

## Lecture 11

# Compiler II: Code Generation

These slides support chapter 11 of the book

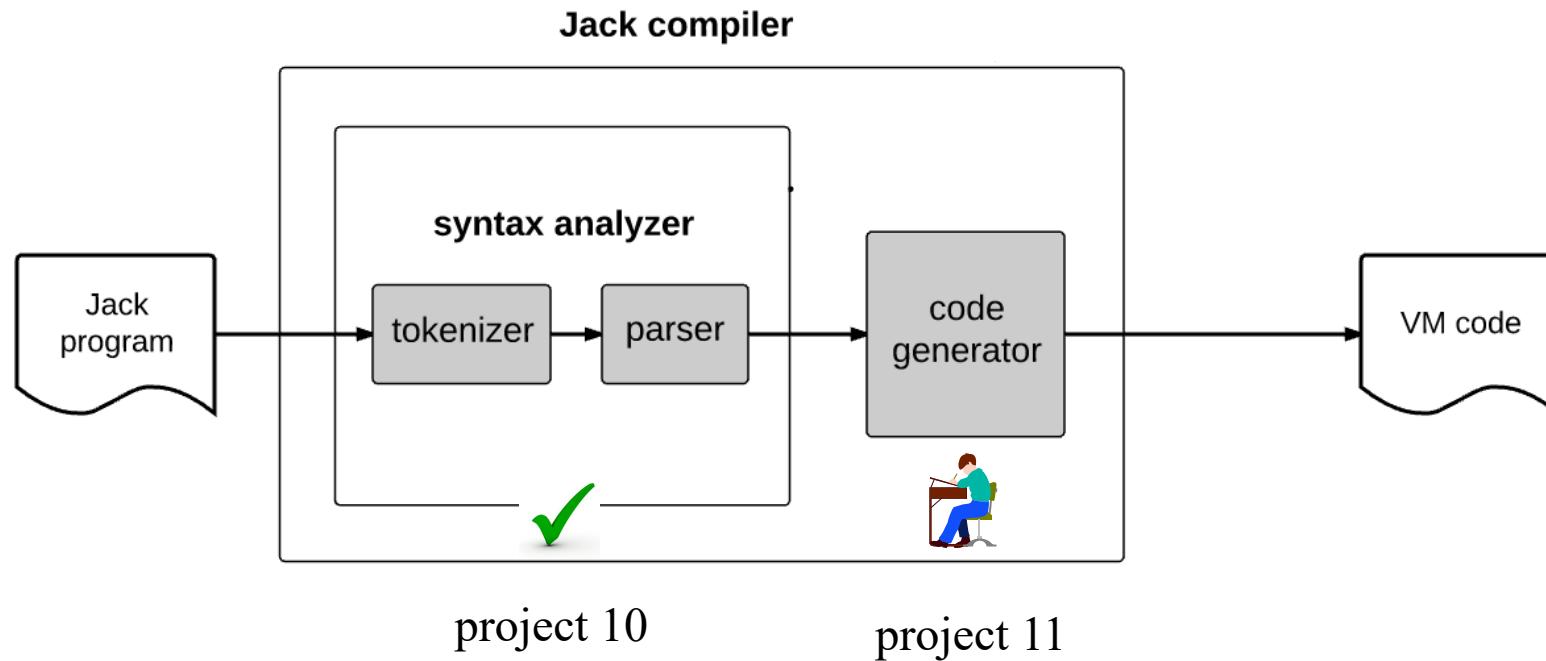
*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press, 2021

# Compiler development roadmap

---



## This lecture

Extending the syntax analyzer to a full-scale compiler.

# Topics

---

## Implementing a procedural programming language



### Variables

- Expressions
- Statements

## Adding object-based programming features

- Objects
- Constructors
- Methods

## Techniques

- Parsing (mostly done)



### Symbol tables

- Compilation engine
- Code generation.

# Symbol table

---

Source code

```
class Point {  
    field int x, y;  
    static int pointCount;  
  
    ...  
  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) + (dy*dy));  
    }  
    ...  
}
```

# Symbol table

---

Source code

```
class Point {  
    field int x, y;  
    static int pointCount;  
  
    ...  
  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) + (dy*dy));  
    }  
    ...  
}
```

## Variable properties

name (identifier)

type (int, char, boolean, class name)

kind (field, static, local, argument)

scope (class level, subroutine level)

# Symbol table

## Source code

```
class Point {  
    field int x, y;  
    static int pointCount;  
  
    ...  
  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) + (dy*dy));  
    }  
    ...  
}
```

|                           | name       | type  | kind     | # |
|---------------------------|------------|-------|----------|---|
| class-level symbol table  | x          | int   | this     | 0 |
|                           | y          | int   | this     | 1 |
|                           | pointCount | int   | static   | 0 |
| method-level symbol table | this       | Point | argument | 0 |
|                           | other      | Point | argument | 1 |
|                           | dx         | int   | local    | 0 |
|                           | dy         | int   | local    | 1 |

this (argument 0) :

a system variable representing the object on which the method operates (discussed later in the lecture)

## Variable properties

name (identifier)

type (int, char, boolean, class name)

kind (field, static, local, argument)

scope (class level, subroutine level)

The Jack compiler represents all the program's variables as entries in two symbol tables;

The tables are constructed and used during the compilation process.

# Symbol table: Construction

## Source code

class declaration

```
class Point {  
    field int x, y;  
    static int pointCount;
```

method declaration

```
...  
method int distance(Point other) {  
    var int dx, dy;  
    let dx = x - other.getx();  
    let dy = y - other.gety();  
    return Math.sqrt((dx*dx) + (dy*dy));  
}  
...  
}
```

## class-level symbol table

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

## method-level symbol table

| name  | type  | kind     | # |
|-------|-------|----------|---|
| this  | Point | argument | 0 |
| other | Point | argument | 1 |
| dx    | int   | local    | 0 |
| dy    | int   | local    | 1 |

this (argument 0) :

a system variable representing the object on which the method operates (discussed later in the lecture)

## Compiling a class declaration

The compiler creates a class-level symbol table;

For each *field* and *static* variable found in the class declaration,  
the compiler adds the variable to the symbol table as *this i* or as *static i*

## Compiling a method declaration

The compiler creates a method-level symbol table, and adds to it the entry <this *className* argument 0>;

For each *parameter* and *local* variable found in the method declaration,  
the compiler adds the variable to the symbol table as *argument i* or as *local i*

The compiler generates the *i* values from running counters, one for each of the four memory segments.

# Symbol table: Usage

## Source code

class declaration

```
class Point {  
    field int x, y;  
    static int pointCount;  
  
    ...  
  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) + (dy*dy));  
    }  
    ...  
}
```

method declaration

## class-level symbol table

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

## method-level symbol table

| name  | type  | kind     | # |
|-------|-------|----------|---|
| this  | Point | argument | 0 |
| other | Point | argument | 1 |
| dx    | int   | local    | 0 |
| dy    | int   | local    | 1 |

this (argument 0):

a system variable representing the object on which the method operates (discussed later in the lecture)

## Compiling statements

For each variable found in the statement:

The compiler looks up the variable in the method-level symbol table;

If found, the variable is replaced with its *segment i* reference;

Else, the compiler looks up the variable in the class-level symbol table;

If found, the variable is replaced with its *segment i* reference;

Else, the compiler throws a compilation error.

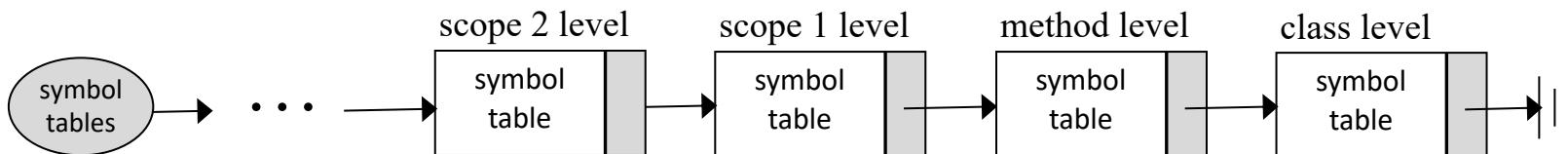
For example, x + dx is compiled into push this 0, push local 0, add.

# Nested scoping

```
class foo {  
    // class-level variable declarations  
    method bar () {  
        // method-level variable declarations  
        ...  
        {  
            // scope-1-level variable declarations  
            ...  
            {  
                // scope-2-level variable declarations  
                ...  
            }  
        }  
    }  
}
```

Some high-level languages (but not Jack) feature nested variable scoping of unlimited depth

Nested scoping can be handled using a linked list of symbol tables:



Variable lookup: The variable is looked up in the first table in the list:  
if not found, the next table is looked up, and so on.

# Variable life cycles

Source code

```
class Point {  
    field int x, y;  
    static int pointCount;  
  
    ...  
  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) + (dy*dy));  
    }  
    ...  
}
```

Static variables: Persist throughout the program's execution

Field variables: Persist as long as the objects are not disposed

Local variables: Each time a subroutine starts running during runtime, it gets a fresh set of local variables; Each time a subroutine returns, its local variables are recycled

Argument variables: Same as local variables

Managing these variable life cycles is a major headache;

But... this is taken care of by the VM implementation (projects 7–8);

The compiler need not worry about it!



# Lecture plan

---

## Compilation



Handling variables



Handling expressions

- Handling statements
- Handling objects
- Handling arrays

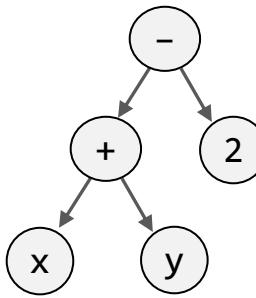
# Compiling expressions

---

infix notation

**x + y - 2**

human  
friendly

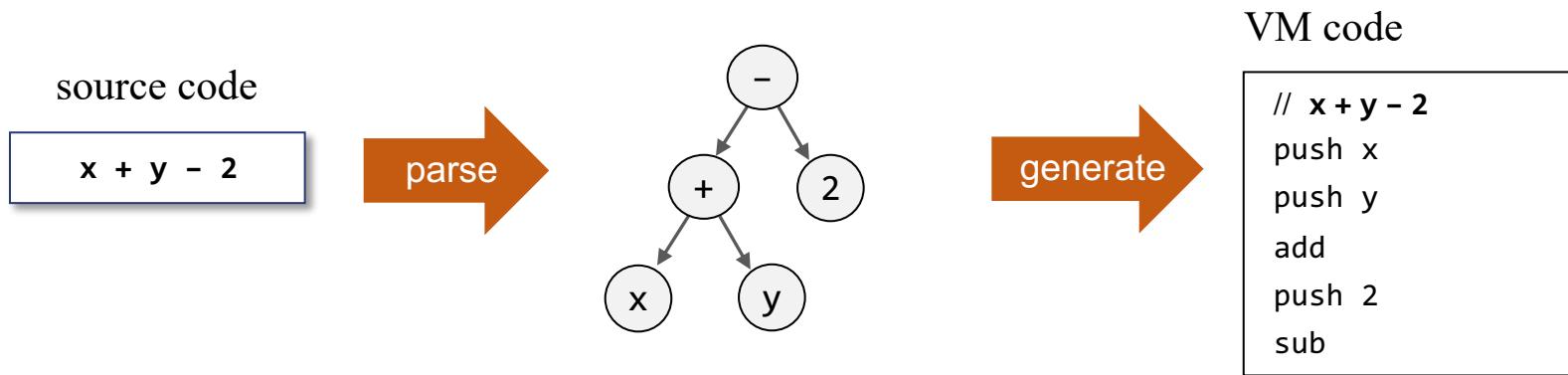


postfix notation

**x y + 2 -**

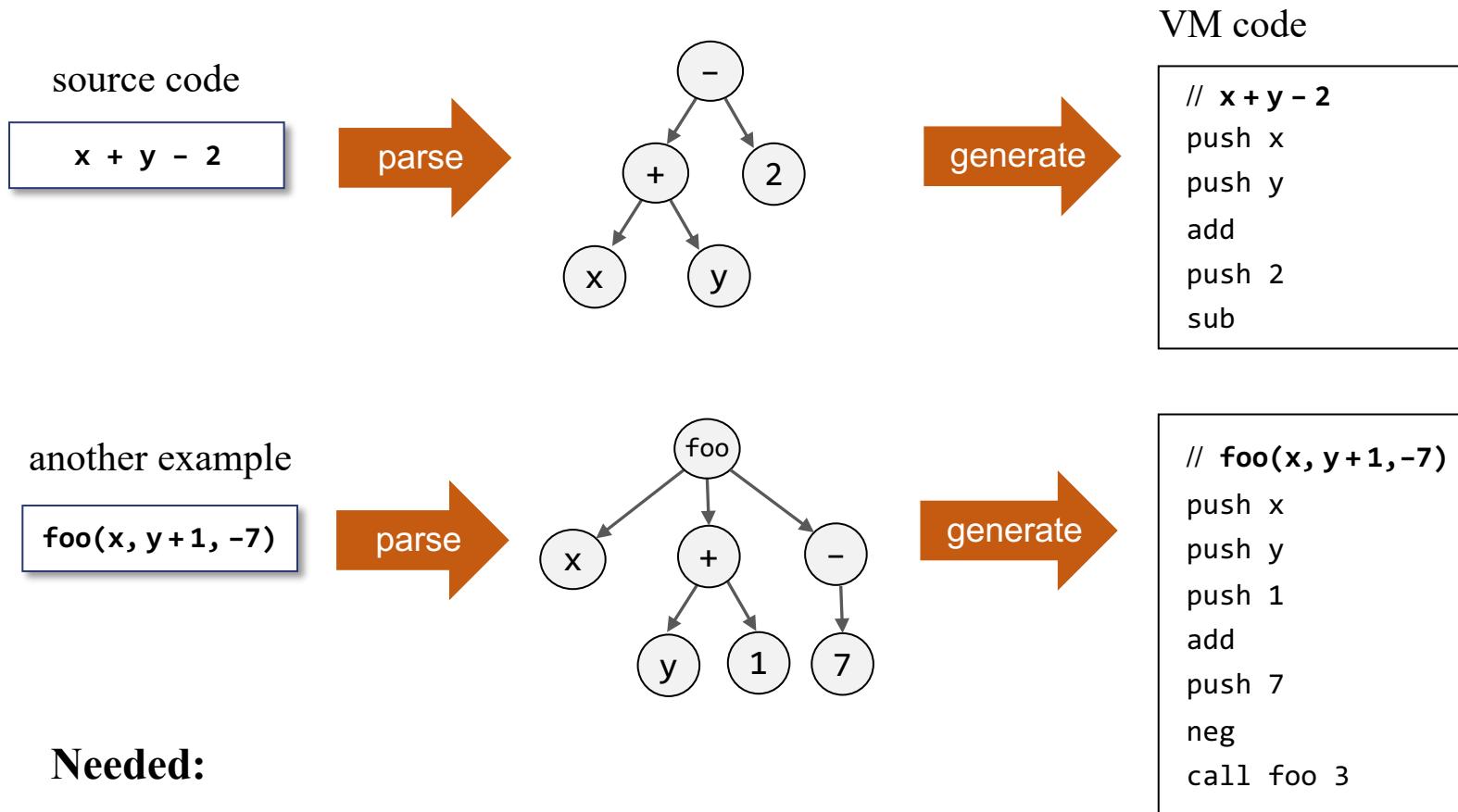
stack automata  
friendly

# Compiling expressions



When executed during runtime,  
the generated code ends up  
leaving the value of the  
expression at the stack's top

# Compiling expressions



**Needed:**

A definition of “expression”;

A general algorithm for compiling expressions.

# Compiling expressions

Recursive algorithm for compiling expressions:

```
compileExpression(exp):  
    if exp is term:  
        compileTerm(term)  
  
    if exp is "term1 op1 term2 op2 term3 op3 ... termn":  
        compileTerm(term1)  
        compileTerm(term2)  
        output "op1"  
        compileTerm(term3)  
        output "op2"  
        ...  
        compileTerm(termn)  
        output "opn-1"
```

```
compileTerm(term):  
    if term is a constant c:  
        output "push c"  
  
    if term is a variable var:  
        output "push var"  
  
    if term is "unaryOp term":  
        compileTerm(term)  
        output "unaryOp"  
  
    if term is "f(exp1, exp2, ...)":  
        compileExpression(exp1)  
        compileExpression(exp2)  
        ...  
        compileExpression(expn)  
        output "call f n"  
  
    if term is "(exp)":  
        compileExpression(exp)
```

Generated code (examples)

```
// x + y - 2  
push x  
push y  
add  
push 2  
sub
```

```
// foo(x, y+1, -7)  
push x  
push y  
push 1  
add  
push 7  
neg  
call foo 3
```

Jack grammar (subset):

```
expression: term1 op1 term2 op2 term3 op3 ... termn  
term: constant | varName | unaryOp term | subroutineCall | ( expression )  
subroutineCall: f ( expression1, expression2, ..., expressionn )
```

# Compiling expressions

Recursive algorithm for compiling expressions:

```
compileExpression(exp):  
    if exp is term:  
        compileTerm(term)  
  
    if exp is "term1 op1 term2 op2  
            term3 op3 ... termn":  
        compileTerm(term1)  
        compileTerm(term2)  
        output "op1"  
        compileTerm(term3)  
        output "op2"  
        ...  
        compileTerm(termn)  
        output "opn-1"
```

```
compileTerm(term):  
    if term is a constant c:  
        output "push c"  
  
    if term is a variable var:  
        output "push var"  
  
    if term is "unaryOp term":  
        compileTerm(term)  
        output "unaryOp"  
  
    if term is "f(exp1, exp2, ...)":  
        compileExpression(exp1)  
        compileExpression(exp2)  
        ...  
        compileExpression(expn)  
        output "call f n"  
  
    if term is "(exp)":  
        compileExpression(exp)
```

Generated code (examples)

```
// x + y - 2  
push x  
push y  
add  
push 2  
sub
```

```
// foo(x, y+1,-7)  
push x  
push y  
push 1  
add  
push 7  
neg  
call foo 3
```

Notes:

The complete algorithm must implement the complete grammar of Jack expressions;

The complete algorithm must generate VM code in which variables are represented as local *i*, argument *i*, etc. (instead of x, y, etc.)

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- Handling statements
  - Handling objects
  - Handling arrays

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- Handling statements

let  
return  
do  
if  
while

We'll describe how to compile each statement, and outline a compilation routine that handles it.

# Lecture plan

---

## Compilation

- ✓ Handling variables
  - ✓ Handling expressions
    - Handling statements
- let  
return  
do  
if  
while
- 

We'll describe how to compile each statement, and outline a compilation routine that handles it.

# Compiling let

---

source code (Jack)

```
...  
let varName = expression;  
...
```

compiler

VM code (generated by `compileLet`)

```
...  
VM code generated by compileExpression  
pop varName  
...
```

When the VM code generated by `compileExpression` will execute during runtime, it will end up leaving the expression's value at the stack's top.

The subsequent `pop` will then store this value in the segment entry.

# Compiling let

source code (Jack)

```
...  
let varName = expression;  
...
```

compiler

VM code (generated by compileLet)

```
...  
VM code generated by compileExpression  
pop varName  
...
```

example

```
let v = g + r2;
```

compiler

```
push g  
push r2  
add  
pop v
```

1. first three commands are generated by compileExpression
2. Instead of symbolic variable names, the generated VM code uses segment entries.

General compilation routine:

compileLet:

```
compileExpression  
output "pop static/this/argument/local i "
```

# Lecture plan

---

## Compilation

- Handling variables
- Handling expressions
- Handling statements

`let`



`return`

`do`

`if`

`while`

# Compiling return expression

---

source code

```
...  
return expression;  
...
```

compiler →

VM code (generated by compileReturn)

```
...  
VM code generated by compileExpression  
return  
...
```

When the VM code generated by `compileExpression` will execute during runtime, it will end up leaving the expression's value at the stack's top;

The VM implementation of the subsequent `return` command will place the return value at the top of caller's working stack.

General compilation routine:

```
compileReturn:  
    compileExpression  
    output "return"
```

# Compiling return

---

source code

```
...  
return;  
...
```

compiler

VM code (generated by compileReturn)

```
...  
push constant 0  
return  
...
```

When compiling a `return` Jack statement with no return value, the `compileReturn` routine generates code that pushes a dummy value onto the stack;

The dummy value will be tossed away by the compiled code of the caller (discussed next).

General compilation routine:

`compileReturn:`

```
    output "push constant 0"  
    output "return"
```

Conventions: (1) A Jack subroutine must end with a `return / return value` statement

(2) The compiled code of a subroutine always returns a value (even if it's `void`).

# Lecture plan

---

## Compilation

- Handling variables
- Handling expressions
- Handling statements

let

return

do

if

while



# Compiling do

source code (Jack)

```
...  
do subroutineName(exp1, exp2, ...)  
...
```

compiler →

VM code (generated by compileDo)

```
...  
VM code generated by compileExpression  
pop temp 0  
...
```

Used to call a function or a method for its effect, ignoring the returned value

The pop gets rid of the return value.

example

```
do Output.putInt(7);
```

compiler →

```
push constant 7  
call Output.putInt 1  
pop temp 0
```

first two commands are generated by compileExpression

General compilation routine:

compileDo:

```
compileExpression  
output "pop temp 0"
```

Explanation:

We compile `do subroutineName(...)` statements as if they were `do expression` statements;

The `compileExpression` routine knows how to handle method calls.



# Lecture plan

---

## Compilation

- Handling variables
- Handling expressions
- Handling statements

let

return

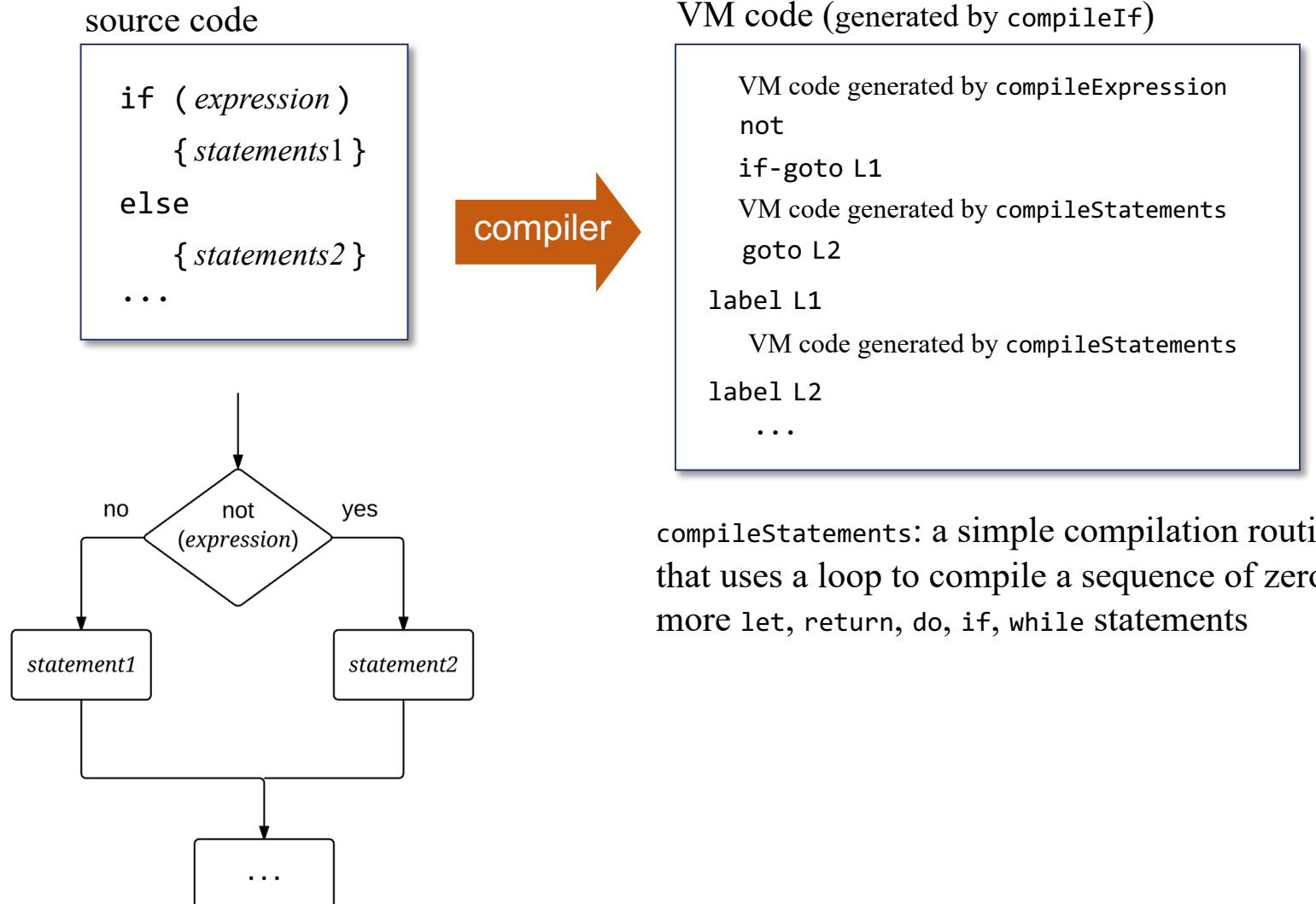
do

if

while



# Compiling if



`compileStatements`: a simple compilation routine that uses a loop to compile a sequence of zero or more `let`, `return`, `do`, `if`, `while` statements

# Compiling if

source code

```
if ( expression )
    { statements1 }
else
    { statements2 }
...
```

compiler

VM code (generated by compileIf)

```
VM code generated by compileExpression
not
if-goto L1
VM code generated by compileStatements
goto L2
label L1
VM code generated by compileStatements
label L2
...
```

example

```
if (x < 0) {
    let y = 0;
} else {
    let y = 1;
}
...
```

compiler

```
push x
push 0
lt
not
if-goto L1
push 0
pop y
goto L2
label L1
push 1
pop y
label L2
...
```

# Compiling if

source code

```
if ( expression )
    { statements1 }
else
    { statements2 }
...
```

compiler →

VM code (generated by compileIf)

```
VM code generated by compileExpression
not
if-goto L1
VM code generated by compileStatements
goto L2
label L1
VM code generated by compileStatements
label L2
...
```

General compilation routine:

**compileIf:**

    compileExpression

    output "not"

    generate a unique label L1, and output "if-goto L1"

    compileStatements

    generate a unique label L2, and output "goto L2"

    output "label L1"

    compileStatements

    output "label L2"

# Compiling while

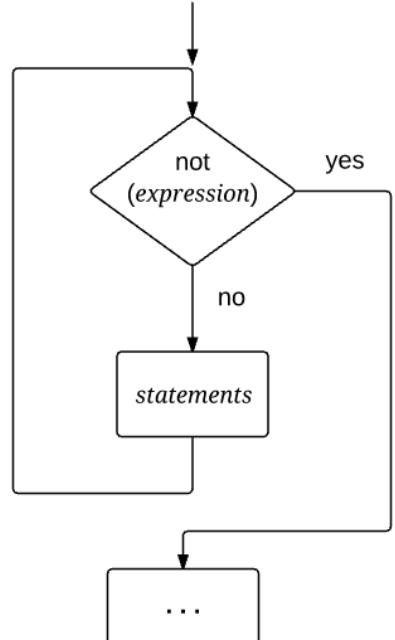
source code

```
while ( expression )
    { statements }
    ...
```

compiler →

VM code (generated by compileWhile)

```
label L1
VM code generated by compileExpression
not
if-goto L2
VM code generated by compileStatements
goto L1
label L2
...
```



# Compiling while

source code

```
while ( expression )
    { statements }
...
```

compiler

VM code (generated by compileWhile)

```
label L1
VM code generated by compileExpression
not
if-goto L2
VM code generated by compileStatements
goto L1
label L2
...
```

example

```
while (g = 0) {
    let x = x + 1;
}
...
```

compiler

```
label L1
push g
push 0
eq
not
if-goto L2
push x
push 1
add
pop x
goto L1
label L2
...
```

(pseudo  
code)

# Compiling while

source code

```
while ( expression )
    { statements }
    ...
```

compiler →

VM code (generated by compileWhile)

```
label L1
VM code generated by compileExpression
not
if-goto L2
VM code generated by compileStatements
goto L1
label L2
...
```

General compilation routine:

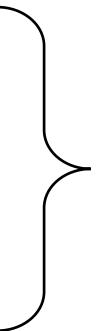
**compileWhile:**

```
generate the unique label L1, and output "label L1"
compileExpression
output "not"
generate the unique label L2, and output "if-goto L2"
compileStatements
output "goto L1"
output "label L2"
```

# Lecture plan

---

## Compilation

- ✓ Handling variables
  - ✓ Handling expressions
  - ✓ Handling statements
- 
- Handling objects
  - Handling arrays
- We have what it takes to write  
a compiler for a procedural  
programming language!

# Lecture plan

---

## Compilation

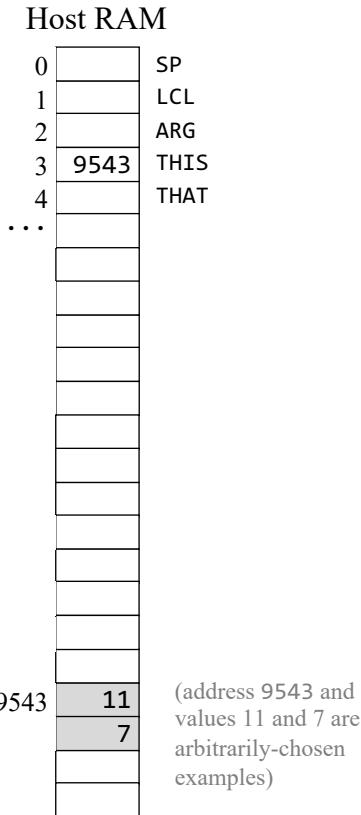
- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
  - Handling objects
    - Structures
      - Constructing objects
      - Manipulating objects
  - Handling arrays

## “Structures”

In the C language, a *structure* is a composite data type, consisting of several *fields*;

In the example shown here, `THIS` points at a two-field structure;

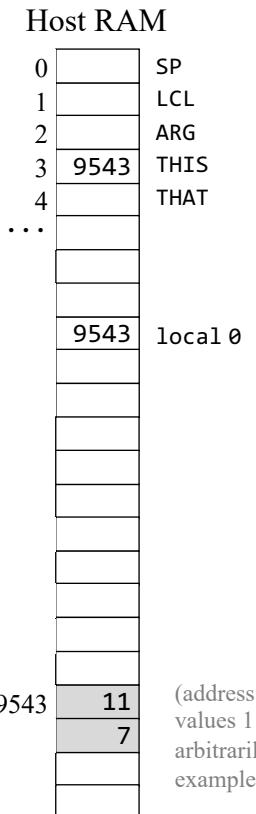
How can we create and manipulate structures using the VM language?



# “Structures”

Creating a structure:

```
// Creates a 2-word structure, initializes it, and makes local 0 refer to it  
push constant 2  
call Memory.alloc 1  
pop pointer 0 // pop THIS  
  
push constant 11  
pop this 0  
push constant 7  
pop this 1  
  
push pointer 0 // push THIS  
pop local 0
```



## Explanation

**Memory.alloc(*n*):** Calls a memory allocation OS function.

alloc finds a memory block of size *n* words, and returns  
(pushes onto the stack) its base address;

**pop pointer 0:** Pops the base address of the new structure into THIS;

Result: Subsequent this *i* references will map on RAM addresses THIS + *i*

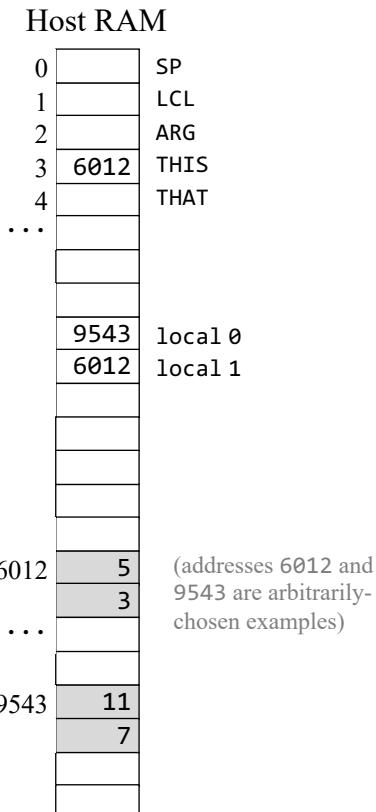
**Last two commands:** Make local 0 point to the new structure.

# “Structures”

Manipulating a structure:

```
// Creates two structures and makes local 0 and local 1 point to them
// (code omitted)
...
// Example of manipulating a structure:
// Sets the fields of p2 to 5 and 3

push local 1
pop pointer 0    // THIS = base address of the target structure
push constant 5
pop this 0
push constant 3
pop this 1
```



## Note

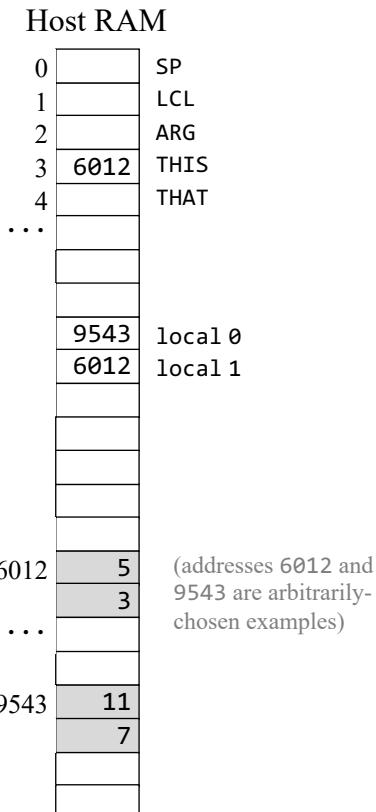
Using this technique, we can get / set the fields of any given structure.

# “Structures”

Manipulating a structure:

```
// Creates two structures and makes local 0 and local 1 point to them
// (code omitted)
...
// Example of manipulating a structure:
// Sets the fields of p2 to 5 and 3

push local 1
pop pointer 0    // THIS = base address of the target structure
push constant 5
pop this 0
push constant 3
pop this 1
```



## From structures to objects

“Objects” are well-dressed structures;

The big step from C to C++ is that C++ compilers allow creating and manipulating objects using a dedicated, object-oriented terminology;

The Jack language and compiler feature such OOP capabilities (next).

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
  - Handling objects
    - Structures
    - Compiling objects construction
      - Compiling objects manipulations
  - Handling arrays

# Compiling constructors

---

Source code

```
class Foo {  
    ...  
    function void bar () {  
        var Point p1;  
        ...  
        // Creates a new Point object  
        let p1 = Point.new(2,3);  
        ...  
        (such constructor calls can appear  
        in any subroutine, in any class)
```

Caller: Creates an object, by calling a constructor

# Compiling constructors

Source code

```
class Foo {  
    ...  
    function void bar () {  
        var Point p1;  
        ...  
        // Creates a new Point object  
        let p1 = Point.new(2,3);  
        ...  
  
(such constructor calls can appear  
in any subroutine, in any class)
```



VM code

```
...  
// let p1 = Point.new(2,3);  
push constant 2  
push constant 3  
call Point.new 2  
pop local 0  
...
```

## Observations

- There's nothing special about compiling *constructor calls*;  
We handle them as if they were regular subroutine calls.
- The calls are serviced by the compiled constructors.

# Compiling constructors

Source code

```
...  
// Creates a new Point object  
let p1 = Point.new(2,3);
```

Caller: Creates an object, by calling a constructor

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
  
    /** Constructs a new point. */  
    constructor Point new(int ax, int ay) {  
        let x = ax;  
        let y = ay;  
        let pointCount = pointCount + 1;  
        return this; /// required in Jack constructors  
    }  
    ...
```

Callee: Does the object construction

Typical client-server, OOP, object construction idiom;

What makes the magic work? The compiled constructor's code.

# Compiling constructors

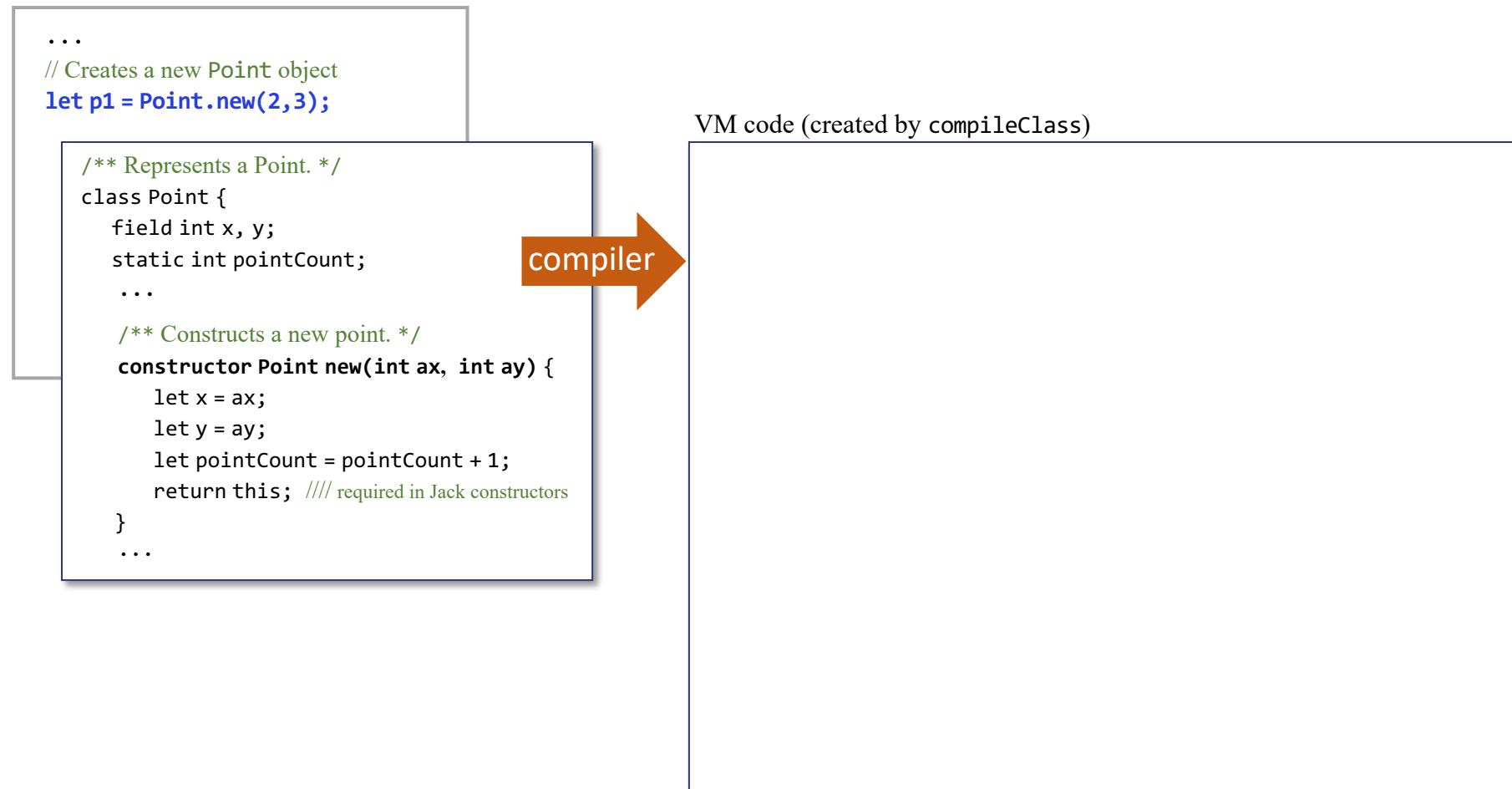
Source code

```
...  
// Creates a new Point object  
let p1 = Point.new(2,3);
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
  
    /** Constructs a new point. */  
    constructor Point new(int ax, int ay) {  
        let x = ax;  
        let y = ay;  
        let pointCount = pointCount + 1;  
        return this; /// required in Jack constructors  
    }  
    ...
```

compiler

VM code (created by compileClass)



# Compiling constructors

Source code

```
...
// Creates a new Point object
let p1 = Point.new(2,3);

/** Represents a Point */
class Point {
    field int x, y;
    static int pointCount;
    ...

    /** Constructs a new point. */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this; /// required in Jack constructors
    }
    ...
}
```

compiler

class-level  
symbol table

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

VM code (created by compileClass)

```
// class Point {
//   field int x, y;
//   static int pointCount;
//   /// compileClass builds the class-level symbol table
```

# Compiling constructors

Source code

```
...  
// Creates a new Point object  
let p1 = Point.new(2,3);
```

```
/** Represents a Point */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
  
    /** Constructs a new point. */  
    constructor Point new(int ax, int ay) {  
        let x = ax;  
        let y = ay;  
        let pointCount = pointCount + 1;  
        return this; /// required in Jack constructors  
    }  
    ...
```

compiler

VM code (created by compileClass)

```
// class Point {  
//     field int x, y;  
//     static int pointCount;  
//     /// compileClass builds the class-level symbol table  
  
//     constructor Point new(int ax, int ay);  
//     /// compileSubroutine builds the subroutine's symbol table, and notes that  
//     /// it is handling a constructor. It generates VM code that declares a VM function,  
//     /// allocates memory for the new object, and sets THIS to its base address:  
function Point.new 0  
    push constant 2  
    call Memory.alloc 1  
    pop pointer 0
```

class-level  
symbol table

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

constructor-level  
symbol table

| name | type | kind     | # |
|------|------|----------|---|
| ax   | int  | argument | 0 |
| ay   | int  | argument | 1 |

# Compiling constructors

Source code

```
...  
// Creates a new Point object  
let p1 = Point.new(2,3);
```

```
/** Represents a Point */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
  
    /** Constructs a new point. */  
    constructor Point new(int ax, int ay) {  
        let x = ax;  
        let y = ay;  
        let pointCount = pointCount + 1;  
        return this; /// required in Jack constructors  
    }  
    ...
```

compiler

VM code (created by compileClass)

```
// class Point {  
//     field int x, y;  
//     static int pointCount;  
//     /// compileClass builds the class-level symbol table  
  
//     constructor Point new(int ax, int ay);  
//     /// compileSubroutine builds the subroutine's symbol table, and notes that  
//     /// it is handling a constructor. It generates VM code that declares a VM function,  
//     /// allocates memory for the new object, and sets THIS to its base address:  
function Point.new 0  
    push constant 2  
    call Memory.alloc 1  
    pop pointer 0  
    /// CompileStatements handles the constructor's body  
    // let x = ax;  
    push argument 0  
    pop this 0  
    ...  
    // return this;  
    push pointer 0  
    return
```

class-level  
symbol table

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

constructor-level  
symbol table

| name | type | kind     | # |
|------|------|----------|---|
| ax   | int  | argument | 0 |
| ay   | int  | argument | 1 |

# Compiling constructors

Source code

```
...
// Creates a new Point object
let p1 = Point.new(2,3);

/** Represents a Point */
class Point {
    field int x, y;
    static int pointCount;
    ...

    /** Constructs a new point. */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this; /// required in Jack constructors
    }
    ...
}
```

compiler

VM code (created by compileClass)

```
// class Point {
//   field int x, y;
//   static int pointCount;
//   /// compileClass builds the class-level symbol table
//   constructor Point new(int ax, int ay);
//   /// compileSubroutine builds the subroutine's symbol table, and notes that
//   /// it is handling a constructor. It generates VM code that declares a VM function,
//   /// allocates memory for the new object, and sets THIS to its base address:
//   function Point.new 0
//     push constant 2
//     call Memory.alloc 1
//     pop pointer 0
//   /// CompileStatements handles the constructor's body
//   let x = ax;
//   push argument 0
//   pop this 0
//   ...
//   return this;
//   push pointer 0
//   return
```

C++, Java, Python, ...

In Jack, every constructor must end with the explicit statement `return this;`

All OOP compilers do the same thing, implicitly:

Before terminating the compiled constructor code, they inject low-level code that returns the address of the newly constructed object; That's how `p1` (in this example) ends up pointing at the new object.

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
  - Handling objects
    - Structures
    - Compiling objects construction
  - Compiling objects manipulations
- Handling arrays

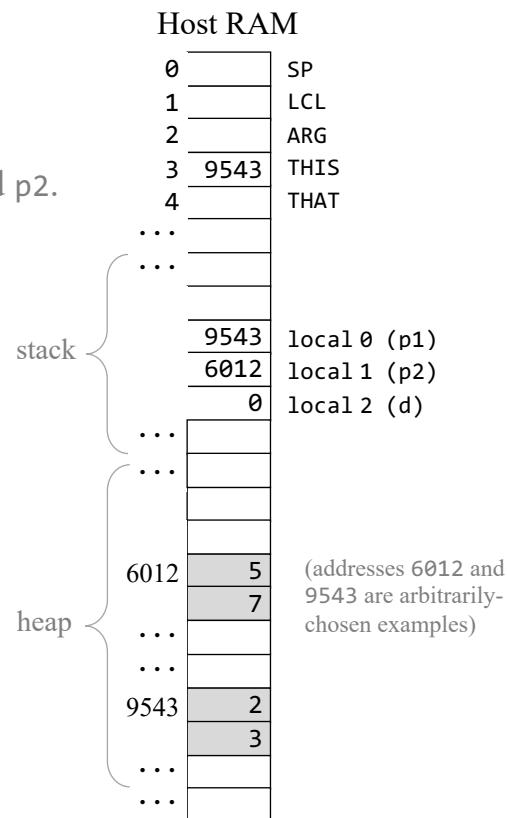
# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

We assume that the caller has already declared the local variables p1, p2, d, and constructed the objects p1 and p2.



# Compiling methods

---

source code

```
...  
let d = p1.distance(p2)  
...
```

Caller: “Applies a method to an object”

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

Callee: “Operates on the target object”

Typical client-server, OOP, object manipulation idioms;

What makes the magic happen? The compiler (next).

# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

compiler

VM code

```
...  
push local 0  
push local 1  
call Point.distance 2  
pop local 2  
...
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

name type kind #

| name | type  | kind  | # |
|------|-------|-------|---|
| p1   | Point | local | 0 |
| p2   | Point | local | 1 |
| d    | int   | local | 2 |

caller's symbol table

## When compiling a *method call*:

The compiler generates code that passes the target object as the *first argument* for the called method

This is the key compilation trick that facilitates the OO “object first” method calling idiom:

*objName.methodName(args)*

# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

compiler

VM code

```
...  
push local 0  
push local 1  
call Point.distance 2  
pop local 2  
...
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

name type kind #

| name | type  | kind  | # |
|------|-------|-------|---|
| p1   | Point | local | 0 |
| p2   | Point | local | 1 |
| d    | int   | local | 2 |

caller's symbol table

compiler

# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

compiler

VM code

```
...  
push local 0  
push local 1  
call Point.distance 2  
pop local 2  
...
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

name

type

kind

#

|    |       |       |   |
|----|-------|-------|---|
| p1 | Point | local | 0 |
| p2 | Point | local | 1 |
| d  | int   | local | 2 |

caller's symbol table

```
// class Point {  
//     field int x, y;  
//     static int pointCount;  
//     /// compileClass builds the class-level symbol table
```

compiler

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

class-level symbol table

# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

compiler

VM code

```
...  
push local 0  
push local 1  
call Point.distance 2  
pop local 2  
...
```

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

| name  | type  | kind     | # |
|-------|-------|----------|---|
| this  | Point | argument | 0 |
| other | Point | argument | 1 |
| dx    | int   | local    | 0 |
| dy    | int   | local    | 1 |

name type kind #

|    |       |       |   |
|----|-------|-------|---|
| p1 | Point | local | 0 |
| p2 | Point | local | 1 |
| d  | int   | local | 2 |

caller's symbol table

```
// class Point {  
//     field int x, y;  
//     static int pointCount;  
//     /// compileClass builds the class-level symbol table
```

```
// method int distance(Point other)  
// var int dx, dy;  
//     /// compileSubroutine builds the subroutine's symbol table, and notes  
//     /// that it is handling a method. It generates VM code that declares a VM  
//     /// function and sets THIS to the object on which the method was called.  
function Point.distance 2  
    push argument 0  
    pop pointer 0
```

class-level symbol table

method-level symbol table

When compiling a *method*: The compiler always adds the following entry to the method's symbol table:

<this, className, argument, 0>

# Compiling methods

source code

```
...  
let d = p1.distance(p2)  
...
```

compiler

```
/** Represents a Point. */  
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    /** Distance from this to the other point */  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx) +  
                        (dy*dy));  
    }  
    ...  
}
```

VM code

```
...  
push local 0  
push local 1  
call Point.distance 2  
pop local 2  
...
```

name type kind #

| name | type  | kind  | # |
|------|-------|-------|---|
| p1   | Point | local | 0 |
| p2   | Point | local | 1 |
| d    | int   | local | 2 |

caller's symbol table

compiler

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | this   | 0 |
| y          | int  | this   | 1 |
| pointCount | int  | static | 0 |

class-level symbol table

| name  | type  | kind     | # |
|-------|-------|----------|---|
| this  | Point | argument | 0 |
| other | Point | argument | 1 |
| dx    | int   | local    | 0 |
| dy    | int   | local    | 1 |

method-level symbol table

```
// class Point {  
//     field int x, y;  
//     static int pointCount;  
//     /// compileClass builds the class-level symbol table  
//     // method int distance(Point other)  
//     // var int dx, dy;  
//     // compileSubroutine builds the subroutine's symbol table, and notes  
//     // that it is handling a method. It generates VM code that declares a VM  
//     // function and sets THIS to the object on which the method was called.  
function Point.distance 2  
    push argument 0  
    pop pointer 0  
  
    // /// CompileStatements handles the method's body  
    // let dx = x - other.getx();  
    push this 0  
    push argument 1  
    call Point.getx 1  
    sub  
    pop local 0  
    ...  
    add  
    call Math.sqrt 1  
    return
```

# Lecture plan

---

## Compilation

✓ Handling variables

✓ Handling expressions

✓ Handling statements

✓ Handling objects

- Handling arrays

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
- Handling arrays
  - Creating arrays
  - Manipulating arrays

# Compiling array construction

## Source code

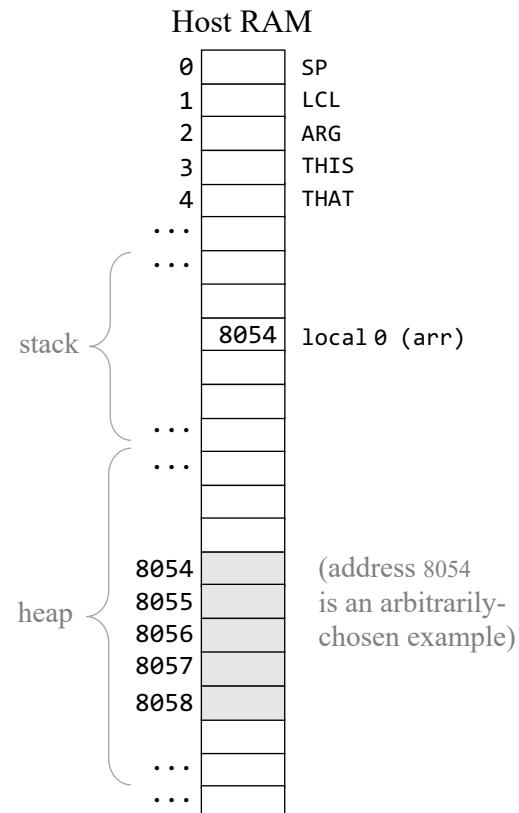
```
// Can appear in any class/subroutine:  
method foo() {  
    // Declares an array variable:  
    var Array arr;  
    ...  
    // Constructs the array:  
    let arr = Array.new(5);  
    ...
```

## Jack conventions:

Arrays are instances of the OS class `Array`

To construct an array, we call `Array.new`

(The `Array` class will be implemented later in the course, when we'll build the OS).



# Compiling array construction

## Source code

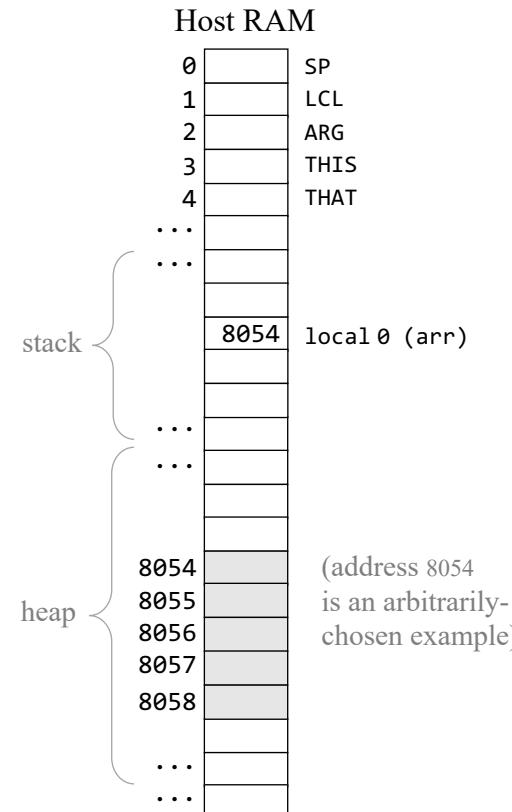
```
// Can appear in any class/subroutine:  
method foo() {  
    // Declares an array variable:  
    var Array arr;  
    ...  
    // Constructs the array:  
    let arr = Array.new(5);  
    ...
```

compile

## VM code

```
// var Array arr;  
/// compileSubroutine builds  
/// the subrout.'s symbol table,  
/// and adds to it variable arr.
```

| name | type  | kind  | # |
|------|-------|-------|---|
| arr  | Array | local | 0 |



# Compiling array construction

## Source code

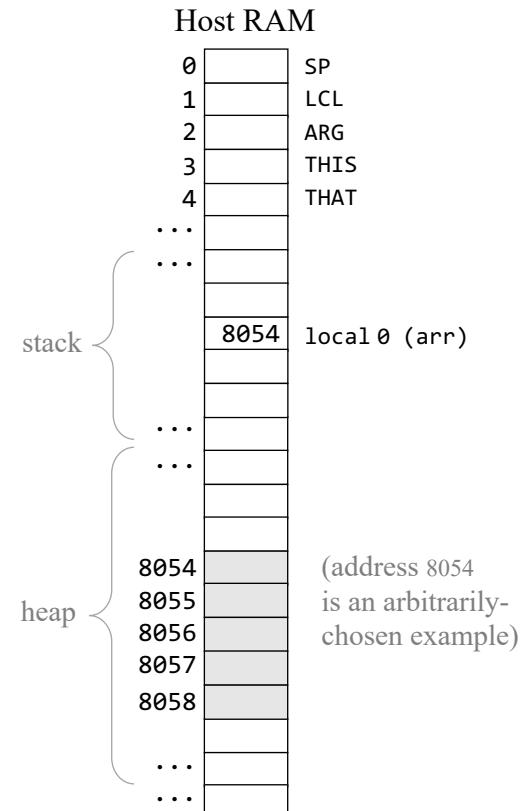
```
// Can appear in any class/subroutine:  
method foo() {  
    // Declares an array variable:  
    var Array arr;  
    ...  
    // Constructs the array:  
    let arr = Array.new(5);  
    ...
```

compile

## VM code

```
// var Array arr;  
/// compileSubroutine builds  
/// the subrout.'s symbol table,  
/// and adds to it variable arr.  
...  
// let arr = Array.new(5);  
push constant 5  
call Array.new 1  
pop local 0  
...
```

| name | type  | kind  | # |
|------|-------|-------|---|
| arr  | Array | local | 0 |



## Compiling array construction:

We simply compile the `let` statement  
(same as with calling an object constructor)

# Lecture plan

---

## Compilation

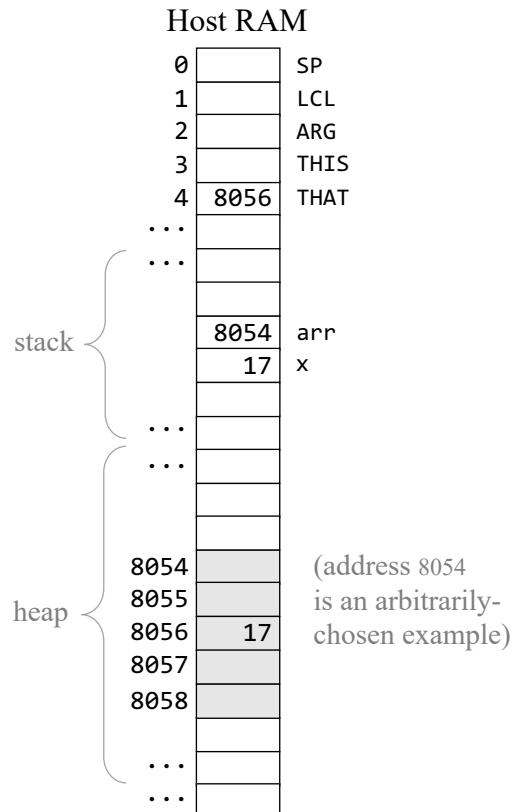
- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
  - Handling arrays
    - ✓ Creating arrays
    - Manipulating arrays

# Accessing array elements

## Examples

```
// let arr[2] = x  
push arr  
push 2  
add  
pop pointer 1 } Sets THAT to  
push x  
pop that 0 the address of  
the target  
array element
```

```
// let x = arr[2]  
push arr  
push 2  
add  
pop pointer 1 } same  
push that 0  
pop x
```



# Compiling array access

source code

```
let arr[i] = exp
```



VM code (generated by compileLet)

```
push arr  
push i  
add  
pop pointer 1 // THAT = arr + 2  
push exp  
pop that 0
```

unfortunately,  
there's a problem



Generalizing:

```
compileLet: (let varName[expression] = expression;)  
    output "push varName"  
    call compileExpression  
    output "add"  
    output "pop pointer 1"  
    call compileExpression  
    output "pop that 0"
```

# Compiling array access

source code (example)

```
let a[i] = b[j]
```



VM code (generated by compileLet)

```
push a  
push i  
add  
pop pointer 1  
  
// Now handle the right hand side  
push b  
push j  
add  
pop pointer 1  
...
```



unfortunately,  
there's a problem



Generalizing:

```
compileLet: (let varName[expression] = expression;)  
    output "push varName"  
    call compileExpression  
    output "add"  
    output "pop pointer 1"  
    call compileExpression  
    output "pop that 0"
```

# Compiling array access

source code (example)

```
let a[i] = b[j]
```



VM code (generated by compileLet)

```
push a  
push i  
add  
pop pointer 1  
  
// Now handle the right hand side  
push b  
push j  
add  
pop pointer 1  
...
```



unfortunately,  
there's a problem



Generalizing:

~~compileLet: (let varName[expression] = expression;)  
output "push varName"  
call compileExpression  
output "add"  
output "pop pointer 1"  
call compileExpression  
output "pop that 0"~~

# Compiling array access

source code (example)

```
let a[i] = b[j]
```



VM code (generated by compileLet)

```
push a  
push i  
add  
push b  
push j  
add  
  
pop pointer 1 // THAT = address of b[j]  
push that 0 // stack top = b[j]  
pop temp 0 // temp 0 = b[j]  
pop pointer 1 // THAT = address of a[i]  
push temp 0 // stack top = b[j]  
pop that 0 // a[i] = b[j]
```



What about compiling, say,

```
let a[a[i]] = a[b[a[b[j]]]] ?
```

No problem...

```
compileLet: (let varName[expression] = expression;)  
// (both expressions can contain array elements, of any depth)  
output "push varName"  
call compileExpression  
output "add"  
call compileExpression  
output "pop pointer 1"  
...  
output "pop that 0"
```

} 6 fixed push/pop commands  
at the bottom of the VM  
code shown above

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
- ✓ Handling arrays

## Implementation

→ Standard mapping

- Proposed design
- Project 11

# Standard mapping

---

So far we described:

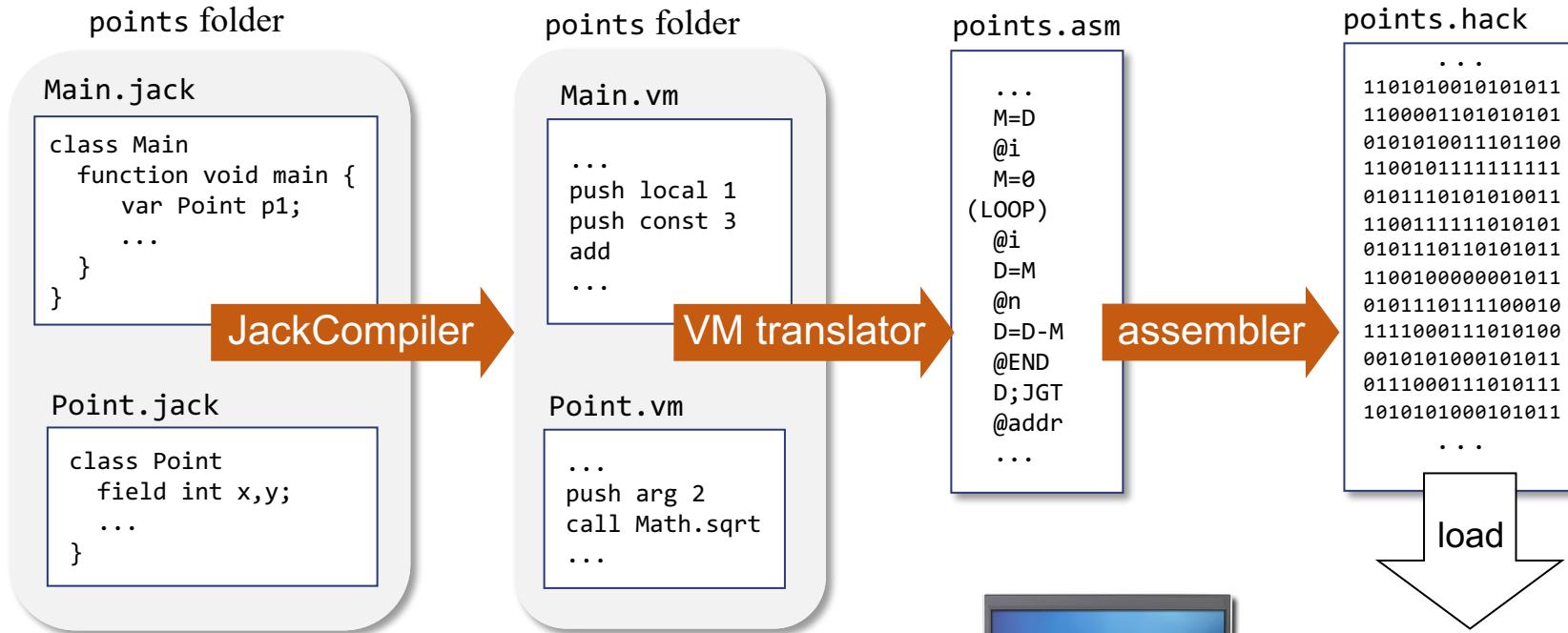
- General compilation techniques
- Compiling Jack code

We now turn to describe:

How to generate code for the specific VM language and OS of the Hack computer

(There will be some repetition of things already described).

# The big picture



A Jack program is a collection of one or more class files, in same folder (here, named `points`)

Each class file `ClassName.jack` is *compiled separately* into the file `ClassName.vm`

# Subroutines (implementation notes)

---

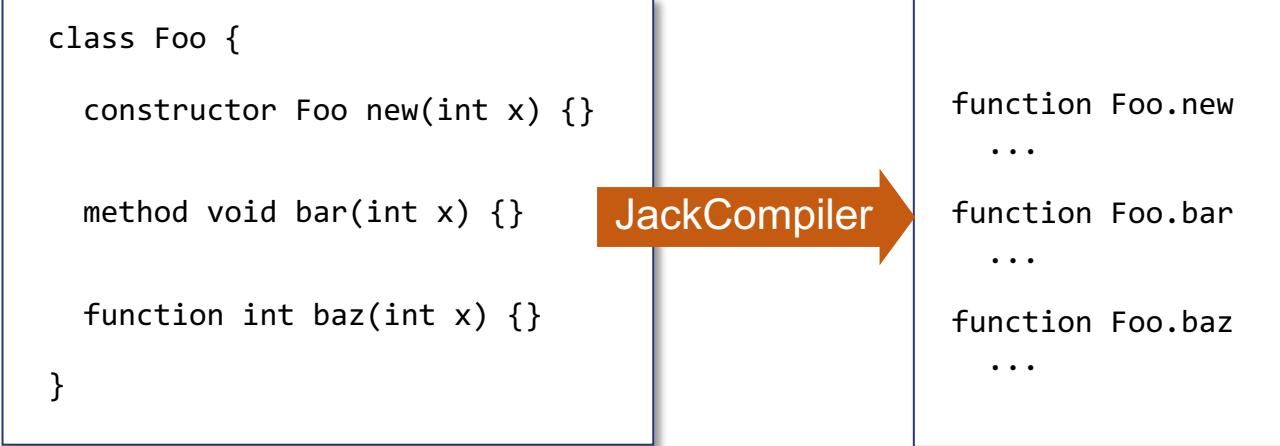
Foo.jack

```
class Foo {  
  
    constructor Foo new(int x) {}  
  
    method void bar(int x) {}  
  
    function int baz(int x) {}  
  
}
```

Foo.vm

```
function Foo.new  
...  
function Foo.bar  
...  
function Foo.baz  
...
```

JackCompiler



Each subroutine (constructor, function, method) *subName* in a file *ClassName.jack* is compiled into a VM function named *ClassName.subName*

# Constants (implementation notes)

---

The Jack language has four constants

`true` is represented in VM code as constant 1, followed by `neg`

`false` is represented in VM code as constant 0

`null` is represented in VM code as constant 0

`this` is represented in VM code as pointer 0

# Variables (implementation notes)

---

## Local variables

Are mapped on `local 0`, `local 1`, `local 2`, ...

## Argument variables

Are mapped on `argument 0`, `argument 1`, `argument 2`, ...

## Static variables

Are mapped on `static 0`, `static 1`, `static 2`, ...

## Field variables

Are mapped on `this 0`, `this 1`, `this 2`, ...

# Arrays (implementation notes)

---

## Access to array element $\text{arr}[i]$

Is implemented by generating VM code that realizes:

set pointer 1 to  $\text{arr} + i$

push / pop that 0

Implementation tip: There is never a need use that  $i$  for  $i$  greater than 0.

# Subroutine calls (implementation notes)

---

(Suppose that we are compiling the file `ClassName.jack` )

Compiling a constructor call or a function call `subName(exp1, exp2, ..., expn)`

The generated VM code pushes the expressions  $exp_1, exp_2, \dots, exp_n$  onto the stack,  
followed by the command `call ClassName.subName n`

Compiling a method call `obj.subName(exp1, exp2, ..., expn)`

The generated VM code pushes  $obj$  and then  $exp_1, exp_2, \dots, exp_n$  onto the stack,  
followed by the command `call ClassName.subName n+1`

If the called subroutine is `void`:

Just after the call, the generated VM code gets rid of the return value using the  
command `pop temp 0`

# Subroutines (implementation notes)

---

## When compiling a Jack method:

- The first entry in the method's symbol table must be a variable named `this` whose *type* is the name of the class to which the method belongs, *kind* is argument, and *index* is 0
- The generated VM code starts by setting pointer 0 to argument 0

## When compiling a Jack constructor:

- The generated VM code starts by:
  - Calling the OS function `Memory.alloc n`, where *n* is the number of fields in the class
  - Setting pointer 0 to `alloc`'s return value
- The generated VM code ends with `return pointer 0`

## When compiling a *void* function or a *void* method:

The generated VM code ends with `push constant 0` and then `return`

# The OS (implementation notes)

---

- The OS is written in Jack (later in the course)
- Implemented as a set of 8 compiled Jack classes:

Math.vm  
Memory.vm  
Screen.vm  
Output.vm  
Keyboard.vm  
String.vm  
Array.vm  
Sys.vm



Available in  
nand2tetris/tools/os

- Every OS subroutine *ClassName.subName* is available as a compiled VM function, and can be called by the generated VM code using the usual call command `call ClassName.subName nArgs`

## Usage

If you execute / test the generated VM code on the supplied VM emulator (recommended in this project), then there is no need to include or translate the OS files: The supplied VM emulator features a built-in implementation of all the OS subroutines.

If you wish to translate the generated VM code to assembly, copy all the `nand2tetris/tools/os/*.vm` files into same folder as the VM files generated by the compiler, and apply the translator to the folder.

# The OS (implementation notes)

---

Some OS routines come to play during compilation. In particular:

## The compiler handles...

Multiplication (\*) by calling the OS function `Math.multiply()`

Division (/) by calling the OS function `Math.divide()`

String constants by calling the OS constructor `String.new(length)`

String assignments like `x = "cc ... c"` by making a sequence of calls to `String.appendChar(c)`

Object construction by calling the OS function `Memory.alloc(size)`

Note: The compiler generates VM code, and the OS routines are implemented as VM functions.  
So, every call above is implemented using the regular VM command `call OSfunction nArgs`

# Lecture plan

---

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
- ✓ Handling arrays

## Implementation

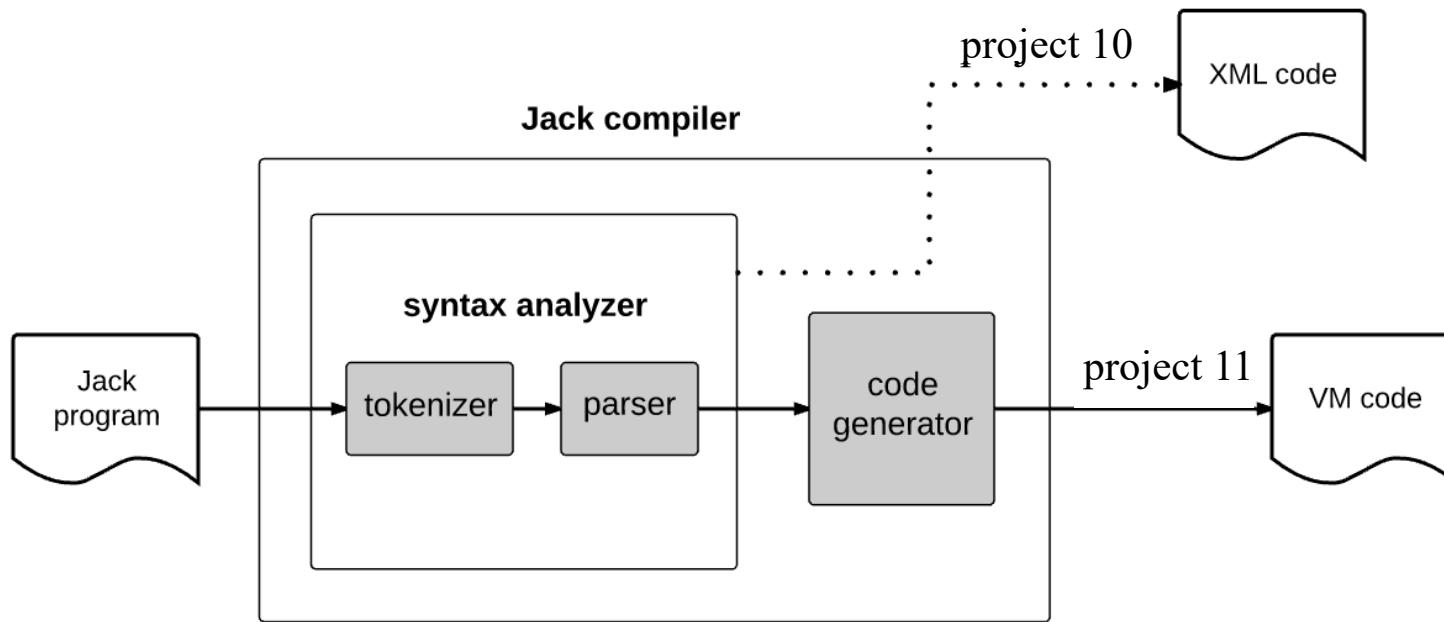
- ✓ Standard mapping

→ Proposed design

- Project 11

# The Jack compiler

---



Project 11: Morphing the syntax analyzer into a full scale compiler, using:

- `JackCompiler`: top-most (“main”) module
- `JackTokenizer`
- `CompilationEngine`
- `SymbolTable`
- `VMWriter`

# The Jack compiler

---

## Usage

```
$ JackCompiler input
```

*input:*    *fileName.jack*: name of a single file containing a Jack class, or

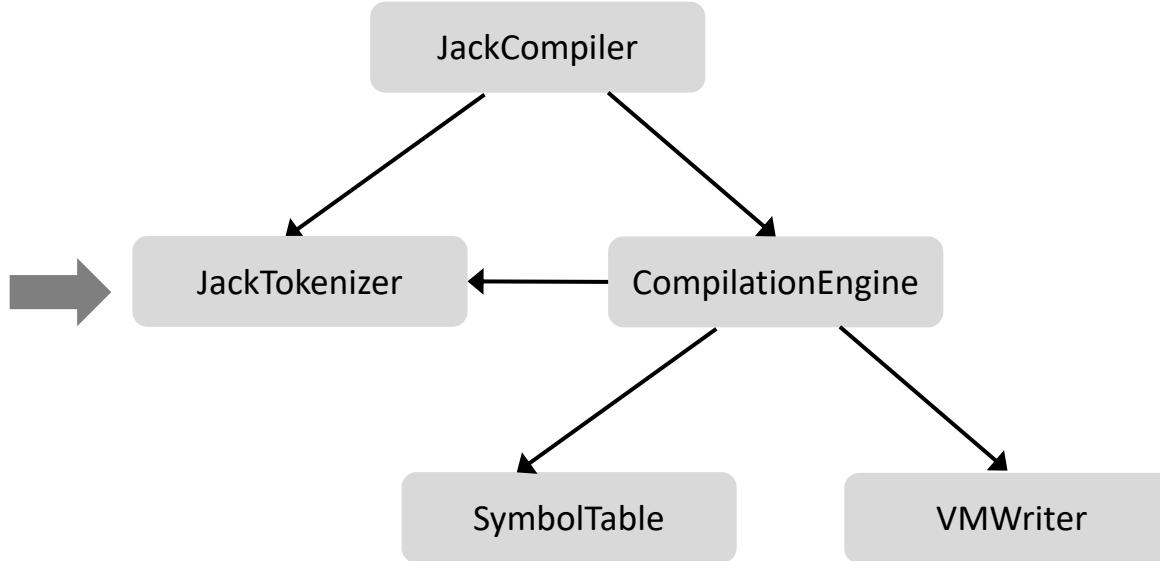
*folderName*:      name of a folder containing one or more .jack files

*output:*    If the input is a single file: *fileName.vm*

                If the input is a folder: One .vm file for every .jack file, stored in that folder

# The Jack compiler

---



- For each source `.jack` file, the compiler creates a `JackTokenizer` and an output `.vm` file
- Next, the compiler uses the `CompilationEngine` to write the VM code into the output `.vm` file.

# JackTokenizer

---

The `JackTokenizer` handles the compiler's input.

Provides services for:

- Skipping white space
- Getting the current token and advancing the input just beyond it
- Getting the type of the current token

(Developed in project 10)

# JackTokenizer (same as in project 10)

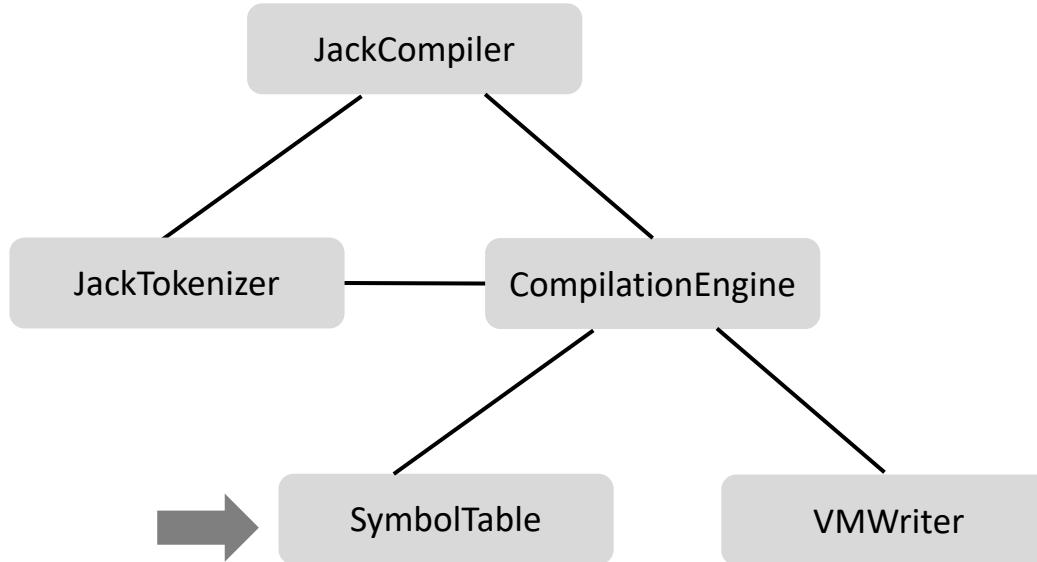
---

**JackTokeinizer:** Ignores all comments and white space, gets the next token, and advances the input just beyond it

| Routine                   | Arguments           | Returns   | Function   |
|---------------------------|---------------------|---|--|
| Constructor / initializer | input file / stream | –   | Opens the input .jack file / stream and gets ready to tokenize it.   |
| hasMoreTokens             | –                   | boolean   | Are there more tokens in the input?  |
| advance                   | –                   | –   | Gets the next token from the input, and makes it the current token.<br>This method should be called only if hasMoreTokens is true.<br>Initially there is no current token. |
| tokenType                 | –                   | KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST  | Returns the type of the current token, as a constant.  |
| keyword                   | –                   | CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS | Returns the keyword which is the current token, as a constant.<br>This method should be called only if tokenType is KEYWORD.   |
| symbol                    | –                   | char  | Returns the character which is the current token. Should be called only if tokenType is SYMBOL.  |
| identifier                | –                   | string  | Returns the string which is the current token. Should be called only if tokenType is IDENTIFIER.   |
| intval                    | –                   | int   | Returns the integer value of the current token. Should be called only if tokenType is INT_CONST.   |
| stringVal                 | –                   | string  | Returns the string value of the current token, without the opening and closing double quotes. Should be called only if tokenType is STRING_CONST.                          |

# The Jack compiler

---



# SymbolTable

Jack source code

```
class Point {  
    field int x, y;  
    static int pointCount;  
    ...  
    method int distance(Point other) {  
        var int dx, dy;  
        let dx = x - other.getx();  
        let dy = y - other.gety();  
        return Math.sqrt((dx*dx)+(dy*dy));  
    }  
    ...  
}
```

| name       | type | kind   | # |
|------------|------|--------|---|
| x          | int  | field  | 0 |
| y          | int  | field  | 1 |
| pointCount | int  | static | 0 |

class-level  
symbol table

| name  | type  | kind     | # |
|-------|-------|----------|---|
| this  | Point | argument | 0 |
| other | Point | argument | 1 |
| dx    | int   | local    | 0 |
| dy    | int   | local    | 1 |

subroutine-level  
symbol table

## Implementation notes

- symbol table = instance of a `SymbolTable` class (next slide)
- When compiling a Jack class, we need one class-level symbol table and one subroutine-level symbol table
- When we start compiling a subroutine, we reset the latter table
- Each variable is assigned a running index within its *scope* (table) and *kind*.  
The index starts at 0, increments by 1 after each time a new symbol is added to the table, and is reset to 0 when starting a new scope (table)
- When compiling an error-free Jack code, each identifier not found in the symbol tables can be assumed to be either a *subroutine name* or a *class name*.

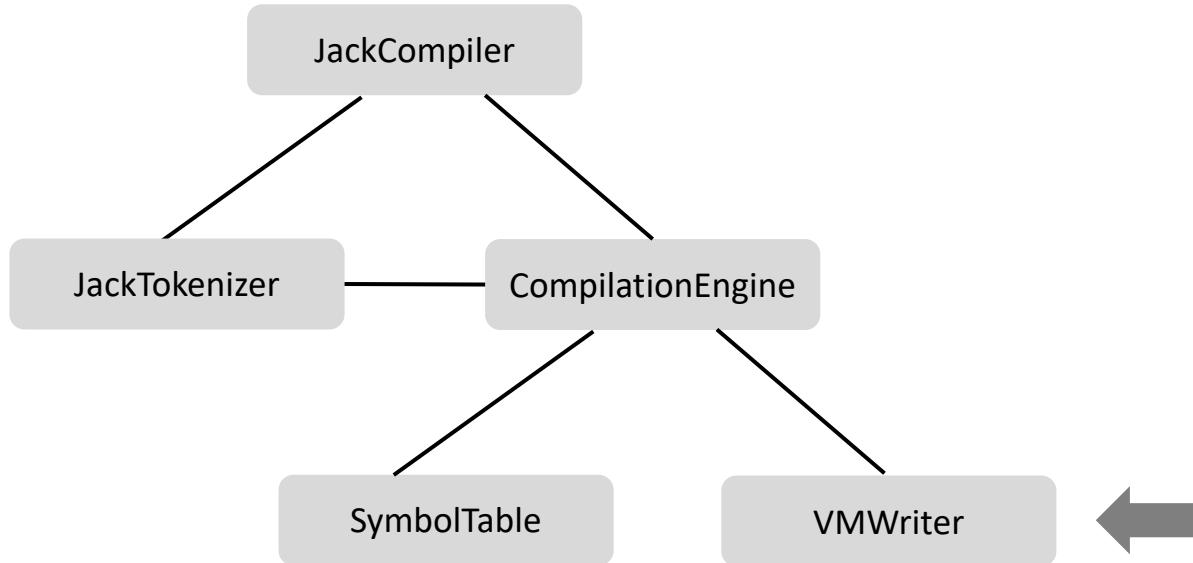
# SymbolTable

---

| <b>Routine</b>            | <b>Arguments</b>  | <b>Returns</b>                  | <b>Function</b>  |
|---------------------------|---|---------------------------------|--|
| Constructor / initializer | —   | —                               | Creates a new symbol table.  |
| reset                     | —   | —                               | Empties the symbol table, and resets the four indexes to 0. Should be called when starting to compile a subroutine declaration.                    |
| define                    | name (string)<br>type (string)<br>kind (STATIC, FIELD, ARG, or VAR) | —                               | Defines (adds to the table) a new variable of the given name, type, and kind. Assigns to it the index value of that kind, and adds 1 to the index. |
| varCount                  | kind (STATIC, FIELD, ARG, or VAR)                                   | int                             | Returns the number of variables of the given kind already defined in the table.  |
| kindOf                    | name (string)   | (STATIC, FIELD, ARG, VAR, NONE) | Returns the kind of the named identifier. If the identifier is not found, returns NONE.  |
| typeof                    | name (string)   | string                          | Returns the type of the named variable.  |
| indexof                   | name (string)   | int                             | Returns the index of the named variable.   |

# The Jack compiler

---



# VMWriter

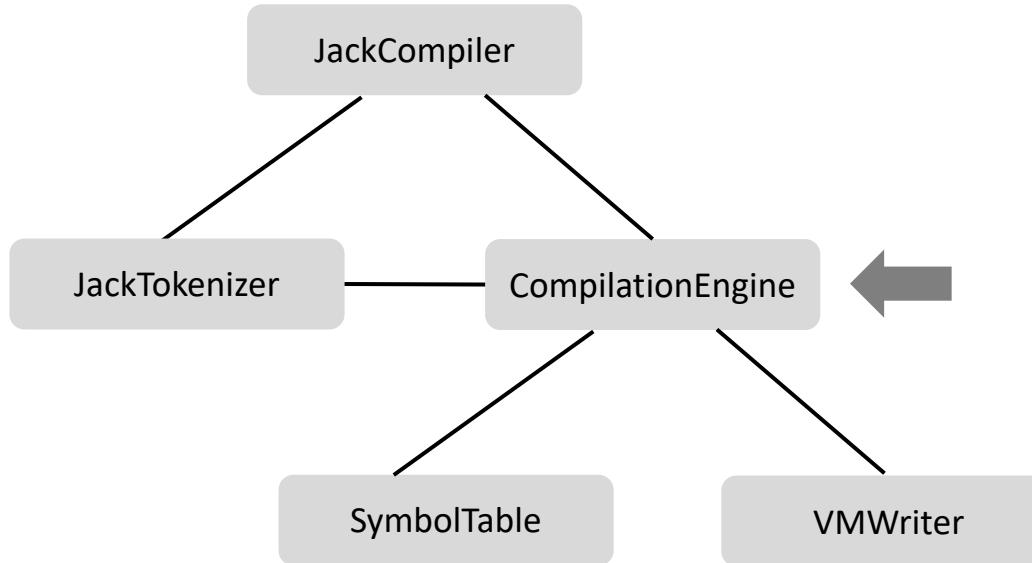
---

| <b>Routine</b>            | <b>Arguments</b>  | <b>Returns</b> | <b>Function</b>  |
|---------------------------|---|----------------|--|
| Constructor / initializer | output file / stream  | —              | Creates a new output .vm file / stream, and prepares it for writing. |
| writePush                 | segment (CONSTANT, ARGUMENT, LOCAL, STATIC, THIS, THAT, POINTER, TEMP)<br>index (int) | —              | Writes a VM push command.  |
| writePop                  | segment (ARGUMENT, LOCAL, STATIC, THIS, THAT, POINTER, TEMP)<br>index (int)           | —              | Writes a VM pop command.   |
| writeArithmetic           | command (ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT)                                     | —              | Writes a VM arithmetic-logical command.                              |
| writeLabel                | label (string)  | —              | Writes a VM label command.   |
| writeGoto                 | label (string)  | —              | Writes a VM goto command.  |
| writeIf                   | label (string)  | —              | Writes a VM if-goto command.   |
| writeCall                 | name (string)<br>nArgs (int)  | —              | Writes a VM call command.  |
| writeFunction             | name (string)<br>nVars (int)  | —              | Writes a VM function command.  |
| writeReturn               | —   | —              | Writes a VM return command.  |
| close                     | —   | —              | Closes the output file / stream.                                     |

A simple module that writes individual VM commands to the output .vm file.

# The Jack compiler

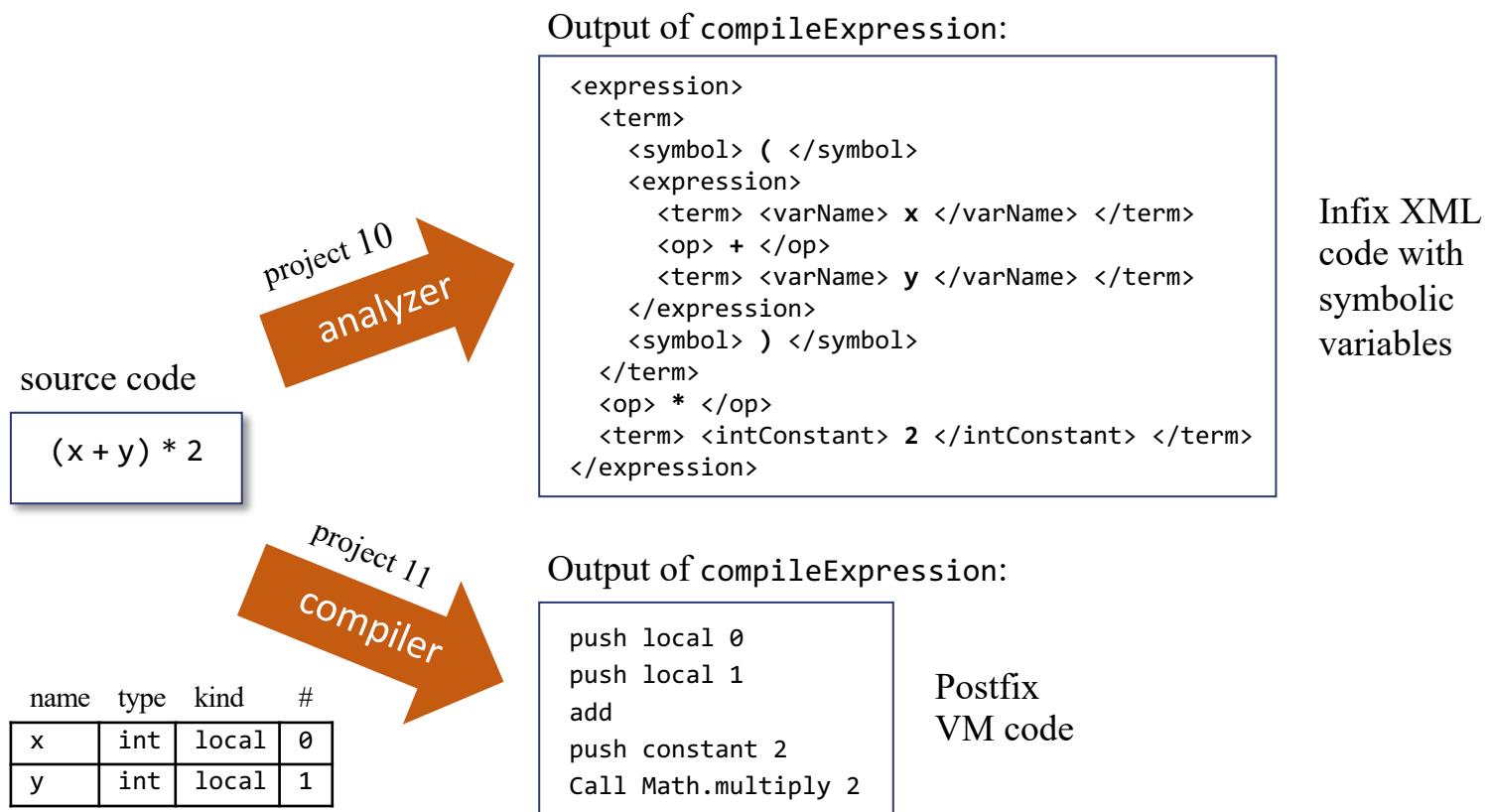
---



# CompilationEngine

The `CompilationEngine` of the *compiler* (project 11) and the *syntax analyzer* (project 10) have the same design and API: a set of `compileXXX` methods

However, the `compileXXX` methods generate different outputs. For example:



# CompilationEngine

---

- Gets its input from a `JackTokenizer` and writes its output using the `VMWriter`
- Organized as a set of `compilexxx` routines, *xxx* being a syntactic element in the Jack grammar:
  - Each `compilexxx` routine should read *xxx* from the input, `advance()` the input exactly beyond *xxx*, and emit to the output VM code effecting the semantics of *xxx*
  - `compilexxx` is called only if *xxx* is the current syntactic element
  - If *xxx* is part of an expression and thus has a value, the emitted VM code computes the value and leaves it at the top of the VM's stack

# CompilationEngine (same API as in project 10)

---

| <i>Routine</i>                     | <i>Arguments</i>                            | <i>Returns</i> | <i>Function</i>  |
|------------------------------------|---|----------------|--|
| Constructor / initializer          | Input file / stream<br>Output file / stream |                | Creates a new compilation engine with the given input and output.<br><br>The next routine called must be <code>compileClass</code> . |
| <code>compileClass</code>          | —   | —              | Compiles a complete class.   |
| <code>compileClassVarDec</code>    | —   | —              | Compiles a static variable declaration, or a field declaration.  |
| <code>compileSubroutine</code>     | —   | —              | Compiles a complete method, function, or constructor.  |
| <code>compileParameterList</code>  | —   | —              | Compiles a (possibly empty) parameter list. Does not handle the enclosing parentheses tokens ( and ).                                |
| <code>compileSubroutineBody</code> | —   | —              | Compiles a subroutine's body.  |
| <code>compileVarDec</code>         | —   | —              | Compiles a var declaration.  |
| <code>compileStatements</code>     | —   | —              | Compiles a sequence of statements. Does not handle the enclosing curly bracket tokens { and }.                                       |

# CompilationEngine (same API as in project 10)

---

| <i>Routine</i>        | <i>Arguments</i> | <i>Returns</i> | <i>Function</i>  |
|-----------------------|------------------|----------------|--|
| compileLet            | —                | —              | Compiles a <code>let</code> statement.   |
| compileIf             | —                | —              | Compiles an <code>if</code> statement, possibly with a trailing <code>else</code> clause.  |
| compileWhile          | —                | —              | Compiles a <code>while</code> statement.   |
| compileDo             | —                | —              | Compiles a <code>do</code> statement.  |
| compileReturn         | —                | —              | Compiles a <code>return</code> statement.  |
| compileExpression     | —                | —              | Compiles an expression.  |
| compileTerm           | —                | —              | Compiles a <i>term</i> . If the current token is an <i>identifier</i> , the routine must resolve it into a <i>variable</i> , an <i>array entry</i> , or a <i>subroutine call</i> . A single lookahead token, which may be <code>[</code> , <code>(</code> , or <code>.</code> , suffices to distinguish between the possibilities. Any other token is not part of this term and should not be advanced over. |
| compileExpressionList | —                | int            | Compiles a (possibly empty) comma-separated list of expressions. Returns the number of expressions in the list.  |

# Lecture plan

---

## Compilation

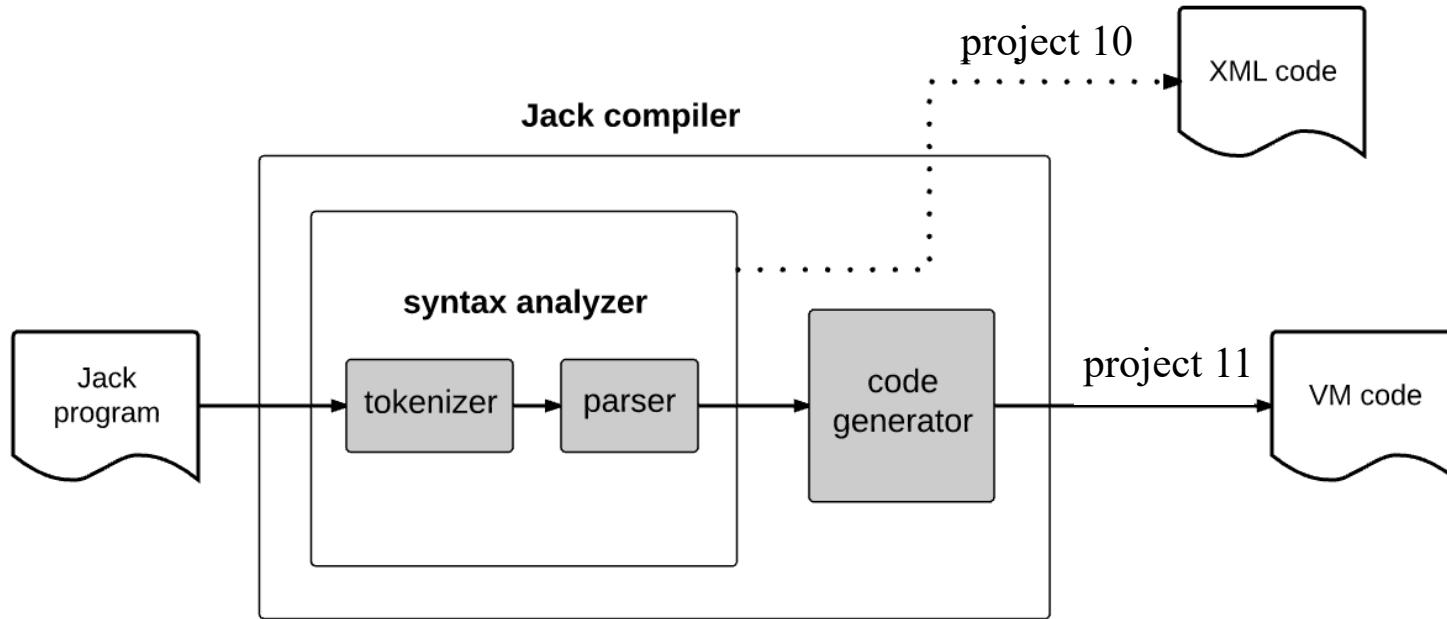
- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
- ✓ Handling arrays

## Implementation

- ✓ Standard mapping
- ✓ Proposed design

→ Project 11

# Compiler development roadmap



Project 11: Extend the syntax analyzer into a full-scale compiler

Stage 0: Syntax analyzer (done)

Stage 1: Symbol table handling

Stage 2: Code generation.

# Symbol table

Output of the syntax analyzer (project 10)

```
...  
<expression>  
  <term>  
    <identifier> count </identifier>  
  </term>  
  <symbol> < </symbol>  
  <term>  
    <intConstant> 100 </intConstant>  
  </term>  
</expression>  
...
```

In the syntax analyzer built in project 10,  
identifiers were handled by outputting  
`<identifier> identifier </identifier>`

Extend the handling of identifiers:

- Output the identifier's category: var, argument, static, field, class, subroutine
- If the identifier's category is var, argument, static, field,  
output also the running index assigned to this variable in the symbol table
- Output whether the identifier is being defined, or being used

Implementation

1. Implement the `SymbolTable` API
2. Extend the syntax analyzer developed in project 10 with the outputs described above  
(plan and use your own output/tags format)
3. Test the extended syntax analyzer by running it on the test programs given in project 10.

# Compiler development roadmap

---

Project 11: Extend the syntax analyzer into a full-scale compiler

- ✓ Stage 0: Syntax analyzer (done)
- ✓ Stage 1: Symbol table handling
- Stage 2: Code generation.

# Code generation

---

## Test programs

Seven  
ConvertToBin  
Square  
Average  
Pong  
ComplexArrays

Test your evolving compiler on the supplied test programs, in the shown order

Each test program is designed to test some of your compiler's capabilities.

## For each test program:

1. Use your compiler to compile the program folder
2. Inspect the generated code;  
If there's a problem, fix your compiler and go to stage 1
3. Load the folder into the VM emulator
4. Run the compiled program, inspect the results
5. If there's a problem, fix your compiler and go to stage 1.

# Test program: Seven

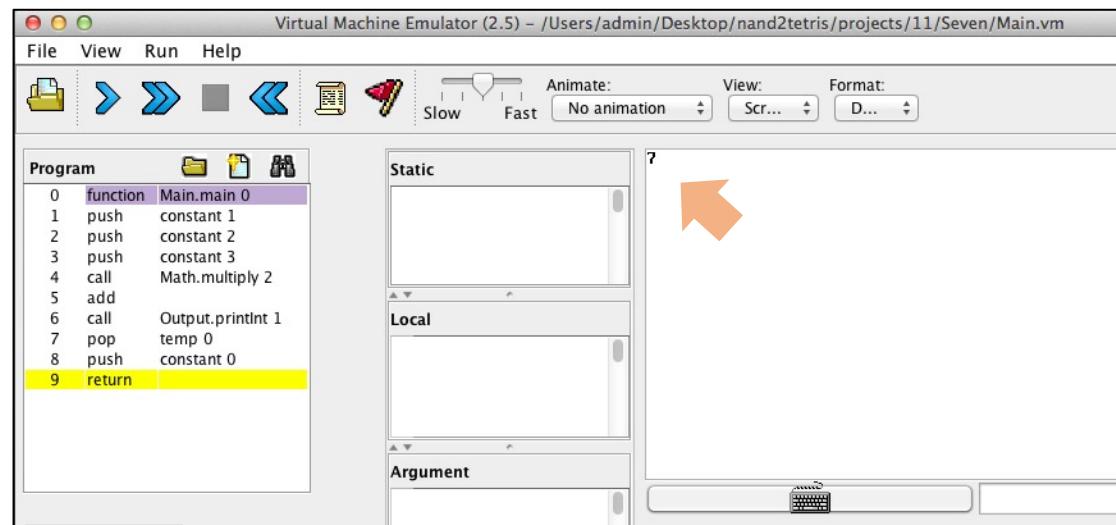
projects/11/Seven/Main.jack

```
/** Computes the value of 1 + (2 * 3)
 * and prints the result at the top-left
 * corner of the screen. */
class Main {
    function void main() {
        do Output.printInt(1 + (2 * 3));
        return;
    }
}
```

JackCompiler

projects/11/Seven/Main.vm

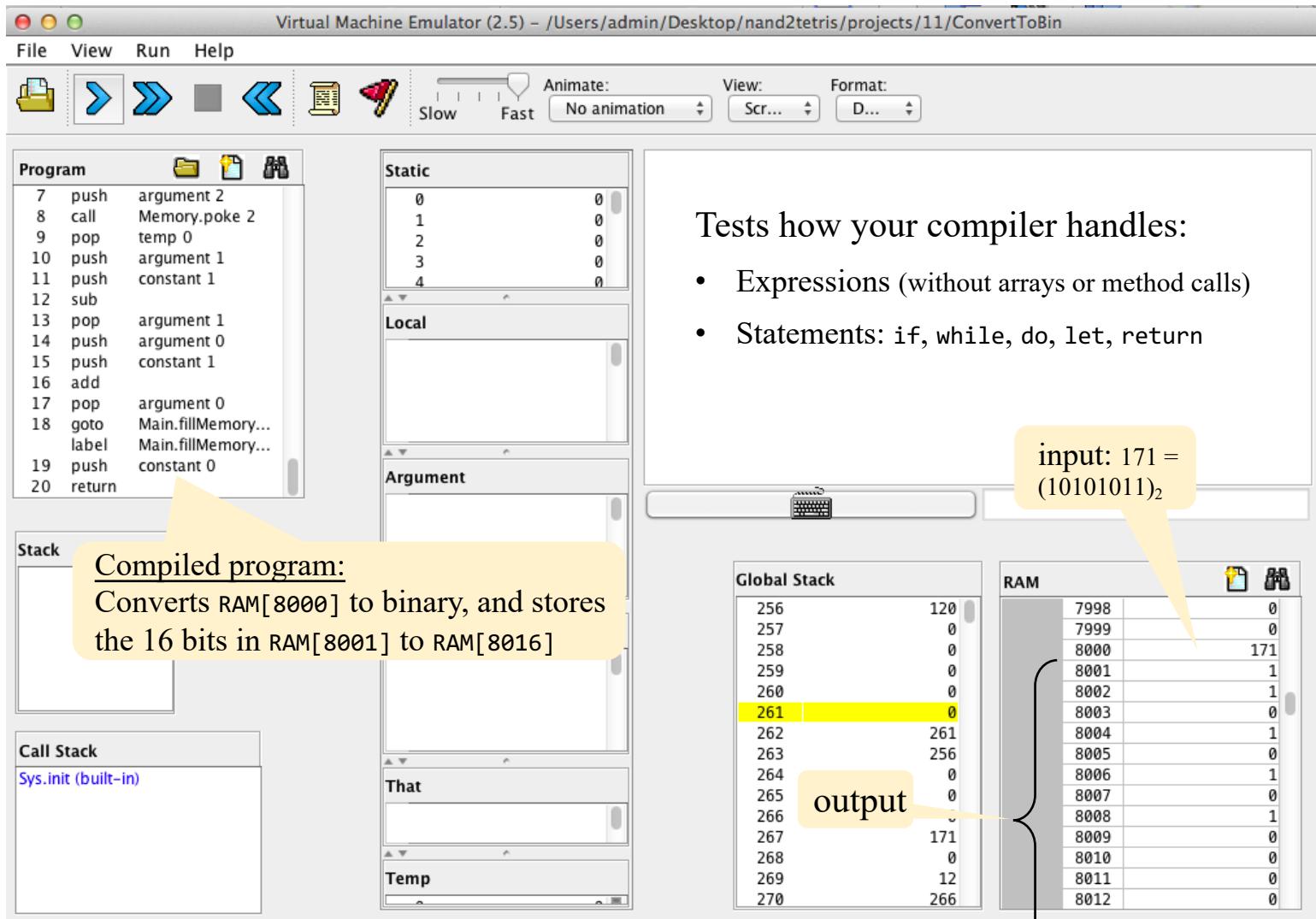
```
function Main.main 0
    push constant 1
    push constant 2
    push constant 3
    call Math.multiply 2
    add
    call Output.printInt 1
    pop temp 0
    push constant 0
    return
```



Tests how your compiler handles:

- A simple program
- An arithmetic expression involving constants only
- A do statement
- A return statement

# Test program: Decimal-to-binary conversion



Tests how your compiler handles:

- Expressions (without arrays or method calls)
  - Statements: `if`, `while`, `do`, `let`, `return`

input: 171 =  
 $(10101011)_2$

Compiled program:  
Converts RAM[8000] to binary, and stores  
the 16 bits in RAM[8001] to RAM[8016]

## output

# Test program: Decimal-to-binary conversion

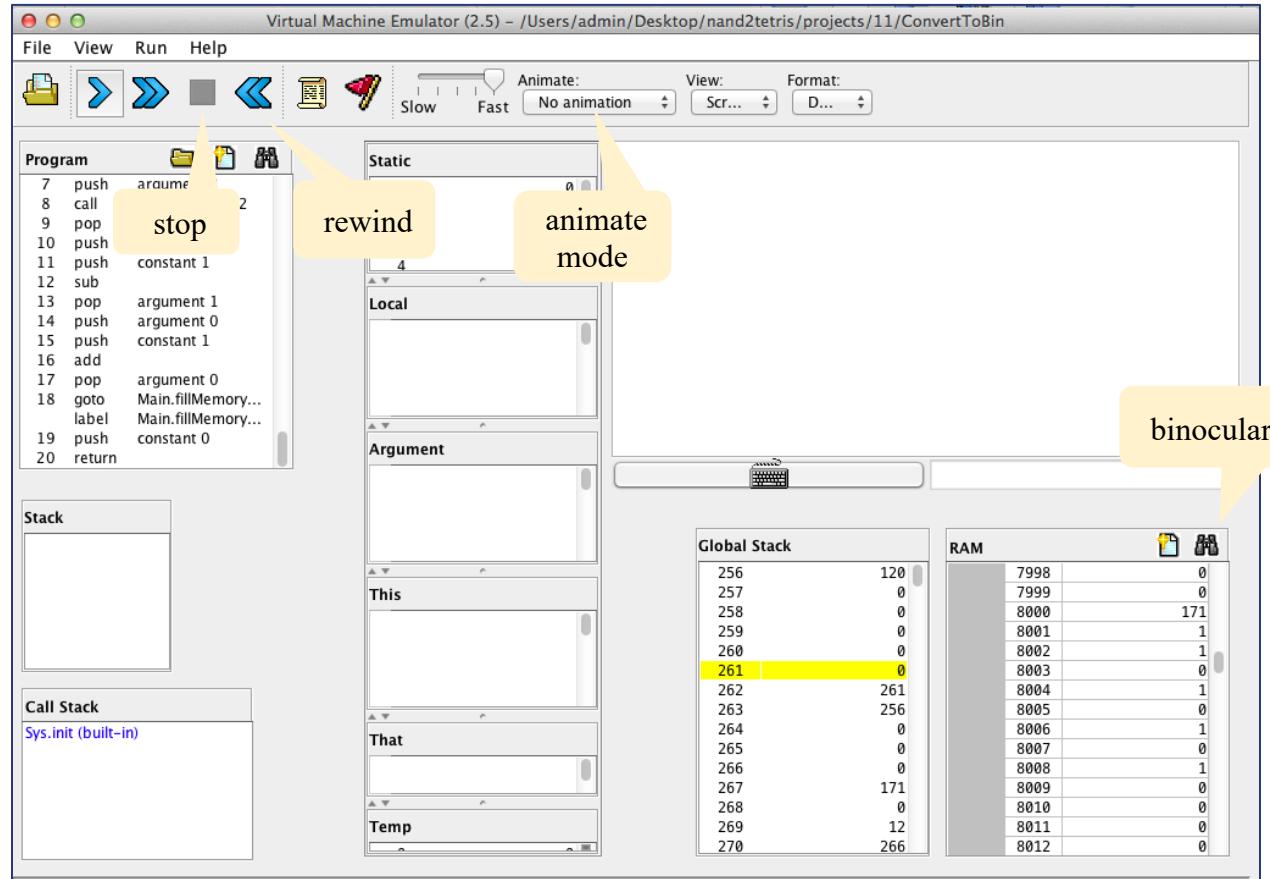
---

```
class Main {  
    // Converts RAM[8000] to binary, putting the resulting bits in RAM[8001]..RAM[8016]  
    function void main() {  
        var int value;  
        do Main.fillMemory(8001, 16, -1); // sets RAM[8001]..RAM[8016] to -1  
        let value = Memory.peek(8000); // gets the input from RAM[8000]  
        do Main.convert(value); // performs the conversion  
        return;  
    }  
  
    // Fills 'length' consecutive memory locations with 'value',  
    // starting at 'startAddress'.  
    function void fillMemory(int startAddress, int length, int value) { // code omitted }  
  
    // Converts the value to binary, and puts the result in RAM[8001]..RAM[8016] */  
    function void convert(int value) { // code omitted }  
  
    // Some more private functions (omitted)  
}
```

Tests how your compiler handles:

- Expressions (without arrays, without method calls)
- Statements: `if`, `while`, `do`, `let`, `return`

# Test program: Decimal-to-binary conversion

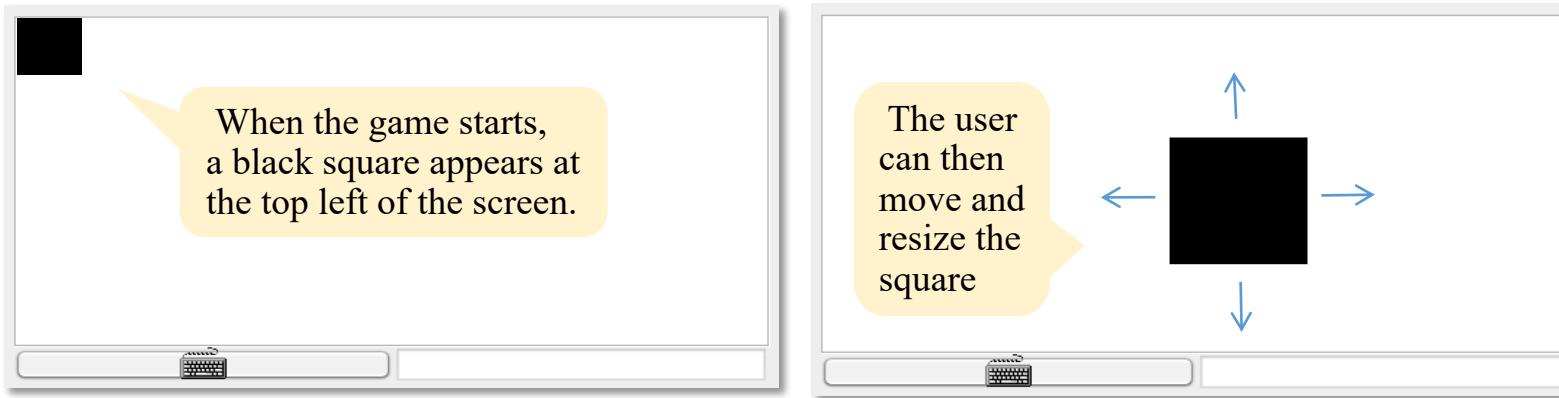


## Testing tips:

- Use the “binocular” control
- Note that the “rewind” control erases the RAM
- Note that you cannot enter input into the RAM in “no animation” mode
- To see the program’s results (RAM state), click the “stop” control

# Test program: Square

---



Tests how your compiler handles object-oriented features of the Jack language:

- Constructors
- Methods
- Expressions that include method calls.

# Test program: Square

projects/11/Square/Square.jack (showing only method signatures)

```
/** Represents a graphical square object */
class Square {

    /** Constructs a new square with a given location and size */
    constructor Square new(int Ax, int Ay, int Asize)

    /** Disposes this square */
    method void dispose()

    /** Draws the square on the screen */
    method void draw()

    /** Erases the square from the screen */
    method void erase()

    /** Increments the square's size by 2 pixels */
    method void incSize()

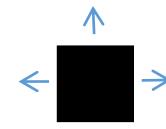
    /** Decrementsthe square's size by 2 pixels */
    method void decSize()

    /** Moves up by 2 pixels */
    method void moveUp()

    /** Moves down by 2 pixels */
    method void moveDown()

    /** Moves left by 2 pixels */
    method void moveLeft()

    /** Moves right by 2 pixels */
    method void moveRight()
}
```



SquareGame.jack

```
/** Represents a square game */
class SquareGame {

    field Square square; // the square
    field int direction; // the square's direction:
                        // 0=none, 1=up, 2=down,
                        // 3=left, 4=right
```

constructor SquareGame new() {

```
    let square = Square.new(0, 0, 30);
    let direction = 0;
    return this;
}
```

method void dispose() {

```
    do square.dispose();
    do Memory.deAlloc(this);
    return;
}
```

...

Main.jack

```
/** Main class of the square game. */
class Main {

    /** Initializes and starts a new game */
    function void main() {
        var SquareGame game;

        let game = SquareGame.new();
        do game.run();
        do game.dispose();
        return;
    }
}
```

# Test program: Average

```
/** Computes the average of a sequence of integers */
class Main {
    function void main() {
        var Array a;
        var int length;
        var int i, sum;

        let length = Keyboard.readInt("How many numbers? ");
        let a = Array.new(length);
        let i = 0;

        while (i < length) {
            let a[i] = Keyboard.readInt("Enter the next number: ");
            let i = i + 1;
        }

        let i = 0; let sum = 0;

        while (i < length) {
            let sum = sum + a[i];
            let i = i + 1;
        }

        do Output.printString("The average is: ");
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}
```

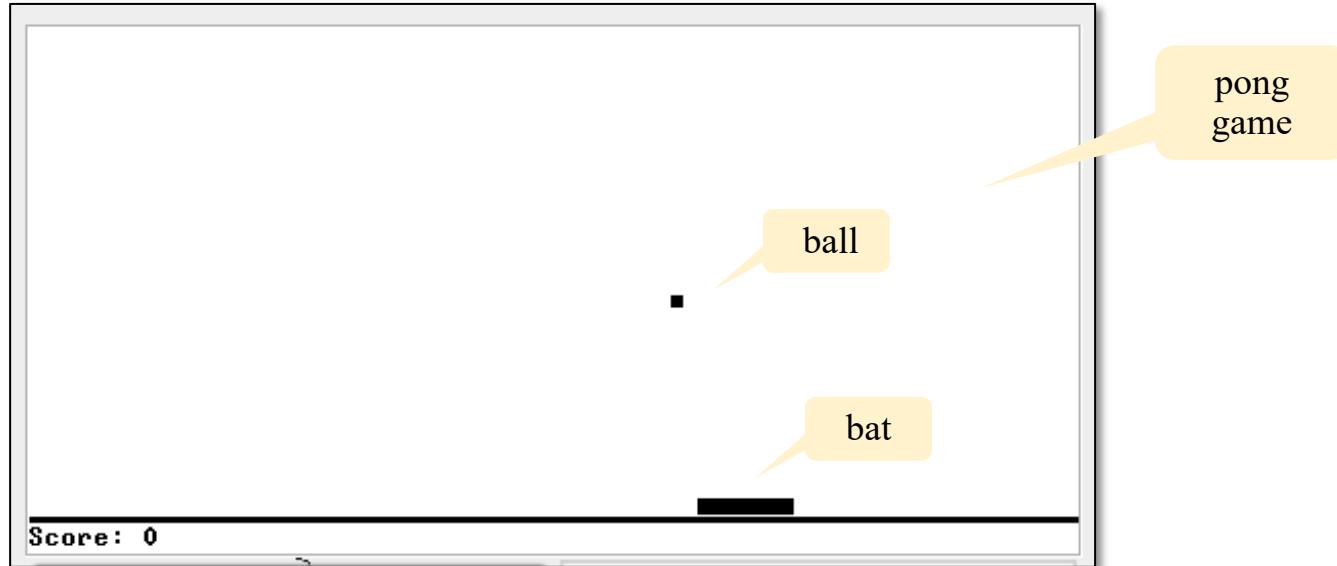
Tests how your compiler handles:

- Arrays
- Strings

```
How many numbers? 3
Enter the next number: 10
Enter the next number: 20
Enter the next number: 30
The average is: 20
```

# Test program: Pong

---



Tests how your compiler handles a complete object-oriented application, including the handling of objects and static variables.

# Test program: Pong

projects/11/Pong/Ball.jack

```
/** A graphic ball, with methods for drawing, erasing  
 * and moving on the screen. */
```

```
class Ball {  
  
    // Ball's  
    field int  
  
    // Distance  
    field int  
  
    // Used  
    field int  
    field boolean  
  
    // Location  
    field int  
  
    // last  
    field int  
  
    /** Constructor  
     * located at  
    construct  
  
    ...  
  
    // More  
}  
  
}  
  
    // More Ball methods
```

Bat.jack

```
    /** A graphic paddle with methods for drawing,  
     * erasing, moving left and right changing width. */
```

```
class Bat {  
  
    // Screen location  
    field int x, y;  
  
    // Bat's width and height  
    field int width, height;  
  
    // Bat's direction of movement  
    field int direction; // 1 = up, -1 = down  
  
    /** Constructs a new bat  
     * at position (x,y) with width w and height h.  
     * direction is the initial direction of movement.  
     */  
    constructor Bat new(int Ax, int Ay, int Aw, int Ah)  
        let x = Ax;  
        let y = Ay;  
        let width = Aw;  
        let height = Ah;  
        let direction = 2;  
        do show();  
        return this;  
    }  
  
    ...  
  
    // More Bat methods
```

PongGame.jack

```
    /** Pong game */  
class PongGame {  
  
    static PongGame instance; // the game  
    field Bat bat; // the bat  
    field Ball ball; // the ball  
    ...  
    /** Creates an instance of a PongGame */  
    function void newInstance() {  
        let instance = PongGame.new();  
        return;
```

Main.jack

```
    }  
    ...  
    /** Runs the game */  
    method void run()  
        var char key;  
        while (~exit)  
            // waits for user input  
            while ((key = readchar()) != '#')  
                let k = key;  
                do batMove(k);  
                do moveBall();  
            }  
            if (key == '#')  
                do quit();  
        }  
        ...  
    }  
}
```

# Test program: ComplexArrays

projects/11/ComplexArrays/Main.jack

```
class Main {  
    function void main() {  
        var Array a, b, c;  
        let a = Array.new(10);  
        let b = Array.new(5);  
        ...  
        // Fills the arrays with some data (omitted)  
        ...  
        // Manipulates the arrays using some complex index expressions  
        let a[b[a[3]]] = a[a[5]] * b[7 - a[3] - Main.double(2) + 1];  
        ...  
        // Prints the expected and the actual values of a[b[a[3]]]  
        ...  
    }  
  
    // A trivial function that tests how the compiler handles a subroutine  
    // call within an expression that evaluates to an array index  
    function int double(int a) {  
        return a * 2;  
    }  
  
    // Creates a two dimensional array  
    function void fill(Array a, int size) {  
        while (size > 0) {  
            let size = size - 1;  
            let a[size] = Array.new(3);  
        }  
        return;  
    }  
}
```

Tests how your compiler handles array manipulations using index expressions that include complex array references

```
Test 1 - Required result: 5, Actual result: 5  
Test 2 - Required result: 40, Actual result: 40  
Test 3 - Required result: 0, Actual result: 0  
Test 4 - Required result: 77, Actual result: 77  
Test 5 - Required result: 110, Actual result: 110
```