CSE 425 / L-21 / 11.11.2024 /

Regular expression:

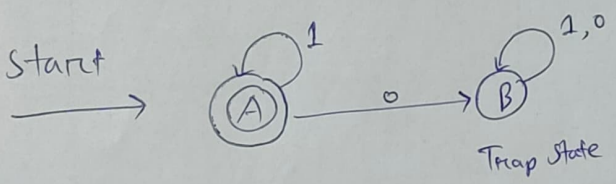
- Kneth Rosen
- CS143
- Python ⇒ regex / regexp

$\Sigma = \{0, 1\}$

$L = \{1^n \mid n = 0, 1, 2, 3, \ldots\}$

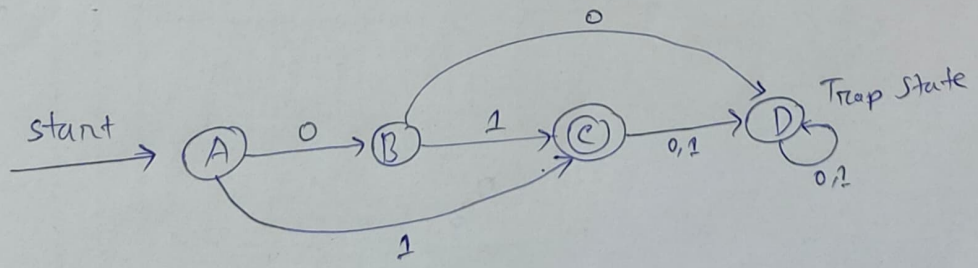
$1^0 = \epsilon / \lambda$

Draw the DFA:



M_0 = That recognize string $\{1, 01\}$ and discard the rest.

⇒

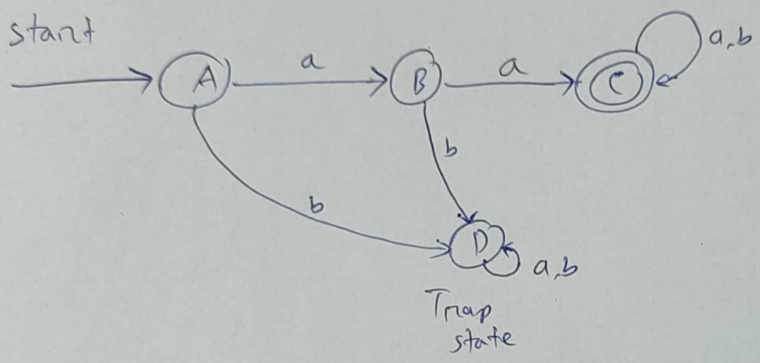


Construct a DFA that recognize a language M, which is the set of string contains two a's as the prefix.

$\Sigma = \{a, b\}$

Sample:
- aa
- aaa
- aab
- aabab
- ...

⇒



DFA: Deterministic Finite Automata

- for single input, the next state is precisely defined.
  - only one possible next state.

NFA: Non-deterministic Finite Automata

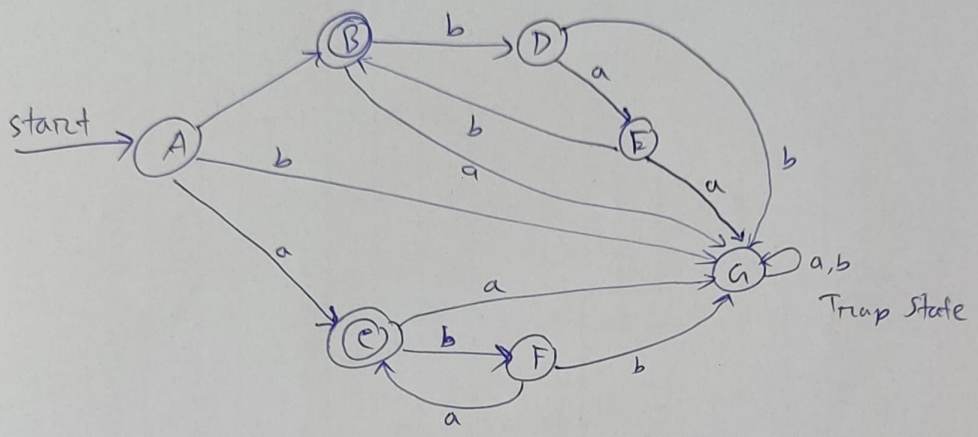
- allows more than one possible next state for a given input.
- ϵ-string / ϵ-transition.

$a(bab)^* \cup a(ba)^*$

$a(bab)^*$ → a, abab, ababbab...

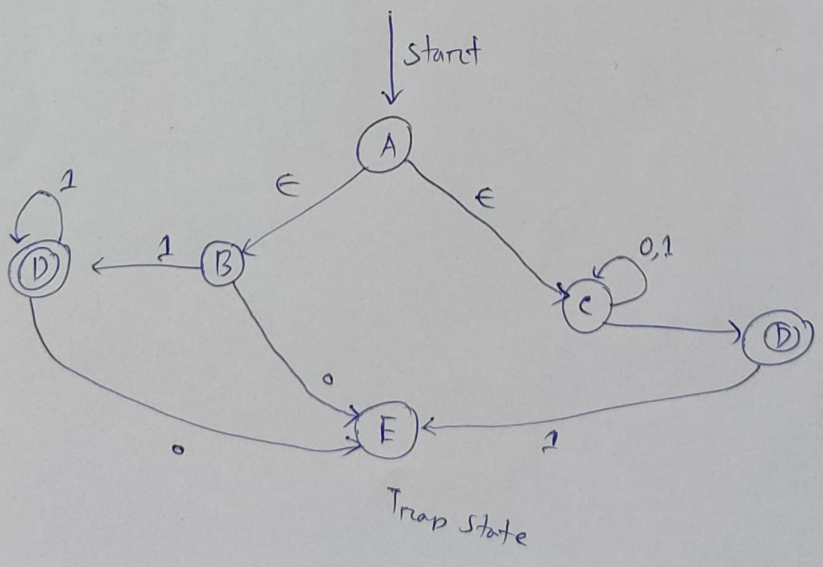
$a(ba)^*$ → a, aba, ababa, ...

⇒



- Practice more from provided handouts.

- ∈- transition: all binary strings where last symbol is "0" or it recognizes strings of 1's only.

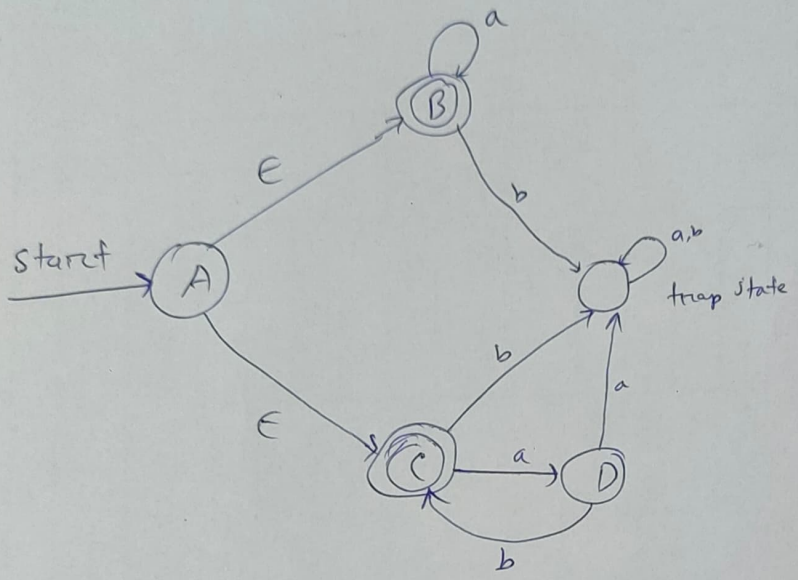


$L = \{ a^* + (ab)^* \}$

a^* → $\epsilon, a, aa \ldots$

$(ab)^*$ → $\epsilon, ab, abab \ldots$

$\Rightarrow$



Next class
name, scope, binding

L-22 / 13.11.2024

Names, Scopes, Binding

Von Neuman Architecture:

Separated
- Memory ⇒ variables ⇒ - abstraction to the physical location
  - has some attributes.
- CPU

- Name
- Address
- Value
- Type
- Lifetime
- Scopes

Name:

- Design issue: 'Length'
- Special Symbol to differentiate between variables
  - Scalar: $
  - Array: @
  - Hashes: %
- PL 'C': Camelhat notation

Address:

- aliasing: pointer, references variables.

Type:

variable → normal, static } Difference?

→ Holds the value once the scope is over and not reinitialize.

```
#include <stdio.h>
int main()
{
    printf("%d", test());
    printf("%d", test());
    return 0;
}
```

```
int test()
{
    static int count = 3;
    or int count = 3;
    count ++;
    return 0;
}
```

What will be the output in static & not static?

⇒

Static	Not - static
4	4
5	4

initialize only one time.

Binding:

- association between an entity and symbol.

↳ operation 'addition' ↳ '+'

- binding time:

(i) language design time binding
- operator symbols to operation.

(ii) Language implementation time
- Floating point type to a representation
  - 64 bit
  - 32 bit

(iii) Compile time binding
- bind a variable to memory cell.

(iv) Load time binding
- static variable to memory cell

(v) Run time:
- bind non-static variable to its type.

Static:

- binding occure before run time and remain unchanged.

Dynamic:

- occurs during execution and can change during the execution.

Syntax → Binding time

```
if (x>0)
  y=x;
```
→ Design time

keywords → Design time

Primitives data types
float, int, char → Design time

Representation of
3.44 → Implementation time

Specification of Type
of variables → compile time

Storage allocation
method for a variable → multiple → Design, Implementation, Compile

load executable in
memory → load time

Non-static allocation or
space to variables → Run time

Type binding:

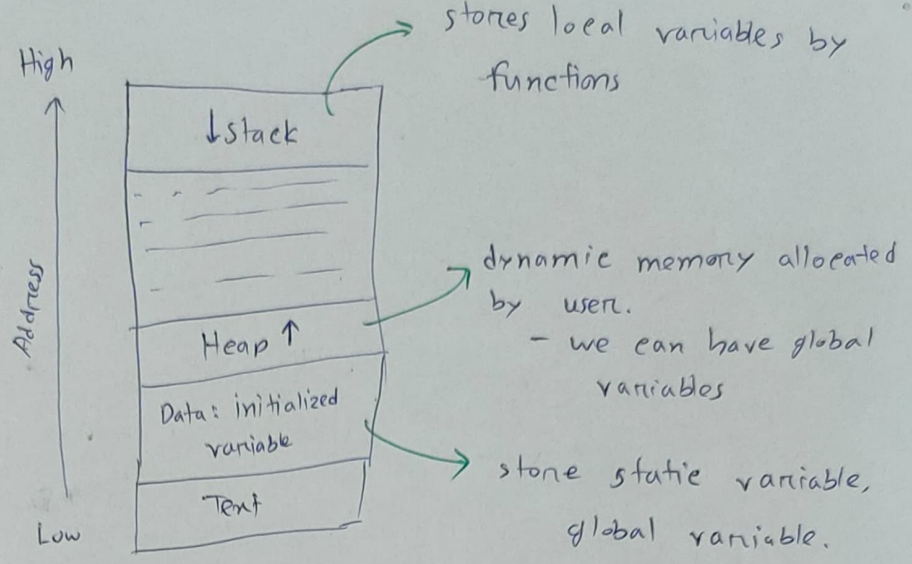
- Explicit ⇒ C: int x
- Implicit ⇒ First appearance of a variable name decides its type.

⇒
```
void main()
{
    int a;
    float b;
    char c;
    b = c;
}
```
→ char will be float for implicit.

Storage Binding:

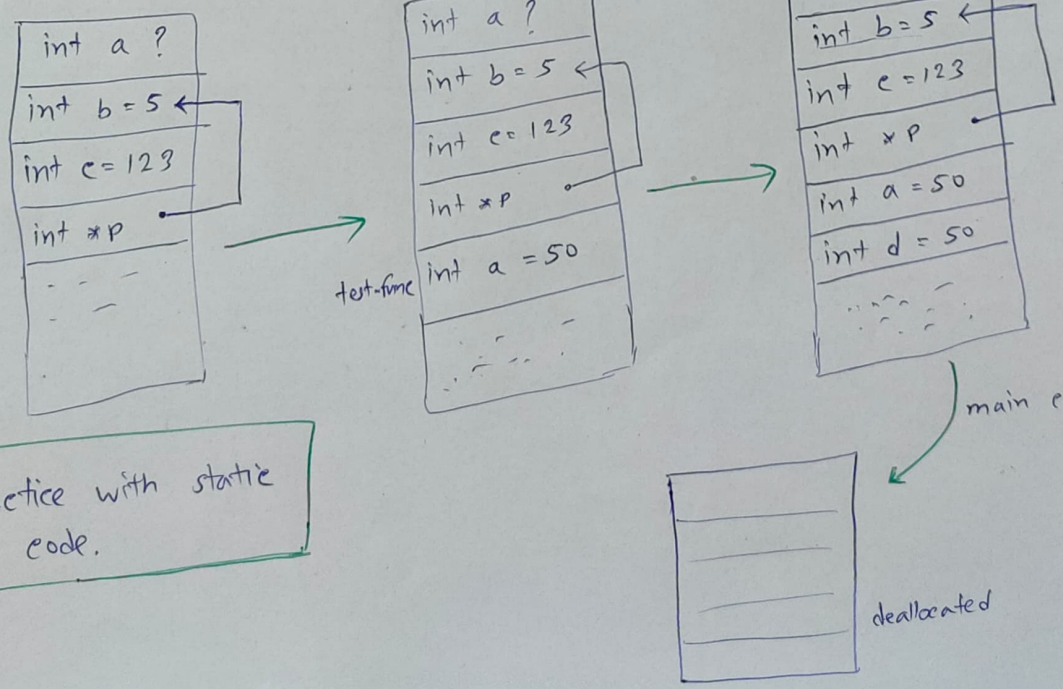
- allocation: assign memory cell from the available pool.
- Dealloeation: place the memory cell to the available pool upon completion/unbound from a variable.
- lifetime: Amount of time that a variable is alloeated to a cell.

Time(allocation) - Time(deallocation)




```
int main()
{
    int a;
    int b = 5;
    int c = 123;
    int *p = &b;
    int d = test-func();
    return 0;
}
```

```
int test-func()
{
    int a = 50;
    return a;
}
```

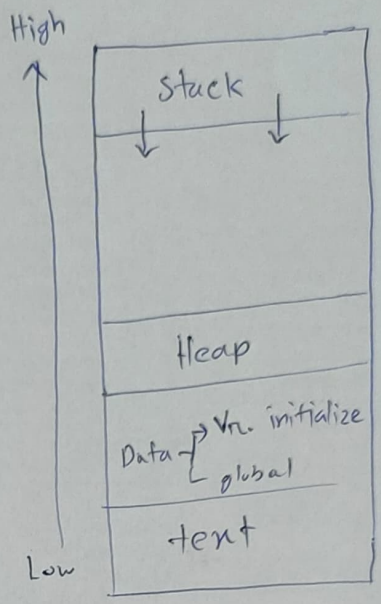


Practice with static code.

L-23 / 18.11.2024 /

Stack variables:

- allocation & deallocation are automatically done
- remain bound memory location and released upon completion of local function operation.



Advantage:

- very fast in access - direct addressing mode is used.
- less/no runtime overhead.

Disadvantage:

- if a language has static variable only, then it doesn't support recursive ~~sub-problem~~ sub-program.
- memory efficiency/usage is poor.
- for instance, static arrays are defined for two sub-problems and they are not running concurrently.

```
#include <stdio.h>

int global = 50;

int main()
{
    static int i = 100;
    return 0;
}
```

MinGW
- install & set the path variable.
- gcc .... -o ..... exe
- size ..... exe

} C/C++

* how to define the static variable.

* stack-dynamic: → binding done

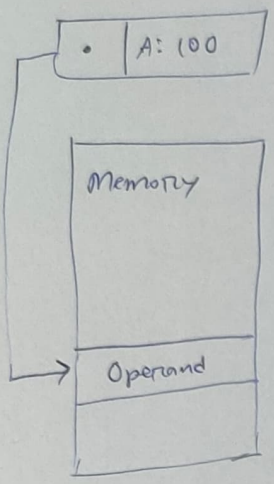
- storage is allocated when the declarations are elaborated.
  → that is the associated code is executed.

⇒ Example:
- local arrays of varing size.
  - here, type is statically board.

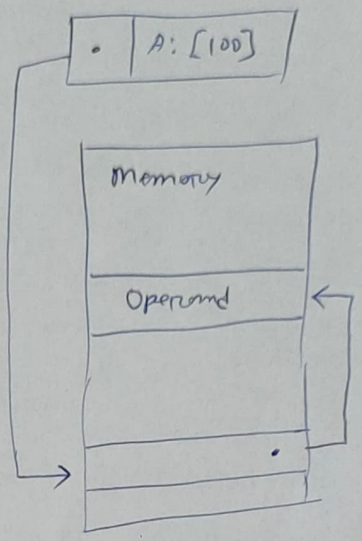
⇒ Advantage:
- Allows recursive sub-process
- conserve memory storage.
  → shared memory.

Direct / Indirect Addressing:

LOAD R1, 100



LOAD R1, [100]



Heap Memory:

- explicitly allocated by user/program
- deallocation to be done as well.
  explicitly by the programmer.

How to use or define?

=> C:

malloc : allocate memory

calloc : allocate with initialization

realloc : extend the memory allocation by reallocation.

free() : deallocation

C++:

new: allocation

delete: deallocation

Memory leak problem : (disadvantage)

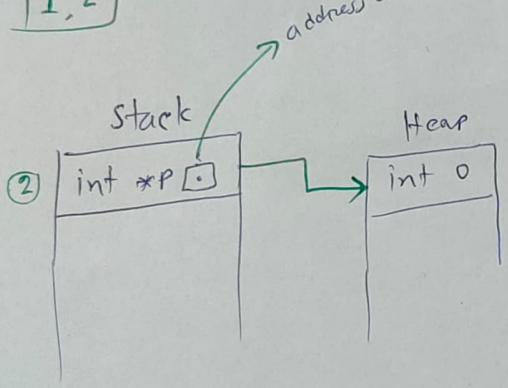
```
1. int main()
   {
2.    int *p = new int;
3.    cube *c1 = new cube();
4.    cube *c2 = c1;
5.    c2 -> set_length(10);
6.    return 0;
   }
```

Valgrid ⇒ for checking memory leakage

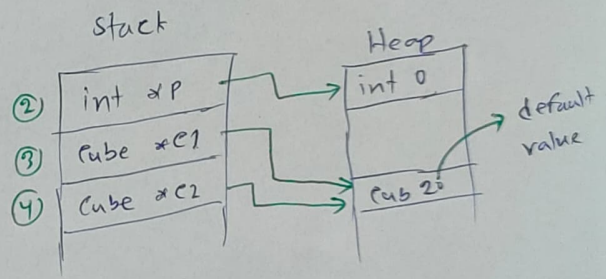
Madhu Sudhan Harvard University

BOAZ BORAK

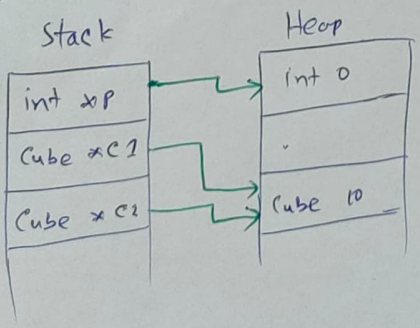
⇒ 1, 2



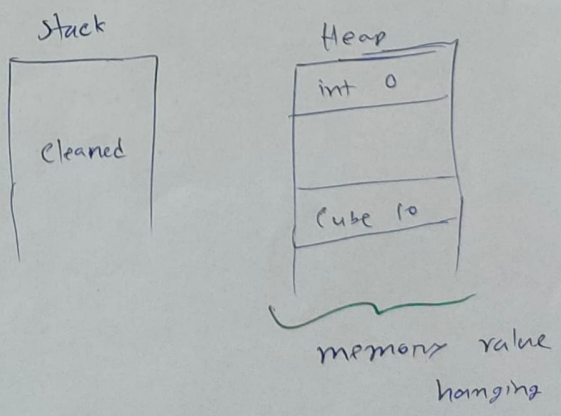
3,4



5



6



⇒ this problem known as memory leakage.

Explicit Heap-dynamic:

- allocation & deallocations
  - are explicitly specified by the user.
  - can only be accessed through pointers/ references.
- Examples:
  - 'new' command in C++.

Implicit Heap-dynamic:

- Bound to storage only when the values are assigned.
- Other attributes are also bound right after the value assignment.

```
array1 = [..char..]
array1 = [..int...]
```

- Advantage:
  - High degree of flexibility.
- Disadvantage:
  - Run time overhead, in order to maintain dynamic attributes.

Dangling Pointer:

- Pointer that contains the address of a heap dynamic variable which is already deallocated.

⇒ Examples:

```
{
    - - - - -
    - - - - -
    int * array-ptr1;
    int * array-ptr2 = new int[100]
    array-ptr1 = array-ptr2;
    delet (array-ptr2);
    - - -
    - - -
    - - -
    return 0;
}
```

Quiz-2
27.11.2024
Project Report
29.11.2024

L-24 / 20.11.2024 /

Data Types:

- Defines a collection of data values and some predefined operations on those.
- Could be primitive:
  - in built, user don't need to define. a new.

~~compiler~~

Primitive data types:

- char
- int
- string
- :
- etc
- complex [optional]
  - ↳ Python, Fortran
    - already available

Complex number in C?

<cmath.h>:

```
x = 10
y = 20
z = complex (10, 20)
  = x + iy
```

⇒

```
typedef struct complex{
    float real;
    float imag;
} complex;

int main()
{
    complex test_num;
    complex A, B;
    test_num.real = A.real + B.real
    test_num.imag = A.imag + B.imag
}
```

- How to define a complex number in C/python.

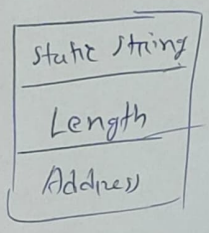
Descriptor:

- used for - Type checking, binding,
  - available on symbol table.
- collection of attributes
- area of the memory that stores the attributes of a variable.

⇒ **Static**

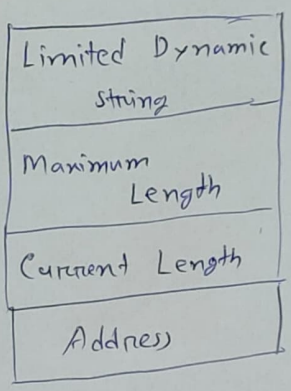
compile time

- when all the attributes are static.



Dynamic

- requires at the run-time



Python Practice
- Jupyter Lab