Abstract data types

- An ADT consists of a set of values and a set of operations
- A common example: integers
  - Values: 1, 2, 3, …
  - Operations: +, -, *, div, …
- In most of the popular uses of ADTs, the values and operations have no state
  - The values are constants
  - The operations have no internal memory (they don't remember anything in between calls)

A stack ADT

- We can implement a stack as an ADT:
  - Values: all possible stacks and elements
  - Operations: NewStack, Push, Pop, IsEmpty
- The operations take (zero or more) stacks and elements as input and return (zero or more) stacks and elements as output
  - S={NewStack}
  - S2={Push S X}
  - S2={Pop S X}
  - {IsEmpty S}
- For example:
  - S={Push {Push {NewStack} a} b} returns the stack S=[b a]
  - S2={Pop S X} returns the stack S2=[a] and the top X=b

Unencapsulated implementation

- The stack we saw before is almost an ADT:
  - **fun** {NewStack} nil **end**
  - **fun** {Push S X} X|S **end**
  - **fun** {Pop S X} X=S.1 S.2 **end**
  - **fun** {IsEmpty S} S==nil **end**
- Here the stack is represented by a list
- But this is not a data abstraction, since the list is not protected
- How can we protect the list, and make this a true ADT?
  - How can we build an abstract data type with encapsulation?
  - We need a way to protect values

Encapsulation using a secure wrapper

- To protect the values, we will use a secure wrapper:
  - The two functions Wrap and Unwrap will "wrap" and "unwrap" a value
  - W={Wrap X}   % Given X, returns a protected version W
  - X={Unwrap W}   % Given W, returns the original value X
- The simplest way to understand this is to consider that Wrap and Unwrap do encryption and decryption using a shared key that is only known by them
- We need a new Wrap/Unwrap pair for each ADT that we want to protect, so we use a procedure that creates them:
  - {NewWrapper Wrap Unwrap} creates the functions Wrap and Unwrap
  - Each call to NewWrapper creates a pair with a new shared key
- We will not explain here how to implement NewWrapper, but if you are curious you can look in the book (Section 3.7.5)

Implementing the stack ADT

- Now we can implement a true stack ADT:

```
local Wrap Unwrap in
    {NewWrapper Wrap Unwrap}

    fun {NewStack} {Wrap nil} end
    fun {Push W X} {Wrap X|{Unwrap W}} end
    fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
    fun {IsEmpty W} {Unwrap W}==nil end
end
```

- How does this work? Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result
- Wrap and Unwrap are hidden from the rest of the program (static scoping)

Final remarks on ADTs

- ADT languages have a long history
  - The language CLU, developed by Barbara Liskov and her students in 1974, is the first
  - This is only a little bit later than the first object-oriented language Simula 67 in 1967
  - Both CLU and Simula 67 strongly influenced later object-oriented languages up to the present day
- ADT languages support a protection concept similar to Wrap/Unwrap
  - CLU has syntactic support that makes the creation of ADTs very easy
- Many object-oriented languages also support ADTs
  - For example, we will see that Java objects are also ADTs