Statement
A BL program consists of some statements …
Statement

- The **Statement** component family allows you to manipulate values that are ASTs for BL statements.
- The mathematical model of a **Statement** value is a tree of **STATEMENT_LABEL** with some constraints.
$S = \begin{cases} 
\text{CALL} \\
\text{WHILE} \\
\text{IF} \\
\text{IF\_ELSE}
\end{cases}$

$\text{BLOCK}$
A Statement variable’s value is a **tree of** `STATEMENT_LABEL` *with some constraints*, so we use □ rather than △ to illustrate its recursive structure.
The kind of statement (based on the root) determines how many and which kinds of children it may have.
The children of a **BLOCK** statement may not be **BLOCK** statements.
The child of an *IF* or *WHILE* statement *must* be a **BLOCK** statement.
The two children of an *IF_ELSE* statement *must* be *BLOCK* statements.
A **CALL** statement has no children.
Interfaces and Classes

```
Standard
       \---
      \   
Statement-
Kernel
       \---
      \   
Statement
       \---
      \   
Statement1
```
Interfaces and Classes

StatementKernel has contracts for 12 methods that involve “assembling” and “disassembling” the various BL statements (similar to Tree methods).

11 January 2015 OSU CSE
Interfaces and Classes

Statement extends Standard extends Statement-Kernel

Statement has these additional methods (not discussed here):

parse
parseBlock
prettyPrint

Statement implements Statement1
Enumerations

• Java has a special construct, `enum` (short for “enumeration”), that easily allows you to use meaningful symbolic names where you might otherwise be inclined to declare `int` variables and give them arbitrary values to participate just in equality tests.
Example: The `Kind` Enum

- The interface `StatementKernel` contains this code:

```cpp
/**
 * The kinds of statements.
 */

`enum` Kind {
    BLOCK, IF, IF_ELSE, WHILE, CALL
}
```
Enum

The interface StatementKernel contains this code:

```java
/**
 * The kinds of statements.
 */
enum Kind {
    BLOCK, IF, IF_ELSE, WHILE, CALL
}
```

This is quite different from:

```java
final int BLOCK = 1;
final int IF = 2;
...
```

You may not do arithmetic with the enum constants, where you could with the int constants above.
Example: The **Condition** Enum

/\**
* The possible conditions for IF, IF_ELSE, 
* and WHILE statements.
*/

```java
enum Condition {
    NEXT_IS_EMPTY, NEXT_IS_NOT_EMPTY,
    NEXT_IS_WALL, NEXT_IS_NOT_WALL,
    NEXT_IS_FRIEND, NEXT_IS_NOT_FRIEND,
    NEXT_IS_ENEMY, NEXT_IS_NOT_ENEMY,
    RANDOM, TRUE
}
```
Mathematical Model (1)

IDENTIFIER is string of character
exemplar id
constraint
  [id starts with a letter 'a'-'z', 'A'-'Z'] and
  [id contains only letters, digits '0'-'9', and '-'] and
  [id is not one of the keywords or conditions in the BL language]
Mathematical Model (2)

\[
\text{STATEMENT\_LABEL is (}
\text{kind: Kind,}
\text{test: Condition,}
\text{call: IDENTIFIER)}
\]

exemplar sl

constraint

[if sl.kind = BLOCK then sl.test and sl.call are irrelevant] \text{ and}
[if sl.kind = IF or sl.kind = IF\_ELSE or sl.kind = WHILE then sl.call is irrelevant] \text{ and}
[if sl.kind = CALL then sl.test is irrelevant]
Mathematical Model

The STATEMENT_LABEL is (kind: Kind, test: Condition, call: IDENTIFIER)

exemplar sl

constraint

[if sl.kind = BLOCK then sl.test and sl.call are irrelevant] and
[if sl.kind = IF or sl.kind = IF_ELSE or sl.kind = WHILE then sl.call is irrelevant] and
[if sl.kind = CALL then sl.test is irrelevant]
Mathematical Model (3)

\text{STATEMENT\_MODEL is tree of STATEMENT\_LABEL exemplar } s \\
\text{constraint} \\
| s | > 0 \quad \text{and} \\
[\text{BLOCK can have 0 or more children, but not another BLOCK as a child}] \quad \text{and} \\
[\text{IF must have exactly one BLOCK child}] \quad \text{and} \\
[\text{IF\_ELSE must have exactly two BLOCK children}] \quad \text{and} \\
[\text{WHILE must have exactly one BLOCK child}] \quad \text{and} \\
[\text{CALL must have no children (must be a leaf)}]
type StatementKernel is modeled by STATEMENTMODEL
No-argument Constructor

• Ensures:

\[ \textit{this} = \textit{compose}((\textit{BLOCK}, ?, ?), <>) \]
No-argument Constructor

• Ensures:

\[ this = compose((BLOCK, ?, ?), <>). \]

The use of ? here means we do not know—and, frankly, do not care about—the values of the 2\textsuperscript{nd} and 3\textsuperscript{rd} tuple components (\textit{test} and \textit{call}); the model says they are irrelevant if the 1\textsuperscript{st} tuple component (\textit{kind}) is \textit{BLOCK}. 
**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Statement s = new Statement1();</code></td>
<td></td>
</tr>
</tbody>
</table>
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Statement s = </code></td>
<td></td>
</tr>
<tr>
<td><code>new Statement1();</code></td>
<td></td>
</tr>
<tr>
<td>$s =$ BLOCK</td>
<td></td>
</tr>
</tbody>
</table>
kind

Statement::Kind kind()

• Reports the kind of statement this is.
• Ensures:

  \( \text{kind} = [\text{the statement kind of this}] \)
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind k = s.kind();</td>
<td>$s = \text{BLOCK}$</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind k = s.kind();</td>
<td><img src="image" alt="" /> $s =$ BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="" /> $s =$ BLOCK</td>
</tr>
<tr>
<td></td>
<td>$k =$ BLOCK</td>
</tr>
</tbody>
</table>
void addToBlock(int pos, Statement s)

- Adds the statement \( s \) at position \( pos \) in \texttt{this BLOCK} statement.
- Updates: \texttt{this}
- Clears: \( s \)
- Requires:
  \[
  \texttt{this} \text{ is a BLOCK statement} \quad \text{and} \quad \\
  \texttt{s} \text{ is not a BLOCK statement} \quad \text{and} \quad \\
  0 \leq pos \leq \text{[length of \texttt{this BLOCK}]}
  \]
- Ensures:
  \[
  \texttt{this} = \texttt{[#this with child \#s inserted at position pos]}
  \]
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="State Diagram" /></td>
</tr>
<tr>
<td><code>s.addToBlock(1, ns);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Diagram 1" /></td>
</tr>
<tr>
<td><code>s = ns =</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>s.addToBlock(1, ns);</code></td>
<td><img src="image" alt="Diagram 2" /></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
removeFromBlock
Statement removeFromBlock(int pos)

• Removes and returns the statement at position pos of this BLOCK statement.

• Updates: this

• Requires:
  
  [this is a BLOCK statement] and
  
  0 <= pos < [length of this BLOCK]

• Ensures:

  this = [#this with child at position pos removed and returned as result]
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Statement ns = s.removeFromBlock(1);</code></td>
<td><code>s = BLOCK</code></td>
</tr>
</tbody>
</table>

Diagram:
- A labeled node labeled `BLOCK` is connected to three smaller nodes, each representing a statement or block of code.
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
<tr>
<td><code>s = ns =</code></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
<tr>
<td><code>Statement ns = s.removeFromBlock(1);</code></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
</tbody>
</table>
lengthOfBlock

**int** lengthOfBlock()

• Reports the number of statements in **this BLOCK** statement.

• Requires:

  > [**this** is a BLOCK statement]

• Ensures:

  \[
  \text{lengthOfBlock} = \text{[the number of children of **this**]}
  \]
void assembleIf(Statement.Condition c, Statement s)

• Assembles in this a statement with root label (IF, c, ?) and only subtree the BLOCK s; the declaration notwithstanding, the dynamic type of s must be the same as the dynamic type of this.

• Replaces: this

• Clears: s

• Requires:
  [s is a BLOCK statement]

• Ensures:
  this = compose((IF, c, ?), <#s>)
## Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.assembleIf(RANDOM, ns);</code></td>
<td><code>s = ?</code>&lt;br&gt;<code>ns =</code></td>
</tr>
</tbody>
</table>

```java
s.assembleIf(RANDOM, ns);
```
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| ```
s = ?
s = ns =
``` | ![State Diagram 1](image1.png) |
| ```
s.assembleIf(RANDOM, ns);
``` | ![State Diagram 2](image2.png) |
disassembleIf

Statement.Condition disassembleIf(
Statement s)

• Disassembles \textit{IF} statement \texttt{this} into its test \texttt{Condition}, which is
returned as the value of the function, and its only subtree, the \texttt{BLOCK} statement \texttt{s}; the declaration notwithstanding, the \texttt{dynamic}
type of \texttt{s} must be the same as the \texttt{dynamic} type of \texttt{this}.

• Replaces: \texttt{s}

• Clears: \texttt{this}

• Requires:

  \texttt{[this is an IF statement]}

• Ensures:

  \texttt{#this = compose((IF, disassembleIf, ?), <s>)}
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$s = \text{disassembleIf}(ns)$</td>
<td>$s = \text{IF TRUE}$ $ns = ?$</td>
</tr>
</tbody>
</table>

Condition $c =$

$s = \text{disassembleIf}(ns);$
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s = \text{IF TRUE}$</td>
</tr>
<tr>
<td></td>
<td>$ns = \text{?}$</td>
</tr>
<tr>
<td>Condition $c =$</td>
<td>$s = \text{BLOCK}$</td>
</tr>
<tr>
<td>$s$.disassembleIf($ns$);</td>
<td>$ns = \phantom{\text{?}}$</td>
</tr>
<tr>
<td></td>
<td>$c = \text{TRUE}$</td>
</tr>
</tbody>
</table>
Other Methods

• See the Javadoc for Statement for details of the other methods:
  - assembleIfElse
  - disassembleIfElse
  - assembleWhile
  - disassembleWhile
  - assembleCall
  - disassembleCall
“Processing” a Statement

```java
if (s.kind() == BLOCK) { ... }
else if (s.kind() == IF) { ... }
else if (s.kind() == IF_ELSE) { ... }
else if (s.kind() == WHILE) { ... }
else if (s.kind() == CALL) { ... }
```
“Processing” a Statement

```java
if (s.kind() == BLOCK) { ... }
else if (s.kind() == IF) { ... }
else if (s.kind() == IF_ELSE) { ... }
else if (s.kind() == WHILE) { ... }
else if (s.kind() == CALL) { ... }
```

Technically, there is no reason you need to test this last condition; because what else could it be?
Java’s `switch` Construct

```java
switch (s.kind()) {
    case BLOCK: { ... ; break; }
    case IF: { ... ; break; }
    case IF_ELSE: { ... ; break; }
    case WHILE: { ... ; break; }
    case CALL: { ... ; break; }
    default: { ... ; break; }
}
```
Java’s `switch` Construct

```
switch (s.kind()) {
    case BLOCK: { ... ; break; }
    case IF: { ... ; break; }
    case IF_ELSE: { ... ; break; }
    case WHILE: { ... ; break; }
    case CALL: { ... ; break; }
    default: { ... ; }
}
```

The `switch` is recommended over a long string of `if-else-if-else-if-...`
Java’s `switch` Construct

```java
switch (s.kind()) {
    case BLOCK: { ... ; break; }
    case IF: { ... ; break; }
    case IF_ELSE: { ... ; break; }
    case WHILE: { ... ; break; }
    case CALL: { ... ; break; }
    default: { ... ; }
}
```

The `default` case is recommended even when technically it is not needed (as here).
Resources

• OSU CSE Components API: Statement
  – http://cse.osu.edu/software/common/doc/

• Big Java Late Objects, Section 3.3
  – http://proquest.safaribooksonline.com.proxy.lib.ohio-
    state.edu/book/programming/java/9781118087886/chapter-3-
    decisions/navpoint-27