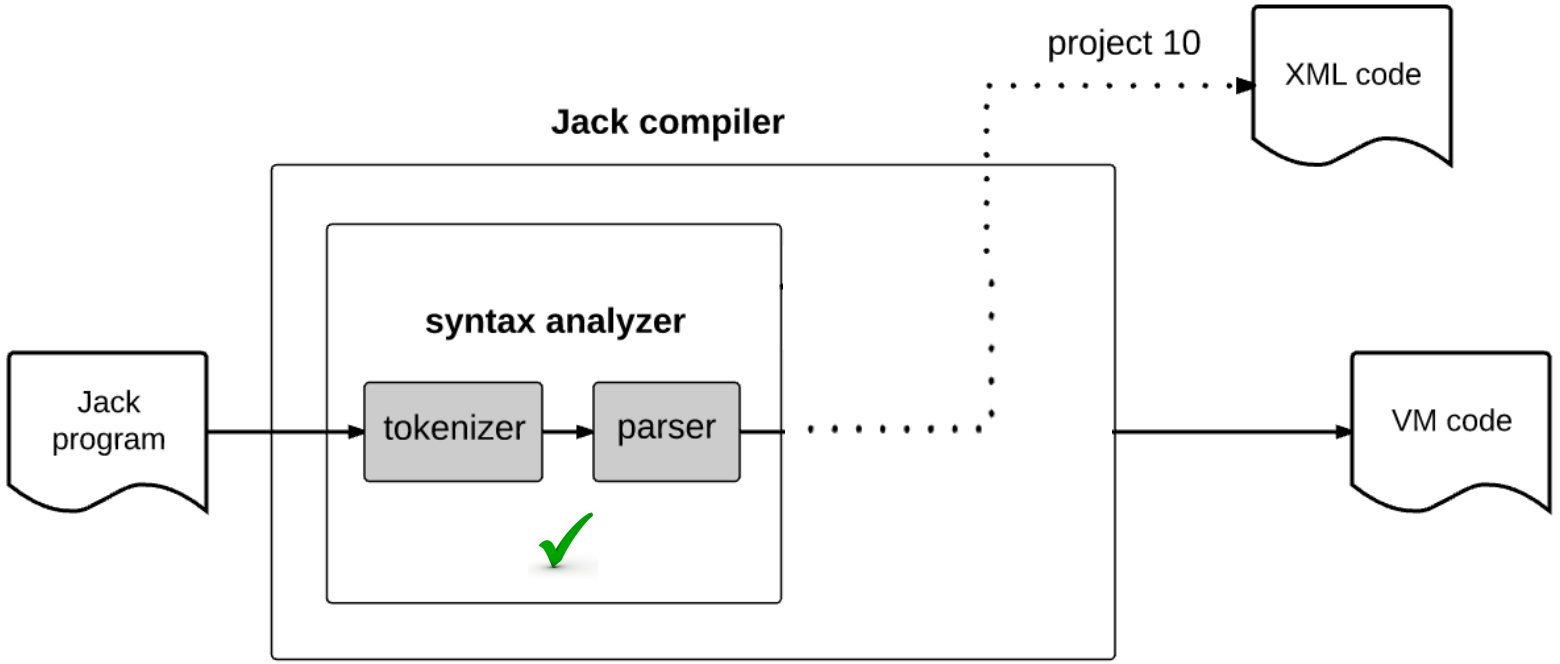
Chapter 11

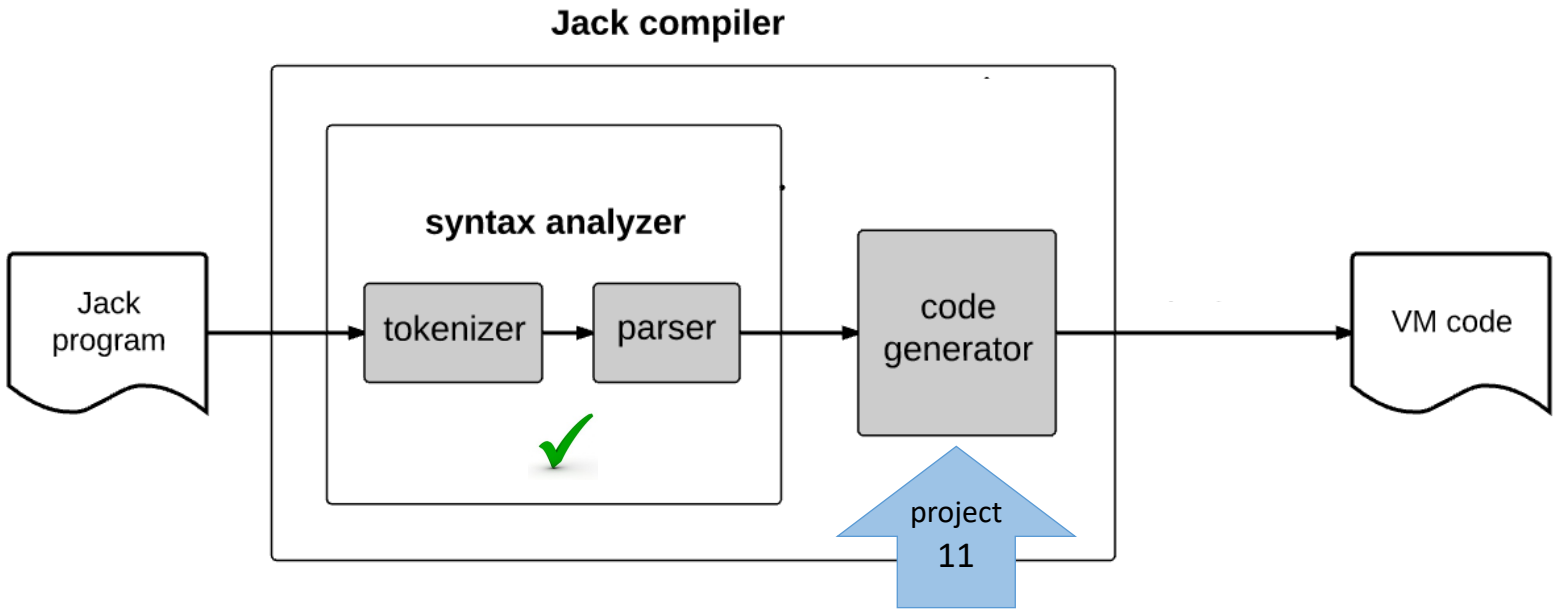
# Compiler II: Code Generation

These slides support chapter 11 of the book
*The Elements of Computing Systems*
By Noam Nisan and Shimon Schocken
MIT Press

# Compiler development roadmap



# Compiler development roadmap




This lecture:

Extending the syntax analyzer to a full-scale compiler.

# Take home lessons

Implementing a procedural programming language

- Variables
- Expressions
- Statements
- Flow of control
- Arrays
- Functions

Implementing an object-based programming language

- Objects
- Constructors
- Methods

Software engineering

- Parsing (mostly done)
- Symbol tables
- Compilation engine
- Code generation
- Memory management.

# Program compilation

High-level source code (Jack)

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

   constructor Point new(int ax, int ay) {}

   method int getx() {}

   method int gety() {}

   function int getPointCount() {}

   method Point plus(Point other) {}

   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 declarations:

- fields
- statics

Subroutine declarations:

- constructors
- methods
- functions

# Program compilation

High-level source code (Jack)

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

   constructor Point new(int ax, int ay) {}

   method int getx() {}

   method int gety() {}

   function int getPointCount() {}

   method Point plus(Point other) {}

   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));
    }
    ...
}
```

Compilation challenges

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

# Variables

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));
    }
    ...
}
```

# Variables

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));
    }
    ...
}
```


VM code (generated by the compiler)

```
...
// return Math.sqrt((dx*dx) + (dy*dy));
push local 0
push local 0
call Math.multiply 2
push local 1
push local 1
call Math.multiply 2
add
call Math.sqrt 1
return
...
```

In order to generate VM code, the compiler must know, for each variable in the program:

## Variable properties

- name (identifier)
- type (`int`, `char`, `boolean`, class name)
- kind (`field`, `static`, `local`, `argument`)
- index (`0`, `1`, `2`, ...)
- scope (class level, subroutine level)

# Symbol tables

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

We'll discuss the variable `this` (`argument 0`) later in the lecture

## Variable properties

- name (identifier)
- type (`int`, `char`, `boolean`, class name)
- kind (`field`, `static`, `local`, `argument`)
- index (`0`, `1`, `2`, ...)
- scope (class level, subroutine level)

# Handling variable declarations

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

Variable declarations occur in class declarations, in method declarations (parameter lists), and in `var` statements.

The compiler handles each…

- `field` variable declaration by adding a `field` *i* entry to the *class-level* symbol table
- `static` variable declaration by adding a `static` *i* entry to the *class-level* symbol table
- parameter variable declaration by adding an `argument` *i* entry to the *subroutine-level* symbol table
- `var` variable declaration by adding a `local` *i* entry to the *subroutine-level* symbol table.

# Handling variable usage

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

Variables are used in *statements* and in *expressions*

The compiler looks up the variable name in the subroutine-level symbol table. If not found, it looks it up in the class-level symbol table.

Each *field*, *static*, *local*, *parameter* variable name in the source code is translated into `this` *i*, `static` *i*, `local` *i*, `argument` *i*, in the generated VM code.

source code

```
let x = x + dx
```

compiler

(example)

VM code

```
push this 0
push local 0
add
pop this 0
```

# Handling variables life cycle

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: Exist throughout the program's execution

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

Field variables: Unique to object-oriented languages; will be discussed later.

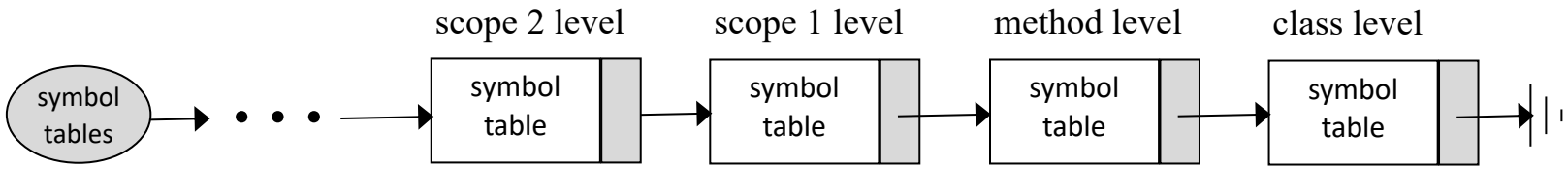
- Managing these variables life cycles is a major headache
- But… this is taken care of by the VM implementation (projects 7–8)
- The compiler need not worry about it!

# Handling 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 unlimited nested variable scoping

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

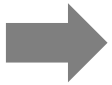



Variable lookup: Start in the first table in the list:
if not found, look up the next table, and so on.

# Lecture plan

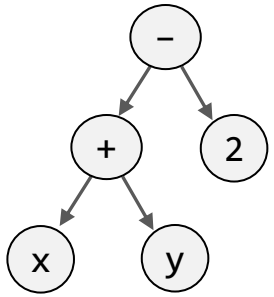
Compilation


Handling variables


Handling expressions

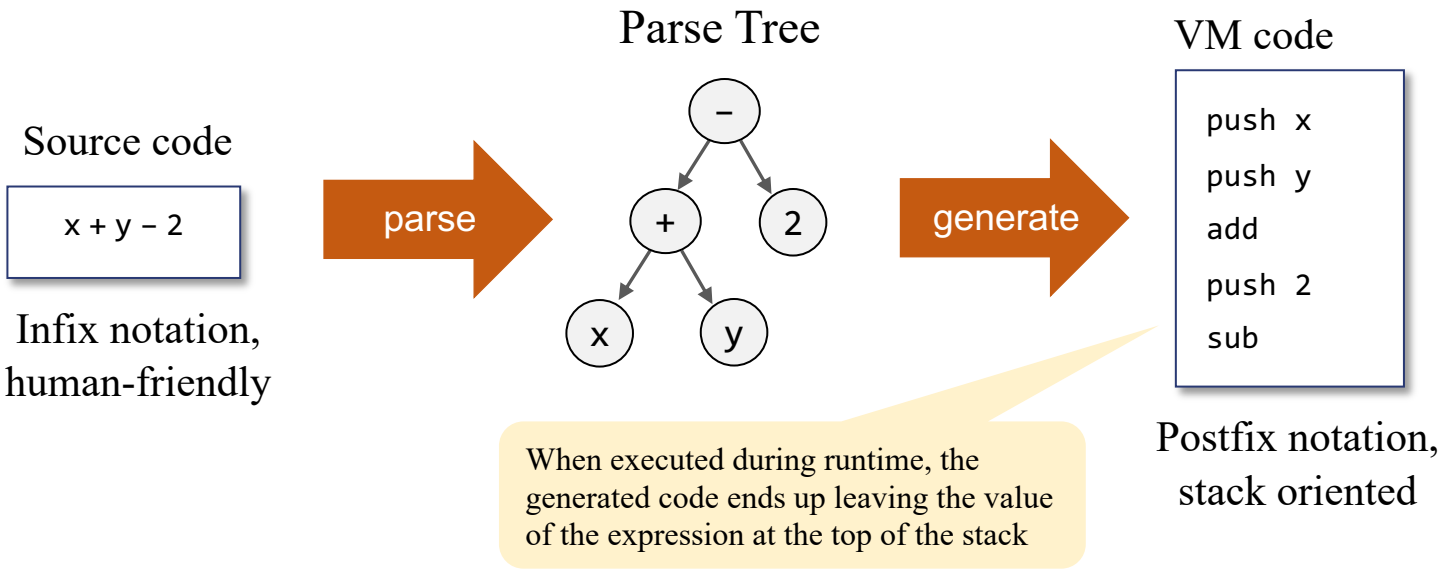
- Handling statements
- Handling objects
- Handling arrays

# Expressions




| Prefix notation | Infix notation | Postfix notation |
|---|---|---|
| `– + x y 2` | `x + y – 2` | `x y + 2 –` |
| (functional) | (human friendly) | (stack oriented) |
| `–(+(x,y),2)` | | |

# Compiling expressions




- Option 1: Parse the infix source expression into a parse tree, then generate the postfix VM code from the tree
- Option 2: Generate the VM code directly from the source expression.

# Compiling expressions

Recursive algorithm for generating postfix VM code from an expression written in infix notation:

**compileExpression**(*exp*):

if *exp* is *term*:
  compileTerm(*term*)

if *exp* is "$term_1$ $op_1$ $term_2$ $op_2$ $term_3$ $op_3$ ... $term_n$":
  compileTerm($term_1$)
  compileTerm($term_2$)
  output "$op_1$"
  compileTerm($term_3$)
  output "$op_2$"
  ...
  compileTerm($term_n$)
  output "$op_{n-1}$"

**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 "(*exp*)":
  compileExpression(*exp*)

if *term* is "*f* ($exp_1$, $exp_2$, ...)":
  compileExpression($exp_1$)
  compileExpression($exp_2$)
  ...
  compileExpression($exp_n$)
  output "call *f*"

The formal definition of *term* was given in chapter 10

The syntactic handling of terms in expressions was done by the Syntax Analyzer (project 10).

# Compiling expressions

Recursive algorithm for generating postfix VM code from an expression written in infix notation:

**compileExpression**(*exp*):

if *exp* is *term*:
  compileTerm(*term*)

if *exp* is "$term_1$ $op_1$ $term_2$ $op_2$ $term_3$ $op_3$ ... $term_n$":
  compileTerm($term_1$)
  compileTerm($term_2$)
  output "$op_1$"
  compileTerm($term_3$)
  output "$op_2$"
  ...
  compileTerm($term_n$)
  output "$op_{n-1}$"

**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 "(*exp*)":
  compileExpression(*exp*)

if *term* is "*f* ($exp_1$, $exp_2$, ...)":
  compileExpression($exp_1$)
  compileExpression($exp_2$)
  ...
  compileExpression($exp_n$)
  output "call *f* n"

Compilation examples (assuming that `x` and `y` are `local 0` and `local 1`)

```
// 5+x
push constant 5
push local 0
add
```

```
// x+y-2
push local 0
push local 1
add
push constant 2
sub
```

```
// foo(x,y+1,-7)
push local 0
push local 1
push constant 1
add
push constant 7
neg
call foo 3
```

Note: This algorithm…

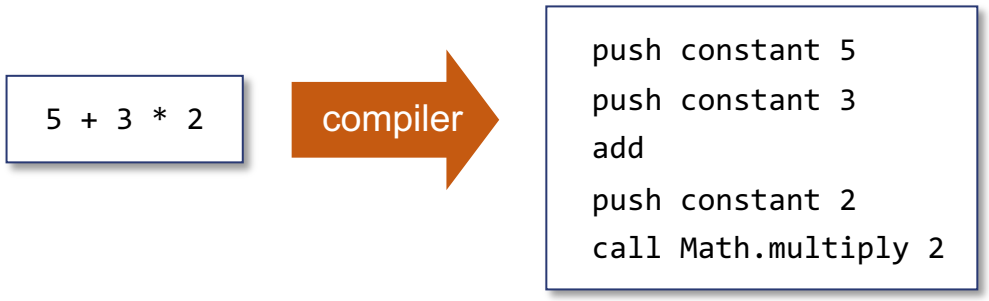
- Does not handle *array elements* (later)
- Supports no *operator priority*.

# Compiling expressions

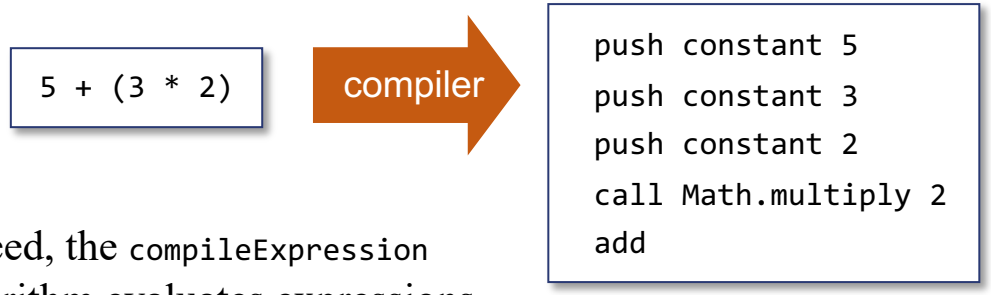
The Jack language specifies no operator priority

(In purpose, to make the compiler simpler)

The default priority is left-to-right:




But, expressions in parentheses are evaluated first:




Indeed, the `compileExpression` algorithm evaluates expressions in parentheses first.

# Compiling expressions

Handling the four constants of the Jack language

- `true` is represented as `-1`
- `false` is represented as `0`
- `null` is represented as `0`
- `this` is represented as `pointer 0`

Compilation examples (in the context of `return` statements)

```
// return true;
push constant 1
neg
return
```

```
// return false;
push constant 0
return
```

```
// return null;
push constant 0
return
```

```
// return this;
push pointer 0
return
```

# 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`

Handled by five compilation routines:
`compileLet`, `compileReturn`, `compileDo`, `compileIf`, `compileWhile`

- Handling objects
- Handling arrays

# Compiling `let`

source code (Jack)

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

compiler

VM code (generated by `compileLet`)

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

To handle a `let` statement, `compileLet` calls `compileExpression` to translate the statement's right hand side. When the compiled VM code will execute during runtime, it will end up leaving the expression's value at the stack's top. The subsequent `pop` command will then assign it to *varName*.

Example:

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

compiler

```
// v = g + r2;
push static 0
push this 1
add
pop local 1
```

The blue code is generated by `compileExpression`

| name | type | kind | # |
|---|---|---|---|
| g | int | static | 0 |
| r1 | int | field | 0 |
| r2 | int | field | 1 |

class-level symbol table

| name | type | kind | # |
|---|---|---|---|
| u | int | local | 0 |
| v | int | local | 1 |

subroutine-level symbol table

# Lecture plan

## Compilation


Handling variables


Handling expressions

- Handling statements
  - `let`
  - `return`
  - `do`
  - `if`
  - `while`
- Handling objects
- Handling arrays

# Compiling `return`

source code

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

compiler

VM code (generated by `compileReturn`)

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

`compileLet` calls `compileExpression`. During runtime, the code generated by `compileExpression` will end up leaving the expression's value at the stack's top. The VM implementation of the subsequent `return` command will place this value at the top of the caller's stack, when the caller resumes its execution.

(The `return` statement has two versions)

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

compiler

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

- A VM function must return a value. When compiling a `return` statement without a return value, the compiler generates code that pushes a dummy value onto the stack
- This value will be tossed away by the compiled code of the caller (discussed next).

# Lecture plan

## Compilation


Handling variables


Handling expressions

- Handling statements
  - `let`
  - `return`
  - `do`
  - `if`
  - `while`
- Handling objects
- Handling arrays

# Compiling do

source code (Jack)

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

compiler

VM code (generated by `compileDo`)

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

The `do` statement is used to call a function or a method for its effect, ignoring the returned value

We recommend handling `do` *subroutineName*`(…)` statements as if they where `do` *expression* statements.

The subsequent `pop` gets rid of the return value.

Example:

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

compiler

```
// do Output.printInt(7);
push constant 7
call Output.printInt 1
pop temp 0
```

Blue code:
Generated by `compileExpression`

# Lecture plan

## Compilation


Handling variables


Handling expressions

- Handling statements
  - `let`
  - `return`
  - `do`
  - `if`
  - `while`
- Handling objects
- Handling arrays

# Compiling `if`

source code

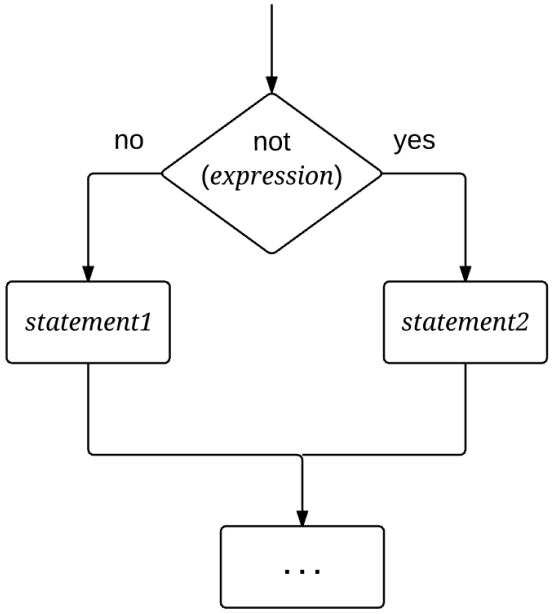
```
if (expression)
    {statements1}
else
    {statements2}
...
```

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
    ...
```

(the labels `L1` and `L2` are generated by the compiler)



# 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
    ...
```

(the labels `L1` and `L2` are generated by the compiler)

Example:

```
if (u > 0) {
    let v = g;
}
...
```

compiler

```
// if (u > 0) { let v = g }
   push u
   push 0
   gt
   not
   if-goto L1
   push g
   pop v
label L1
   ...
```

# Compiling `while`

source code

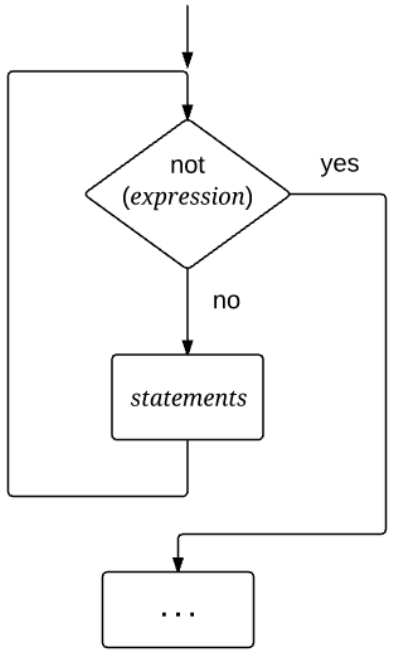
```
while ( expression )
    { statements }
...
```

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
   ...
```

(the labels `L1` and `L2` are generated by the compiler)



# 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

(the labels `L1` and `L2` are generated by the compiler)

```
// while (g = 0) { let x = x + 1 }
label L1
  push g
  push 0
  eq
  not
  if-goto L2
  push x
  push 1
  add
  pop x
  goto L1
label L2
  ...
```

# Lecture plan

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements

We have what it takes to write a compiler for a procedural programming language!

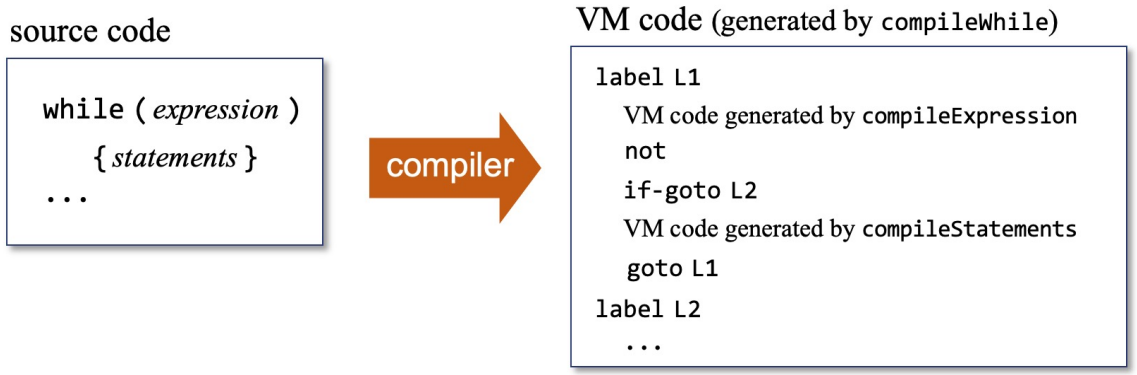
- Handling objects
- Handling arrays

# Before proceeding, a side note

## Our goal: Teaching compilation fundamentals

We don't want to rob from you the pleasure of writing a compiler on your own;

We describe the major building blocks of a simple compiler, and expect you to figure out how to put them together. For example, consider our explanation on how to handle `while`:




## In project 11, you will write the `compileWhile` routine that realizes this translation

Your `compileWhile` code will have to generates unique labels like `L1` and `L2` (possibly using other names). It will have to remember where in the output to emit which label. It will have to call `compileExpression`, and then emit to the output the `not` and the `if-goto L2` commands, and then call `compileStatements`, and so on.

Of course you will also have to write the `compileExpression` and `compileStatements` routines, following similar explanations.

Back to the lecture…

# Lecture plan

## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements

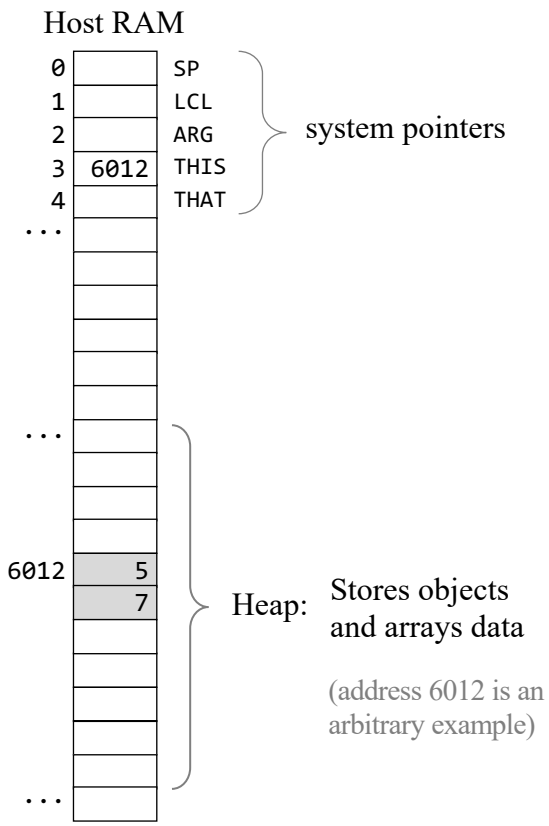
We have what it takes to write a compiler for a procedural programming language!

- ➡ Handling objects
- Handling arrays

# Memory management primer

Exercise: Create, and initialize, a 2-words object:

```
// Creates a 2-words object, and aligns the virtual
// segment this with its base address:
push constant 2
call Memory.alloc 1
pop pointer 0
```




To "create a new object": We need to *allocate memory*.
This can be done by calling an OS function named `Memory.alloc`

Like all the OS functions, `Memory.alloc` is implemented, and is accessible as, a compiled VM function.

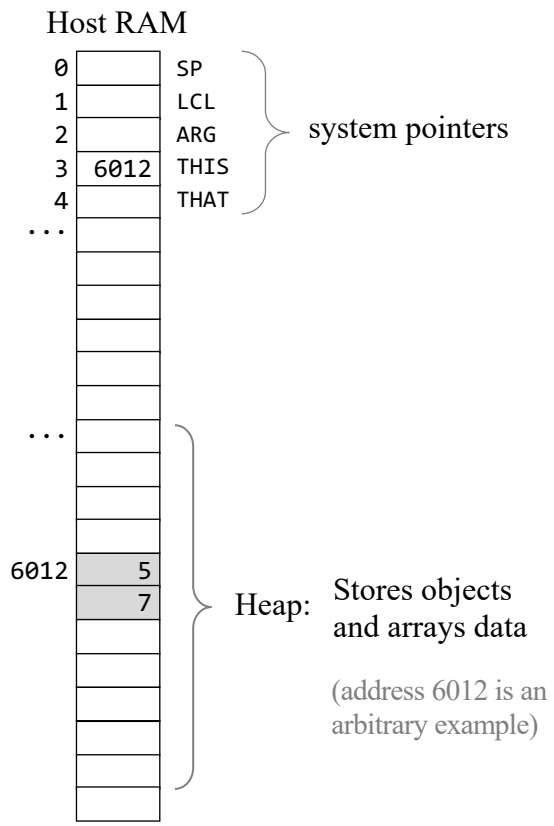
`Memory.alloc(n)`:
Finds a free memory block of size *n* words, and returns the base address of the found block.

`Memory.alloc` uses clever memory management algorithms, learned and implemented in chapter 12

# Memory management primer

Exercise: Create, and initialize, a 2-words object:

```
// Creates a 2-words object, and aligns the virtual
// segment this with its base address:
push constant 2
call Memory.alloc 1
pop pointer 0

// Initializes the object's fields to 5 and 7:
push constant 5
pop this 0
push constant 7
pop this 1
```




`pop pointer 0`:

(Reminder: `pointer 0` and `pointer 1` represent `THIS` and `THAT`).

In the above context, this command sets `THIS` to the base address of the newly constructed object.

This setting ensures that the `this` segment will be properly aligned with the object's base address in memory.

And this, in turn, ensures that in the subsequent VM-to-assembly translation, every `this` *i* reference in the VM code will be mapped correctly on the memory addresses (`THIS` + *i*).

Using this technique, we can apply a method's code to *any* object; All we need is its base address.

Perhaps the most important "enabling trick" in object-oriented programming.

# Creating objects

Source code

```
// Can appear in any class:                    caller
...
// Declares three local variables:
var Point p1, p2;
var int d;
...
// Constructs two objects:
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

Caller: Creates two Point objects, and makes `p1` and `p2` refer to them

```
/** Represents a Point (API). */
class Point {
    ...
    /** Constructs a point. */
    constructor Point new(int ax, int ay)
    // ... More Point methods follow
}                                               callee
```

Callee: Creates a Point object, and returns its base memory address

**Abstraction**

("objects")

## Object creation

is typically a 2-stage process:

- Declaring an object variable,
- Creating the object and assigning it to the variable

Typically involves:

- Caller
- Callee (constructor)

# Creating objects

Source code

```
// Can appear in any class:                    caller
...
// Declares three local variables:
var Point p1, p2;
var int d;
...
// Constructs two objects:
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

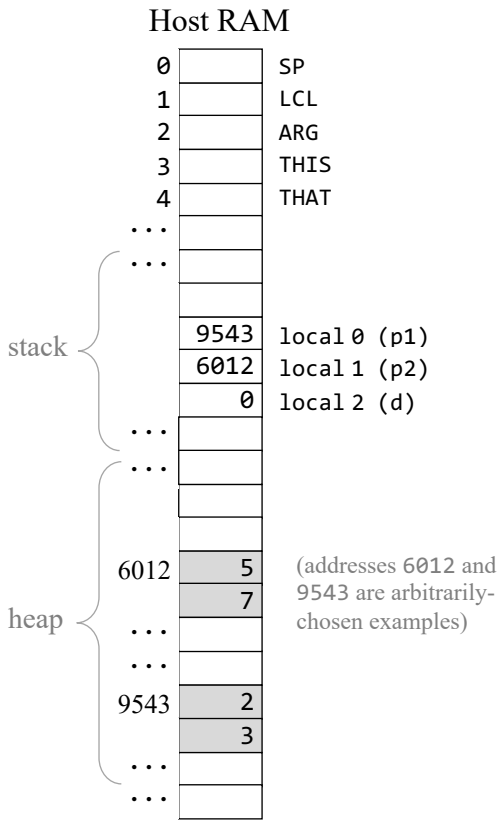
Caller: Creates two Point objects, and makes `p1` and `p2` refer to them

```
/** Represents a Point (API). */
class Point {
    ...
    /** Constructs a point. */
    constructor Point new(int ax, int ay)
    // ... More Point methods follow
}                                               callee
```

Callee: Creates a Point object, and returns its base memory address

**Abstraction**

("objects")




**Realization**

(managed memory blocks)

Who realizes the object creation abstraction?

The code that the compiler generates from the *caller* and the *callee*.

# Compiling constructor calls

Source code

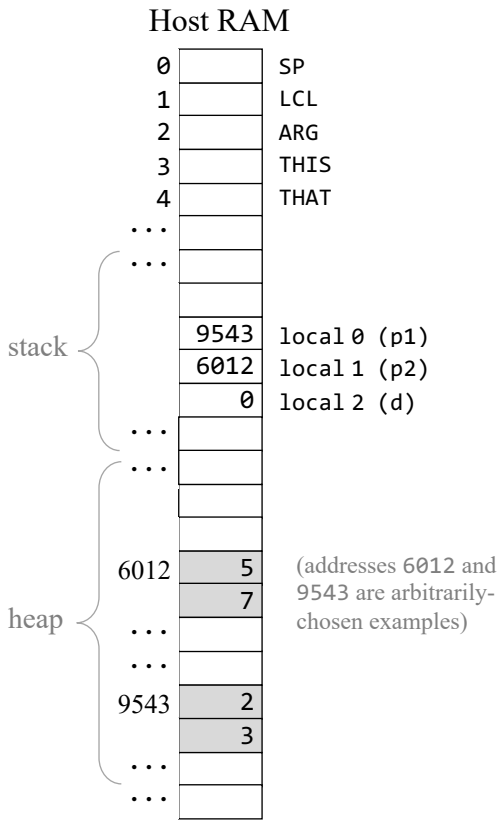
```
// Can appear in any class:        caller
...
// Declares three local variables:
var Point p1, p2;
var int d;
...
// Constructs two objects:
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

compile

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

VM code

```
// var Point p1, p2;
// var int d;
//// The compiler builds the
////  method's symbol table.
...
// let p1 = Point.new(2,3);
push constant 2
push constant 3
call Point.new 2
pop local 0
// let p2 = Point.new(5,7);
push constant 5
push constant 7
call Point.new 2
pop local 1
...
```




The source code calls a constructor, *which is a subroutine*, for its effect.

The caller's expectations from the constructor:
Create, initialize, and return (the base address of) a new object.

If we trust this expected behavior abstractly, there's nothing special here;

The compiler simply calls the `compileLet` routine, which generates the VM code.

The heavy lifting will be done by the compiled constructor code (next).

# Compiling constructors

High-level code

```
// Can appear in any class:
var Point p1;
...
let p1 = Point.new(2,3);
...
```

caller

```
/** 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;
   }
   ...
```

callee

Expected to create a Point object, initialize its fields, and return its base memory address.

# Compiling constructors

High-level code

```
// Can appear in any class:
var Point p1;
...
let p1 = Point.new(2,3);
...
```

caller

When the caller resumes its execution, `p1` will be assigned the base address of the new object.

```
/** 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;
   }
   ...
```

callee

compile

VM code

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

// constructor Point new(int ax, int ay);
//// The compiler builds the constructor's symbol table,
//// adds ax and ay to it, and handles all the var statements,
//// of which there are none in this constructor.

//// Next, the compiler generates the commands:

function Point.new 0
   push constant 2
   call Memory.alloc 1
   pop pointer 0

//// Next, the compiler handles the constructor's body.

//// Note: The compilation of return this generates
//// the commands  push pointer 0  and return.
```

class-level symbol table

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

constructor-level symbol table

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

When executed, the generated constructor's code will construct an object of the right size, align `this` with the new object's base address, and return the base address to the caller.

# Lecture plan

## Compilation

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

# Method calling primer

high-level code

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

caller

The caller "applies a method on an object"

How to pass the target object from the caller to the callee?

```
class Point
  ...
  /** Distance from this to the other point. */
  method int distance(Point other) {
      ...
  }
  ...
}
```

callee

The called(method) responds by "operating on the given object"

Designed to operate on *any object* of the class type

# Method calling primer

high-level code

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


VM (pseudo) code

```
...
push p1
push p2
call distance
pop d
...
```

Compilation conventions:

1. When compiling a method call, we always pass the target object as the *first argument* for the called method

```
class Point
  ...
  /** Distance from this to the other point. */
  method int distance(Point other) {
      ...
  }
  ...
}
```
callee


```
...
function distance
push argument 0
pop pointer 0
...
```

2. When compiling a method, we always start with "`this = argument 0`". This ensures that all the subsequent "`this i`" references will map on the fields of the target object.

This is the key compilation trick

that enables the OO "object first" method calling syntax:

*objName*`.`*methodName*`(`*args*`)`

# Method calling primer

And now for the details…

# Compiling methods calls / Compiling methods

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

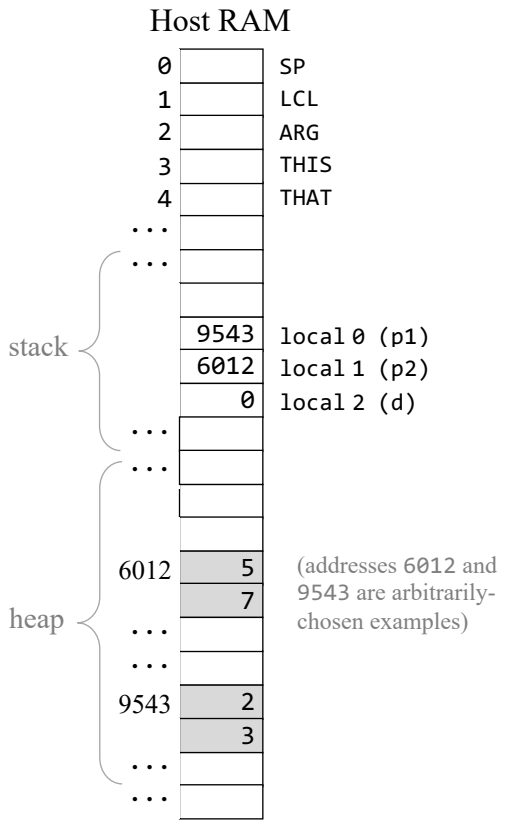
Method call abstraction:

Applies the method `distance` to the object `p1`

```
/** 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

Method abstraction:

Applies the method to the current object (`this`)




## Who realizes the method calling abstraction?

The code that the compiler generates from the caller and the callee.

# Compiling methods calls / Compiling methods

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

compiler

VM code

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

| name | type | kind | # |
|---|---|---|---|
| p1 | Point | local | 0 |
| p2 | Point | local | 1 |
| d | int | 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));
  }
  ...
}
```
callee

compiler

VM code

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

//// compileClass builds the class-level symbol table,
//// and adds to it the variables x, y, and pointCount.

// method int distance(Point other)
// var int dx, dy;

//// compileSubroutine builds the method's symbol
//// table, and adds to it the variables this, other, dx, dy.
//// Next, it generates the code:

function Point.distance 2
  push argument 0
  pop pointer 0

//// Next, compileStatements is called to handle the method's
//// statements.
```

| 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 |

method-level symbol table

# Compiling methods calls / Compiling methods

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

compiler

VM code

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

| name | type | kind | # |
|---|---|---|---|
| p1 | Point | local | 0 |
| p2 | Point | local | 1 |
| d | int | 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));
  }
  ...
}
```
callee

compiler

VM code

```
// method int distance(Point other) {
// var int dx, dy;
function Point.distance 2
  push argument 0
  pop pointer 0
  // let dx = x - other.getx();
  push this 0
  push argument 1
  call Point.getx 1
  sub
  pop local 0
  // let dy = y - other.gety();
  //// Similar, code omitted.
  // return Math.sqrt((dx*dx)+(dy*dy));
  push local 0
  push local 0
  call Math.multiply 2
  push local 1
  push local 1
  call Math.multiply 2
  add
  call Math.sqrt 1
  return
```

| 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 |

method-level symbol table

# 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 array elements

# Creating arrays

Source code

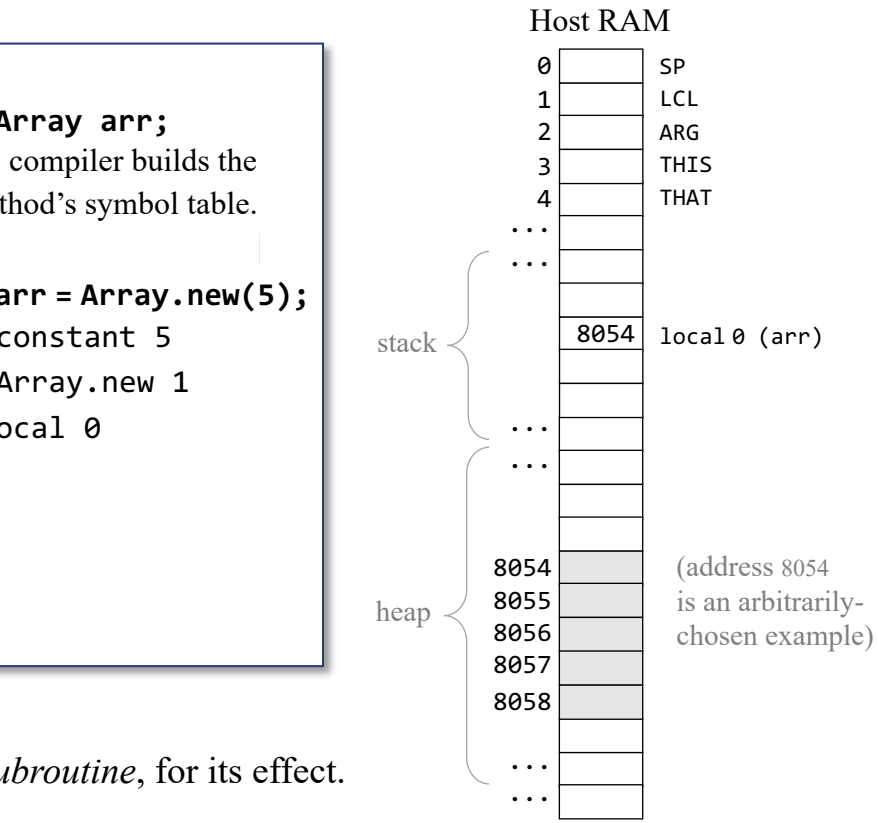
```
// Can appear in any class:
...
// Declares an array variable:
var Array arr;
...
// Constructs the array:
let arr = Array.new(5);
...
// Manipulates the array
let arr[2] = 17;
...
```

compile

VM code

```
...
// var Array arr;
//// The compiler builds the
////  method's symbol table.
...
// let arr = Array.new(5);
push constant 5
call Array.new 1
pop local 0
...
```

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




The source code calls `Array.new`, *which is a subroutine*, for its effect.

The caller's expectations:
Create, initialize, and return (the base address of) a new array.

If we trust this expected behavior abstractly, there's nothing special here;

The compiler simply calls the `compileLet` routine, which generates the VM code.

# Lecture plan

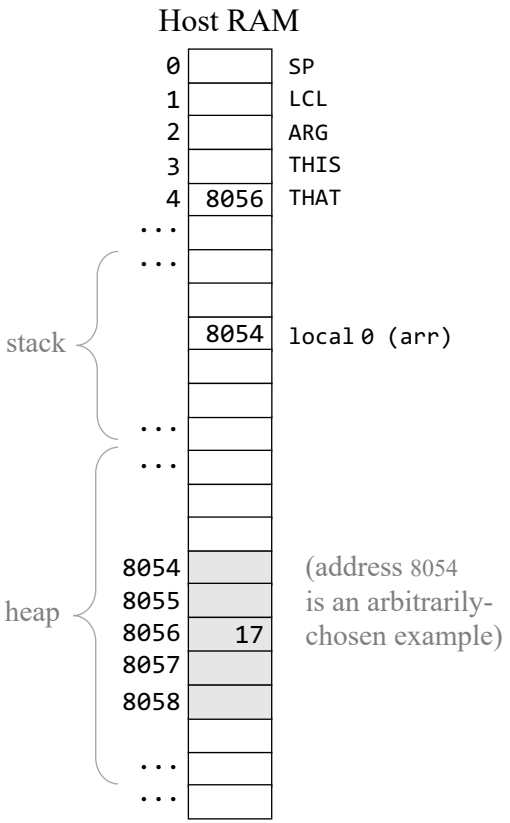
## Compilation

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling statements
- ✓ Handling objects
- Handling arrays
  - ✓ Creating arrays
  - ➡ Manipulating array elements

# Manipulating array elements

Example:

```
// let arr[2] = 17
push arr
push 2
add
pop pointer 1    // THAT = arr + 2
push 17
pop that 0
```




Reminder (VM implementation):

- `pointer 0` represents `THIS`
- `pointer 1` represents `THAT`

Suppose we set `pointer 1` = *addr*

In the subsequent VM-to-Assembly translation, every `that` *i* in the VM code will map on the machine-level address *addr* + *i*

In particular, `that 0` will map on address *addr*.

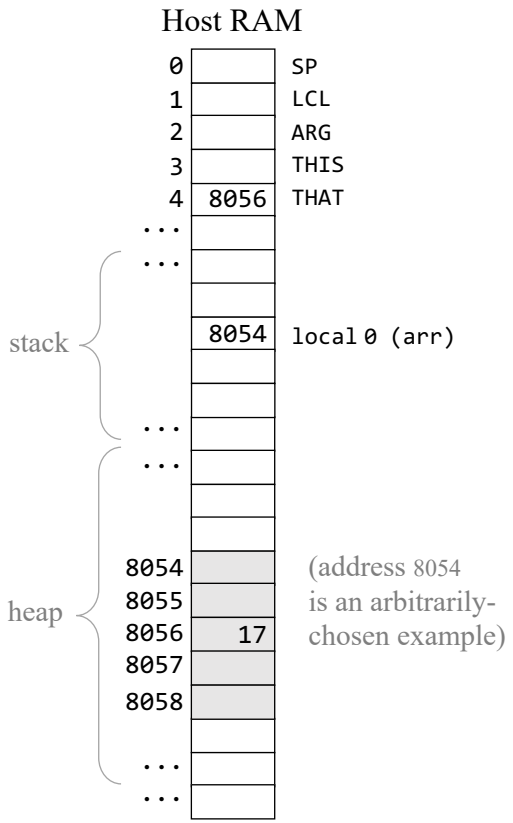
# Manipulating array elements

Example:

```
// let arr[2] = 17
push arr
push 2
add
pop pointer 1    // THAT = arr + 2
push 17
pop that 0
```

Generalizing into a compilation strategy (first try…):

```
// let arr[expression1] = expression2
push arr
push expression1
add
pop pointer 1
push expression2
pop that 0
```



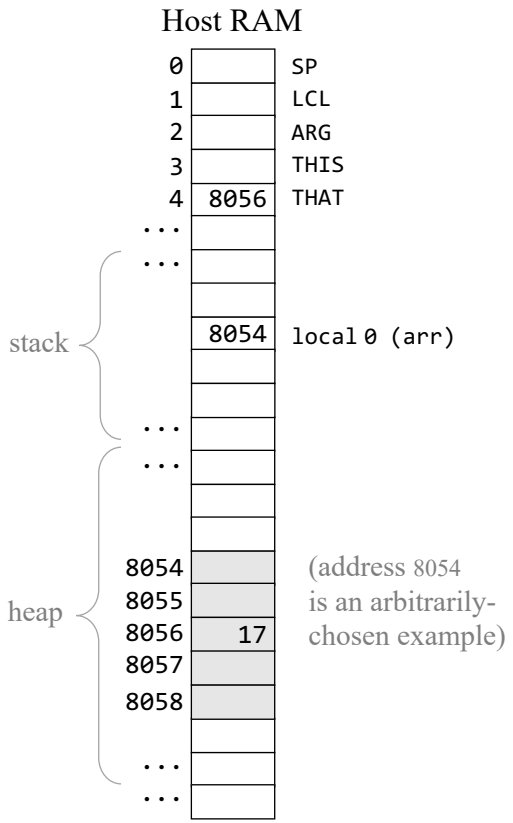

Note:

"`push` *expression*" is pseudo code for

//// call `compileExpression` (which computes,
////  and pushes the value of, *expression*)

# Manipulating array elements

unfortunately,
there's a problem




General compilation strategy (first try…)

```
// let arr[expression1] = expression2
push arr
push expression1
add
pop pointer 1
push expression2
pop that 0
```

Note:

"`push` *expression*" is pseudo code for

//// call `compileExpression` (which computes,
////  and pushes the value of, *expression*)

# Manipulating array elements

## The problem, illustrated

```
// let a[i] = b[j]
push a
push i
add
pop pointer 1

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


## Solution

```
// let a[i] = b[j]
push a
push i
add
push b
push j
add
pop pointer 1
push that 0
pop temp 0
pop pointer 1
push temp 0
pop that 0
```

At this point, during runtime:

- The stack topmost value is the address of `a[i]`
- `temp 0` contains the value of `b[j]`

# Manipulating array elements

Solution (general)

```
// let arr[expression1] = expression2
push arr
//// call compileExpression to compute and push expression1
add                 // top stack value = address of arr[expression1]
//// call compileExpression to compute and push expression2
pop temp 0      // temp 0 = the value of expression2
pop pointer 1
push temp 0
pop that 0
```

What about handling, say, `let a[a[i]] = a[b[a[b[j]]]]` ?

No problem... Thx to the stack's LIFO behavior.

# 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:

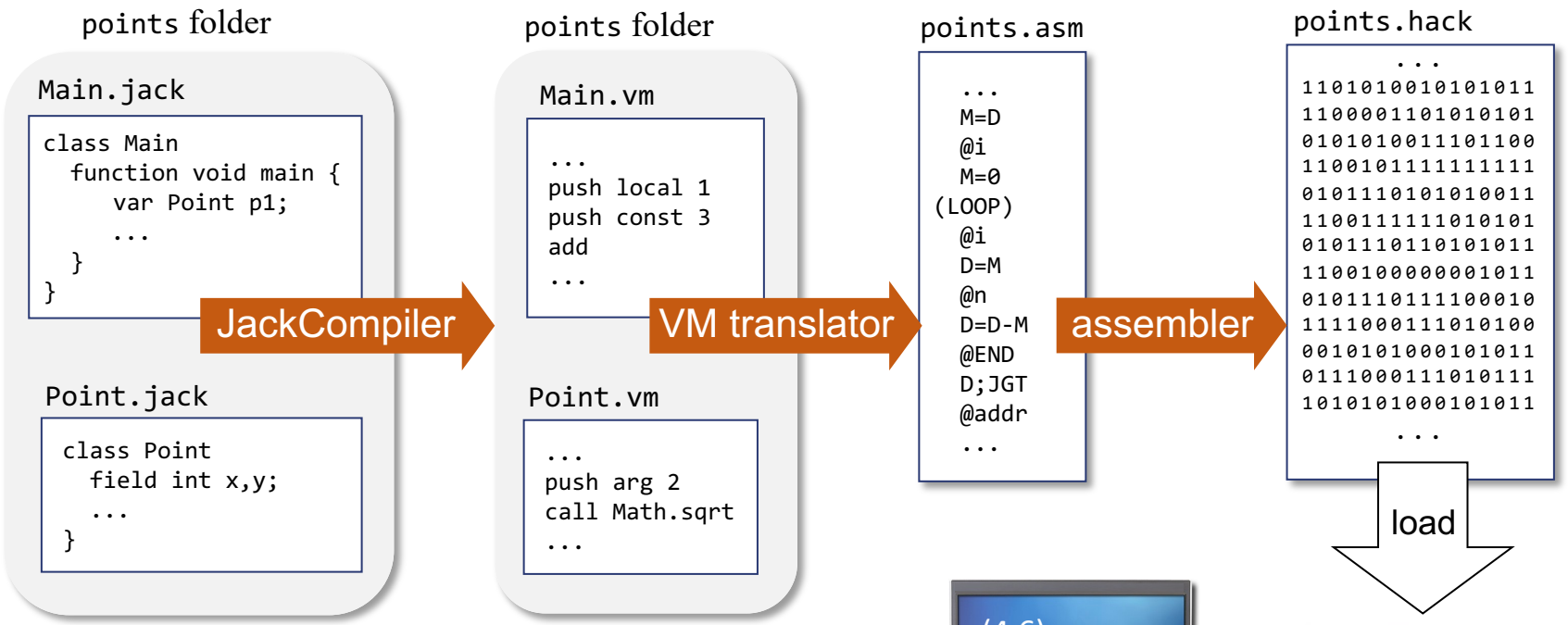
- General compilation techniques
- Compiling Jack code

We now turn to describe:

How to generate code for the 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 / app 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
  ...
```

A Jack class is a collection of one or more subroutines

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 `arr[i]`

Is implemented by generating VM code that realizes:

set `pointer 1` to `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 or a function call *subName*(*exp*$_1$, *exp*$_2$, ..., *exp*$_n$)

The generated VM code pushes the expressions *exp*$_1$, *exp*$_2$, ..., *exp*$_n$ onto the stack, followed by the command `call` *ClassName*`.`*subName* *n*

Compiling a method call *obj*`.`*subName*(*exp*$_1$, *exp*$_2$, ..., *exp*$_n$)

The generated VM code pushes *obj* and then *exp*$_1$, *exp*$_2$, ..., *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 call 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` (`this`) 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

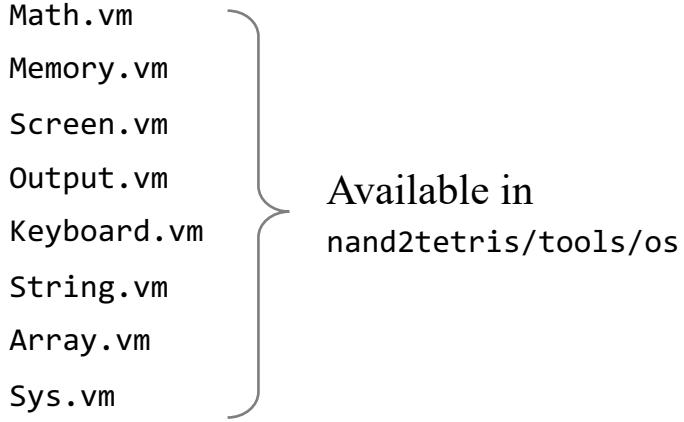
  Setting `pointer 0` (`this`) to `alloc`'s return value
- The generated VM code ends with `return pointer 0`

## 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 (chapter 12)
- The OS is implemented as a set of 8 compiled Jack classes:




- 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 wish to translate the generated VM code to assembly, put the eight OS `*.vm` files in the same folder as the VM files generated by the compiler, and apply the translator to the folder.

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

# The OS (implementation notes)

The generated VM code 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*`)`

# Lecture plan

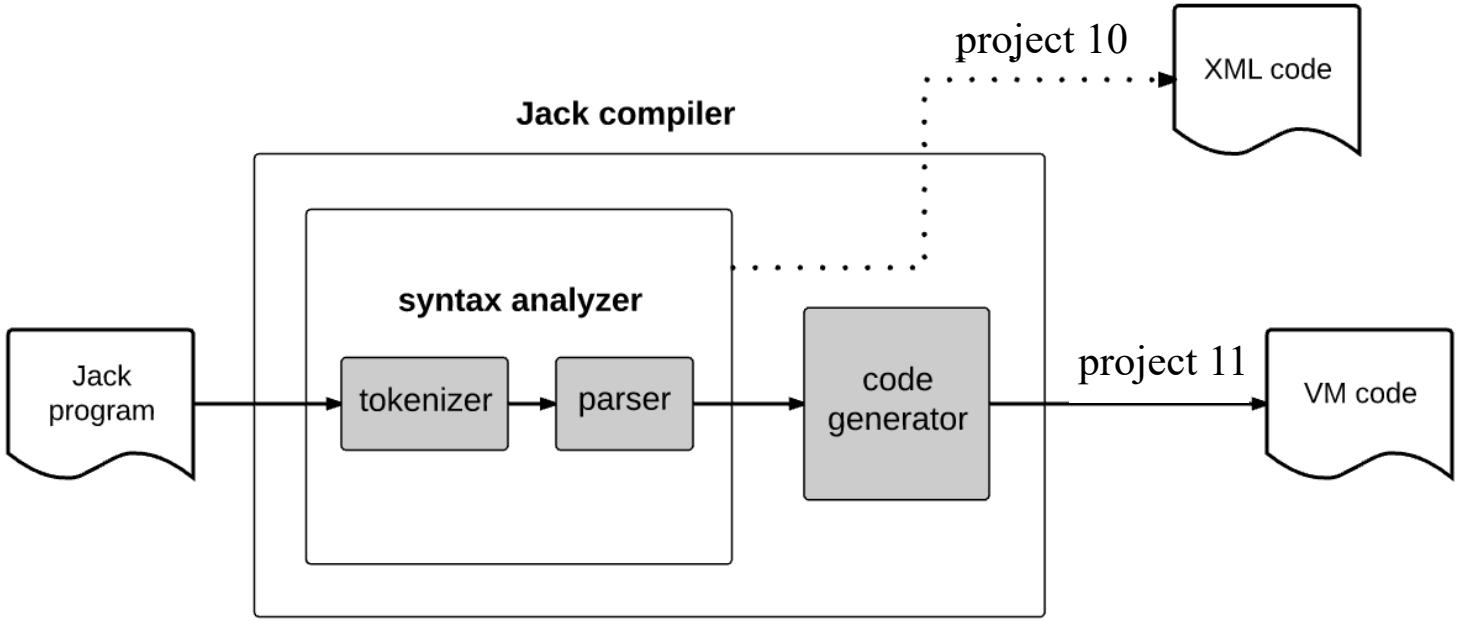
## Compilation

- ✓ 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 the modules:

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

# The Jack compiler

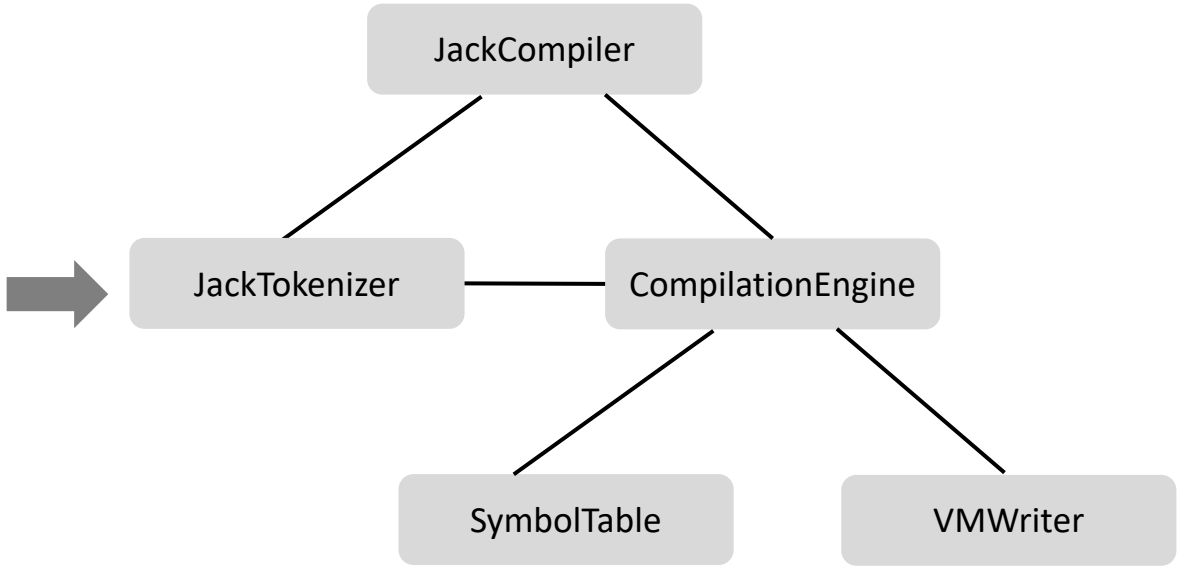
## Usage:

```
prompt> JackCompiler input
```

Input:
- *fileName*`.jack`: name of a single source file, or
- *folderName*: name of a folder containing one or more `.jack` source 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 the same 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:

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

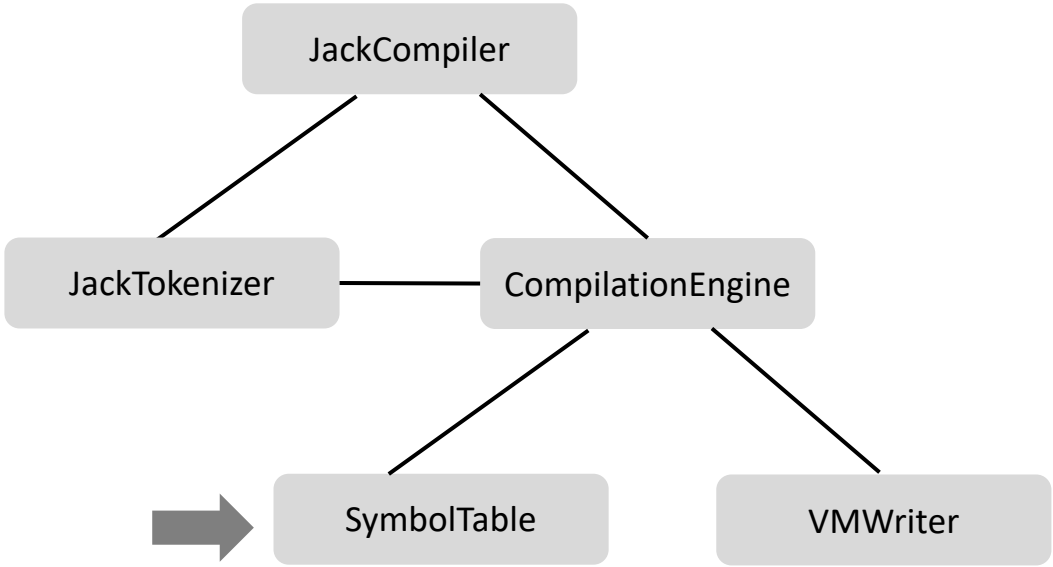
(Developed in project 10)

# 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

# SymbolTable

Implementation notes:

- Each symbol table can be implemented as a separate instance of `SymbolTable`
- When compiling a Jack class, we can build one class-level symbol table and one subroutine-level symbol table
- We can reset the latter table each time we start compiling a new subroutine

| 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

`SymbolTable` API

| *Routine* | *Arguments* | *Returns* | *Function* |
|---|---|---|---|
| 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.<br>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. |

# SymbolTable

Implementation notes:

- 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 symbol not found in the symbol tables can be assumed to be either a *subroutine name* or a *class name*.

| 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

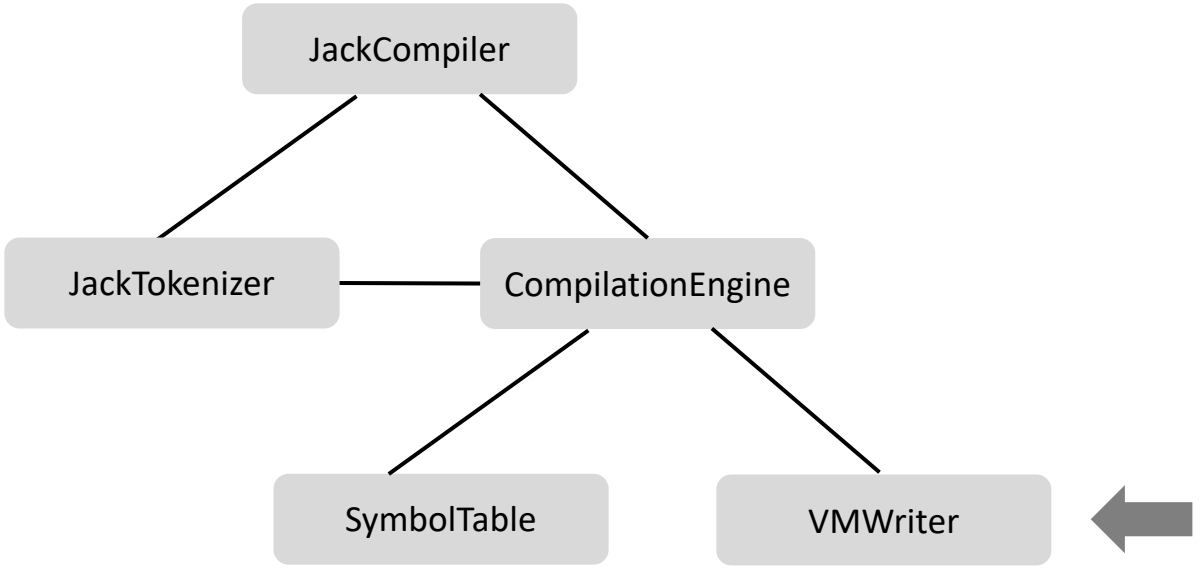
`SymbolTable` API

| ***Routine*** | ***Arguments*** | ***Returns*** | ***Function*** |
|---|---|---|---|
| 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.<br>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. |

# SymbolTable (same as previous slide)

| *Routine* | *Arguments* | *Returns* | *Function* |
|---|---|---|---|
| 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

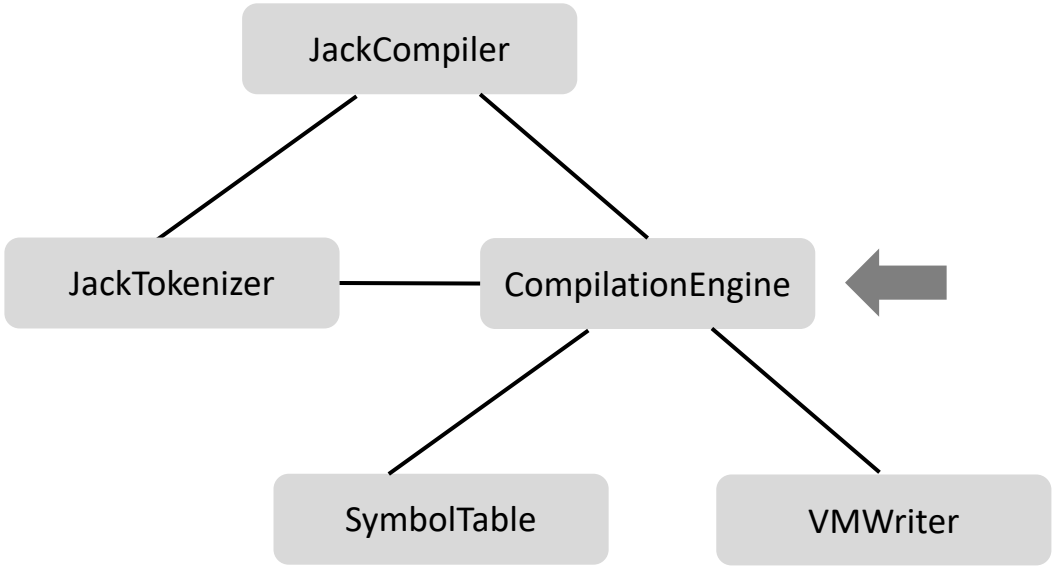


# VMWriter

| *Routine* | *Arguments* | *Returns* | *Function* |
|---|---|---|---|
| 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.

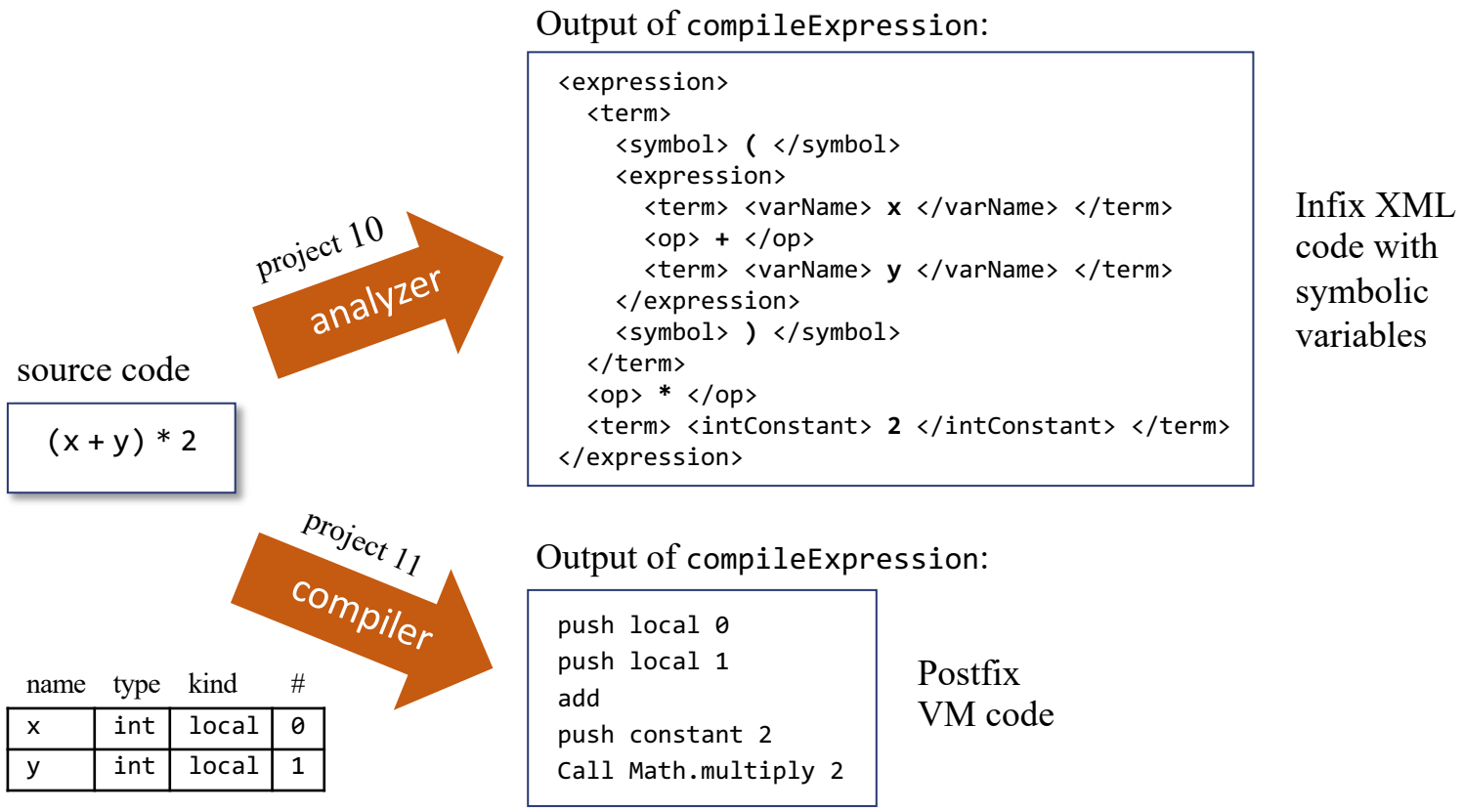
# The Jack compiler



# CompilationEngine

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

However, the `compile`*xxx* methods generate different outputs. For example:



# CompilationEngine

- Gets its input from a `JackTokenizer` and writes its output using the `VMWriter`
- Organized as a series of `compile`*xxx* routines, *xxx* being a syntactic element in the Jack grammar:
  - Each `compile`*xxx* 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*
  - `compile`*xxx* 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 should compute this value and leave it at the top of the VM's stack

# CompilationEngine (same API as in project 10)

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

# CompilationEngine (same API as in project 10)

| *Routine* | *Arguments* | *Returns* | *Function* |
|---|---|---|---|
| `compileLet` | – | – | Compiles a `let` statement. |
| `compileIf` | – | – | Compiles an `if` statement, possibly with a trailing `else` clause. |
| `compileWhile` | – | – | Compiles a `while` statement. |
| `compileDo` | – | – | Compiles a `do` statement. |
| `compileReturn` | – | – | Compiles a `return` statement. |
| `compileExpression` | – | – | Compiles an expression. |
| `compileTerm` | – | – | Compiles a *term*. If the current token is an *identifier*, the routine must resolve it into a *variable*, an *array entry*, or a *subroutine call*. A single lookahead token, which may be `[`, `(`, or `.`, 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

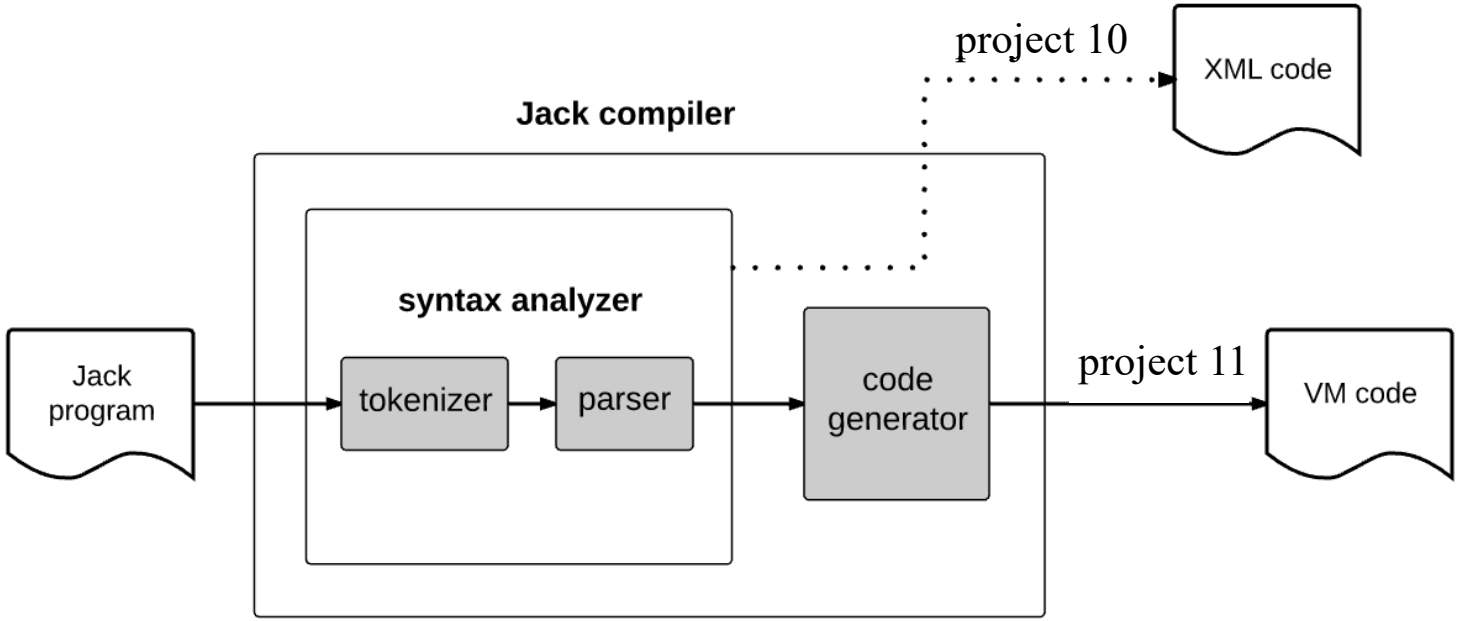
## Compilation

- ✓ 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>`

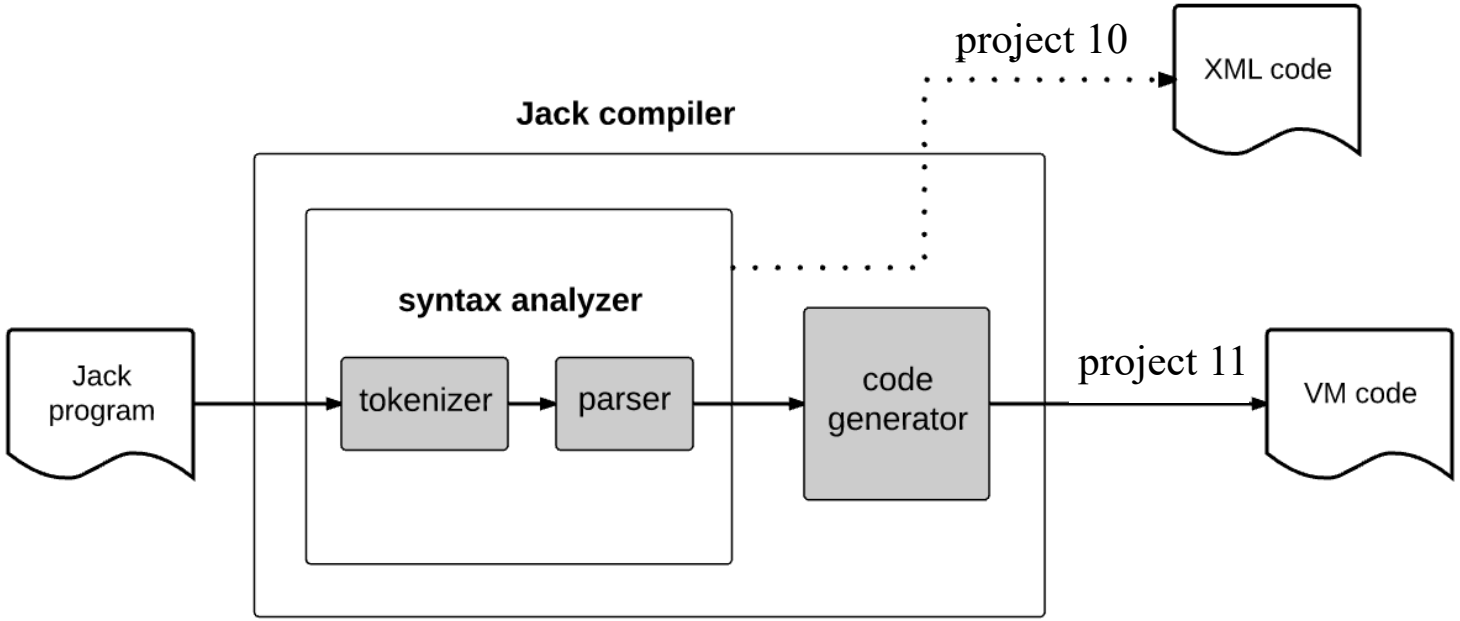
## Extend the handling of identifiers:

- 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`

Unit testing:

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

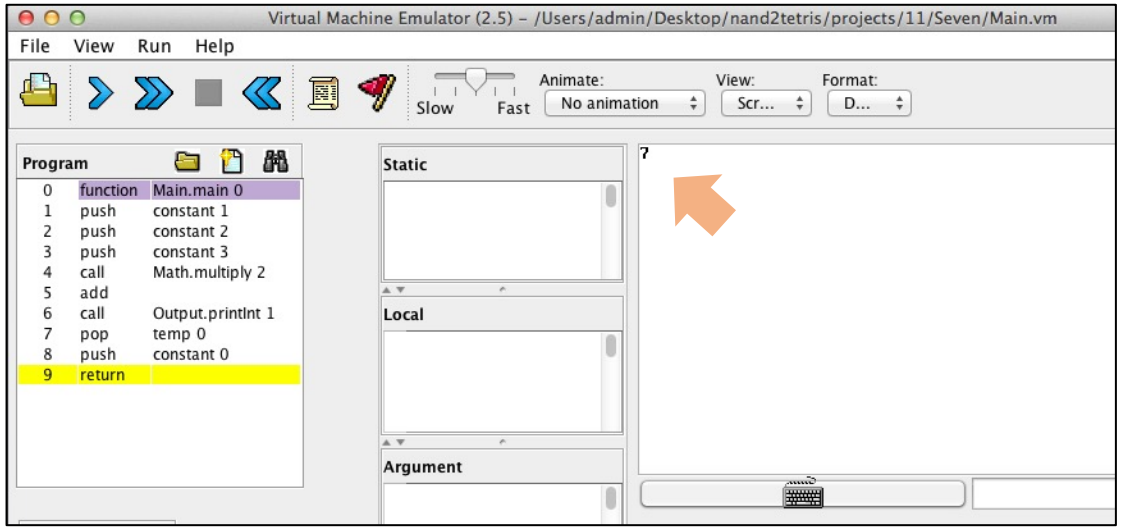
projects/11/Seven/Main.jack

```
/** 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

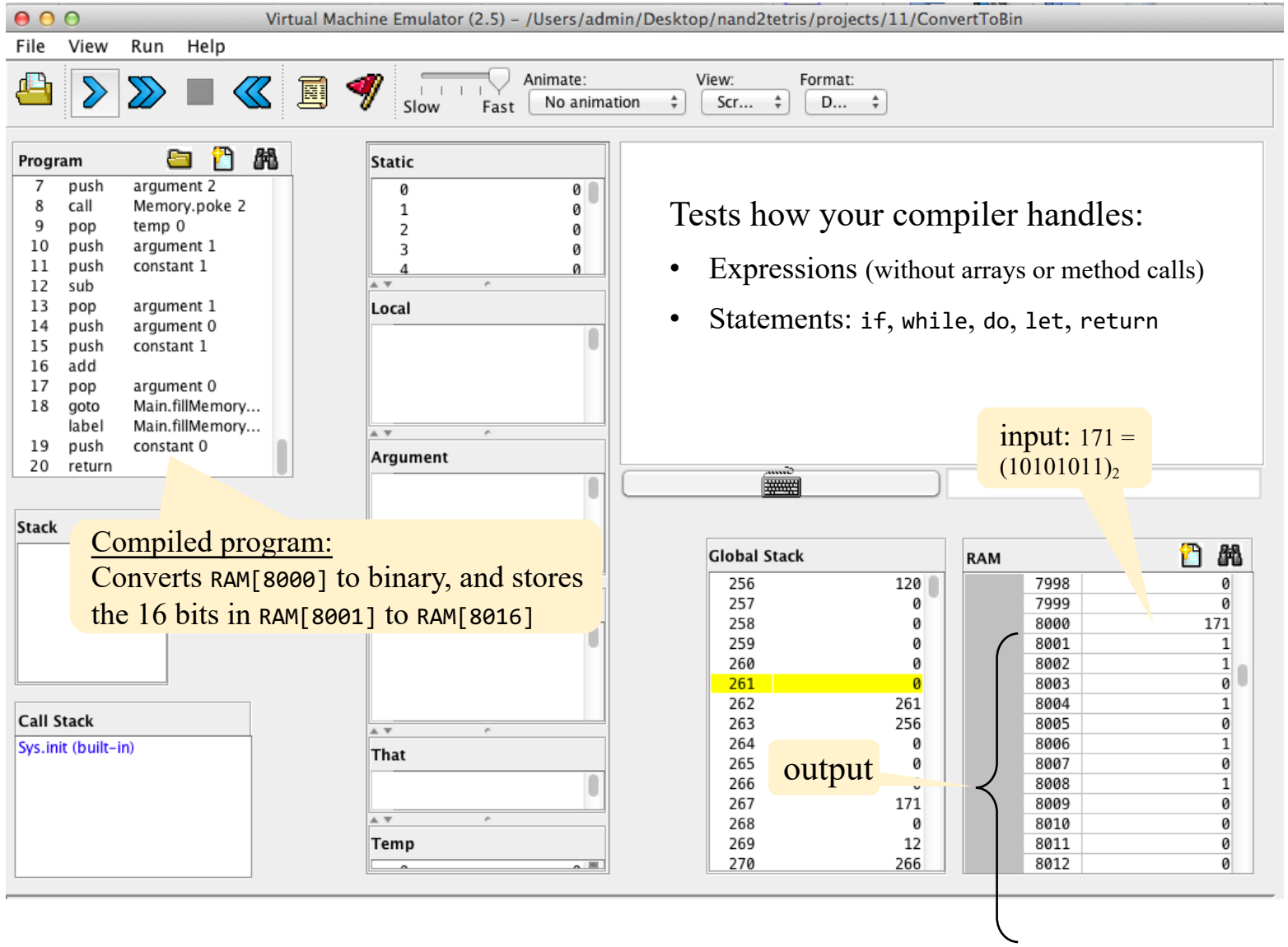
```
function Main.main 0
  push constant 1
  push constant 2
  push constant 3
  call Math.multiply 2
  add
  call Output.printInt 1
  pop temp 0
  push constant 0
  return
```




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



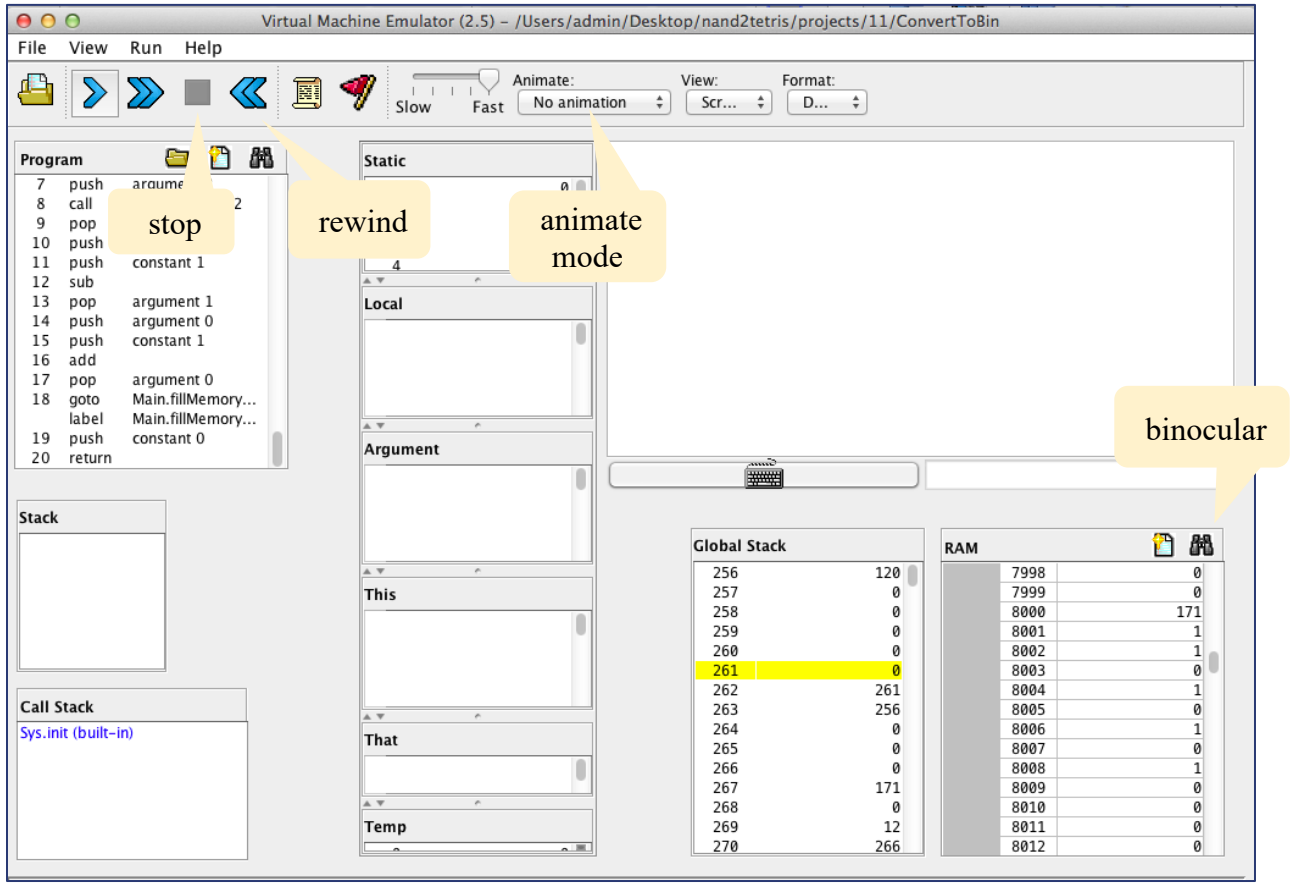
# Test program: Decimal-to-binary conversion

```
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`

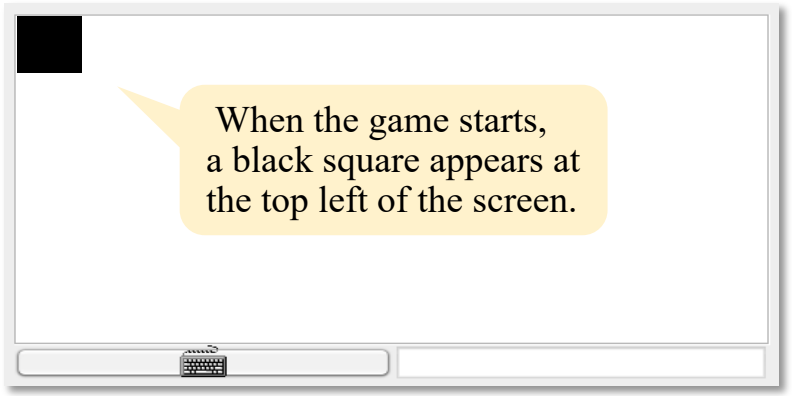
# Test program: Decimal-to-binary conversion

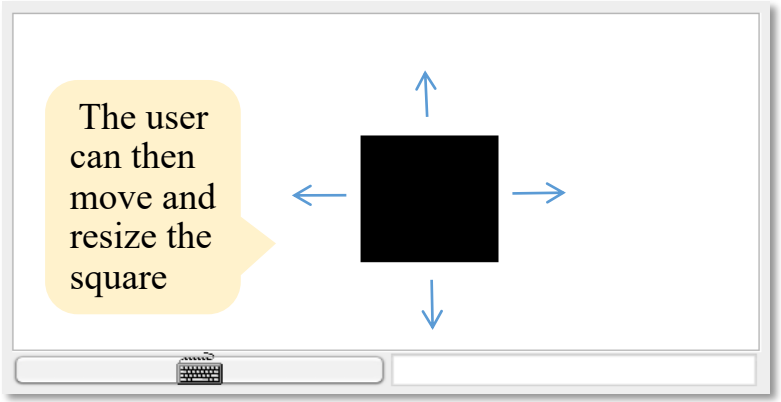



## Testing tips:

- 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

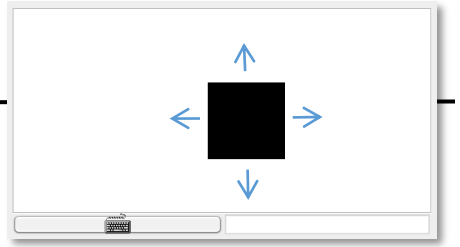



Tests how your compiler handles object-oriented features of the Jack language:

- 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 scree
   method void erase()

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

   /** Decrements the square's size by
   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

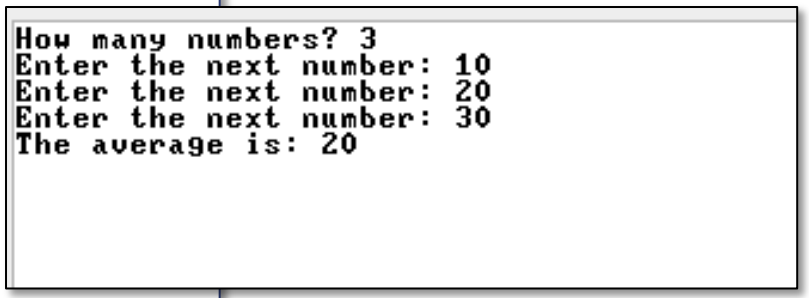
```
/** Represents a square game */
class SquareGame {

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

   /** Constructs a new Square Game */
   constructor SquareGame new() {
      let square = Square.new(0, 0, 30);
      let direction = 0;
      return this;
   }

   /** Disposes this game */
   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

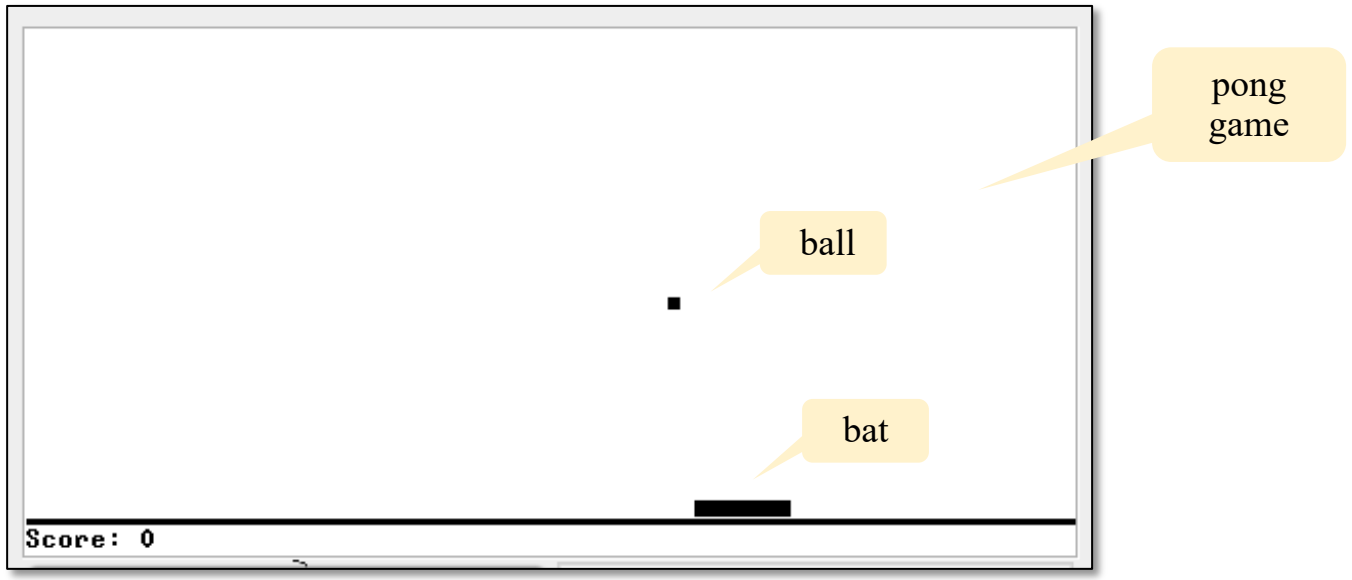
```
/** Computes the average of a sequence of integers */
class Main {
   function void main() {
      var Array a;
      var int length;
      var int i, sum;

      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

# 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'
    field in

    // Dista
    field in

    // Used
    field in
    field bo

    // Locat
    field in

    // last
    field in

    /** Cons
     *  loca
    construc

    ...

    // More

}
```

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 mov
    field int direction; // 1

    /** Constructs a new bat
    constructor Bat new(int A
                        int A
        let x = Ax;
        let y = Ay;
        let width = Awidth;
        let height = Aheight;
        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;
    }
    ...
    /** Runs the g
    method void ru
       var char ke
       while (~exi
          // waits
          while ((
             let k
             do ba
             do mo
          }

          if (key
                dc
          }
          ...
       }
    }
    ...
}
```

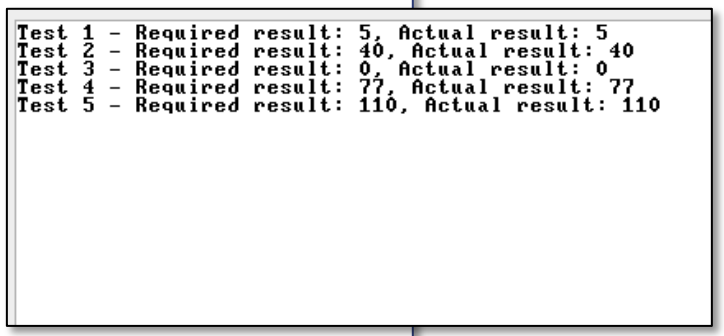
Main.jack

```
/** Main class of the Pong game */
class Main {
   /** Initializes a Pong game and
   starts running it. */
   function void main() {
      var PongGame game;
      do PongGame.newInstance();
      let game = PongGame.getInstance();
      do game.run();
      do game.dispose();
      return;
   }
}
```

# Test program: ComplexArrays

projects/11/ComlpexArrays/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

# Lecture plan

## Compilation

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

## Implementation

- ✓ Standard mapping
- ✓ Proposed design
- ✓ Project 11

Chapter 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