

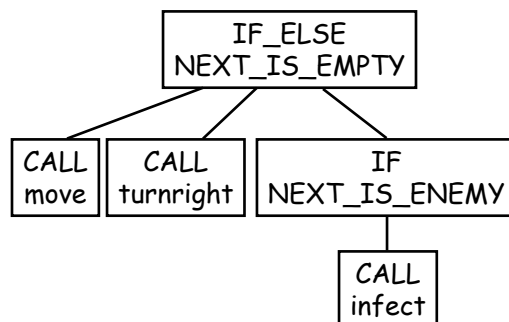
Statement Component

How can we use *tree* to model a *BL statement*?

```
IF next-is-empty THEN
  move
  turnright
ELSE
  IF next-is-enemy THEN
    infect
  END IF
END IF
```

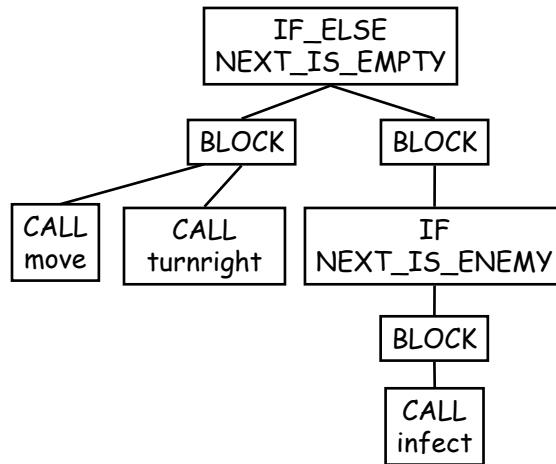
Statement Component

What's wrong with

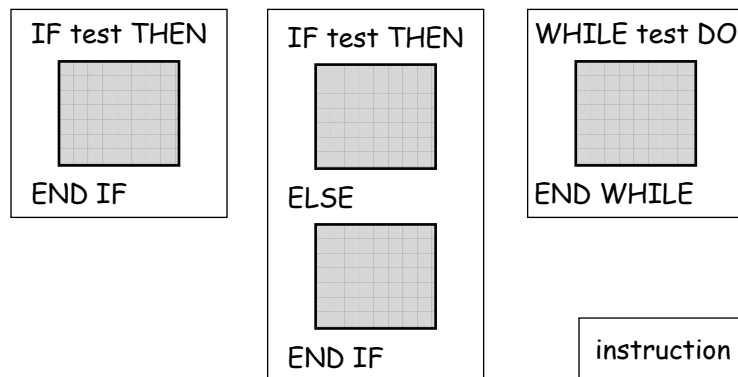


Statement Component

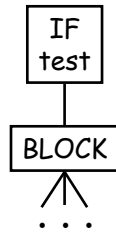
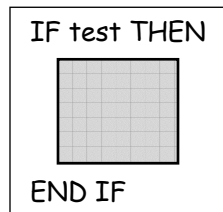
An Abstract Syntax Tree



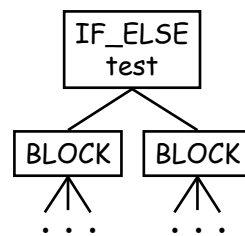
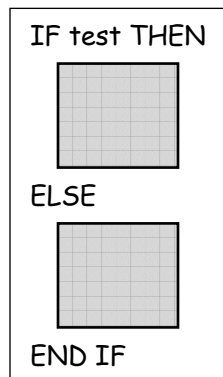
BL Statements



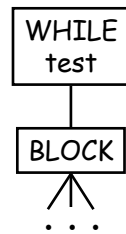
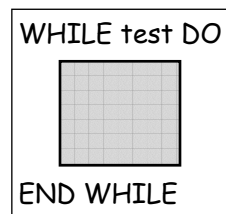
BL Statements: IF



BL Statements: IF_ELSE



BL Statements: WHILE



BL Statements: CALL

instruction

CALL
instruction

An Example

```
WHILE true DO
  IF next-is-enemy THEN
    infect
  ELSE
    IF next-is-empty THEN
      move
    ELSE
      turnleft
    END IF
  END IF
END WHILE
```

Statement Continued...

- Type
 - Statement_Kernel is modeled by STATEMENT
- Initial Value
 - IS_INITIAL_STATEMENT (self)

Statement Continued...

- **math subtype** STATEMENT_LABEL is (
 kind: Statement_Kind
 test: Condition
 instruction: IDENTIFIER
)
- constraint** ...
- **math subtype** STATEMENT is
 tree of STATEMENT_LABEL
 exemplar s
 constraint IS_LEGAL_STATEMENT (s)

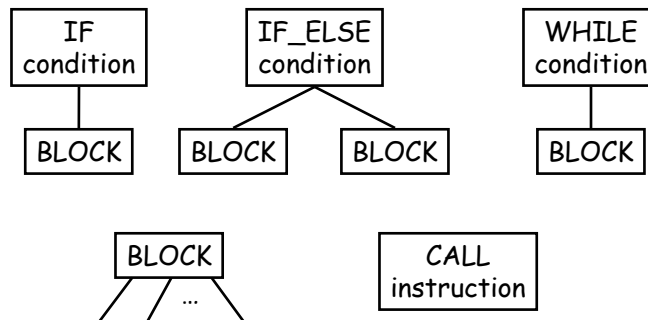
Statement Continued...

- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">▪ Statement_Kind ▪ BLOCK ▪ IF ▪ IF_ELSE ▪ WHILE ▪ CALL | <ul style="list-style-type: none">▪ 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 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

These are all integer values, but we use the names because they are more meaningful than arbitrary integer values.

Statement Continued...

■ IS_LEGAL_STATEMENT?



Statement Operations

■ Operations

- s.Add_To_Block (pos, statement)
- s.Remove_From_Block (pos, statement)
- s.Length_Of_Block ()
- s.Compose_If (cond, block)
- s.Decompose_If (cond, block)
- s.Compose_If_Else (cond, if_block, else_block)
- s.Decompose_If_Else (cond, if_block, else_block)
- s.Compose_While (cond, block)
- s.Decompose_While (cond, block)
- s.Compose_Call (inst)
- s.Decompose_Call (inst)
- s.Kind ()



Statement Block Operations

- `s.Add_To_Block (pos, statement)`
- `s.Remove_From_Block (pos, statement)`
- `s.Length_Of_Block ()`



Statement If Operations

- `s.Compose_If (cond, block)`
- `s.Decompose_If (cond, block)`



Statement If_Else Operations

- `s.Compose_If_Else (cond, if_block, else_block)`
- `s.Decompose_If_Else (cond, if_block, else_block)`



Statement While Operations

- `s.Compose_While (cond, block)`
- `s.Decompose_While (cond, block)`



Statement Call Operations

- `s.Compose_Call (inst)`
- `s.Decompose_Call (inst)`

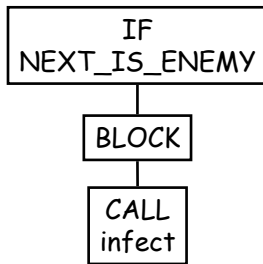


Statement Other Operations

- `s.Kind ()`

Using Statement Operations

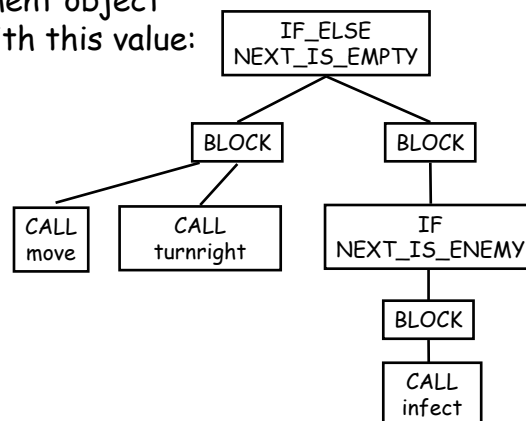
- What operations are needed to produce Statement *if_stmt* =



object Statement call, block, if_stmt;
object Text inst = "infect";
object Integer cond = NEXT_IS_ENEMY;
call.Compose_Call (inst);
block.Add_To_Block (0, call);
if_stmt.Compose_If (cond, block);

Using Statement Operations Continued...

Consider Statement object
If_Else_stmt with this value:



Using Statement Operations Continued...

Show the operations to produce
If_Else_stmt on the previous slide:

Practice Operation

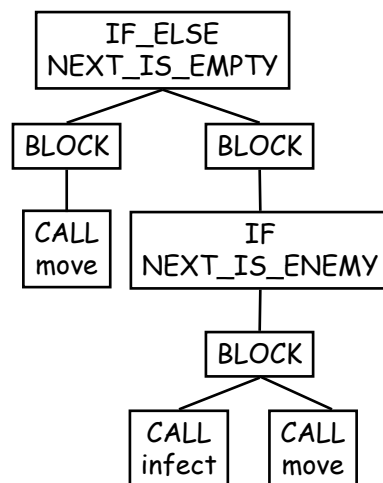
- Most operations on Statement have to be recursive
- Use 5 step process to recursion:
 0. State the problem
 1. Visualize recursive structure
 2. Verify that visualized recursive structure can be leveraged into an implementation
 3. Visualize a recursive implementation
 4. Write a skeleton for the operation body
 5. Refine the skeleton into an operation body

Step 0: State the Problem

- Procedure Demobilize replaces every occurrence of the "move" instruction in statement s with a "skip" instruction.

```
global_procedure Demobilize (  
  alters Statement& s  
);  
/*!  
  ensures  
  s = DEMOBILIZE (#s)  
!*/
```

Demobilize this Statement



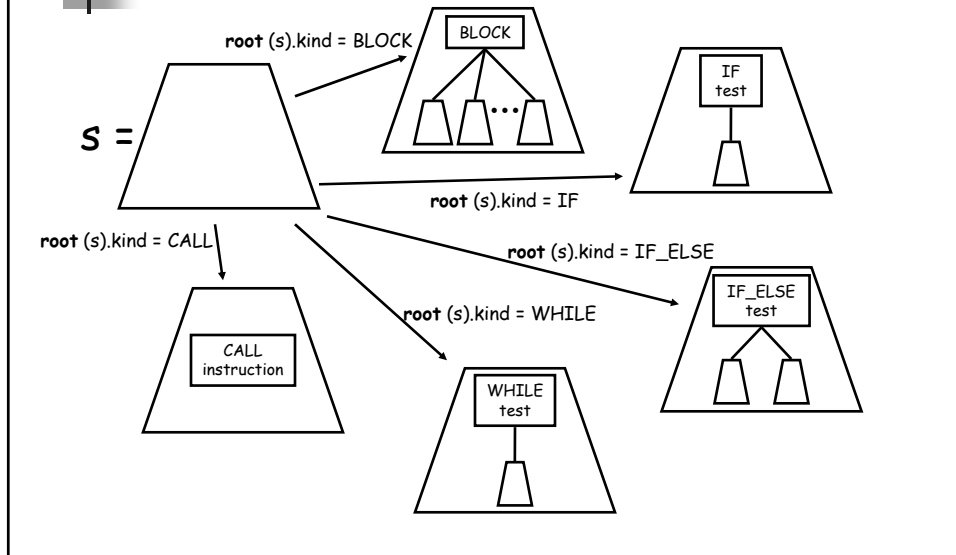
Step 0: State the Problem Continued...

```
math definition DEMOBILIZE (  
  s: STATEMENT  
) : STATEMENT satisfies  
  if root (s).kind = CALL  
  then  
    if root (s).instruction = "move"  
    then DEMOBILIZE (s) =  
      compose ((CALL, TRUE, "skip"), empty_string)  
    else DEMOBILIZE (s) = s  
  else  
    there exists label: STATEMENT_LABEL,  
      nested_stmts: string of STATEMENT  
    (s = compose (label, nested_stmts) and  
     DEMOBILIZE (s) =  
      compose (label, STRING_DEMOBILIZE (nested_stmts)))
```

Step 0: State the Problem Continued...

```
math definition STRING_DEMOBILIZE (  
  str: string of STATEMENT  
) : string of STATEMENT satisfies  
  if str = empty_string  
  then STRING_DEMOBILIZE (str) = empty_string  
  else there exists s: STATEMENT,  
    rest: string of STATEMENT  
  (str = <s> * rest and  
   STRING_DEMOBILIZE (str) =  
     DEMOBILIZE (s) * STRING_DEMOBILIZE (rest))
```

Step 1: Visualize Recursive Structure



Step 2: Verify That Leveraging Works

- Ask yourself: If Demobilize could get a helper to demobilize the nested statements in each of the five (four?) cases, could it take advantage of this generous offer?
- Yes! Once you know how to demobilize the nested statements, you can demobilize the entire statement.

Step 2 $\frac{1}{2}$:

Let's Jump Ahead

```
procedure_body Demobilize (  
    alters Statement& s  
)  
{  
    case_select (s.Kind ())  
    {  
        case BLOCK: {Demobilize_Block (s);} break;  
        case IF: {Demobilize_If (s);} break;  
        case IF_ELSE: {Demobilize_If_Else (s);} break;  
        case WHILE: {Demobilize_While (s);} break;  
        case CALL:  
        { object Text inst, skipinst = "skip";  
          s.Decompose_Call (inst);  
          if ( inst == "move")  
            { inst &= skipinst; }  
          s.Compose_Call (inst);  
        } break;  
    }  
}
```

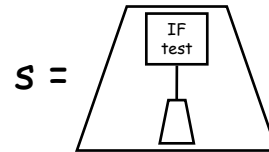
Step 2 $\frac{3}{4}$ (or back to Step 0 ☹):

State the Problem

```
global_procedure Demobilize_Block (  
    alters Statement& s  
);  
/*!  
    requires  
        root (s).kind = BLOCK  
    ensures  
        s = DEMOBILIZE (#s)  
!*/
```

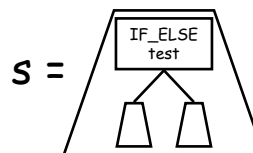

Steps 1-5 Condensed (IF)

```
procedure_body Demobilize_If (  
    alters Statement& s  
)  
{  
  
  
  
  
  
  
}
```



Steps 1-5 Condensed (IF_ELSE)

```
procedure_body Demobilize_If_Else (  
    alters Statement& s  
)  
{ // You try it!  
  
  
  
  
  
  
}
```



Statement: Representation

- The representation for *Statement_Kernel_1* has one field:
 - *tree_rep*: a *Tree_Of_Tree_Node* where a *Tree_Node* is a *Record* with three fields:
 - *kind*: Integer
 - *test*: Integer
 - *instruction*: Text

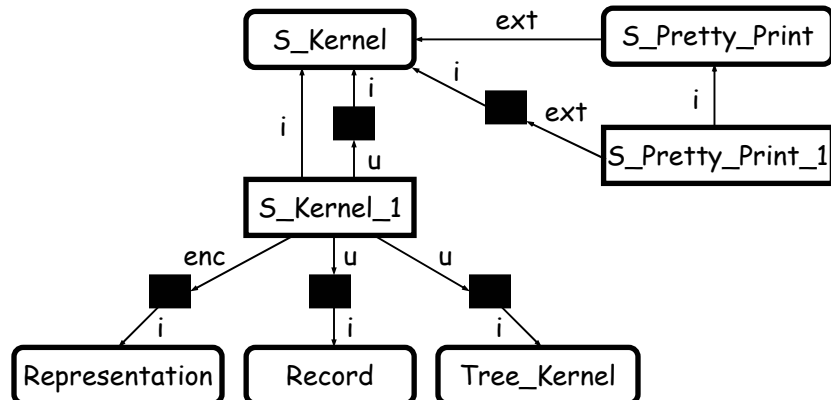
Statement: Correspondence

- *self*, the *Statement*, is equal to *self.tree_rep*, the *Tree* used to represent it.

Statement: Convention

- All the labels in the tree, *tree_rep*, are legal statement labels (i.e., they satisfy the constraint for the definition of STATEMENT_LABEL);
- The tree itself is a legal statement (i.e., it satisfies the constraint for the definition of STATEMENT).

Statement: CCD



Statement: Compose_If

```
object Tree_Of_Tree_Node new_tree_rep, t;

// make t a valid representation for an initial value of Statement
t[current][kind] = BLOCK;
t[current][test] = TRUE;

// extract representation tree from block and consume block
block[tree_rep] &= t;

// construct new representation tree
new_tree_rep[current][kind] = IF;
new_tree_rep[current][test] &= cond; // consume cond
new_tree_rep.Add (0, t);

// produce self
self[tree_rep] &= new_tree_rep;
```

Statement: Decompose_If

```
object Tree_Of_Tree_Node new_tree_rep, t;

// make new_tree_rep a valid representation for
// an initial value of Statement
new_tree_rep[current][kind] = BLOCK;
new_tree_rep[current][test] = TRUE;

// consume self
self[tree_rep] &= new_tree_rep;

// extract condition and body from representation of if
new_tree_rep[current][test] &= cond;
new_tree_rep.Remove (0, block[tree_rep]);
```