Statement
BL Compiler Structure

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 =

CALL instruction

WHILE condition

IF condition

IF_ELSE condition

BLOCK

S =

CALL instruction

WHILE condition

IF condition

IF_ELSE condition

BLOCK
A Statement variable’s value is a tree of \texttt{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 extends Standard

Statement implements Statement-Kernel

Statement1
Interfaces and Classes

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

Statement has these additional methods (not discussed here):
parse
parseBlock
prettyPrint
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:

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

enum Kind {
    BLOCK, IF, IF_ELSE, WHILE, CALL
}
```
Example: The Kind 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.

```java
enum Kind {
    BLOCK, IF, IF_ELSE, WHILE, CALL
}
```
Example: The **Condition** Enum

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

class 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]
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 (2)

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)

STATEMENT_MODEL is tree of STATEMENT_LABEL exemplar s

constraint

|s| > 0 and

[BLOCK can have 0 or more children, but not another BLOCK as a child] and

[IF must have exactly one BLOCK child] and

[IF_ELSE must have exactly two BLOCK children] and

[WHILE must have exactly one BLOCK child] and

[CALL must have no children (must be a leaf)]
Mathematical Model (4)

type StatementKernel is modeled by

STATEMENT_MODEL
No-argument Constructor

• Ensures:

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

• Ensures:

\[
\textbf{this} = \text{compose}((\text{BLOCK, ?}, ?), <>)
\]

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><strong>Statement s =</strong></td>
<td><strong>$S = \text{BLOCK}$</strong></td>
</tr>
<tr>
<td><code>new Statement1();</code></td>
<td></td>
</tr>
</tbody>
</table>
kind

Statement.Kind kind()

• Reports the kind of statement this is.

• Ensures:

  kind = [the statement kind of this]
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind k = s.kind();</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td></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="image1" alt="State Diagram" /></td>
</tr>
<tr>
<td></td>
<td>$s = \text{BLOCK}$</td>
</tr>
<tr>
<td></td>
<td>$k = \text{BLOCK}$</td>
</tr>
</tbody>
</table>

The code snippet `Kind k = s.kind();` sets the `kind` attribute of `s` to `BLOCK`. The state diagram illustrates the initial state of `s` and its transition to the `BLOCK` state after the assignment.
void addToBlock(int pos, Statement s)

• Adds the statement \( s \) at position \( pos \) in \textit{this} \textit{BLOCK} statement.

• Updates: \textit{this}

• Clears: \( s \)

• Requires:

\[ \text{[this is a BLOCK statement]} \quad \text{and} \quad \text{[s is not a BLOCK statement]} \quad \text{and} \quad 0 \leq pos \leq \text{[length of this BLOCK]} \]

• Ensures:

\[ \text{this} = [#this \text{ with child } #s \text{ inserted at position } pos] \]
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.addToBlock(1, ns);</td>
<td>( s = )</td>
</tr>
</tbody>
</table>

**Example**
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$s = ns = \text{BLOCK}$</td>
<td>$s = \text{BLOCK}$</td>
</tr>
<tr>
<td>$s\text{.addToBlock}(1, ns);$</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:

\[ \text{this is a BLOCK statement} \quad \text{and} \quad 0 \leq pos < \text{[length of this BLOCK]} \]

• Ensures:

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

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

27 February 2019 OSU CSE
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| s = Block
| ns = s.removeFromBlock(1); |
| s = Block
| ns = |
lengthOfBlock

int lengthOfBlock()

• Reports the number of statements in this BLOCK statement.

• Requires:

  [this is a BLOCK statement]

• Ensures:

  lengthOfBlock =

  [the number of children of this]
assembleIf

```java
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>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(s = ?)</td>
</tr>
<tr>
<td>\texttt{s.assembleIf( RANDOM, ns);}</td>
<td>(n_s = )</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s = ?$</td>
</tr>
<tr>
<td></td>
<td>$ns =$</td>
</tr>
<tr>
<td>s.assembleIf(RANDOM, ns);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s =$</td>
</tr>
<tr>
<td></td>
<td>$ns =$</td>
</tr>
<tr>
<td></td>
<td>IF RANDOM</td>
</tr>
<tr>
<td></td>
<td>BLOCK</td>
</tr>
</tbody>
</table>
**disassembleIf**

Statement.Condition disassembleIf(
   Statement s)

- Disassembles *IF* statement *this* into its test *Condition*, which is returned as the value of the function, and its only subtree, the *BLOCK* statement *s*; the declaration notwithstanding, the *dynamic* type of *s* must be the same as the *dynamic* type of *this*.
- Replaces: *s*
- Clears: *this*
- Requires:
  
  
  *this is an IF statement*

- Ensures:

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

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = ns = ?</code></td>
<td><code>s = IF TRUE</code></td>
</tr>
<tr>
<td><code>Condition c = s.disassembleIf(ns);</code></td>
<td><code>ns = ?</code></td>
</tr>
</tbody>
</table>

**Code:**

```java
s = ns = ?
Condition c = s.disassembleIf(ns);
```
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Condition c = s.disassembleIf(ns);</code></td>
<td>![State Diagram]($s = \text{IF TRUE}$, $ns = \text{?}$)</td>
</tr>
<tr>
<td></td>
<td>![State Diagram]($s = \text{BLOCK}$, $ns = \text{?}$, $c = \text{TRUE}$)</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

```java
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, Section 5.3
  – http://osu.worldcat.org/title/big-java/oclc/754642794