (Java): When you make a String, it cannot be changed later
  - Mutable: change string and string size later
  - Dynamic size (up to a bound)
  - Dynamic, unlimited size
  - Constant size

Associative Arrays

- Arrays with indices representing keys, like strings

  grades["Moustafa"] = "C"
  grades["Chris"] = "F"
  ⋮

  } syntactically simpler than hash tables

- Arrays implemented as Hash Tables

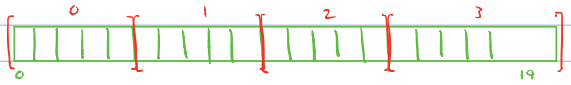
Compilers, interpreters, virtual machines

Sequential Multidimensional Array

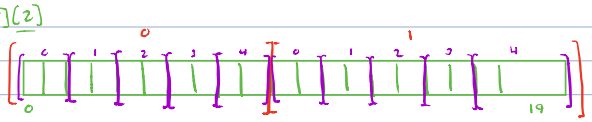
array[4][5]

↓

array[20]



array[2][5][2]



Pointers and Reference Types (Dynamic Allocation)

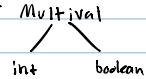
- lifetime
  - Default value ?
  - Deallocation
    - Dangling - very bad
    - Garbage Collection
      - Mark and Sweep
      - Generational Algorithms

Enumeration Types

- Type Fruit: Apple, Banana, Durian, Peach
  0 1 2 3 = internal values

Unions

- Multival




```
union Multival {
  int x
  bool b
}
Multival mv := 4
mv := false
```

List

- Functional languages use lists, not arrays
- append, head, tail
- Used mostly as a default Data Structure in functional languages
- Operations:
  - Append, prepend, concatenate, head, tail, sublist
  - Appending: constant-time operation

Records and Tuples

```
new record R:          R rec = new R
  int x          =>    rec.x = 4        => can perform destructive updates
  boolean b            rec.b = false
end record

tuple t = (0, "Hello", 42, "Forty Two")   (common in functional languages)
      ↳ int x String x int x String
t.0 => ?
t.2 == 42 ?
```

Type Compatibility and Equivalence

- Assignment Compatibility => A is assignment compatible to B if a value type B can be assigned to a variable of type A, without coercion or conversion.
- Coercion and conversion
- Structural Equivalence
- Name Equivalence

```
char c = 'a';
int x = 42 + c;
```
} Structural Equivalence (in C)

- coercion: truncating, conversion: syntactically typecasting

Expressions and Assignments

- Expressions: Any combination of values, operators, variables that yields a value
- Arithmetic Expressions
  - Unary vs. Binary vs Ternary operators
  - Binary:
    - +, plus
    - *, times
    - /, div, //
    - -, minus
    - **, ^ (Exponentiation)
    - %, mod
  - Unary: -, ~ (negative)
    ++ ⇒ side-effects
    --
  - Pure Expressions
    - An expression with no side-effects
  - Referential Transparency
    - replacing an output with the expression, w/o changing program behavior
  - Ternary Operators