Exercises

- Adapt the preceding specification of stacks (LIFO, Last-In First-Out) to describe queues instead (FIFO).

- Adapt the preceding specification of stacks to account for bounded stacks, of maximum size capacity.
  - Hint: put becomes a partial function.

Expressed differently

```plaintext
value = item (remove (put (put (put (put (new, x8), x7), x6), x5), x4)), x2), x1))
```

Expression reduction

```plaintext
value = item { remove { put { put { put { put { new, x8}, x7}, x6, x5, x4 }, x3, x2, x1, x0, x9, x10 }, x11 } }
```

Stack 1

```plaintext
value = item { remove { put { put { put { put { new, x8}, x7, x6 }, x5, x4 }, x3, x2, x1, x0, x9, x10 }, x11 } }
```
Expression reduction

value = item {
    remove {
        put {
            put {
                remove {
                    put (put (new, x8), x7), x6
                        item {
                            remove {
                                put (put (new, x5), x4)
                                        put (new, x8)
                                    }, x1)
                                }, x2)
                            }, x1)
                        }, x2)
                    }, x1)
                }, x2)
            }, x1)
        }, x2)
    }, x1)
}
Expression reduction

value = item {
  remove {
    put {
      put {
        put {
          put {put (put (new, x8), x7), x6}
        }, item {
          remove {
            put (put (new, x5), x4)
        }
      }, x1
    }, x2
  }, x1
}
Expression reduction

```plaintext
value = item { remove { put { remove { put (put (put (new, x8), x7), x6) }, item { remove { put (put (new, x5), x4) }, x2 } }, x1 } }, x2
}
```

Expression reduction

```plaintext
value = item { remove { put { remove { put (put (put (new, x8), x7), x6) }, item { remove { put (put (new, x5), x4) }, x2 } }, x1 } }, x2
}
```

Expression reduction

```plaintext
value = item { remove { put { remove { put (put (put (new, x8), x7), x6) }, item { remove { put (put (new, x5), x4) }, x2 } }, x1 } }, x2
}
```

Expression reduction

```plaintext
value = item { remove { put { remove { put (put (put (new, x8), x7), x6) }, item { remove { put (put (new, x5), x4) }, x2 } }, x1 } }, x2
}
```

An operational view of the expression

```plaintext
value = item { remove { put { remove { put (put (put (new, x8), x7), x6) }, item { remove { put (put (new, x5), x4) }, x2 } }, x1 } }, x2
}
```

Sufficient completeness

- Three forms of functions in the specification of an ADT \( T \):
  - Creators: \( \text{OTHER} \rightarrow T \) e.g. `new`
  - Queries: \( T \times \ldots \rightarrow \text{OTHER} \) e.g. `item, empty`
  - Commands: \( T \times \ldots \rightarrow T \) e.g. `put, remove`

Sufficiently complete specification: a "Query Expression" of the form:

\[
f(\ldots)
\]

where \( f \) is a query, may be reduced through application of the axioms to a form not involving \( f \)
Stack: An Abstract Data Type

- **Types:**
  
  \[ \text{STACK} [G] \]
  
  → \( G \): Formal generic parameter

- **Functions (Operations):**
  
  - `put`: \( \text{STACK} [G] \times G \rightarrow \text{STACK} [G] \)
  
  - `remove`: \( \text{STACK} [G] \rightarrow \text{STACK} [G] \)
  
  - `item`: \( \text{STACK} [G] \rightarrow G \)
  
  - `empty`: \( \text{STACK} [G] \rightarrow \text{BOOLEAN} \)
  
  - `new`: \( \text{STACK} [G] \)

ADTs and software architecture

Abstract data types provide an ideal basis for modularizing software.

- Build each module as an implementation of an ADT:
  
  - Implements a set of objects with the same interface
  
  - Interface is defined by a set of operations (the ADT’s functions) constrained by abstract properties (its axioms and preconditions).

- The module consists of:
  
  - A representation for the ADT
  
  - An implementation for each of its operations
  
  - Possibly, auxiliary operations

Implementing an ADT

- **Three components:**
  
  - (E1) The ADT’s specification: functions, axioms, preconditions.
    
    (Example: stacks.)
  
  - (E2) Some representation choice.
    
    (Example: \( \langle \text{representation}, \text{count} \rangle \).)
  
  - (E3) A set of subprograms (routines) and attributes, each implementing one of the functions of the ADT specification (E1) in terms of the chosen representation (E2).
    
    (Example: routines `put`, `remove`, `item`, `empty`, `new`.)

A choice of stack representation

- **Capacity**

  - “Push” operation:
    
    \[ \text{count} := \text{count} + 1 \]
    
    \[ \text{representation}[\text{count}] := x \]

  - `capacity`
  
  - `representation`
  
  - `count`
  
  - `1`

Application to information hiding

- **Public part:**
  
  ADT specification (**E1**)

- **Secret part:**
  
  - Choice of representation (**E2**)
  
  - Implementation of functions by features (**E3**)

Object technology: A first definition

- Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — not on “the” function they achieve.
Object technology: More precise definition

- Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.

Classes: The fundamental structure

- Merging of the notions of module and type:
  - Module = Unit of decomposition: set of services
  - Type = Description of a set of run-time objects ("instances" of the type)

- The connection:
  - The services offered by the class, viewed as a module, are the operations available on the instances of the class, viewed as a type.

Class relations

- Two relations:
  - Client
  - Heir

Overall system structure

A very deferred class

```plaintext
def deferred class COUNTER
  feature
    item: INTEGER is
      deferred
      up la
        -- Increase item by 1.
down la
        -- Decrease item by 1.
  end
  deferred
    item = old item + 1
  ensure
    item > 0
  invariant
    item >= 0
  end
end
```

End of lecture 4