CMPT 280

Tutorial: Specifications

Stacks: A Refresher

Refresher

- A "last-in, first out" (LIFO) container
- Three fundamental operations:
  - Put new item on top of stack
  - Look at item at top of stack
  - Remove the item on the top of the stack.
- Stacks are also a type of *dispenser* (defined in Chapter 10 of the textbook).

Stack Specifications

What does a stack actually need?

Methods

- newStack
- Push
- Pop
- Top
- isEmpty
- isFull

Sets

- set of all stacks
- set of items that can be in a stack
- booleans: $\{\mathbf{true}, \mathbf{false}\}$

Stack Specifications

Name: Stack<G>

Sets:
S: set of all stacks containing elements from G
G: set of items that can be in the stack S
B: $\{\mathbf{true}, \mathbf{false}\}$

Signatures:
newStack<G>: $\to S$
S.isEmpty: $\to B$
S.isFull: $\to B$
S.push(g): $G \nrightarrow S$
S.pop: $\nrightarrow S$
S.top: $\nrightarrow G$

Preconditions: $\forall s \in S, g \in G$
newStack<G> : none
s.isEmpty : none
s.isFull : none
s.push : s is not full
s.pop : s is not empty
s.top : s is not empty

Semantics: $\forall s \in S, g \in G$
newStack<G> : Construct a new stack that can store elements of G
s.isEmpty: return $\mathbf{true}$ if s is empty, $\mathbf{false}$ otherwise
s.isFull: return $\mathbf{true}$ if s is full, $\mathbf{false}$ otherwise
s.push(g): push g onto top of stack s.pop: remove top element g from stack s.top: fetch top element g from stack

Stack Specifications

Name: Stack<G>

Sets:
S: set of all stacks containing elements from G
G: set of items that can be in the stack S
B: $\{\mathbf{true}, \mathbf{false}\}$

Signatures:
newStack<G>: $\to S$
S.isEmpty: $\to B$
S.isFull: $\to B$
S.push(g): $G \not\to S$
S.pop: $\not\to S$
S.top: $\not\to G$

Preconditions: $\forall s \in S, g \in G$
newStack<G> : none
s.isEmpty : none
s.isFull : none
s.push : s is not full
s.pop : s is not empty
s.top : s is not empty

Semantics: $\forall s \in S, g \in G$
newStack<G> : Construct a new stack that can store elements of G
s.isEmpty: return $\mathbf{true}$ if s is empty, $\mathbf{false}$ otherwise
s.isFull: return $\mathbf{true}$ if s is full, $\mathbf{false}$ otherwise
s.push(g): push g onto top of stack s.pop: remove top element g from stack s.top: fetch top element g from stack

Stack Specifications

Name: Stack<G>

Sets:
S: set of all stacks containing elements from G
G: set of items that can be in the stack S
B: $\{\mathbf{true}, \mathbf{false}\}$

Signatures:
newStack<G>: $\to S$
S.isEmpty: $\to B$
S.isFull: $\to B$
S.push(g): $G \not\to S$
S.pop: $\not\to S$
S.top: $\not\to G$

Preconditions: $\forall s \in S, g \in G$
newStack<G> : none
s.isEmpty : none
s.isFull : none
s.push : s is not full
s.pop : s is not empty
s.top : s is not empty

Semantics: $\forall s \in S, g \in G$
newStack<G> : Construct a new stack that can store elements of G
s.isEmpty: return **true** if s is empty, **false** otherwise
s.isFull: return **true** if s is full, **false** otherwise
s.push(g): push g onto top of stack s.pop: remove top element g from stack s.top: fetch top element g from stack

Stack Specifications

Name: Stack<G>

Sets:
S: set of all stacks containing elements from G
G: set of items that can be in the stack S
B: $\{\mathbf{true}, \mathbf{false}\}$

Signatures:
newStack<G>: $\rightarrow S$
S.isEmpty: $\rightarrow B$
S.isFull: $\rightarrow B$
S.push(g): $G \not\rightarrow S$
S.pop: $\not\rightarrow S$
S.top: $\not\rightarrow G$

Preconditions: $\forall s \in S, g \in G$
newStack<G> : none
s.isEmpty : none
s.isFull : none
s.push : s is not full
s.pop : s is not empty
s.top : s is not empty

Semantics: $\forall s \in S, g \in G$
newStack<G> : Construct a new stack that can store elements of G
s.isEmpty: return **true** if s is empty, **false** otherwise
s.isFull: return **true** if s is full, **false** otherwise
s.push(g): push g onto top of stack s.pop: remove top element g from stack s.top: fetch top element g from stack

Implementation

- Specifications done in this way can be translated into any language.
- We only happen to be using Java.

Implementation of Signatures

Signatures translate to method headers.

- $\text{newStack}{<}G{>} \rightarrow S$
- $S.\text{isEmpty} \rightarrow B$
- $S.\text{isFull} \rightarrow B$
- $S.\text{push}(g)\ G \nrightarrow S$
- $S.\text{pop} \nrightarrow S$
- $S.\text{top} \nrightarrow G$

$\Rightarrow$

```
public class Stack<G> {

    public Stack() {}

    public boolean isEmpty() {}

    public boolean isFull() {}

    public void push(G g) {}

    public void pop() {}

    public G top() {}

}
```

Implementation of Signatures

Signatures translate to method headers.

- newStack$<G> \to S$
- S.isEmpty $\to B$
- S.isFull $\to B$
- S.push(g) $G \nrightarrow S$
- S.pop $\nrightarrow S$
- S.top $\nrightarrow G$

$\Rightarrow$

```java
public class Stack<G> {

    public Stack() {}

    public boolean isEmpty() {}

    public boolean isFull() {}

    public void push(G g) {}

    public void pop() {}

    public G top() {}

}
```

Implementation of Preconditions

Preconditions are javadocs, exceptions, and if-statements.

- newStack<G> : none
- s.isEmpty : none
- s.isFull : none

- s.push : s is not full
- s.pop : s is not empty
- s.top : s is not empty

$\Rightarrow$

```
/** @precond the stack is not full */
public void push(G g)
  throws IllegalStateException {
    if (this.isFull()) {
        throw new
        IllegalStateException();
}

/** @precond the stack is not empty */
public void pop()
  throws IllegalStateException {
    if (this.isEmpty()) {
        throw new
        IllegalStateException();
}

/** @precond the stack is not empty */
public G top()
  throws IllegalStateException {
    if (this.isEmpty()) {
        throw new
        IllegalStateException();
}
```

Implementation of Preconditions

Preconditions are javadocs, exceptions, and if-statements.

- newStack$<G>$: none
- s.isEmpty : none
- s.isFull : none

- s.push : s is not full
- s.pop : s is not empty
- s.top : s is not empty

$\Rightarrow$

```java
/** @precond the stack is not full */
public void push(G g)
  throws IllegalStateException {
    if (this.isFull()) {
        throw new
        IllegalStateException();
}

/** @precond the stack is not empty */
public void pop()
  throws IllegalStateException {
    if (this.isEmpty()) {
        throw new
        IllegalStateException();
}

/** @precond the stack is not empty */
public G top()
  throws IllegalStateException {
    if (this.isEmpty()) {
        throw new
        IllegalStateException();
}
```

Implementation of Semantics

Semantics become the javadoc comments (and later, code).

- newStack$<G>$:
  Construct a new stack to hold $g \in G$
- s.isEmpty: return **true** if s is empty, **false** otherwise
- s.isFull: return **true** if s is full, **false** otherwise

$\Rightarrow$

```java
public class Stack<G> {

    /**
     * Create a new Stack
     */
    public Stack() {}

    /**
     * Tests whether the stack is empty
     * @returns true if the stack is empty,
     *    false otherwise
     */
    public boolean isEmpty() {}

    /**
     * Tests whether the stack is full
     * @returns true if the stack is full,
     *     false otherwise
     */
    public boolean isFull() {}

  [...]
```

Implementation of Semantics

Semantics become the javadoc comments (and later, code).

- newStack<G> : Construct a new stack to hold $g \in G$
- s.isEmpty: return $\mathbf{true}$ if s is empty, $\mathbf{false}$ otherwise
- s.isFull: return $\mathbf{true}$ if s is full, $\mathbf{false}$ otherwise

$\Rightarrow$

```java
public class Stack<G> {

    /**
     * Create a new Stack
     */
    public Stack() {}

    /**
     * Tests whether the stack is empty
     * @returns true if the stack is empty,
     *    false otherwise
     */
    public boolean isEmpty() {}

    /**
     * Tests whether the stack is full
     * @returns true if the stack is full,
     *     false otherwise
     */
    public boolean isFull() {}

  [...]
```

Implementation of Semantics

- s.push(g): push g onto top of stack
- s.pop: remove top element g from stack

$\Rightarrow$

```
[...]

  /**
   * Pushes element g onto
   *   the top of the stack
   * @precond the stack is not full
   */
  public void push(G g)
    throws IllegalStateException {
      if (this.isFull()) {
          throw new
          IllegalStateException();
  }

  /**
   * Removes the top element
   *   from the stack
   * @precond the stack is not empty
   */
  public void pop()
    throws IllegalStateException {
      if (this.isEmpty()) {
          throw new
          IllegalStateException();
  }

[...]
```

Implementation of Semantics

- s.push(g): push g onto top of stack
- s.pop: remove top element g from stack

$\Rightarrow$

```
[...]

  /**
   * Pushes element g onto
   *   the top of the stack
   * @precond the stack is not full
   */
  public void push(G g)
    throws IllegalStateException {
      if (this.isFull()) {
          throw new
          IllegalStateException();
  }

  /**
   * Removes the top element
   *   from the stack
   * @precond the stack is not empty
   */
  public void pop()
    throws IllegalStateException {
      if (this.isEmpty()) {
          throw new
          IllegalStateException();
  }

[...]
```

Implementation of Semantics

- s.top: fetch top element g from stack $\Rightarrow$

```
  [...]

    /**
     * Returns the top element
     *   from the stack
     * @precond the stack is not empty
     * @return the top element of the stack
     */
    public G top()
      throws IllegalStateException {
        if (this.isEmpty()) {
            throw new
            IllegalStateException();
    }


}
```

Implementation of Semantics

- s.top: fetch top element g from stack

$\Rightarrow$

```
  [...]

    /**
     * Returns the top element
     *   from the stack
     * @precond the stack is not empty
     * @return the top element of the stack
     */
    public G top()
      throws IllegalStateException {
        if (this.isEmpty()) {
            throw new
            IllegalStateException();
    }


}
```

Implementation - the rest

- Add the data structures and method implementations
- Don't forget to actually check your preconditions
- ADTs can be implemented in any language, using any implementation