Code Generation
BL Compiler Structure

Tokenizer → Parser → Code Generator

String of characters (source code) → String of tokens (“words”) → Abstract program → String of integers (object code)

The code generator is the last step.
Executing a BL Program

• There are two qualitatively different ways one might execute a BL program, given a value of type Program that has been constructed from BL source code:
  
  – *Interpret* the Program directly
  
  – *Compile* the Program into object code ("byte code") that is executed by a *virtual machine*
Executing a BL Program

• There are two qualitatively different ways one might execute a value of type `Program` constructed from its source code:
  – **Interpret** the `Program` directly
  – **Compile** the `Program` into *object code* ("byte code") that is executed by a *virtual machine*

This is what the BL compiler will actually do; and this is how Java itself works (recall the JVM and its “byte codes”).
Executing a BL Program

- There are two qualitatively different ways one might execute a program, given a value of type Program that has been constructed from its source code:
  - Interpret the Program directly
  - Compile the Program into object code ("byte code") that is executed by a virtual machine

Let’s first see how this might be done ...
Time Lines of Execution

- Directly interpreting a Program:
  - Tokenize
  - Parse
  - Execute by interpreting the Program directly

- Compiling and then executing a Program:
  - Tokenize
  - Parse
  - Generate code
  - Execute by interpreting generated code on VM
Time Lines of Execution

- **Directly interpreting a Program:**
  - Tokenize
  - Parse
  - Execute by interpreting the Program directly

- **Compiling and then executing a Program:**
  - Tokenize
  - Parse
  - Generate code
  - Execute by interpreting generated code on VM

At this point, you have a Program value to use.
## Time Lines

- **Directly interpreting a Program:**
  - **Tokenize**
  - **Parse**
  - **Execute by interpreting the Program directly**

- **Compiling and then executing a Program:**
  - **Tokenize**
  - **Parse**
  - **Generate code**
  - **Execute by interpreting generated code on VM**

"Execution-time" or "run-time" means **here**.
Interpreting a Program

• The structure of a Program and, within it, the recursive structure of a Statement, directly dictate how to execute a Program by interpretation

• Without contracts and other details, the following few slides indicate the structure of such code
executeProgram

```
public static void executeProgram(Program p) {
    Statement body = p.newBody();
    Map<String, Statement> context = p.newContext();
    p.swapBody(body);
    p.swapContext(context);
    executeStatement(body, context);
    p.swapBody(body);
    p.swapContext(context);
}
```
$s =$

[Diagram showing a flowchart with nodes labeled as follows:
- BLOCK
- IF
- WHILE
- IF_ELSE

Nodes include:
- CALL instruction
- WHILE condition
- IF condition
- IF_ELSE condition]
public static void executeStatement(Statement s, Map<String, Statement> context) {
    switch (s.kind()) {
    case BLOCK: {
        for (int i = 0; i < s.lengthOfBlock(); i++) {
            Statement ns = s.removeFromBlock(i);
            executeStatement(ns, context);
            s.addToBlock(i, ns);
        }
        break;
    }
    case IF: {...}
    ...
    }

It’s recursive just like everything else to do with Statement; context is needed for case CALL.
executeStatement

• Non-**BLOCK** cases are different in kind:
  – For **IF**, **IF_ELSE**, and **WHILE**, you need to decide whether the condition being tested as the BL program executes is *(now)* true or false
    • This requires a call to some method that knows the state of BugsWorld, i.e., what the bug “sees”
  – For **CALL**, you need to execute a primitive instruction, e.g., **MOVE**, **INFECT**, etc.
    • This requires a call to some method that updates the state of BugsWorld
The State of BugsWorld
For example, when executing this bug’s Program in this state, $next-is-empty$ is true.
Where Is The State of BugsWorld?

A client executes a particular bug’s program, and tells the server to execute primitive instructions.

The server knows about all the bugs, and can report to a client what a particular bug “sees”.

19 March 2019

OSU CSE
executeStatement

• Surprisingly, perhaps, executing a call to a user-defined instruction is straightforward:
  – You simply make a recursive call to `executeStatement` and pass it the body of that user-defined instruction, which is obtained from the context
Compiling a Program

• As noted earlier, we are instead going to compile a Program, and the last step for a BL compiler is to generate code.

• The structure of a program to do this is similar to that of an interpreter of a Program, except that it processes each Statement once rather than once every time it happens to be executed at run-time.
Code Generation

• **Code generation** is translating a **Program** to a **linear** (non-nested) structure, i.e., to a string of low-level instructions or **“byte codes”** of a BL **virtual machine** that can do the following:
  – Update the state of BugsWorld
  – “Jump around” in the string to execute the right **“byte codes”** under the right conditions, depending on the state of BugsWorld
Code Generation

- **Code generation** is translating a Program to a linear structure, i.e., to a string of low-level instructions or "byte codes" of a BL virtual machine that can do the following:
  - Update the state of BugsWorld
  - "Jump around" in the string to execute the right "byte codes" under the right conditions, depending on the state of BugsWorld

Primitive BL instructions are translated into these "byte codes".
Code Generation

- **Code generation** is translating a Program to a **linear** structure, i.e., to a string of low-level instructions or “byte codes” of a BL virtual machine that can do the following:
  - Update the state of BugsWorld
  - “Jump around” in the string to execute the right “byte codes” under the right conditions, depending on the state of BugsWorld.
**Example Statement**

IF next-is-wall THEN
  turnleft
ELSE
  move
END IF

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>JUMP_IF_NOT_NEXT_IS_WALL</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>TURNLEFT</td>
</tr>
<tr>
<td>3</td>
<td>JUMP</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>MOVE</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
</tr>
</tbody>
</table>
Example Statement

IF next-is-wall THEN
  turnleft
ELSE
  move
END IF

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (&quot;byte code&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
</tr>
</tbody>
</table>
BugsWorld Virtual Machine

• The *virtual machine* for BugsWorld has three main features:
  – Memory
  – Instruction set
  – Program counter
BugsWorld Virtual Machine

- The *virtual machine* for BugsWorld has three main features:
  - Memory
  - Instruction set
  - Program counter

A *string of integers* that contains the “byte codes” generated from a *Program*. 
• The **virtual machine** for BugsWorld has three main features:
  – Memory
  – *Instruction set*
  – Program counter

A *finite set of integers* that are the "byte codes" for the primitive instructions of the BugsWorld VM.
BugsWorld Virtual Machine

- The *virtual machine* for BugsWorld has three main features:
  - Memory
  - Instruction set
  - Program counter

Each instruction is given a **symbolic name** here, for ease of reading, but the VM knows only about integer “byte codes”.
BugsWorld Virtual Machine

• The *virtual machine* for BugsWorld has three main features:
  – Memory
  – Instruction set
  – *Program counter*

An *integer* that designates the location/position/address in memory of the “byte code” to be executed next.
BugsWorld Virtual Machine

• The **virtual machine** for BugsWorld has three main features:
  – Memory
  – Instruction set
  – *Program counter*

Normal execution increments the program counter by 1 or 2 after each instruction, so execution proceeds sequentially.
Instruction Set

• The *instruction set*, or *target language*, for code generation has two types of instructions:
  – Primitive instructions
  – Jump instructions
Instruction Set

• The *instruction set*, or *target language*, for code generation has two types of instructions:
  – Primitive instructions
  – Jump instructions

Each of these occupies one memory location.
Instruction Set

• The *instruction set*, or *target language*, for code generation has two types of instructions:
  – Primitive instructions
  – Jump instructions

Each of these occupies two memory locations: the second one is the location to jump to.
Primitive Instructions

- MOVE (0)
- TURNLEFT (1)
- TURNRIGHT (2)
- INFECT (3)
- SKIP (4)
- HALT (5)
Primitive Instructions

- MOVE (0)
- TURNLEFT (1)
- TURNRIGHT (2)
- INFECT (3)
- SKIP (4)
- HALT (5)

This is the “byte code” corresponding to the symbolic name for each instruction.
Primitive Instructions

- MOVE (0)
- TURNLEFT (1)
- TURNRIGHT (2)
- INFECT (3)
- SKIP (4)
- HALT (5)

This instruction halts program execution, and is the last instruction to be executed.
Jump Instructions

• JUMP (6)
• JUMP_IF_NOT_NEXT_IS_EMPTY (7)
• JUMP_IF_NOT_NEXT_IS_NOT_EMPTY (8)
• JUMP_IF_NOT_NEXT_IS_WALL (9)
• JUMP_IF_NOT_NEXT_IS_NOT_WALL (10)
• JUMP_IF_NOT_NEXT_IS_FRIEND (11)
• JUMP_IF_NOT_NEXT_IS_NOT_FRIEND (12)
• JUMP_IF_NOT_NEXT_IS_ENEMY (13)
• JUMP_IF_NOT_NEXT_IS_NOT_ENEMY (14)
• JUMP_IF_NOT_RANDOM (15)
• JUMP_IF_NOT_TRUE (16)
Jump Instructions

- **JUMP** (6)
- **JUMP_IF_NOT_NEXT_IS_EMPTY** (7)
- **JUMP_IF_NOT_NEXT_IS_NOT_EMPTY** (8)
- **JUMP_IF_NOT_NEXT_IS_WALL** (9)
- **JUMP_IF_NOT_NEXT_IS_NOT_WALL** (10)
- **JUMP_IF_NOT_NEXT_IS_FRIEND** (11)
- **JUMP_IF_NOT_NEXT_IS_NOT_FRIEND** (12)
- **JUMP_IF_NOT_NEXT_IS_ENEMY** (13)
- **JUMP_IF_NOT_NEXT_IS_NOT_ENEMY** (14)
- **JUMP_IF_NOT_RANDOM** (15)
- **JUMP_IF_NOT_TRUE** (16)

This *unconditional jump* instruction causes the program counter to be set to the value in the memory location following the **JUMP**.
Jump Instructions

- JUMP (6)
- JUMP_IF_NOT_NEXT_IS_EMPTY (7)
- JUMP_IF_NOT_NEXT_IS_NOT_EMPTY (8)
- JUMP_IF_NOT_NEXT_IS_WALL (9)
- JUMP_IF_NOT_NEXT_IS_FRIEND (10)
- JUMP_IF_NOT_NEXT_IS_NOT_FRIEND (11)
- JUMP_IF_NOT_NEXT_IS_ENEMY (12)
- JUMP_IF_NOT_NEXT_IS_NOT_ENEMY (13)
- JUMP_IF_NOT_RANDOM (15)
- JUMP_IF_NOT_TRUE (16)

This *conditional jump* instruction causes the program counter to be set to the value in the memory location following the instruction iff it is *not* the case that the cell in front of the bug is a wall.
Handling an **IF** Statement

**IF condition THEN**

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k)</td>
<td>JUMP_IF_NOT_condition</td>
</tr>
<tr>
<td>(k+1)</td>
<td>(k+n+2)</td>
</tr>
<tr>
<td>(k+2)</td>
<td>block (n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(k+n+1)</td>
<td></td>
</tr>
<tr>
<td>(k+n+2)</td>
<td>...</td>
</tr>
</tbody>
</table>

**END IF**
Handling an **IFELSE** Statement

**IF** condition **THEN**

*block1*

**ELSE**

*block2*

**END IF**

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>JUMP_IF_NOT_condition</td>
</tr>
<tr>
<td>k+1</td>
<td>k+n1+4</td>
</tr>
<tr>
<td>k+2</td>
<td>block1 (n1 instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n1+2</td>
<td>JUMP</td>
</tr>
<tr>
<td>k+n1+3</td>
<td>k+n1+n2+4</td>
</tr>
<tr>
<td>k+n1+4</td>
<td>block2 (n2 instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n1+n2+4</td>
<td>...</td>
</tr>
</tbody>
</table>
Handling a **WHILE** Statement

**WHILE** condition **DO**

*block*

**END WHILE**

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>JUMP_IF_NOT_condition</td>
</tr>
<tr>
<td>k+1</td>
<td>k+n+4</td>
</tr>
<tr>
<td>k+2</td>
<td>block (n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n+2</td>
<td>JUMP</td>
</tr>
<tr>
<td>k+n+3</td>
<td>k</td>
</tr>
<tr>
<td>k+n+4</td>
<td>...</td>
</tr>
</tbody>
</table>
Handling a \textit{CALL} Statement

\begin{itemize}
  \item \textit{move}
  \begin{table}[h]
    \begin{tabular}{|c|c|}
      \hline
      Loc & Instruction (symbolic name) \\
      \hline
      k   & MOVE     \\
      \hline
    \end{tabular}
  \end{table}

  \item \textit{turnleft}
  \begin{table}[h]
    \begin{tabular}{|c|c|}
      \hline
      Loc & Instruction (symbolic name) \\
      \hline
      k   & TURNLEFT \\
      \hline
    \end{tabular}
  \end{table}

  \item (etc.)
\end{itemize}
### Handling a **CALL** Statement

**INSTRUCTION**

```
my-instruction IS
  block
END my-instruction
```

### Table

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>block (of n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n-1</td>
<td></td>
</tr>
<tr>
<td>k+n</td>
<td>...</td>
</tr>
</tbody>
</table>

**my-instruction**
Handling a **CALL** Statement

```
INSTRUCTION
  my-instruction IS block
END my-instruction
```

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>block (of n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n-1</td>
<td></td>
</tr>
<tr>
<td>k+n</td>
<td>...</td>
</tr>
</tbody>
</table>

A call to *my-instruction* generates a block of “byte codes” for the body of *my-instruction*. 
Handling a **CALL Statement**

```
INSTRUCTION
  my-instruction IS block
END my-instruction

my-instruction
```

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>block (of n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n-1</td>
<td></td>
</tr>
<tr>
<td>k+n</td>
<td>...</td>
</tr>
</tbody>
</table>

This way of generating code for a call to a user-defined instruction is called **in-lining**.
Handling a **CALL** Statement

**INSTRUCTION**

```plaintext
call
my-instruction IS block
END my-instruction
```

What would happen with in-lining if BL allowed recursion? How else might you handle calls to user-defined instructions?

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction (symbolic name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>block (of n instructions)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k+n-1</td>
<td></td>
</tr>
<tr>
<td>k+n</td>
<td>...</td>
</tr>
</tbody>
</table>

19 March 2019

OSU CSE
Handling a **BLOCK Statement**

- The “byte codes” generated for individual Statements in a block (a sequence of Statements) are placed sequentially, one after the other, in memory.
- Remember: at the end of the body block of the Program, there must be a **HALT** instruction.
Aside: More On Java \texttt{enum}

• Recall: the Java \texttt{enum} construct allows you to give meaningful symbolic names to values for which you might instead have used \textit{arbitrary} \texttt{int} constants

• This construct has some other valuable features that allow you to associate symbolic names (e.g., for VM instructions) with \textit{specific} \texttt{int} constants (e.g., their “byte codes”)

19 March 2019

OSU CSE
The Instruction Enum

- The interface Program contains this code:

```java
/**
 * BugsWorld VM instructions and "byte codes".
 */
enum Instruction {
    MOVE(0), TURNLEFT(1), ...
;

    ...
}
```

- An instance variable, a constructor, and an accessor method ...
The **Instruction** Enum

- The interface **Program** contains this code:

```java
enum Instruction {
    MOVE(0), TURNLEFT(1), ...;

    private int blByteCode;

    private Instruction(int code) {
        this.blByteCode = code;
    }

    public int byteCode() {
        return this.blByteCode;
    }
}
```
Every Instruction (e.g., MOVE) has an int instance variable called blByteCode.
The Instruction

• The interface Program contains this code:

```java
enum Instruction {
    MOVE(0), TURNLEFT(...);

    private int blByteCode;

    private Instruction(int code) {
        this.blByteCode = code;
    }

    public int byteCode() {
        return this.blByteCode;
    }
}
```

This constructor makes each Instruction’s “argument” (in parens) the value of its associated blByteCode.
The **Instruction**

- The interface `Program` contains this code:

```java
enum Instruction {
    MOVE(0), TURNLEFT(1), ...
};

private int blByteCode;

private Instruction(int code) {
    this.blByteCode = code;
}

public int byteCode() {
    return this.blByteCode;
}
```

This access method (an instance method) allows a client to access an Instruction's associated `blByteCode`. 
Using This Feature

• In client code using `Instruction`, one might write something like this:

```
Instruction i = Instruction.TURNLEFT;
...
int code = i.byteCode();
```

or even:

```
... Instruction.TURNLEFT.byteCode() ...
```

• The “byte code” thus obtained is what needs to be put into the generated code
Resources

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

• Java Tutorials: Enum Types
  – http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html