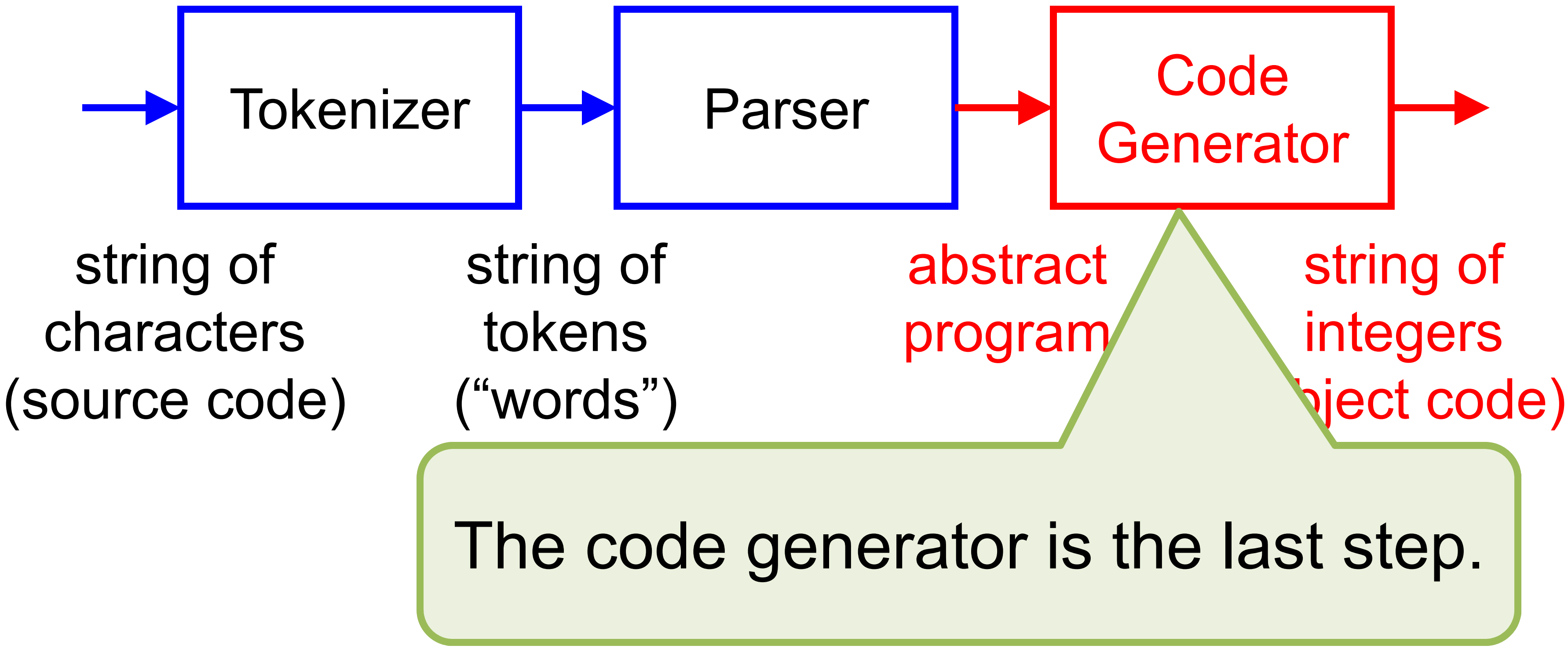
Code Generation

BL Compiler Structure



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

- There are two qua
one might ***execu***
value of type `P`
constructed f om 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 …

> Let's first see how this might be done ...

- There are two qua… 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***

Time Lines of Execution

- Directly interpreting a `Program`:

Tokenize	Parse	Execute by interpreting the `Program` directly

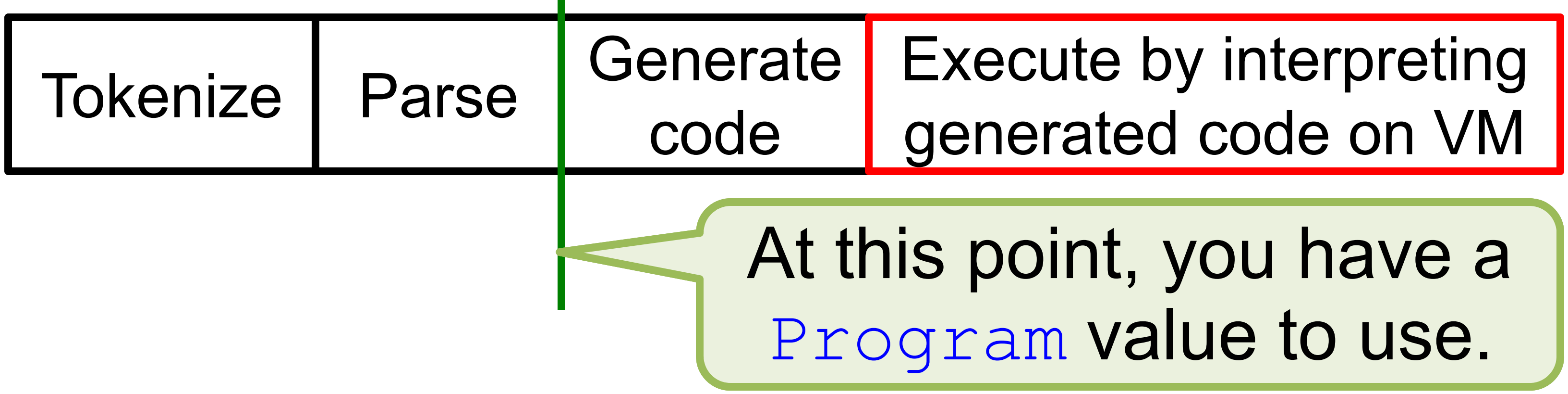
- Compiling and then executing a `Program`:

Tokenize	Parse	Generate code	Execute by interpreting generated code on VM

Time Lines of Execution

- Directly interpreting a `Program`:

- Compiling and then executing a `Program`:

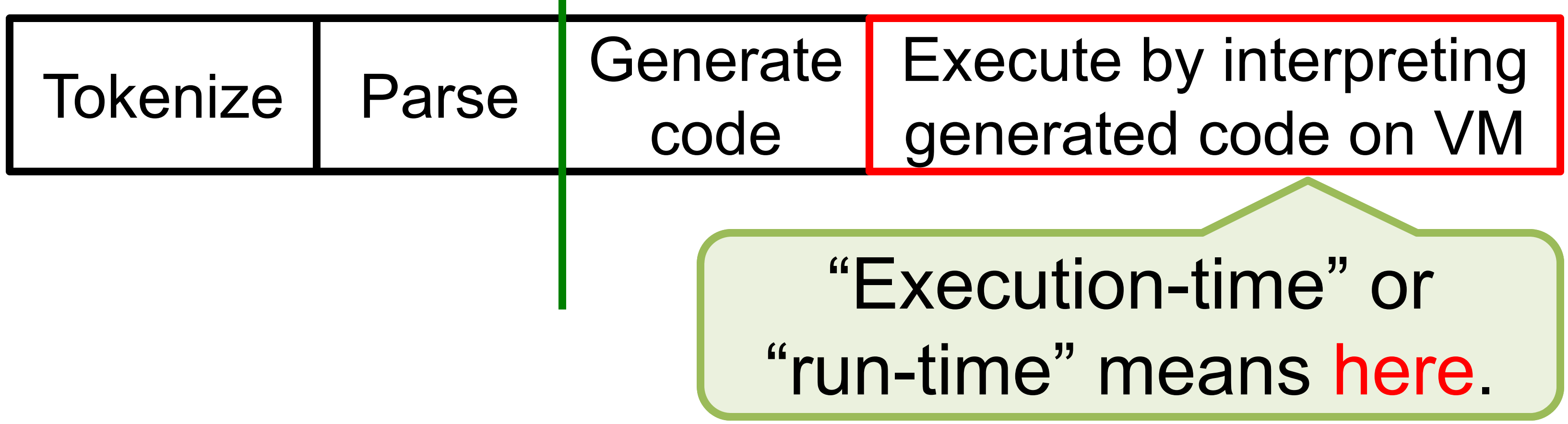


Time Lines

"Execution-time" or "run-time" means here.

- Directly interpreting a `Program`:

- Compiling and then executing a `Program`:

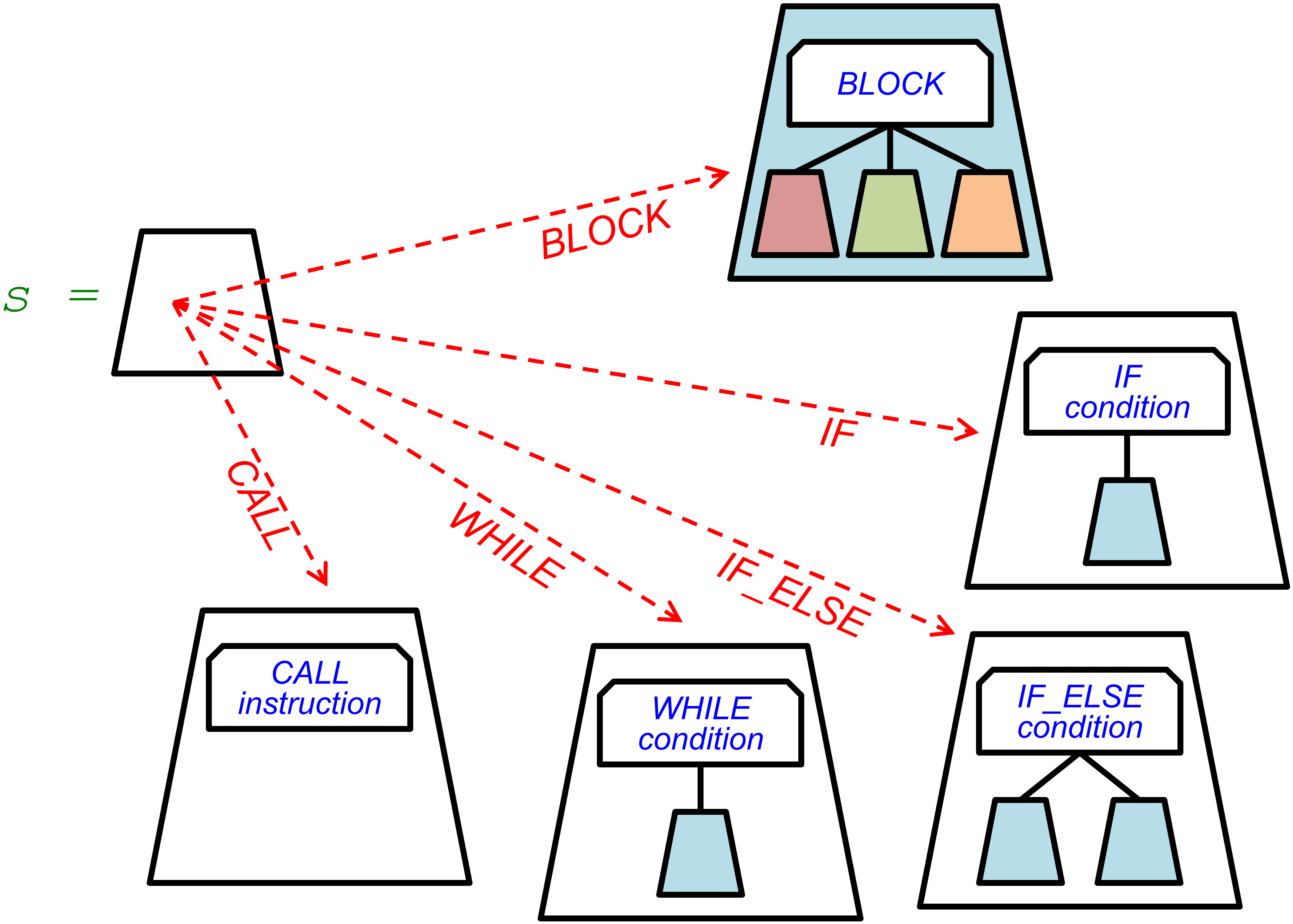


Interpreting a `Program`

- 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 =
BLOCK
BLOCK
IF
IF
condition
CALL
WHILE
IF_ELSE
CALL
instruction
WHILE
condition
IF_ELSE
condition

executeStatement

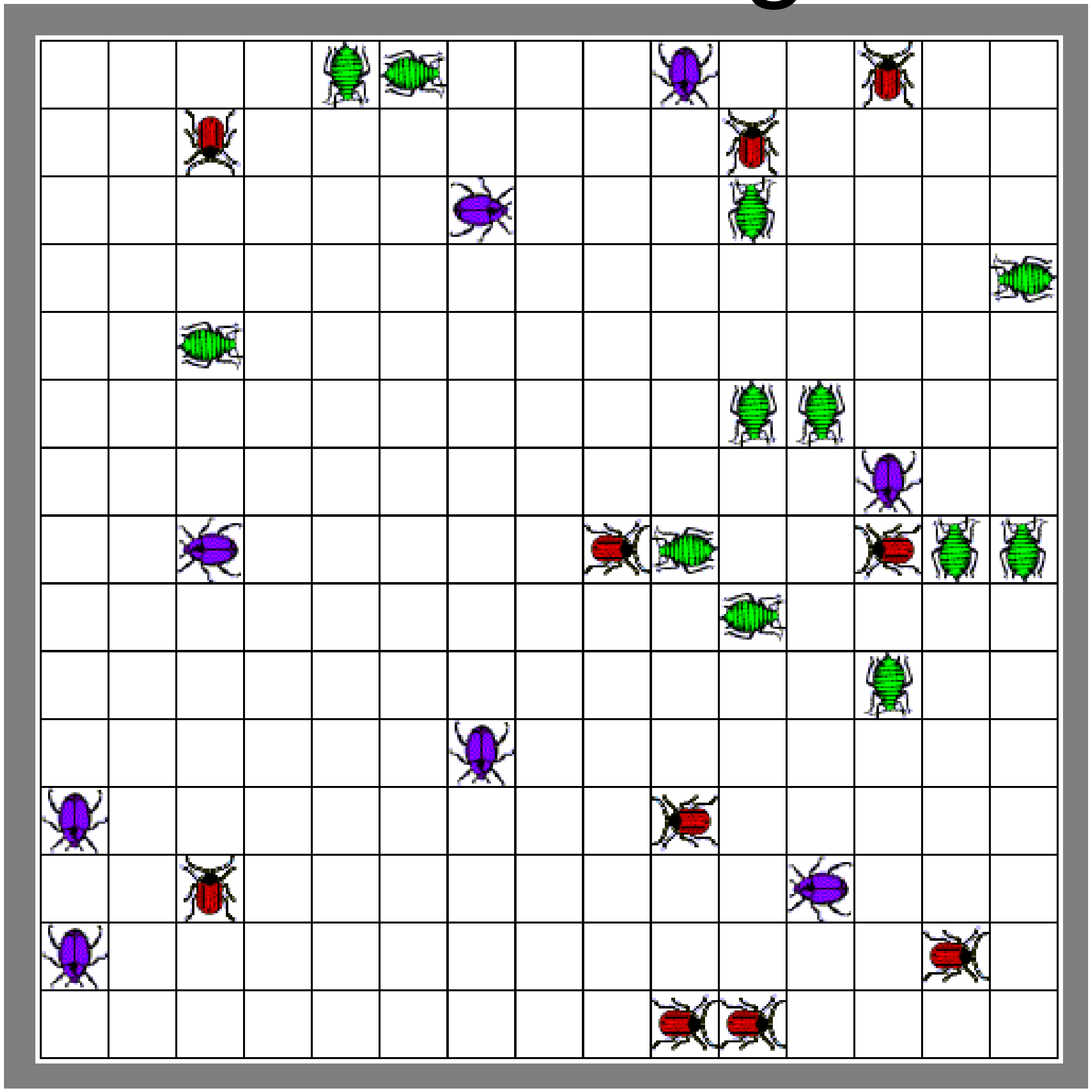
```
public static void executeStatement(Statement s,
    Map<String, Statement> context) {
  switch (s.kind()) {
    case BLOCK: {
      for (int i = 0; i < s.lengthOfBlock(); i++) {
        Statement ns = s.removeFromBlock(i);
        executeStatement(ns, context);
        s.addToBlock(i, ns);
      }
      break;
    }
    case IF: {...}
    ...
}
```

It's recursive just like everything else to do with `Statement`; *context* is needed for case `CALL`.

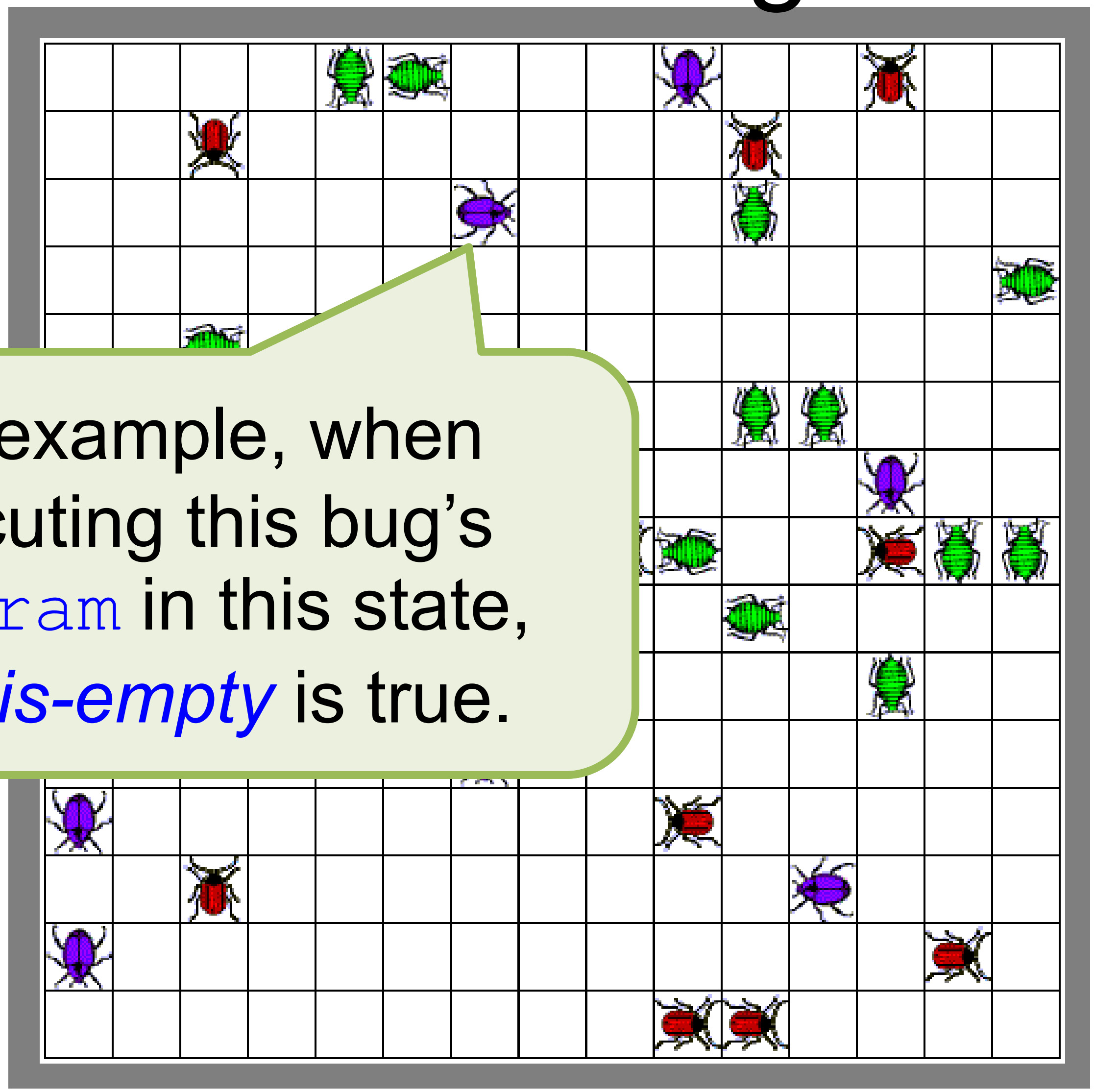
`executeStatement`

- 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



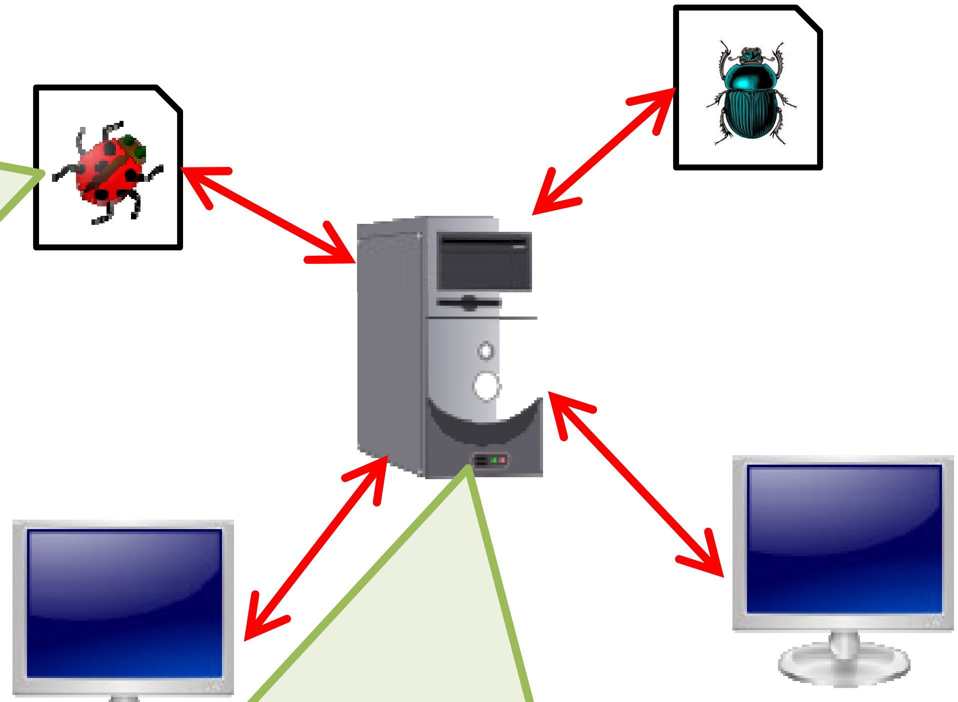
The State of BugsWorld



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".

`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 G

Primitive BL instructions are translated into these “byte codes”.

- ***Code generation*** i `Program` to a ***line*** structure, i.e., to a str of low-level instructions or ***“b te codes”*** of a BL ***virtual mac ine*** 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

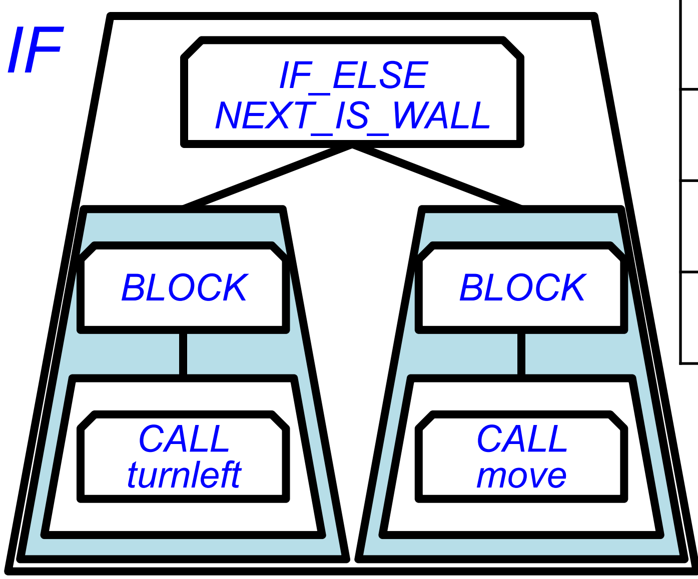
Code G

BL control constructs that check conditions are translated into these "byte codes".

- ***Code generation*** i `Program` to a ***linea*** structure, i.e., to a stri low-level instructions or ***"by codes"*** of a BL ***virtual machi*** that can do the following:
  - Update th 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



Loc	Instruction (symbolic name)
0	`JUMP_IF_NOT_NEXT_IS_WALL`
1	`5`
2	`TURNLEFT`
3	`JUMP`
4	`6`
5	`MOVE`
6	`...`

Example `Statement`

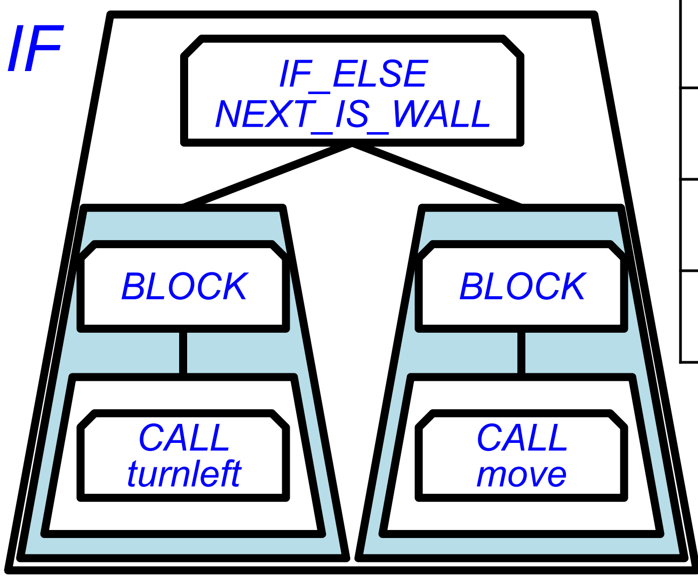
IF next-is-wall THEN

turnleft

ELSE

move

END IF



Loc	Instruction ("byte code")
0	9
1	5
2	1
3	6
4	6
5	0
6	. . .

BugsWorld Virtual Machine

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

BugsWo

A `string of integer`s that contains the “byte codes” generated from a `Program`.

- The ***virtual m***
  three mai... fe...
  - ***Memory***
  - Instruction set
  - Program counter

BugsWo

- The ***virtual m*** three main fe
  - Memory
  - ***Instruction set***
  - Program counter

A ***`finite set of integer`***s that are the ***"byte codes"*** for the primitive instructions of the BugsWorld VM.

BugsWo

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

- The ***virtual m*** three main fe
  - Memory
  - ***Instruction set***
  - Program counter

BugsWo

- The ***virtual m*** three main fe
  - Memory
  - Instruction set
  - ***Program counter***

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

BugsWo

- The ***virtual m***
  three main fe
  - 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 code.

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_...
- JUMP_IF_NOT_N...
- JUMP_IF_NOT_N...
- JUMP_IF_NOT_N...
- JUMP_IF_NOT_N...
- 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` code.

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
- JUMP_IF_
- JUMP_IF_NO
- JUMP_IF_NO
- JUMP_IF_NO
- JUMP_IF_NO
- JUMP_IF_NO

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

Handling an *IF* `Statement`

IF condition THEN
    block
END IF

Loc	Instruction (symbolic name)
k	`JUMP_IF_NOT_condition`
k+1	`k+n+2`
k+2	`block (n instructions)` ↓
...	
k+n+1	
k+n+2	`...`

Handling an *IF_ELSE* `Statement`

IF condition THEN
    block1
ELSE
    block2
END IF

Loc	Instruction (symbolic name)
k	`JUMP_IF_NOT_condition`
k+1	`k+n1+4`
k+2	`block1 (n1 instructions)` ↓
...	
k+n1+2	`JUMP`
k+n1+3	`k+n1+n2+4`
k+n1+4	`block2 (n2 instructions)` ↓
...	
k+n1+n2+4	`...`

Handling a *WHILE* `Statement`

WHILE condition DO

block

END WHILE

Loc	Instruction (symbolic name)
k	`JUMP_IF_NOT_condition`
k+1	`k+n+4`
k+2	`block (n instructions)`
...	↓
k+n+2	`JUMP`
k+n+3	`k`
k+n+4	`...`

Handling a *CALL* `Statement`

move

Loc	Instruction (symbolic name)
k	`MOVE`

turnleft

Loc	Instruction (symbolic name)
k	`TURNLEFT`

(etc.)

Handling a *CALL* `Statement`

INSTRUCTION
  my-instruction IS
    block
END my-instruction

my-instruction

Loc	Instruction (symbolic name)
k	`block (of n instructions)` ↓
...	
k+n-1	
k+n	`...`

Handling a *CALL* `Statement`

INSTRUCTION
my-instruction IS
block
END my-instruction

Loc	Instruction (symbolic name)
k	`block (of n instructions)`
...	↓
k+n-1	
k+n	`...`

my-instruction

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

Loc	Instruction (symbolic name)
k	`block (of n instructions)`
...	
k+n-1	
k+n	`...`

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

Handling a *CALL* `Statement`

INSTRUCTION
  my-instruction IS
    block
END my-instruction

my-instruction

Loc	Instruction (symbolic name)
k	`block (of n instructions)`
...	
k+n-1	
k+n	`...`

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

Handling a *BLOCK* `Statement`

- The “byte codes” generated for individual `Statement`s in a block (a sequence of `Statement`s) 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 `enum`

- Recall: the Java ***enum*** construct allows you to give meaningful symbolic names to values for which you might instead have used *arbitrary* `int` constants
- This construct has some other valuable features that allow you to associate symbolic names (e.g., for VM instructions) with *specific* `int` constants (e.g., their "byte codes")

The `Instruction` Enum

- The interface `Program` contains this code:

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

    ...

}
```

plus 15 more instructions

An instance variable, a constructor, and an accessor method ...

The `Instruction` Enum

- The interface `Program` contains this code:

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

    private int blByteCode;

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

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

The Instru

Every Instruction (e.g., MOVE) has an **int** ***instance variable*** called blByteCode.

- The interface Prog

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

    private int blByteCode;

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

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

The Instr

- The interface Pro

```
enum Instruction {
    MOVE(0), TURNLEFT(

    private int blByteC          ;

    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 Instru

- The interface Prog

```
enum Instruction {
    MOVE(0), TURNLEFT(

    private int blByteCode

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

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

This ***accessor 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