Chapter 11

# Compiler II: Code Genaration

These slides support chapter 11 of the book

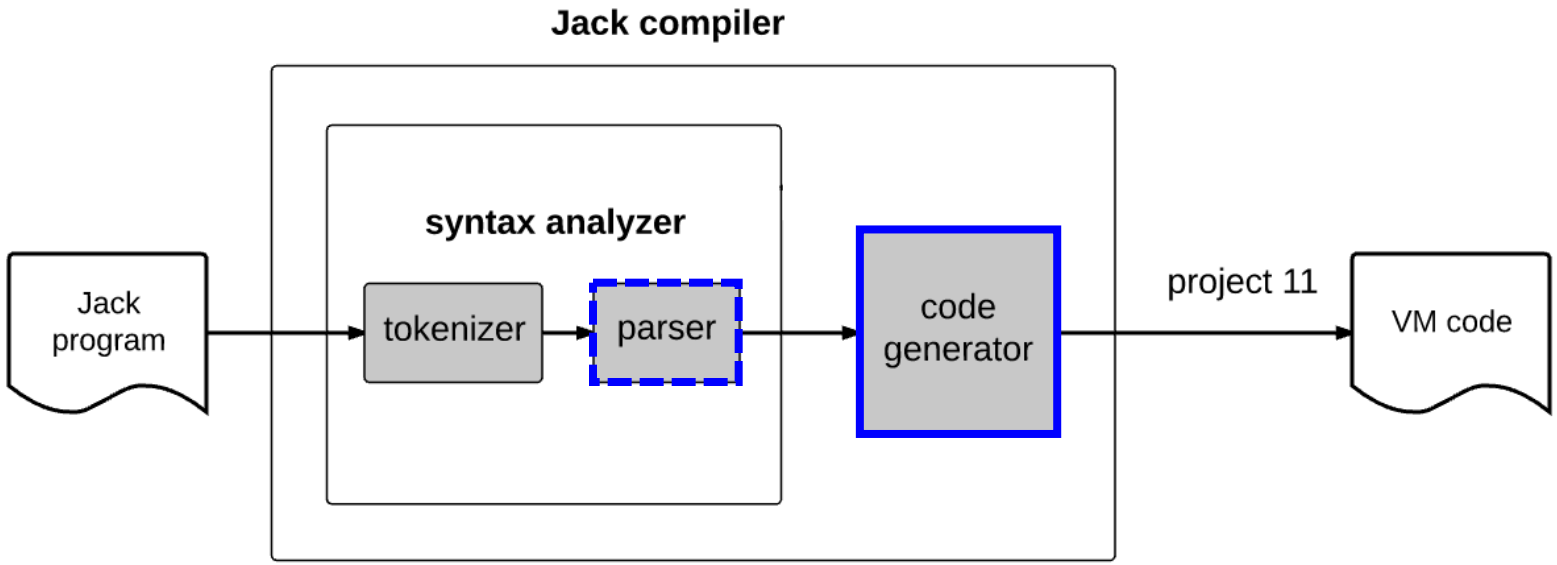
*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press

# Compiler development roadmap

Objective: developing a full-scale compiler




Methodolgy:

- Extending the basic syntax analyzer
- Adding code generation capabilities.

# Program compilation

source code

```
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     // should print (4,6)
     do Output.println();
     do Output.printInt(p1.distance(p3));
     // should print 5
     return;
  }
}
```

```
/** 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;
  }
 // ... more Point methods
```


## Compilation

Each class is compiled separately

# Program compilation

```
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));
    }
    method void print() {}
}
```

class declaration

subroutines:

- constructors
- methods
- functions

# Program compilation

```
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));
    }
    method void print() {}
}
```

Compilation

- class-level code
- subroutine-level code
  - constructors
  - methods
  - functions

# Compilation challenges

- Handling variables
- Handling expressions
- Handling flow of control
- Handling objects
- Handling arrays

The challenge: expressing the above semantics in the VM language.

# Take home lessons

How to implement:

- Procedural programming
  - variables
  - expressions
  - flow of control
- Arrays
- Objects

Techniques:

- Parsing (mostly done)
- Recursive compilation
- Code generation
- Symbol tables
- Memory management.

# Compilation challenges

- **Handling variables**
- Handling expressions
- Handling flow of control
- Handling objects
- Handling arrays

# Variables

High-level (Jack) code

```
class Foo {
   ...
   field int a, b, c;
   static int x, y;
   ...
   method int bar(int a1; int a2)
    {
        var int v1, v2, v3;
        ...
        let c = a2 + (x – v3);
        ...
    }
    ...
}
```

compiler

VM code (pseudo)

```
...
// let c = a2 + (x – v3)
push a2
push x
push v3
sub
add
pop c
...
```

VM code

```
...
push argument 1
push static 0
push local 2
sub
add
pop this 2
...
```

In order to generate VM code,
the compiler must know (among other things):

- Weather each variable is a
  *field*, *static*, *local*, or *argument*
- Weather each variable is the
  *first*, *second*, *third*... variable of its kind

# Variables

High-level (Jack) 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`)
- index (`0`, `1`, `2`, ...)
- scope (class level, subroutine level)

Variable properties:

- Needed for code generation
- Can be recorded and managed using a **symbol table**

# Symbol tables: construction

High-level (Jack) 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 (`field`, `static`)

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

subroutine-level symbol table (`argument`, `local`)

In methods only (not in constructors and functions):

- in addition to the explicit arguments, there is always an implicit argument:
- `argument 0` is always named `this`, and its type is always set to the class name
- Generated by the compiler

# Symbol tables: construction

High-level (Jack) 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 |

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

## Handling variable declarations:

- Each time the compiler detects a variable declaration (`field`, `static`, `argument`, `local`), it adds an entry to the symbol table
- The entry records the variable properties.

# Symbol tables: usage

High-level (Jack) 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 |

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

Handling variables that appear in statements and expressions:

- Each time the compiler detects a variable in some statement or expression, it looks up the variable in the subroutine-level symbol table
- If not found, it looks it up in the class-level symbol table.

# Symbol tables: usage

High-level (Jack) 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 |

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

look-up

source code example:

```
...
let y = y + dy;
...
```

compiler

VM code

```
push this 1     // y
push local 1    // dy
add
pop this 1      // y
```

# Handling variables's life cycle

Static variables:
Seen by all the class subroutines; must exist throughout the program's execution

Local variables:
During run-time, each time a subroutine is invoked, it must get a fresh set of local variables; each time a subroutine returns, its local variables must be recycled

Argument variables:
Same as local variables

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

High-level (Jack) 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));
    }
    ...
}
```

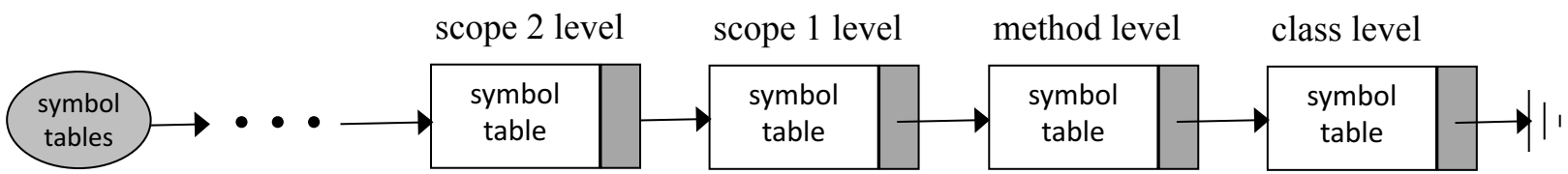
Implementation note

The variable life cycle is managed by the VM implementation during run-time; the compiler need not worry about it!

# End note: handling nested scoping

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

- Some high-level languages feature unlimited nested variable 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.

# Compilation challenges

- Handling variables
- **Handling expressions**
- Handling flow of control
- Handling objects
- Handling arrays

# Expressions

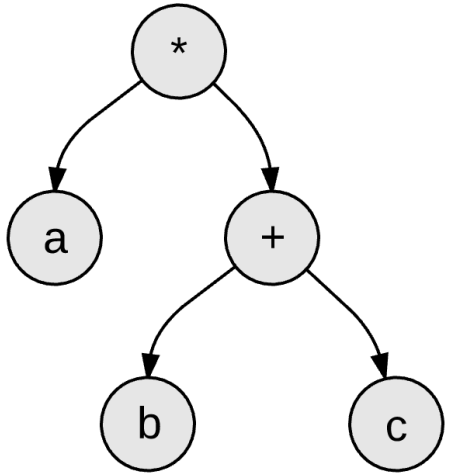
## Definition:

| | |
|---|---|
| expression: | term (op term)* |
| term: | integerConstant \| stringConstant \| keywordConstant \| varName \| varName '[' expression ']' \| subroutineCall \| '(' expression ')' \| unaryOp term |
| subroutineCall: | subroutineName '(' expressionList ')' \| ( className \| varName) '.' subroutineName '(' expressionList ')' |
| expressionList: | (expression (',' expression)* )? |
| op: | '+' \| '-' \| '*' \| '/' \| '&' \| '\|' \| '<' \| '>' \| '=' |
| unaryOp: | '-' \| '~' |
| KeywordConstant: | 'true' \| 'false' \| 'null' \| 'this' |

## Examples:

```
5
x
x + 5
Math.abs(x + 5)
arr[Math.abs(x + 5)]
foo(arr[Math.abs(x + 5)])
...
```

# Parse tree




| prefix | infix | postfix |
|---|---|---|
| `* a + b c` | `a * (b + c)` | `a b c + *` |
| functional | human oriented | stack oriented |

`*(a,+(b,c))`

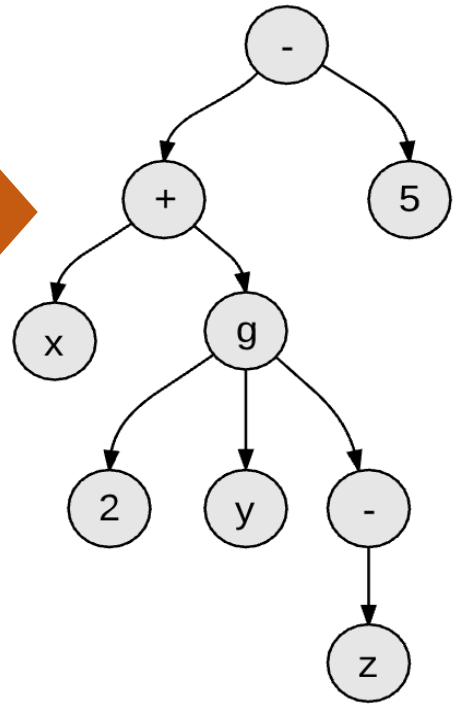
# Generating code for expressions: a two-stage approach

source code

```
x + g(2,y,-z) - 5
```

(infix)

parse tree




code generation

stack-machine code

```
push x
push 2
push y
push z
-
call g
push 5
*
+
```

(postfix)

When executed, the generated code ends up leaving the value of the expression at the top of the stack.

# Generating code for expressions: a one-stage approach

source code

```
x + g(2,y,-z) * 5
```

`codeWrite(`*exp*`)`:

if *exp* is a number *n*:
output "`push` *n*"

if *exp* is a variable *var*:
output "`push` *var*"

if *exp* is "$exp_1$ `op` $exp_2$":
`codeWrite(`$exp_1$`)`,
`codeWrite(`$exp_2$`)`,
output "`op`"

if *exp* is "`op` *exp*":
`codeWrite(`*exp*`)`,
output "`op`"

if *exp* is "$f$ ($exp_1$, $exp_2$, ...)":
`codeWrite(`$exp_1$`)`,
`codeWrite(`$exp_2$`)`, ...,
output "`call` $f$"

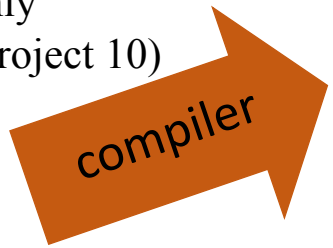
resulting code

stack-machine code

```
push x
push 2
push y
push z
-
call g
push 5
*
+
```

# End note 1: parsing and code generation

parsing
only
(project 10)




source code example:

```
...
let x = a + b - c;
...
```

XML code

```
<letStatement>
   <keyword> let </keyword>
   <identifier> x </identifier>
   <symbol> = </symbol>
   <expression>
      <term> <identifier> a </identifier> </term>
      <symbol> + </symbol>
      <term> <identifier> b </identifier> </term>
      <symbol> - </symbol>
      <term> <identifier> c </identifier> </term>
    <symbol> ; </symbol>
   </expression>
</letStatement>
```

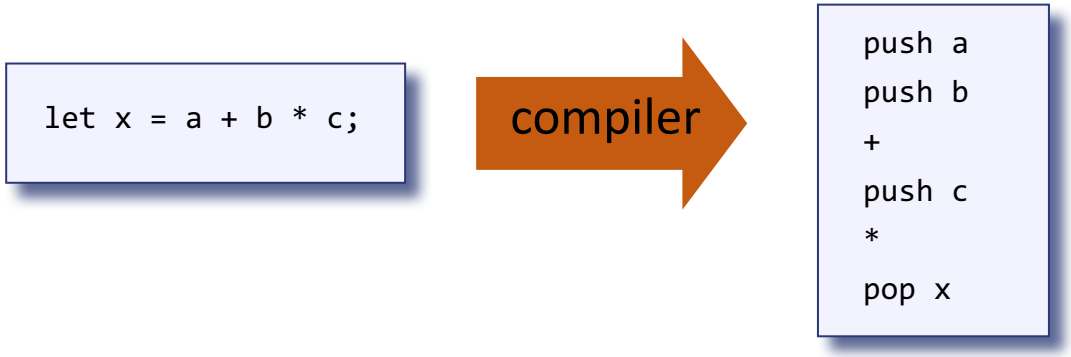

parsing
*and*
code generation
(project 11)

VM (pseudo) code

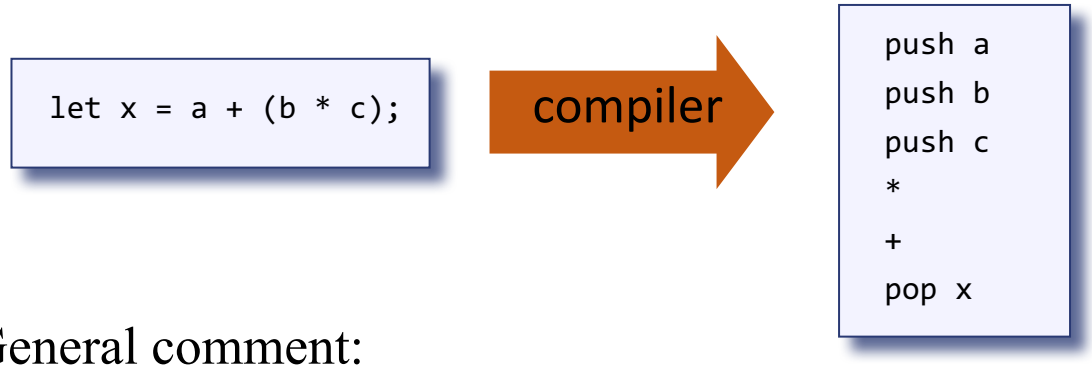
```
push a
push b
+
push c
-
pop x
```

# End note 2: operator priority

- The Jack language definition specifies no operator priority;
- To simplify the compiler writing, we assume left-to-right priority:




The Jack language definition specifies that expressions in parentheses are evaluated first:




General comment:

The Jack language designers decided to leave the question of how to handle operator order priority up to the compiler's implementation.

# Compilation challenges

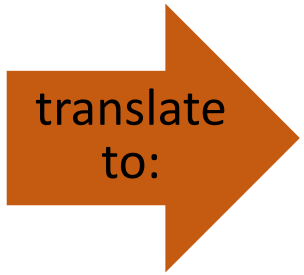
- Handling variables

- Handling expressions
- **Handling flow of control**
- Handling objects
- Handling arrays

# The challenge

high-level code

```
let low = 0;
let high = x;
while ((high – low) > epsilon) {
   let mid = (high + low) / 2;
   if ((mid * mid) > x) {
      high = mid;
   }
   else {
      let low = mid;
   }
}
return low;
```



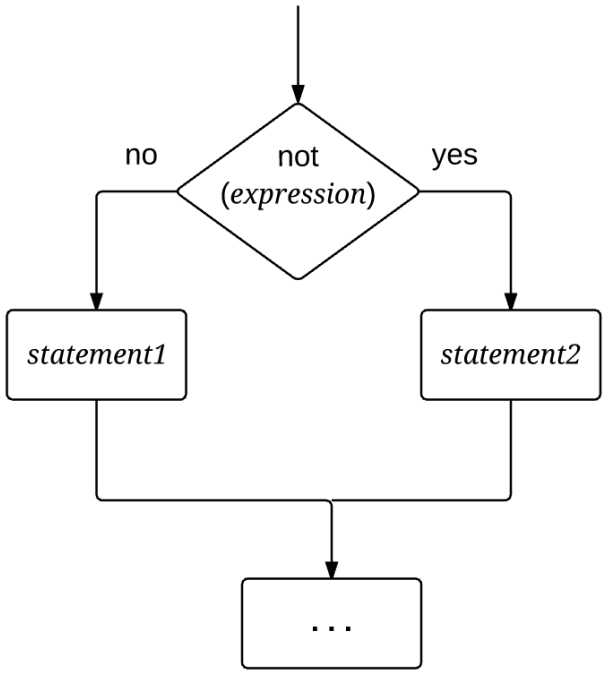

VM code, using:

- `goto`
- `if-goto`
- `label`

# Compiling `if` statements

source code

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




compiler

VM code

```
    compiled (expression)
    not
    if-goto L1
    compiled (statements1)
    goto L2
label L1
    compiled (statements2)
label L2
    ...
```

the labels are generated by the compiler

# Compiling `while` statements

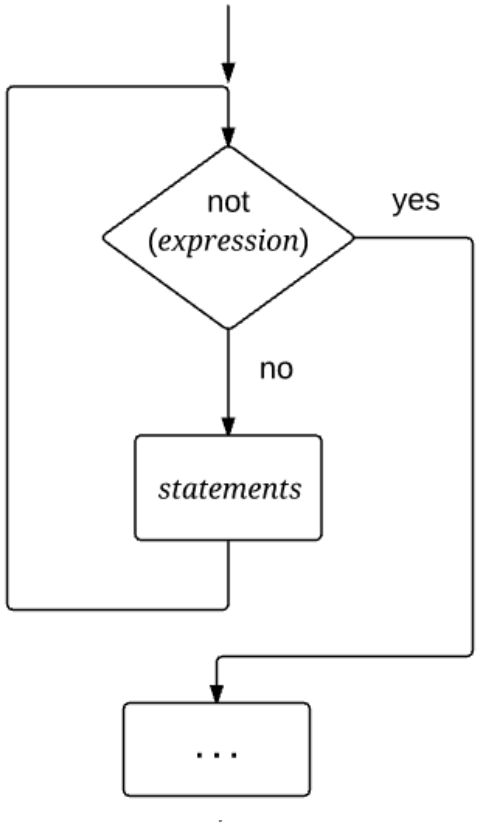
source code

```
while (expression)
    statements
...
```

compiler

VM code

```
label L1
    compiled (expression)
    not
    if-goto L2
    compiled (statements)
    goto L1
label L2
...
```



# Recap

high-level code

```
let low = 0;
let high = x;
while ((high – low) > epsilon) {
   let mid = (high + low) / 2;
   if ((mid * mid) > x) {
      high = mid;
   }
   else {
      let low = mid;
   }
}
return low;
```

Our compiler (front-end) knows how to handle

- Variables
- Expressions
- Flow of control

Our compiler (back-end) knows how to handle

- Functions
- Function call-and-return
- Memory allocation

Conclusion:

We can write a compiler for a simple procedural language!

# Compilation challenges

- Handling variables

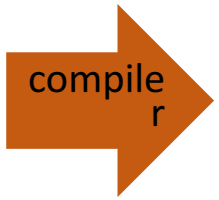
- Handling expressions

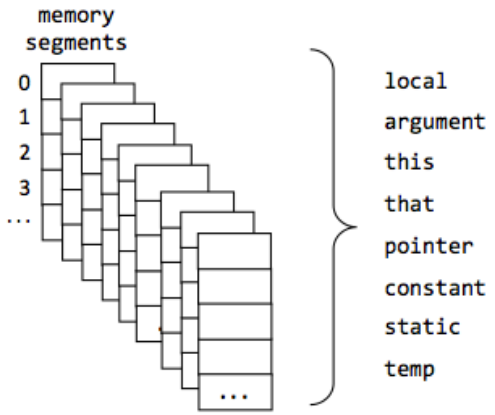
- Handling flow of control
- **Handling objects**
- Handling arrays

# Handling variables: the big picture

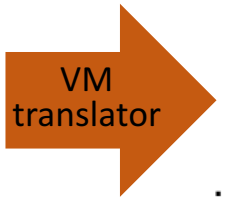
```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax,
                        int   ay)
     {
      let x = ax;
      let y = ay;
      let pointCount =
          pointCount + 1;
      return this;
  }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

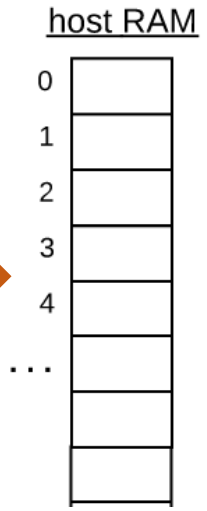
high-level view




intermediate-level view




low-level view

- High-level OO programs operate on high-level, symbolic variables
- Mid-level VM programs operate on virtual memory segments
- Low-level machine programs operate directly on the RAM

The compilation challenge: bridging the gaps.

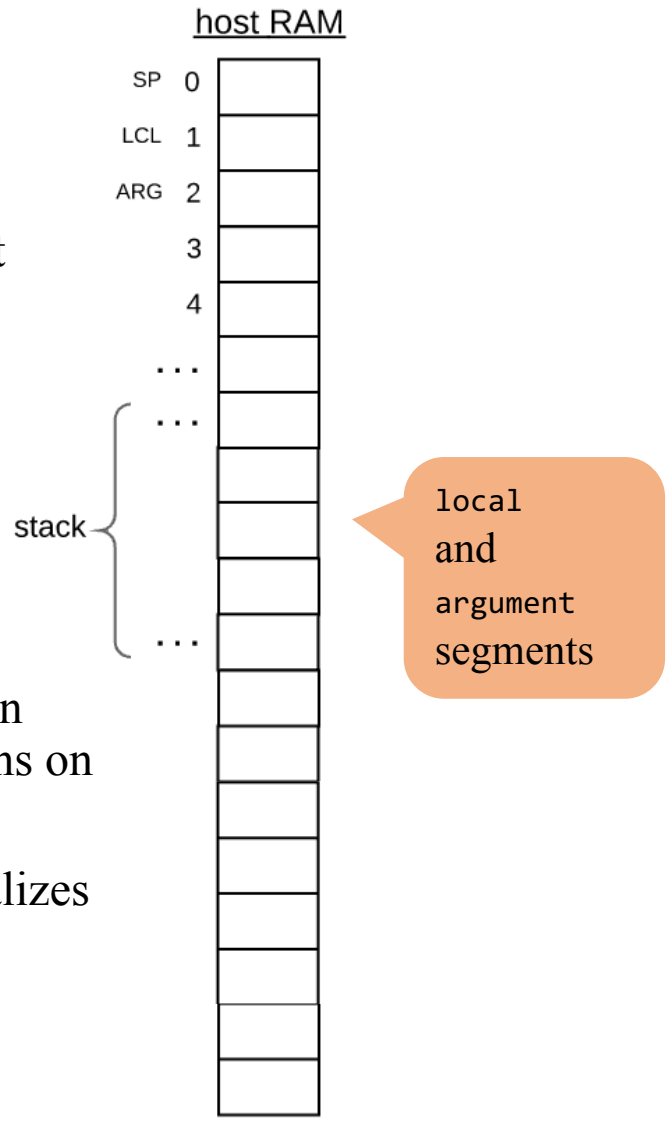
# Handling local and argument variables (review)

Compilation challenge:

- How to represent local and argument variables using the host RAM
- How to initialize and recycle local and argument variables when methods start / return

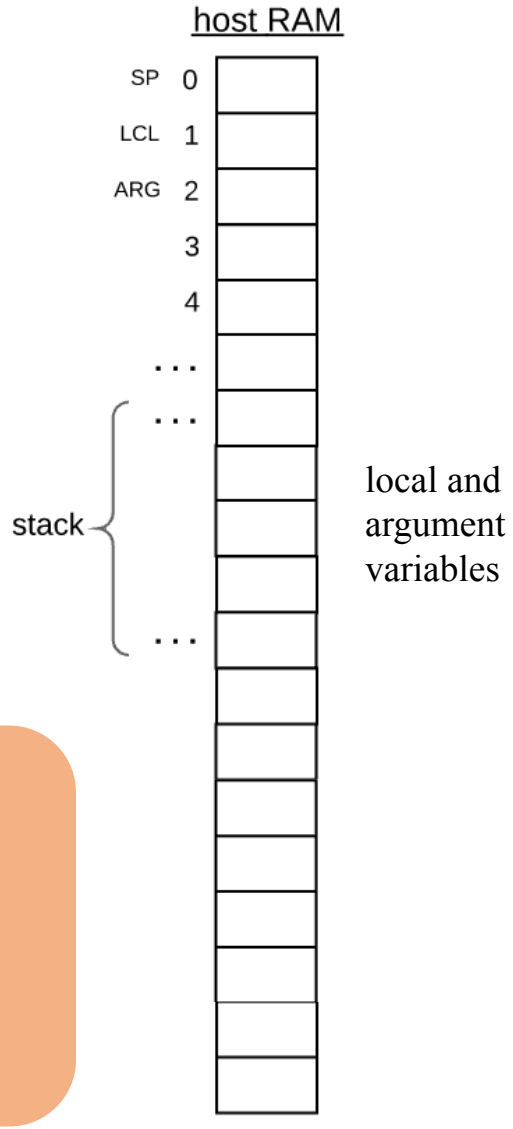
Solution

- The compiler maps the high-level local and argument variables on `local 0`, `1`, ... and on `argument 0`, `1`, ...
- The compiler translates high-level operations on local and argument variables into VM operations on on `local 0`,`1`,... and on `argument 0`,`1`,...
- During run-time, the VM implementation initializes and recycles the `local` and `argument` segments, as needed, using the global stack.



# Handling field variables (object data)

```
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));
    }
    ...
}
```




- There may be many objects (class instances) of type `Point`
- Each represented by its own (also called *private*) set of `x,y` values
- As long as the program is running, all these objects must be managed.

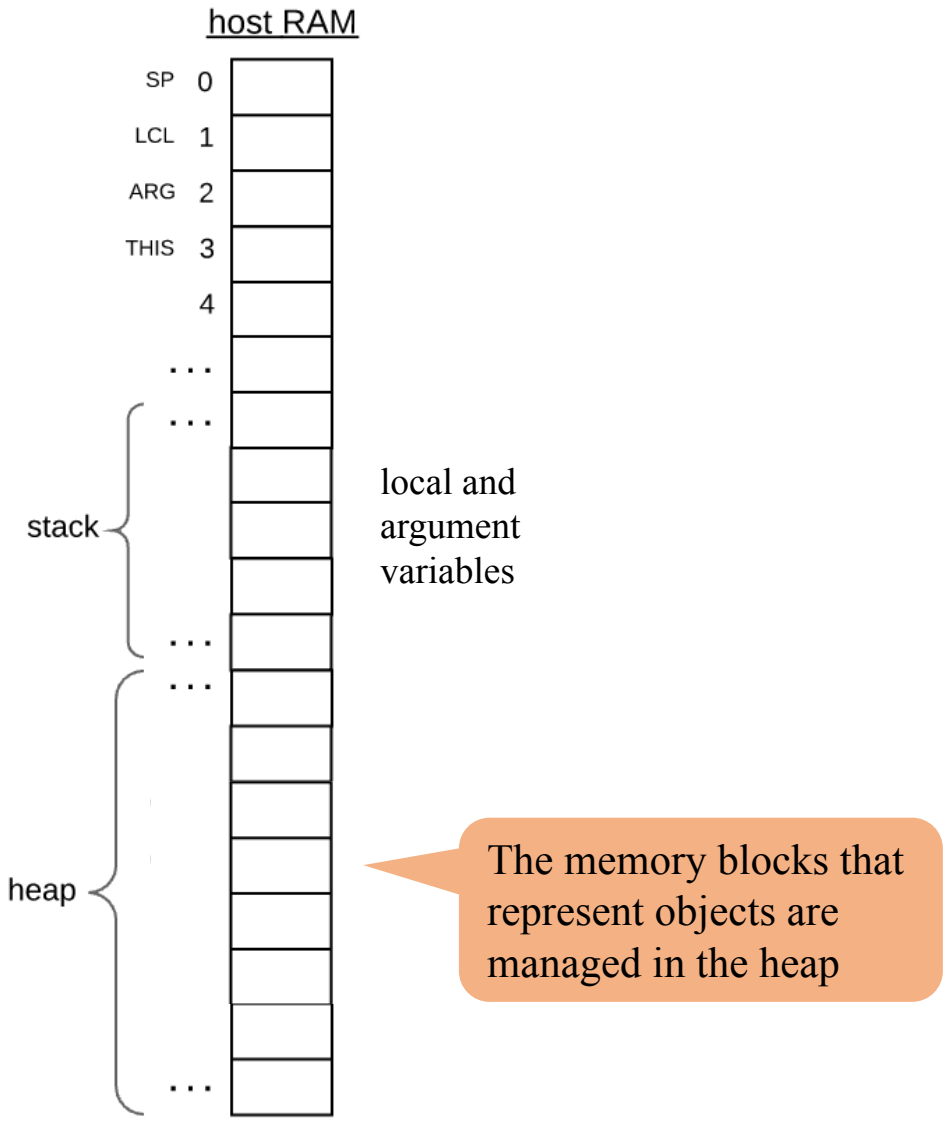
# Handling field variables (object data)

## Abstraction:

- Each object is represented by a bundle of field variable values (each object has a private copy of these values)
- There may be many objects of the same type
- When I call a method, say `bar.foo()`, I want `foo` to operate on a specific object, in this case `bar`

## Implementation

- Each object is managed as a separate memory block, stored in a RAM area named *heap*
- The current object is represented by the segment `this`
- Basic compilation strategy for accessing object data:
  - when we want to operate on a particular object, we set `THIS` to its base address
  - From this point onward, high-level code that uses field *i* of the current object can be translated into VM code that operates on `this` *i*.



# Accessing the RAM using `this`

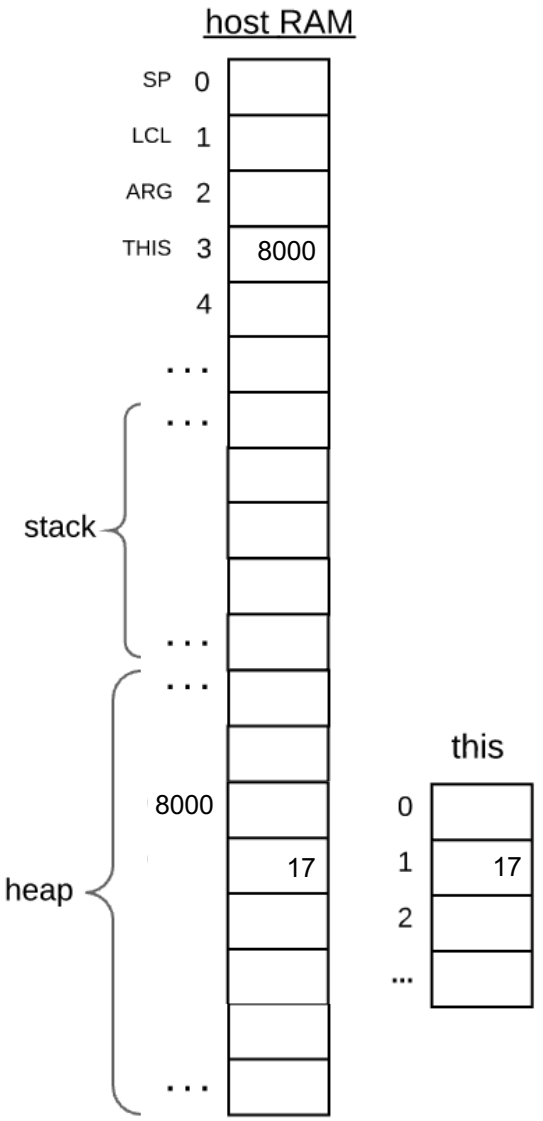
Suppose we wish to access RAM words 8000, 8001, 8002...

| VM code (commands) | Resulting effect |
|---|---|
| `push 8000`<br>`pop pointer 0` | sets `THIS` to `8000` |
| `push/pop this 0` | accesses `RAM[8000]` |
| `push/pop this 1` | accesses `RAM[8001]` |
| `...` | `...` |
| `push/pop this` *i* | accesses `RAM[8000+`*i*`]` |

basic technique for accessing object data

Example:

```
// Sets the second element
// of the memory block
// beginning at 8000 to 17:
push 8000
pop pointer 0
push 17
pop this 1
```



# Compilation challenges


Handling variables


Handling expressions


Handling flow of control

- Handling objects
  - **Construction**
    - Compiling “new”
    - Compiling constructors
  - Manipulation
- Handling arrays

# Object construction: the big picture

some class

```
...
var Point p1;
...
let p1 = Point.new(2,3);
...
```

caller

Point class

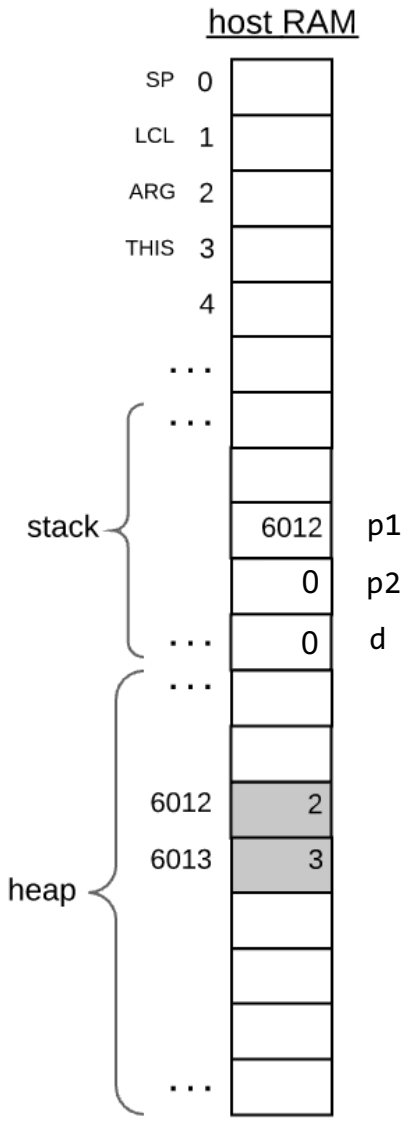
```
class Point {
   ...
   constructor Point new(...)
   ...
}
```

callee

# Object construction: the caller's side

source code

```
...
var Point p1, p2;
var int d;
...
let p1 = Point.new(2,3);
...
```



# Object construction: the caller's side

source code

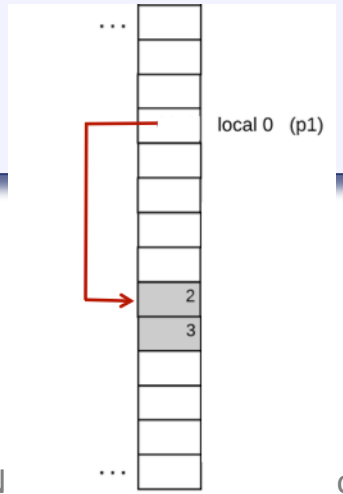
```
...
var Point p1, p2;
var int d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

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

compiled VM (pseudo) code

```
// var Point p1, p2
// var int d;
// The compiler updates the subroutine's symbol table.
// No code is generated.
...
// let p1 = Point.new(2,3)
// Subroutine call:
push 2
push 3
call Point.new
pop p1    // p1 = base address of the new object
// let p2 = Point.new(5,7)
// Similar, code omitted.
...
```

Contract: the caller assumes that the constructor's code (i) arranges a memory block to store the new object, and (ii) returns its base address to the caller.

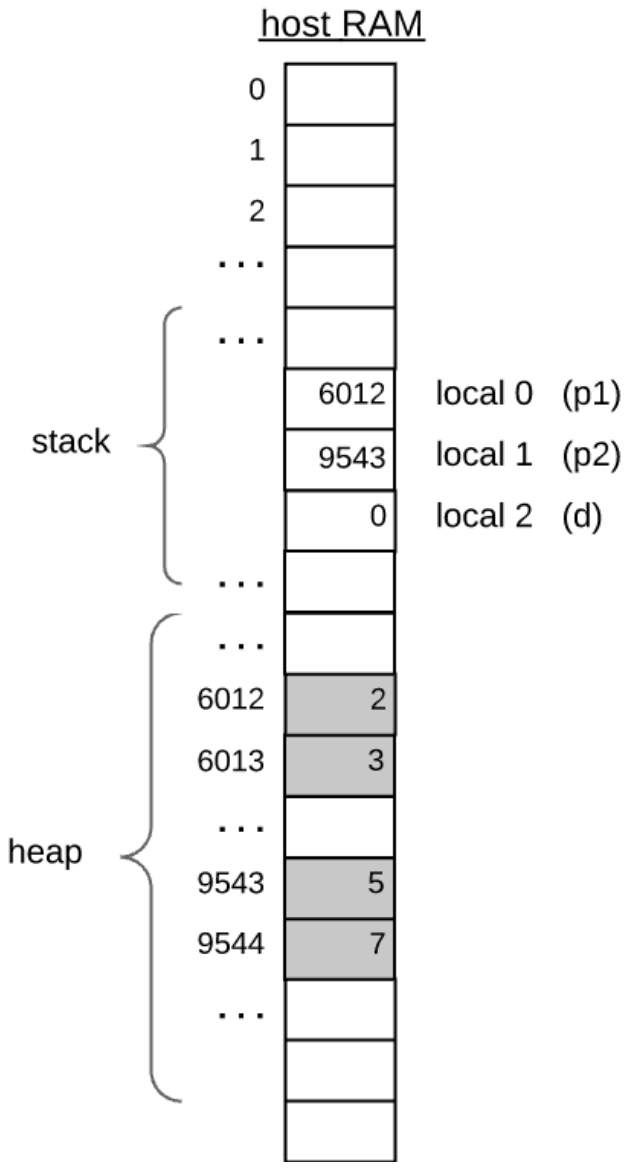


# Resulting impact

source code

```
...
var Point p1, p2;
var int d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

- During compile-time,
  the compiler maps `p1` on `local 0`,
  `p2` on `local 1`, and `d` on `local 2`.
- During run-time, the execution of the constructor's code effects the creation of the objects themselves, on the heap.



# Object construction: the big picture

some class

```
var Point p1;
...
let p1 = Point.new(2,3);
```

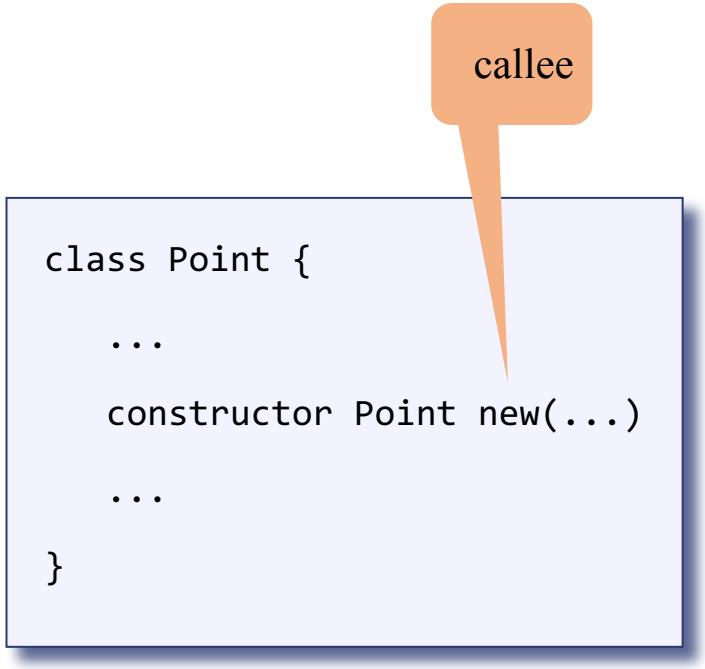
caller

Point class

```
class Point {
   ...
   constructor Point new(...)
   ...
}
```

callee

# Object construction: the big picture



# Object construction: the big picture

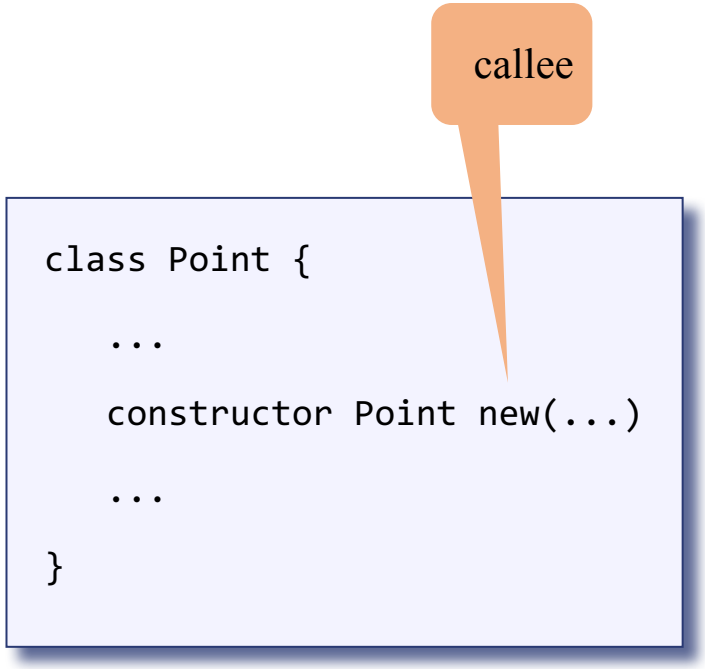
Requirements:

- The constructor must create a new object
- Just like any method, the constructor must be able to manipulate the fields of the current object
- in the case of construcors, the current object is the newly constructed object

Implementation:

The compiled constructor's code ...

- starts by creating a memory block for representing the new object (details later)
- Next, it sets `THIS` to the base address of this block (using `pointer`)
- From now on, the constructor's code can access the object's fields using the `this` segment.



# Compiling constructors

caller's code

```
var Point p1;
...
let p1 = Point.new(2,3);
...
```

Caller's code: compiled separately, shown for context

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

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

class-level

compiled constructor's code

 Copyright © Noam Nisan and Shimon Schocken 

# Compiling constructors

caller's code

```
var Point p1;
...
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;
  }
  ...
```

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

class-level

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

constructor-level

compiled constructor's code

```
...
// constructor Point new(int ax, int ay)
// The compiler creates the subroutine's symbol table.
// The compiler figures out the size of an object of this
// class (n), and writes code that calls Memory.alloc(n).
// This OS method finds a memory block of the required
// size, and returns its base address.
push 2    // two 16-bit words are required (x and y)
call Memory.alloc 1    // one argument
pop pointer 0 // anchors this at the base address
// let x = ax; let y = ay;
push argument 0
pop this 0
push argument 1
pop this 1
// let pointCount = pointCount + 1;
push static 0
push 1
add
pop static 0
// return this
push pointer 0
return    // returns the base address of the new object
```

# Compiling constructors

caller's code

```
var Point p1;
...
let p1 = Point.new(2,3);
...
```

compiled constructor's code

```
...
// constructor Point new(int ax, int ay)
// The compiler creates the subroutine's symbol table.
// The compiler figures out the size of an object of this
// class (n), and writes code that calls Memory.alloc(n).
// This OS method finds a memory block of the required
// size, and returns its base address.
push 2   // two 16-bit words are required (x and y)
call Memory.alloc 1   // one argument
pop pointer 0 // anchors this at the base address
// let x = ax; let y = ay;
push argument 0
pop this 0
push argument 1
pop this 1
// let pointCount = pointCount + 1;
push static 0
push 1
add
pop static 0
// return this
push pointer 0
return   // returns the base address of the new object
```

 Copyright © Noam Nisan and Shimon Schocken 

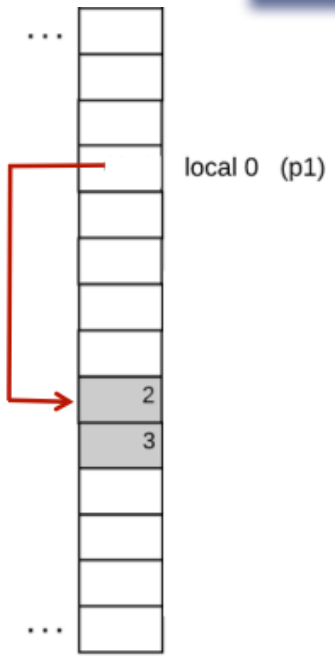
# Compiling constructors

caller's code

```
var Point p1;
...
let p1 = Point.new(2,3);
...
```

compiled caller's code

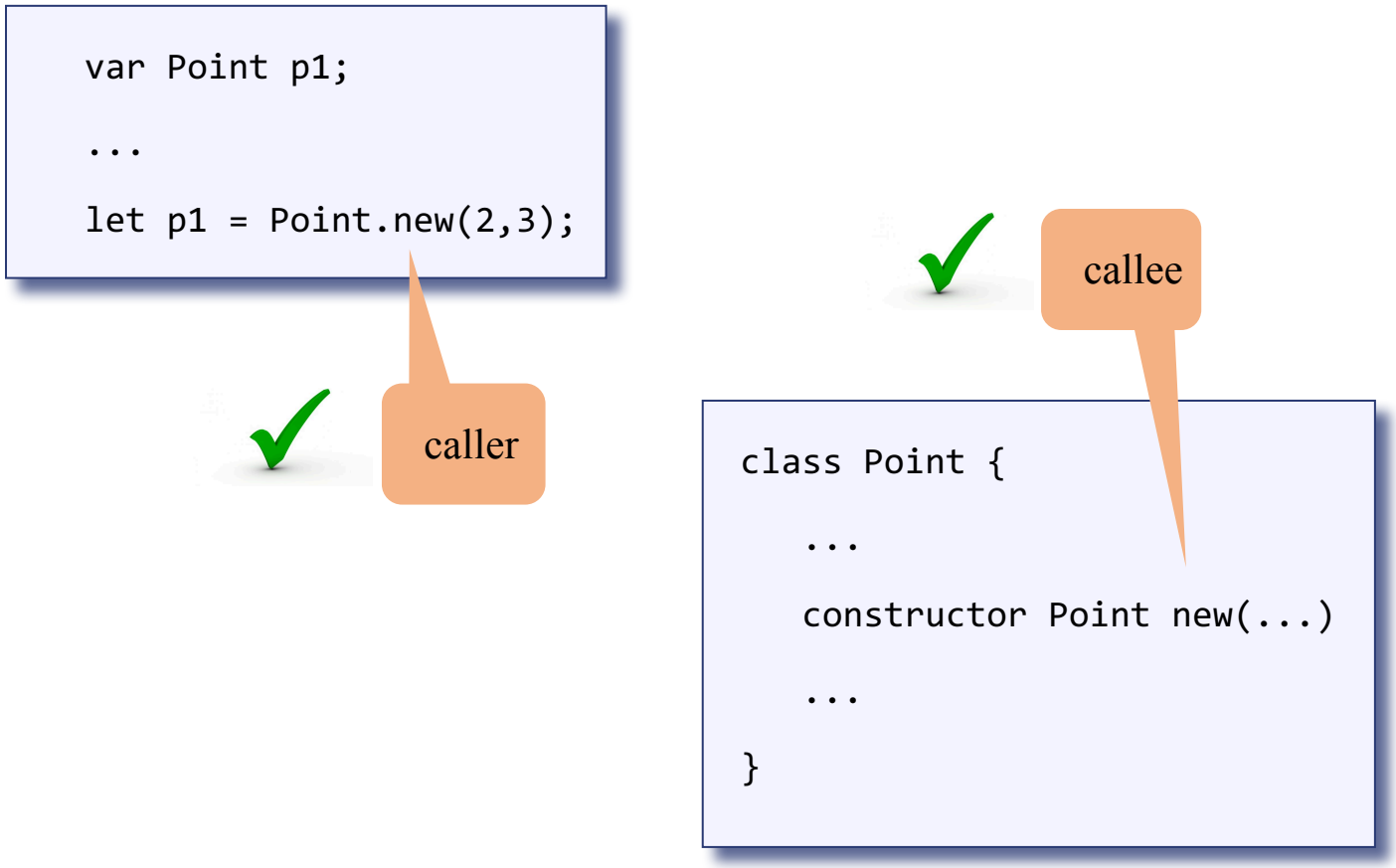
```
// let p1 = Point.new(2,3)
push 2
push 3
call Point.new
pop p1
```




compiled constructor's code

```
...
// constructor Point new(int ax, int ay)
// The compiler creates the subroutine's symbol table.
// The compiler figures out the size of an object of this
// class (n), and writes code that calls Memory.alloc(n).
// This OS method finds a memory block of the required
// size, and returns its base address.
push 2   // two 16-bit words are required (x and y)
call Memory.alloc 1   // one argument
pop pointer 0 // anchors this at the base address
// let x = ax; let y = ay;
push argument 0
pop this 0
push argument 1
pop this 1
// let pointCount = pointCount + 1;
push static 0
push 1
add
pop static 0
// return this
push pointer 0
return   // returns the base address of the new object
```

# Object construction: the big picture



# Compilation challenges

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling flow of control
- Handling objects
  - ✓ Construction
    - **Manipulation**
      - Compiling method calls (caller side)
      - Compiling methods (callee side)
- Handling arrays

# Object manipulation: high-level

some class

```
...
var Point p1, p2, p3;
var int x, d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
let x = p1.getx();
let p3 = p1.plus(p2);
let d = p1.distance(p2);
...
```

caller

Point class

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

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

   method int getx() {}

   method int gety() {}

   method 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));
    }
    method void print() {}
}
```

callee

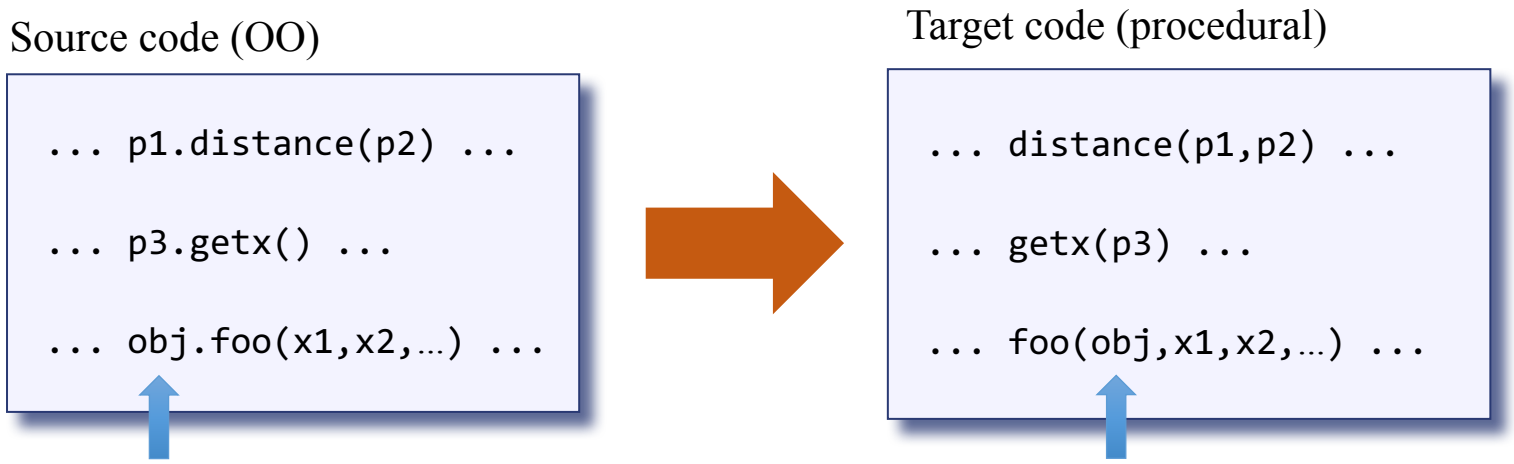
# Compiling method calls

some class

```
...
var Point p1, p2, p3;
var int x, d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
let x = p1.getx();
let p3 = p1.plus(p2);
let d = p1.distance(p2);
...
```

caller

# Compiling method calls

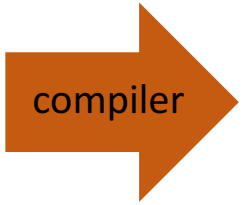



- The target language is *procedural*
- Therefore, the compiler must generate procedural code
- Solution: when calling a method, the object on which the method is called to operate is always passed as a first, implicit argument.

# Compiling method calls: the general technique

source code

```
obj.foo(x1,x2,...)
```

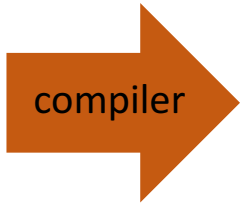



generated VM (pseudo) code

```
// obj.foo(x1,x2,...)
// Pushes the object on which the method
// is called to operate (implicit argument),
// then pushes the method arguments,
// then calls the method for its effect.
push obj
push x1
push x2
push ...
call foo
```

example:

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




generated VM (pseudo) code

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

# Object manipulation: the big picture

some class

```
...
var Point p1, p2, p3;
var int x, d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
let x = p1.getx();
let p3 = p1.plus(p2);
let d = p1.distance(p2);
...
```

caller

Point class

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

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

   method int getx() {}

   method int gety() {}

   method 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));
    }
    method void print() {}
}
```

callee

# Compiling methods

Methods are designed to operate on the current object (`this`):

Therefore, each method's code needs access to the object's *fields*

How to access the object's fields:

- The method's code can access the object's *i*-th field by accessing `this` *i*
- To enable this, the method's code must anchor the `this` segment on the object's data, using `pointer`

`Point` class

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

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

   method int getx() {}

   method int gety() {}

   method 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));
    }
    method void print() {}
}
```

# Compiling methods

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

Caller's code: compiled separately, shown for context

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Distance 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 | field | 0 |
| y | int | field | 1 |
| pointCount | int | static | 0 |

class-level

compiled VM code

# Compiling methods

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

Caller's code: compiled separately, shown for context

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Distance 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 | field | 0 |
| y | int | field | 1 |
| pointCount | int | static | 0 |

class-level

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

method-level

## compiled VM code

```
...
// method int distance(Point other)
// var int dx,dy
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method was called to operate.
push argument 0
pop pointer 0   // THIS = argument 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
```

# Compiling methods

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

Caller's code: compiled separately, shown for context

compiled VM code

```
...
// method int distance(Point other)
// var int dx,dy
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method was called to operate.
push argument 0
pop pointer 0   // THIS = argument 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
```

# Compiling methods

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

Caller's code: compiled separately, shown for context

compiled caller (pseudo) code

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

compiled VM code

```
...
// method int distance(Point other)
// var int dx,dy
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method was called to operate.
push argument 0
pop pointer 0   // THIS = argument 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
```

# Compiling void methods

```
...
do p1.print();
...
```

Caller's code: compiled separately, shown for context

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Prints the point as (x,y) */
  method void print() {
     ...
  }
  ...
}
```

compiled VM code

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

class-level

# Compiling void methods

```
...
do p1.print();
...
```

Caller's code: compiled separately, shown for context

```
/** Represents a Point. */
class Point {
  field int x, y;
  static int pointCount;
  ...
  /** Prints the point as (x,y) */
  method void print() {
     ...
  }
  ...
}
```

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

class-level

| name | type | kind | # |
|---|---|---|---|
| this | Point | argument | 0 |

method-level

compiled VM code

```
...
// method void print()
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method is called to operate.
push argument 0
pop pointer 0   // THIS = argument 0
...
// Returns a value (all methods must return a value)
push constant 0
return
```

# Compiling void methods

```
...
do p1.print();
...
```

Caller's code: compiled separately, shown for context

compiled VM code

```
...
// method void print()
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method is called to operate.
push argument 0
pop pointer 0   // THIS = argument 0
...
// Methods must return a value.
push constant 0
return
```

# Compiling void methods

```
...
do p1.print();
...
```

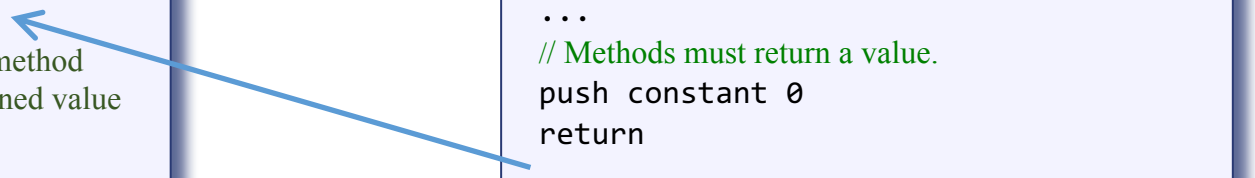
Caller's code: compiled separately, shown for context

compiled VM code

```
...
// method void print()
// The compiler constructs the method's
// symbol table. No code is generated.
// Next, it generates code that associates the
// this memory segment with the object on
// which the method is called to operate.
push argument 0
pop pointer 0   // THIS = argument 0
...
// Methods must return a value.
push constant 0
return
```

compiled caller (pseudo) code

```
...
push p1
call Point.print
// the caller of a void method
// must dump the returned value
pop temp 0
...
```



## Compiler writers, heads up:

**Calle's contract** (compiled code): before terminating,
I must must return a value

(by convention, void methods return 0)

**Caller's contract** (compiled code): after calling a voide method,
I must dump the topmost stack value

# Object manipulation: the big picture

some class

```
...
var Point p1, p2, p3;
var int x, d;
...
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
let x = p1.getx();
let p3 = p1.plus(p2);
let d = p1.distance(p2);
...
```

caller

Point class

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

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

   method int getx() {}

   method int gety() {}

   method 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));
    }
    method void print() {}
}
```

callee

# Compilation challenges

- Handling variables

- Handling expressions

- Handling flow of control

- Handling objects
- **Handling arrays**

# Compilation challenges

- Handling variables

- Handling expressions

- Handling flow of control

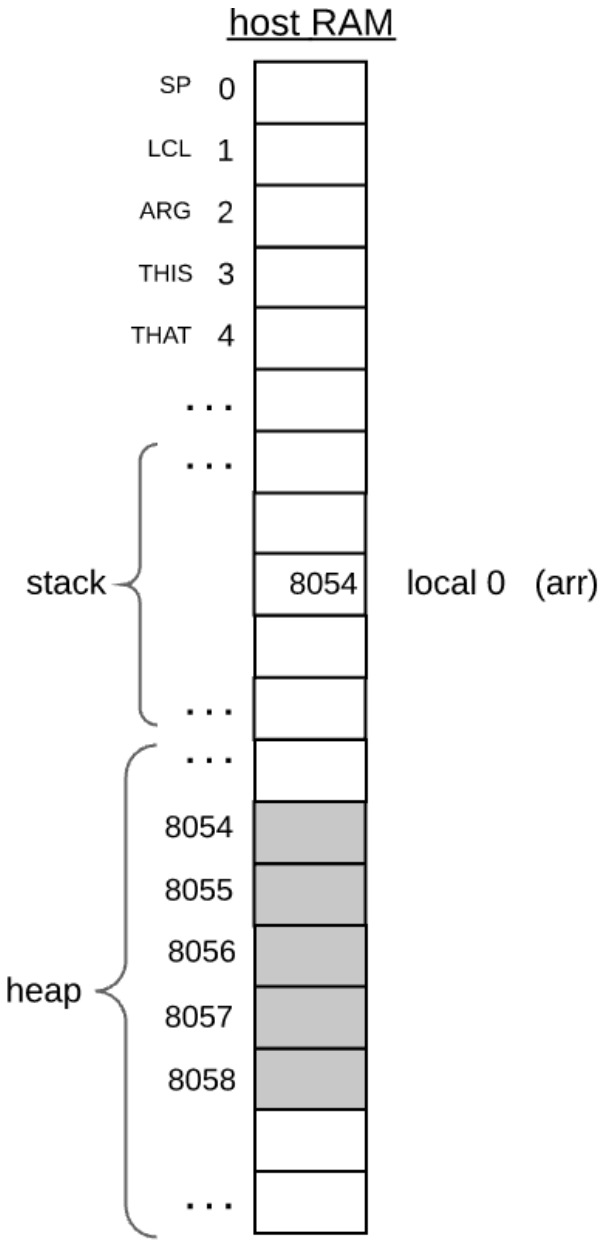
- Handling objects
- **Handling arrays**
  - Array construction
  - Array manipulation

# Array construction

```
var Array arr;
...
let arr = Array.new(5);
...
```

## Code generation

- `var Array arr`:
  generates no code;
  only effects the symbol table
- `let arr = Array.new(`*n*`)`:
  from the caller's perspective,
  handled exactly like object construction



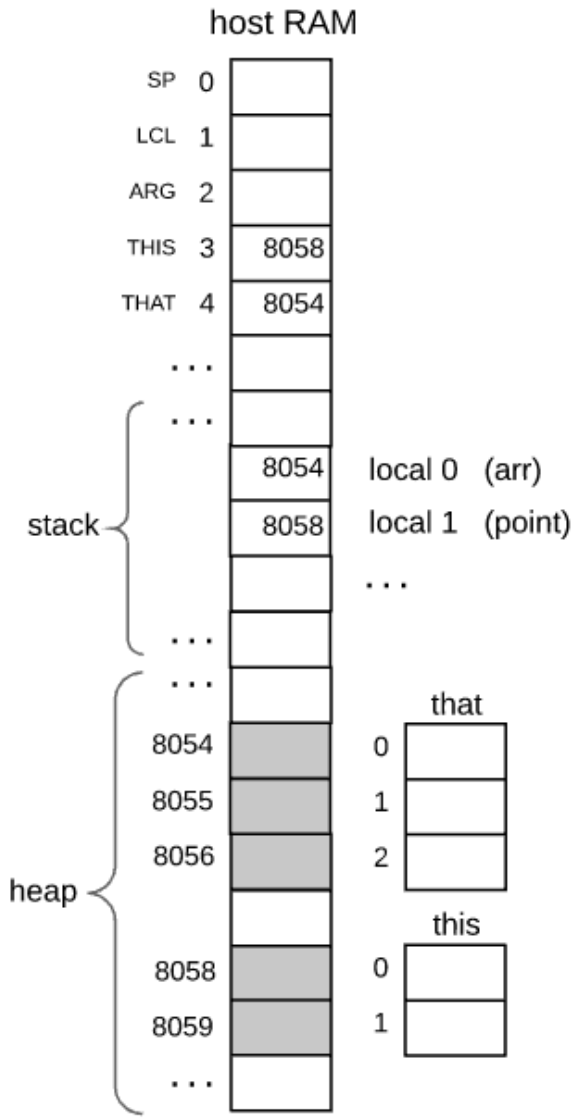
# Compilation challenges

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling flow of control
- ✓ Handling objects
- **Handling arrays**
  - ✓ Array construction
  - Array manipulation

# `this` and `that` (reminder)

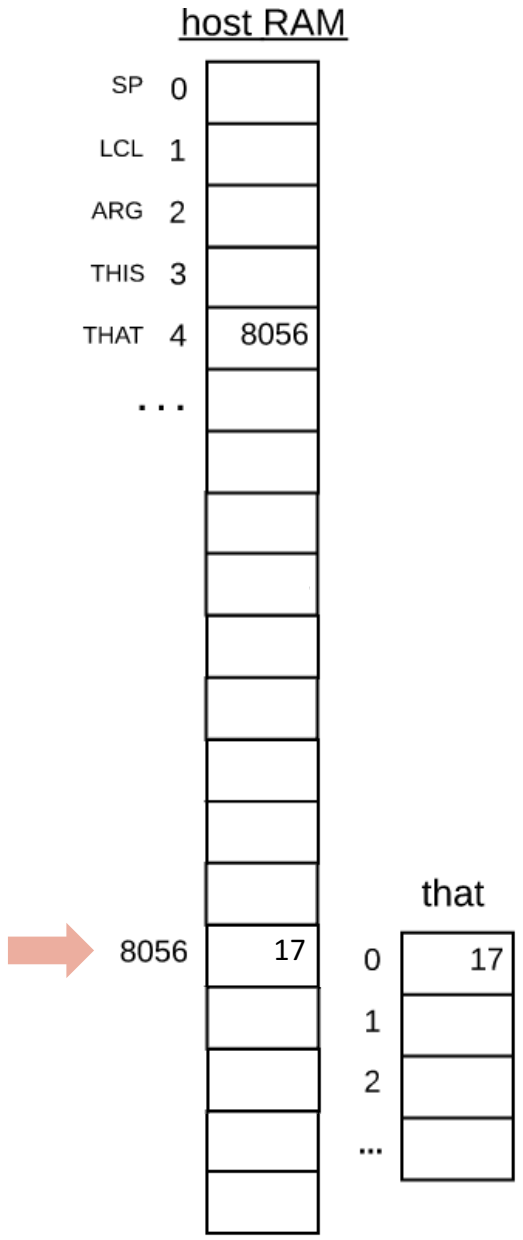
Two "portable" virtual memory segments that can be aligned to different RAM addresses

| | `this` | `that` |
|---|---|---|
| VM use: | used to represent the values of the current *object* | used to represent the values of the current *array* |
| pointer (base address) | `THIS` | `THAT` |
| how to set: | `pop pointer 0` (sets `THIS`) | `pop pointer 1` (sets `THAT`) |



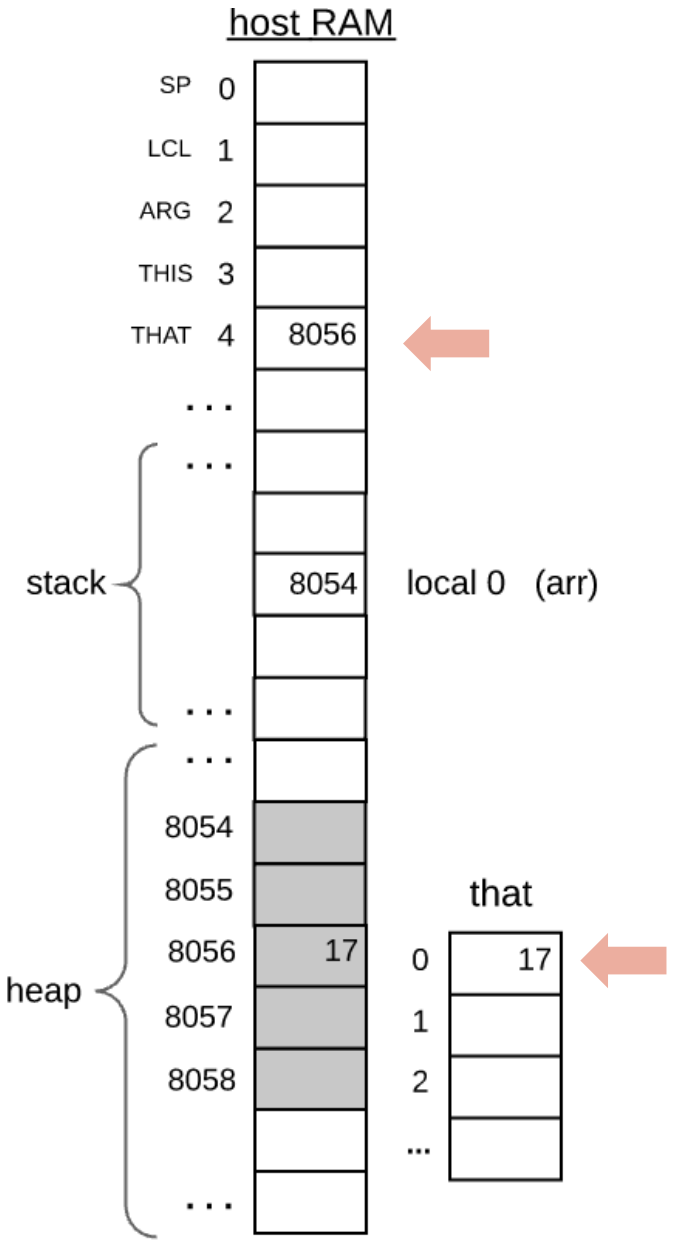
# Example: RAM access using `that`

```
// RAM[8056] = 17
push 8056
pop pointer 1
push 17
pop that 0
```



# Array access

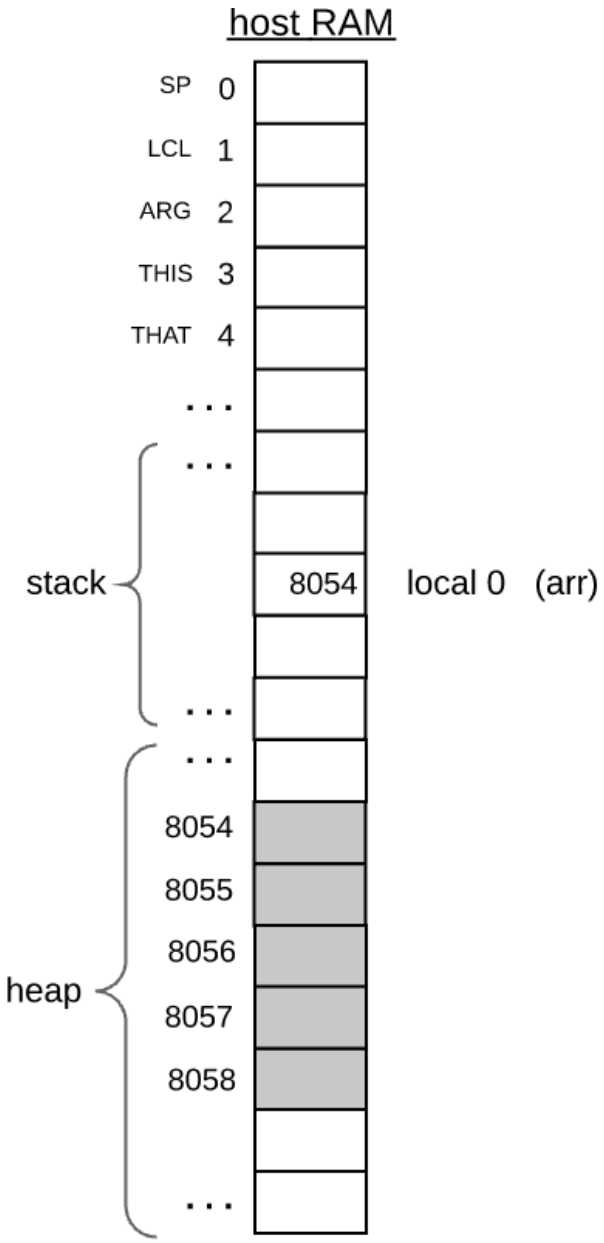
```
// let arr[2] = 17
push arr // base address
push 2   // offset
add
pop pointer 1
push 17
pop that 0
```



# Array access

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

unfortunately,
there's a problem



# Array access

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
```


- Our compilation strategy is generating code on the fly, left to right
- The term within the square brackets is treated as an expression, and compiled as such.

There are many other scenarios that will cause similar problems, e.g.

```
let a[a[i]+a[j+a[a[3]]] = a[j+2]
```

# Array access

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:

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

# Array access

General solution for generating array access code

```
// let arr[expression1] = expression2
push arr
VM code for computing and pushing the value of expression1
add                 // top stack value = RAM address of arr[expression1]
VM code for computing and pushing the value of expression2
pop temp 0      // temp 0 = the value of expression2
pop pointer 1
push temp 0
pop that 0
```

If needed, the evaluation of *expression*$_2$ can set and use `pointer 1` and `that 0` safely

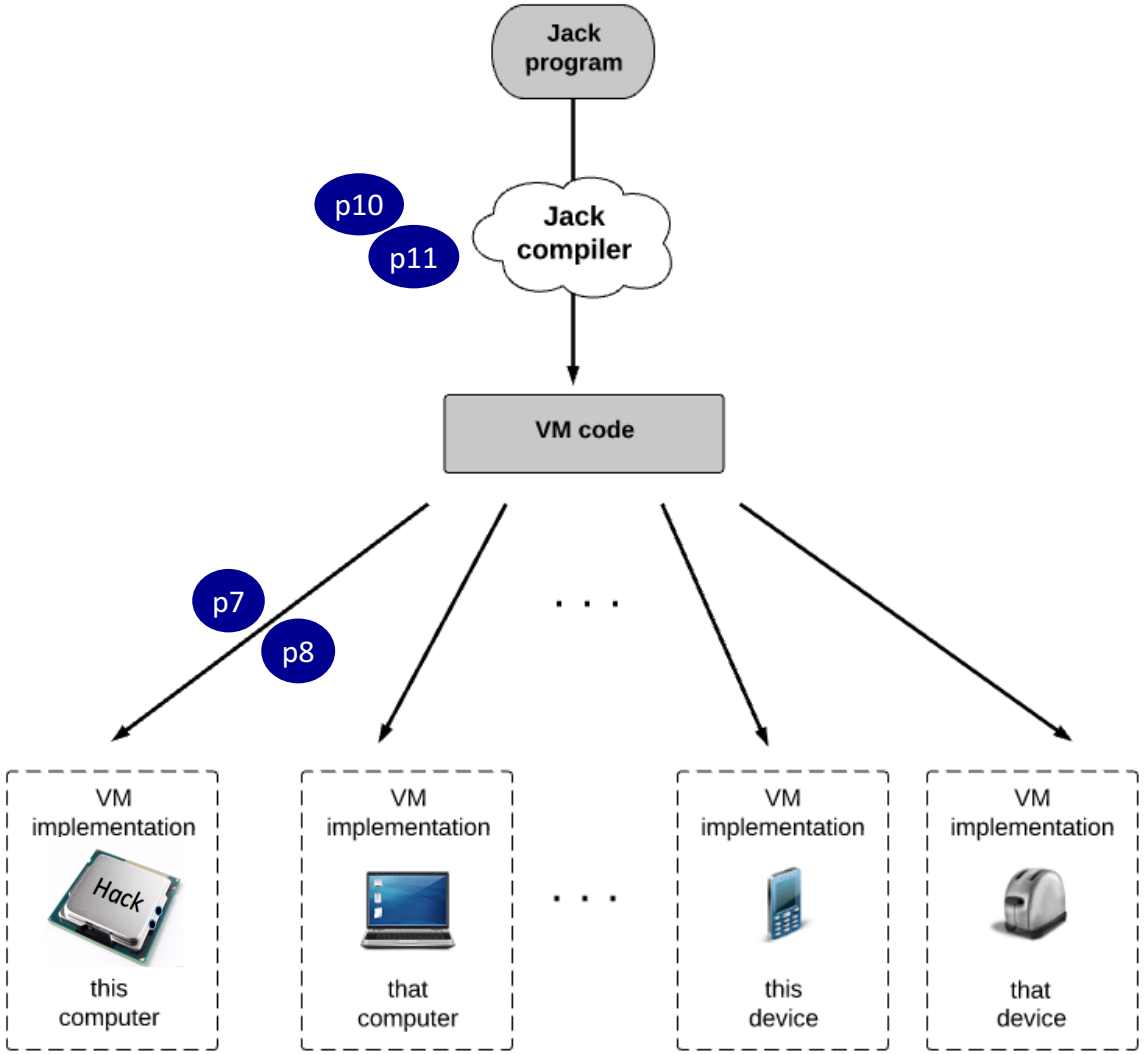
What about handling, say, `let a[a[i]] = a[b[a[b[j]]]]` ?

No problem...

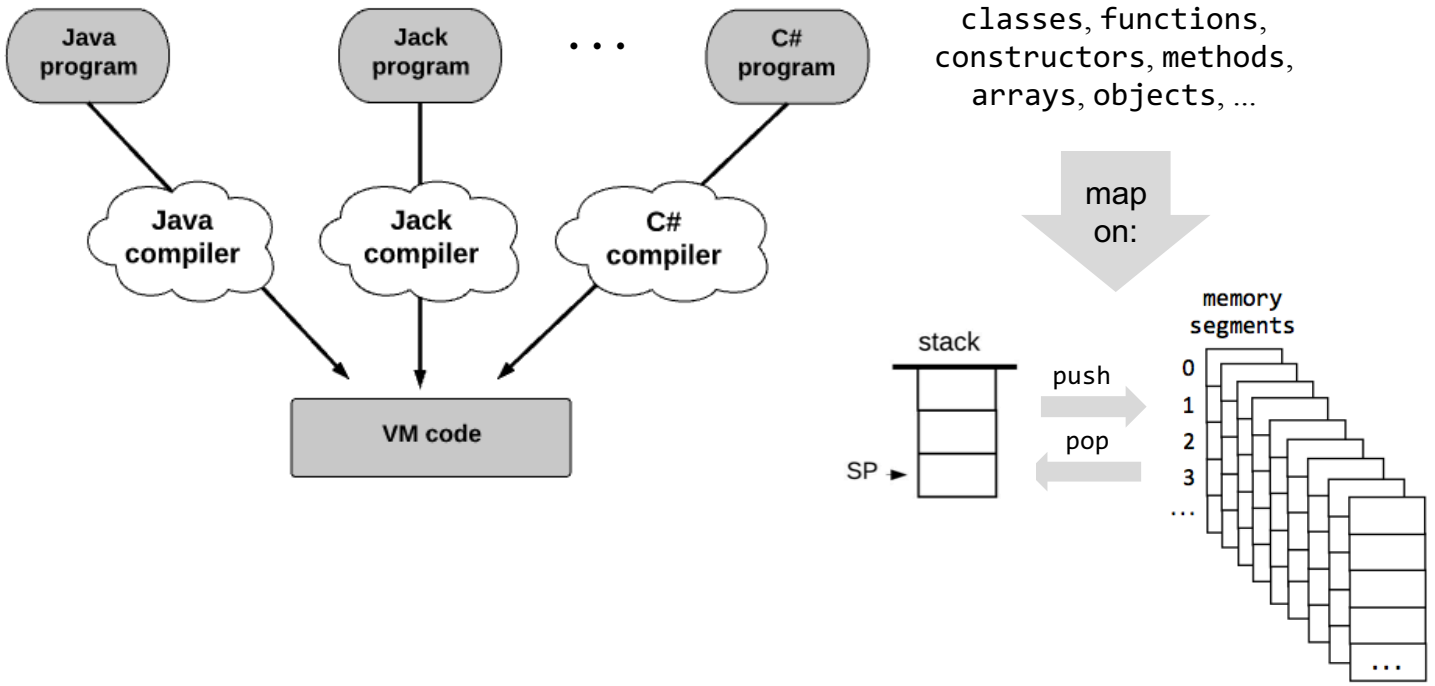
# Compilation challenges

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling flow of control
- ✓ Handling objects
- ✓ Handling arrays
- Standard mapping
- Proposed implementation
- Project 11
- Perspectives

# The big picture



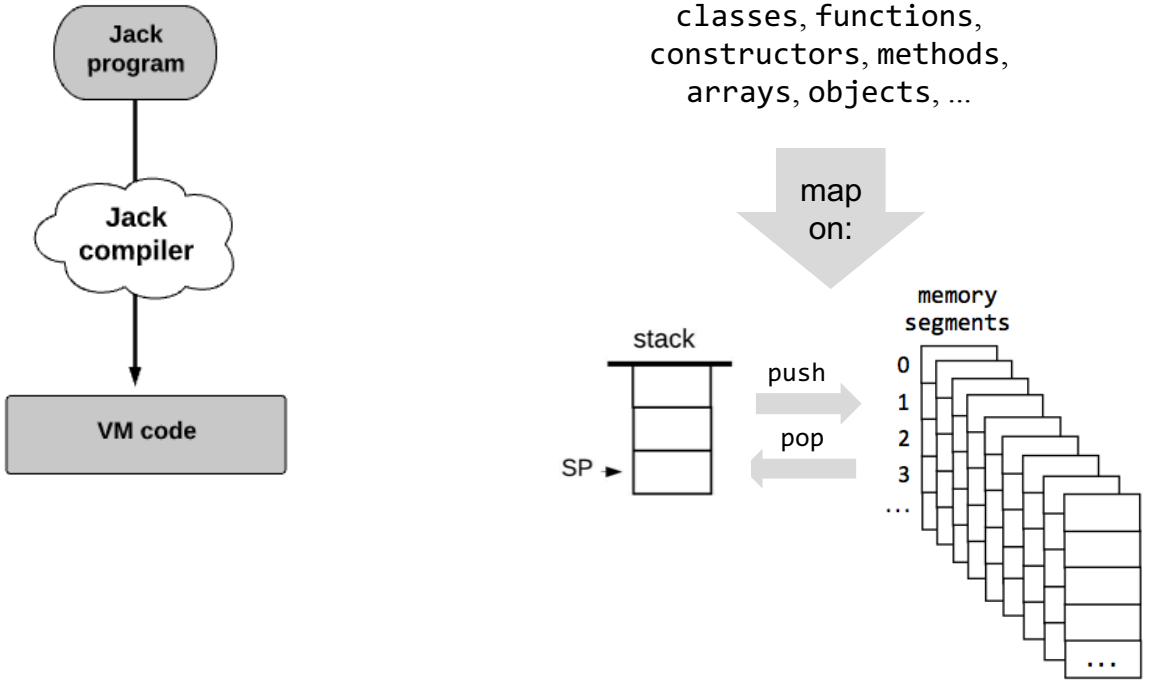
# The big picture




Standard Mapping on the VM platform:

Specifies how to map the constructs of a given high-level language on the constructs of the virtual machine

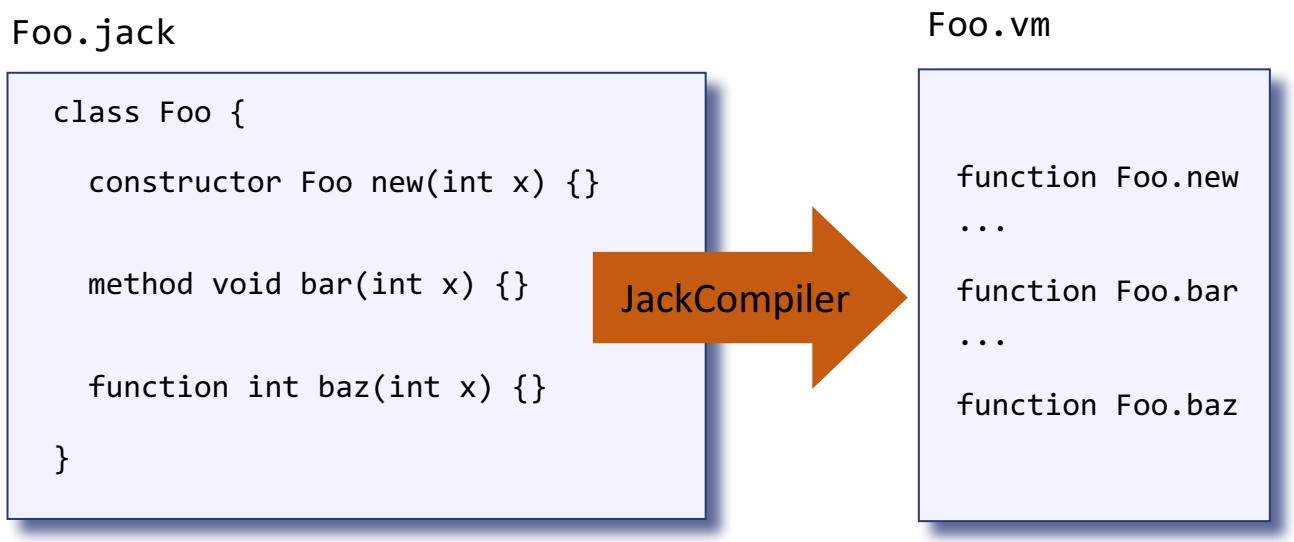
# The big picture




Standard Mapping of Jack on the VM platform:

Specifies how to map the constructs of the Jack language on the constructs of the virtual machine.

# Files and subroutines mapping




- Each file *filename*`.jack` is compiled separately into the file *fileName*`.vm`
- Each subroutine *subName* in *fileName*`.jack` is compiled into a VM function *fileName*.*subName*
- A Jack *constructor* / *function* with *k* arguments is compiled into a VM function that operates on *k* arguments
- A Jack *method* with *k* arguments is compiled into a VM function that operates on *k* + 1 arguments.

# Variables mapping

The *local* variables
of a Jack subroutine are mapped on the virtual segment `local`

The *argument* variables
of a Jack subroutine are mapped on the virtual segment `argument`

The *static* variables
of a `.jack` class file are mapped on the virtual memory segment `static`
of the compiled `.vm` class file

The *field* variables
of the current object are accessed as follows:

- `pointer 0` is set to `argument 0` (which represents the current object, or `this`)
- the *i*-th field of this object is mapped on `this i`

# Arrays mapping

Accessing array entries:

Access to any *array entry* `arr[i]` is realized as follows:

- first, set `pointer 1` to the entry's address (`arr + i`)
- access the entry by accessing `that 0`

# Compiling subroutines

When compiling a Jack method:

- the compiled VM code must set the base of the `this` segment to `argument 0`

When compiling a Jack constructor:

- the compiled VM code must allocate a memory block for the new object, and then set the base of the `this` segment to the new object's base address
- the compiled VM code must return the object's base address to the caller

When compiling a void function or a void method:

- The compiled VM code must return the value `constant 0`.

# Compiling subroutine calls

Compiling a subroutine call *subName*(*arg1*, *arg2*, ...):

The caller (a VM function) must push the arguments *arg1*, *arg2*, ...
onto the stack, and then call the subroutine

If the called subroutine is a *method*,

The caller (a VM function) must first push a reference to the object on which the method is supposed to operate;
next, the caller must push *arg1*, *arg2*, ... and then call the method

If the called subroutine is *void*:

A void Jack subroutine is not required to return a value;
However, VM functions are required to always return a value.
Therefore, when compiling the Jack statement `do` *subName*,
following the call the caller must pop (and ignore) the returned value.

# Compiling constants

- `null` is mapped on the constant `0`
- `false` is mapped on the constant `0`
- `true` is mapped on the constant `-1`

can be obtained using `push 1` followed by `neg`

# OS classes and subroutines

- The OS is written in Jack
- 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`
- Any VM function can call any OS function for its effect.

## Usage

- The eight OS `*.vm` files must reside in the same directory as the VM files generated by the compiler

Or:

- If the generated VM code is executed / tested on the supplied VM emulator, there is no need to include any OS files, since the supplied emulator features a built-in implementation of the OS (written in Java).

# Special OS services

Multiplication is handled using the OS function `Math.multiply()`

Division is handled using the OS function `Math.divide()`

String constants are created using the OS constructor `String.new(length)`

String assignments like `x="cc...c"` are handled using a series of calls to `String.appendChar(c)`

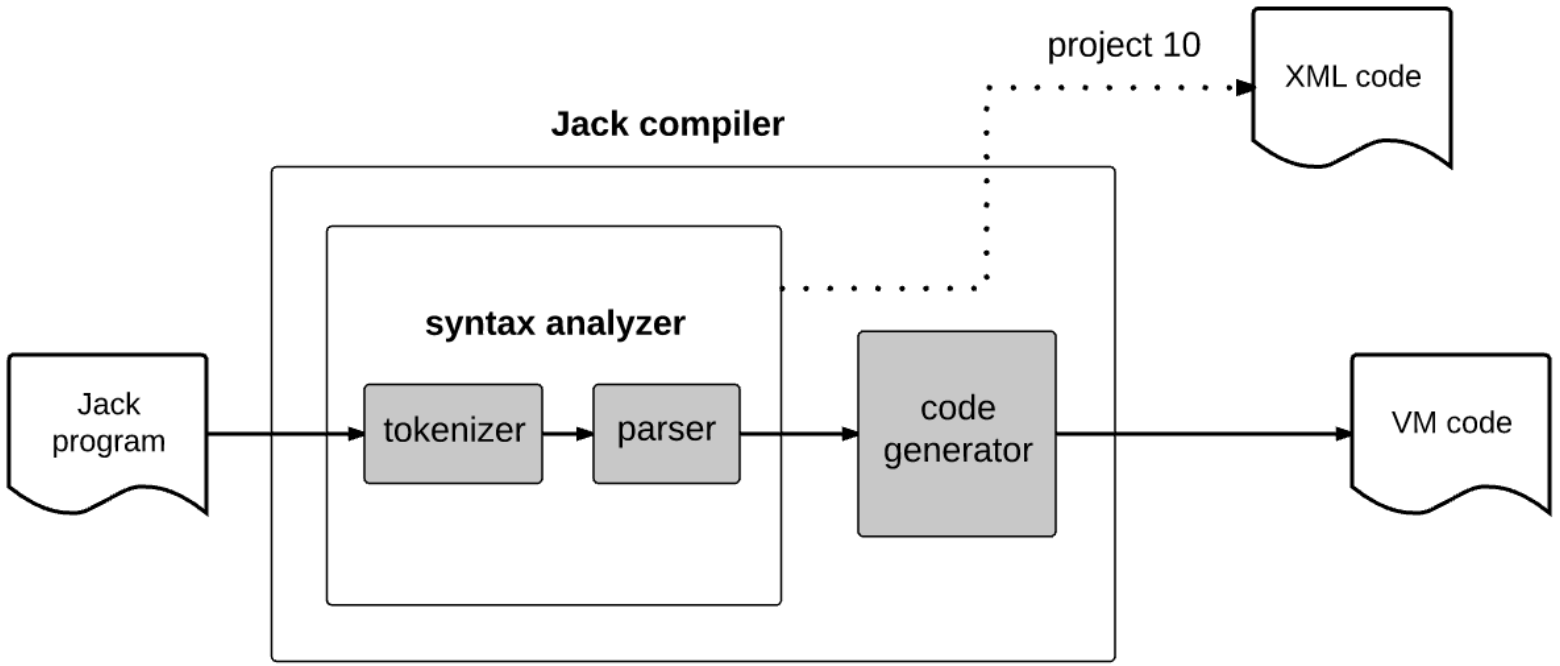
Object construction requires allocating space for the new object using the OS function `Memory.alloc(`*size*`)`

Object recycling is handled using the OS function `Memory.deAlloc(`*object*`).`

# Compilation challenges

- ✓ Handling variables
- ✓ Handling expressions
- ✓ Handling flow of control
- ✓ Handling objects
- ✓ Handling arrays
- ✓ Standard mapping
- **Proposed implementation**
- Project 11
- Perspectives

# Compiler development roadmap




## Modules:

- `JackCompiler`: top-most ("main") module
- `JackTokenizer`
- `SymbolTable`
- `VMWriter`: generates VM code
- `CompilationEngine`: recursive top-down compilation engine

main focus of this project

# JackCompiler

## Usage:

```
prompt> JackCompiler input
```

Input:
- *fileName*`.jack`: name of a single source file, or
- *directoryName*: name of a directory containing one or more `.jack` source files

Output:
- if the input is a single file: *fileName*`.vm`
- if the input is a directory: one `.vm` file for every `.jack` file, stored in the same directory

## Implementation notes:

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

# JackTokenizer

Developed in project 10.

# SymbolTable

```
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 |

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

Can be implemented using two hash tables

Implementation notes:

- Each variable is assigned a running index within its *scope* (table) and *kind*
- The index starts at 0, increments by 1 each time a new symbol is added to the table, and is reset to 0 when starting a new scope

  (the operations described above are performed by the `SymbolTable` routines)
- 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*.

# SymbolTable

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| Constructor | — | — | Creates a new symbol table. |
| `startSubroutine` | — | — | Starts a new subroutine scope (i.e., resets the subroutine's symbol table). |
| `define` | `name` (String)<br>`type` (String)<br>`kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`) | — | Defines a new identifier of the given *name*, *type*, and *kind*, and assigns it a running index.<br>`STATIC` and `FIELD` identifiers have a class scope, while `ARG` and `VAR` identifiers have a subroutine scope. |
| `VarCount` | `kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`) | int | Returns the number of variables of the given *kind* already defined in the current scope. |
| `KindOf` | `name` (String) | (`STATIC`, `FIELD`, `ARG`, `VAR`, `NONE`) | Returns the *kind* of the named identifier in the current scope. If the identifier is unknown in the current scope, returns NONE. |
| `TypeOf` | `name` (String) | String | Returns the *type* of the named identifier in the current scope. |
| `IndexOf` | `name` (String) | int | Returns the *index* assigned to the named identifier. |

# VMWriter

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| constructor | output file/stream | — | Creates a new output `.vm` file and prepares it for writing. |
| `writePush` | `segment` (`CONST`, `ARG`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`)<br>`index` (int) | — | Writes a VM `push` command. |
| `writePop` | `segment` (`ARG`, `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>`nLocals` (int) | — | Writes a VM `function` command. |
| `writeReturn` | — | — | Writes a VM `return` command. |
| `close` | — | — | Closes the output file. |

# 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 language
- Contract:
  - 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*
  - Thus `compile`*xxx* should be 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 as in project 10)

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| Constructor | Input stream/file<br>Output stream/file | | 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. |
| `CompileSubroutineDec` | — | — | Compiles a complete method, function, or constructor. |
| `compileParameterList` | — | — | Compiles a (possibly empty) parameter list. Does not handle the enclosing "`()`". |
| `compileSubroutineBody` | — | — | Compiles a subroutine's body. |
| `compileVarDec` | — | — | Compiles a `var` declaration. |
| `compileStatements` | — | — | Compiles a sequence of statements. Does not handle the enclosing "`{}`". |

# CompilationEngine (same 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 distinguish between a *variable*, an *array entry*, or a *subroutine call*. A single look-ahead token, which may be one of "`[`", "`(`", or "`.`", suffices to distinguish between the possibilities. Any other token is not part of this term and should not be advanced over. |
| `CompileExpressionList` | -- | -- | Compiles a (possibly empty) comma-separated list of expressions. |

# Compilation challenges


- Handling variables


- Handling expressions


- Handling flow of control


- Handling objects


- Handling arrays

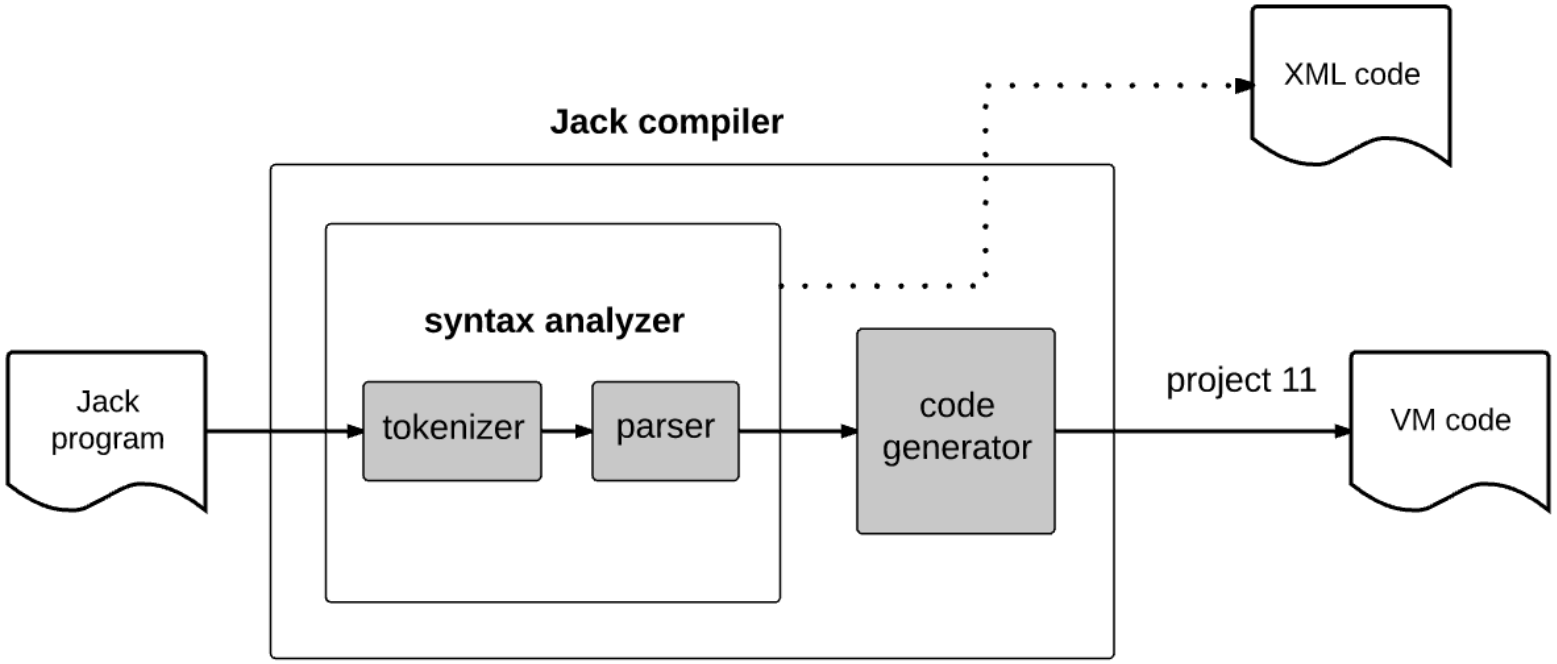

- Standard mapping


- Proposed implementation

- **Project 11**

- Perspectives

# 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>` tags

# Symbol table

Extend the handling of identifiers

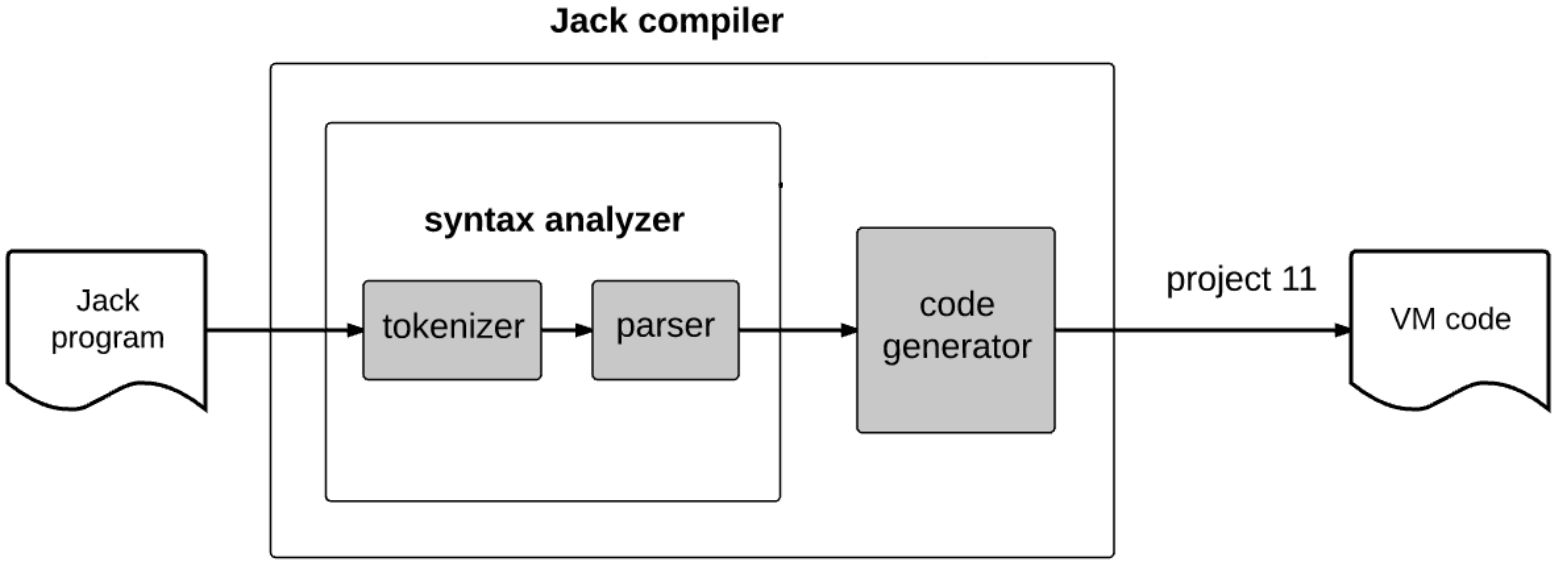
- output the identifier's category: `var`, `argument`, `static`, `field`, `class`, `subroutine`
- if the identifier's category is `var`, `argument`, `static`, `field`, output also the running index assigned to this variable in the symbol table
- output whether the identifier is being defined, or being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

Implementation

1. implement the `SymbolTable` API
2. extend the syntax analyzer (developed in project 01) with the identifier handling described above

   (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
- ✓ Stage 1: symbol table handling
- Stage 2: code generation

# Code generation

Test programs

- `Seven`
- `ConvertToBin`
- `Square`
- `Average`
- `Pong`
- `ComplexArrays`

Development plan: unit testing

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

For each test program:

1. Use your compiler to compile the program directory
2. Inspect the generated code;
   If there's a problem, fix your compiler and go to stage 1
3. Load the directory 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;
   }
}
```


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: Seven

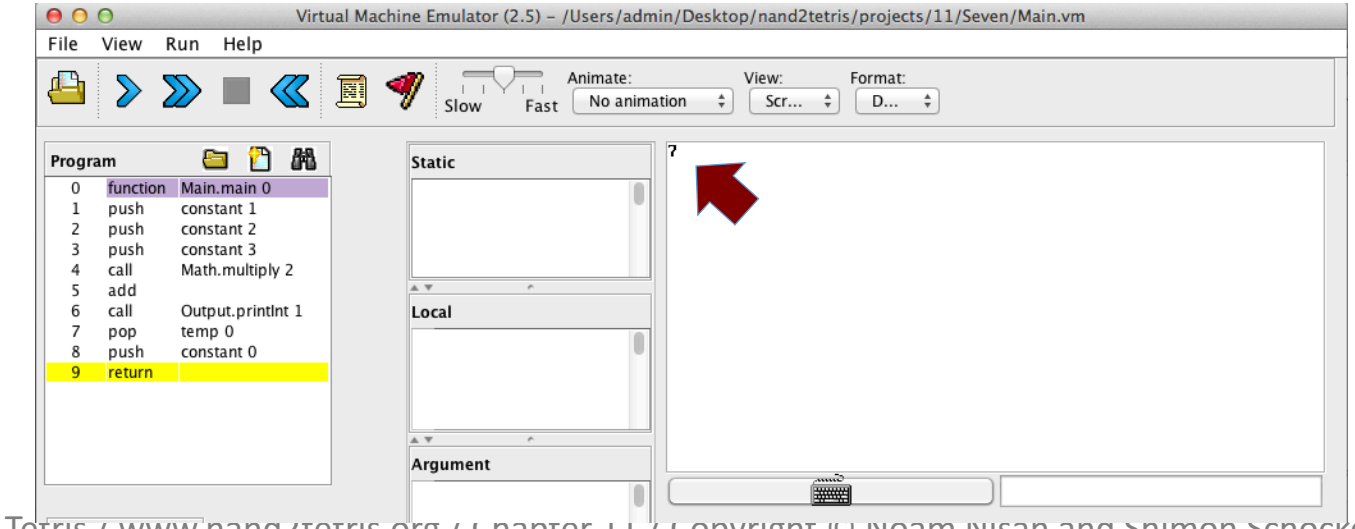
projects/11/Seven/Main.jack

```
/**
 * 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
```



# Quiz

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
```

Inspect the VM code, focusing on the two instructions highlighted in red.
Select the correct observation:

The two highlighted instructions ...

a) are designed to clean up the memory segments before the function returns

b) are related to the function call-and-return contract

c) are example of unnecessary code that is sometimes generated by the compiler

# Quiz answer

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;
   }
}
```


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
```

Inspect the VM code, focusing on the two instructions highlighted in blue.
Select the correct observation:

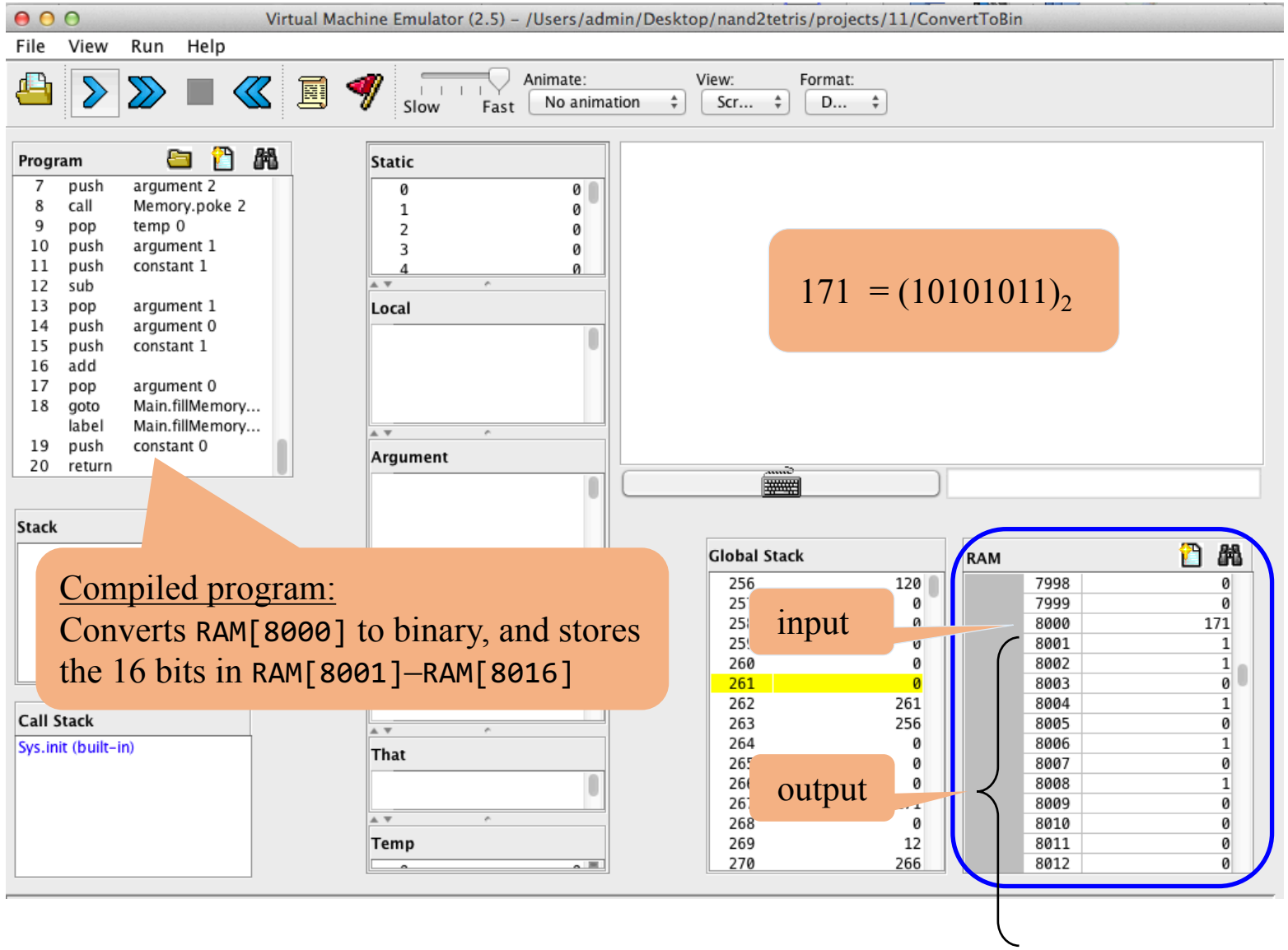
The two highlighted instructions ...

a) are designed to clean up the memory segments before the function returns

b) are related to the function call-and-return contract

c) are example of unnecessary code that is sometimes generated by the compiler

# Test program: decimal-to-binary conversion



# Test program: decimal-to-binary conversion

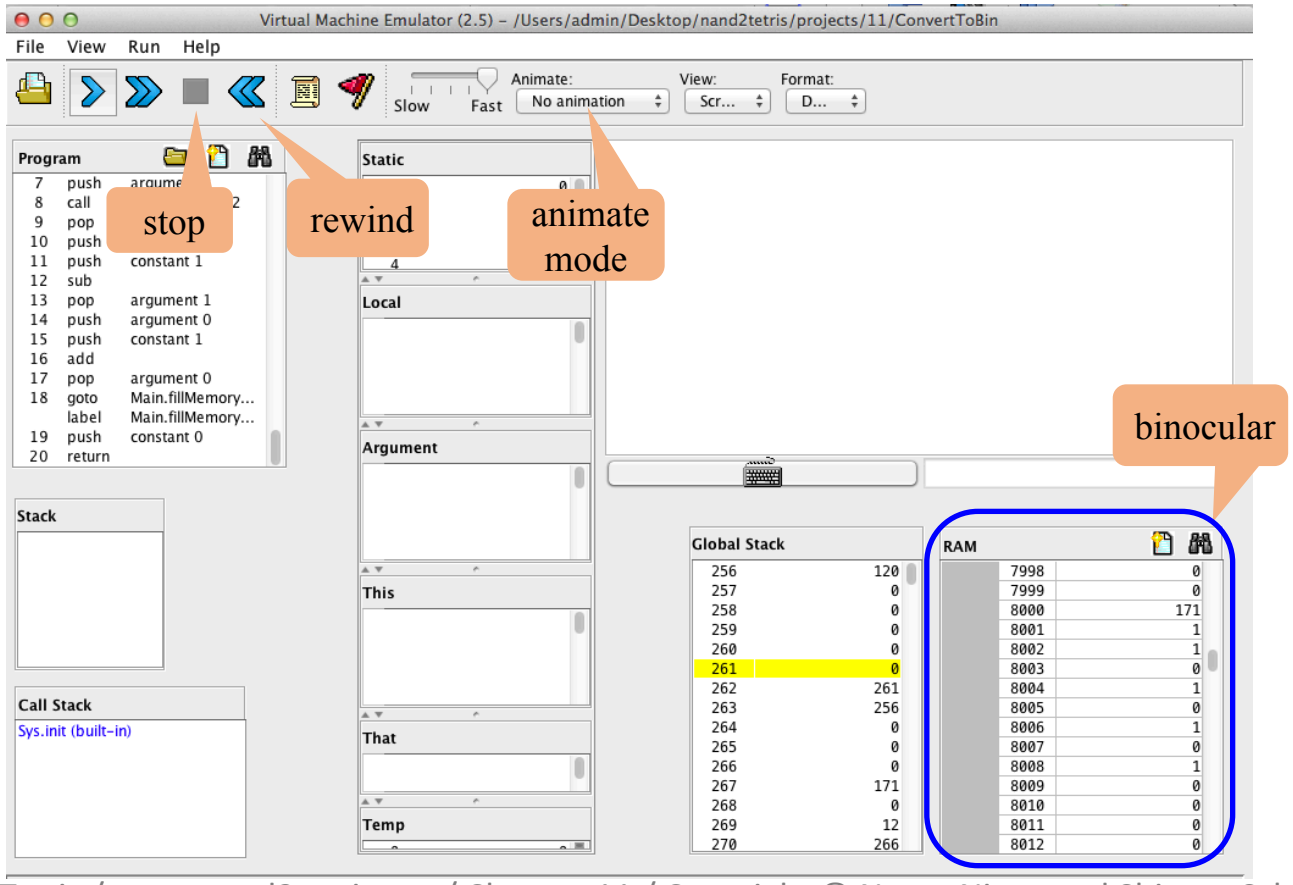
```
class Main {
   // Converts RAM[8000] to binary, putting the resulting bits in RAM[8001]..RAM[8016]
   function void main() {
      var int value;
      do Main.fillMemory(8001, 16, -1);  // sets RAM[8001]..RAM[8016] to -1
      let value = Memory.peek(8000);     // gets the input from RAM[8000]
       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) {}

   // Converts the value to binary, and puts the result in RAM[8001]..RAM[8016] */
   function void convert(int value) {}

   // Some more private functions (omitted)
}
```

Tests how your compiler handles:

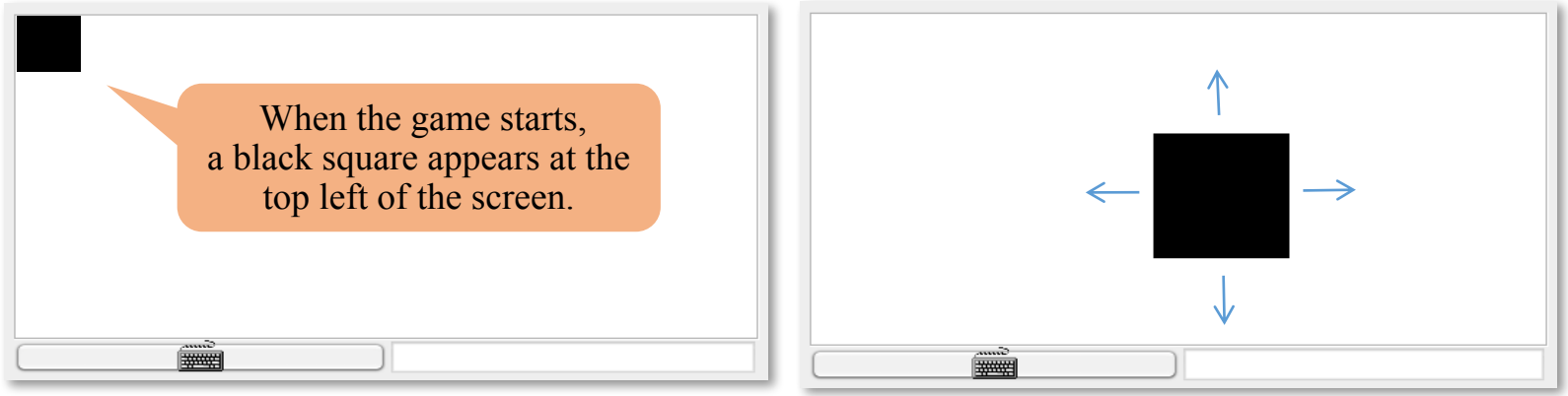
- expressions (without arrays or method calls)
- procedural constructs: `if`, `while`, `do`, `let`, `return`

# 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



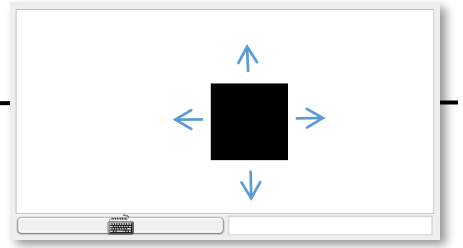
# Test app: Square




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

- constructors
- methods
- expressions that include method calls.

# Test app: Square



projects/11/Square/Square.jack

```
/** Implements a graphical square */
class Square {

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

   /** Dispose
   method void

   /** Draws t
   method void

   /** Erases
   method void

   /** Increme
   method void

   /** Decreme
   method void

   /** Moves u
   method void

   /** Moves d
   method void

   /** Moves l
   method void

   /** Moves r
   method void
}
```

SquareGame.jack

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

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

   /** Constructs a new S
   constructor SquareGame
      let square = Square
      let direction = 0;
      return this;
   }

   /** Disposes this game
   method void dispose()
      do square.dispose()
      do Memory.deAlloc(t
      return;
   }

   ...
```

Main.jack

```
/**
 * Main class of the square game.
 * initializes a new game and starts it
 */
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

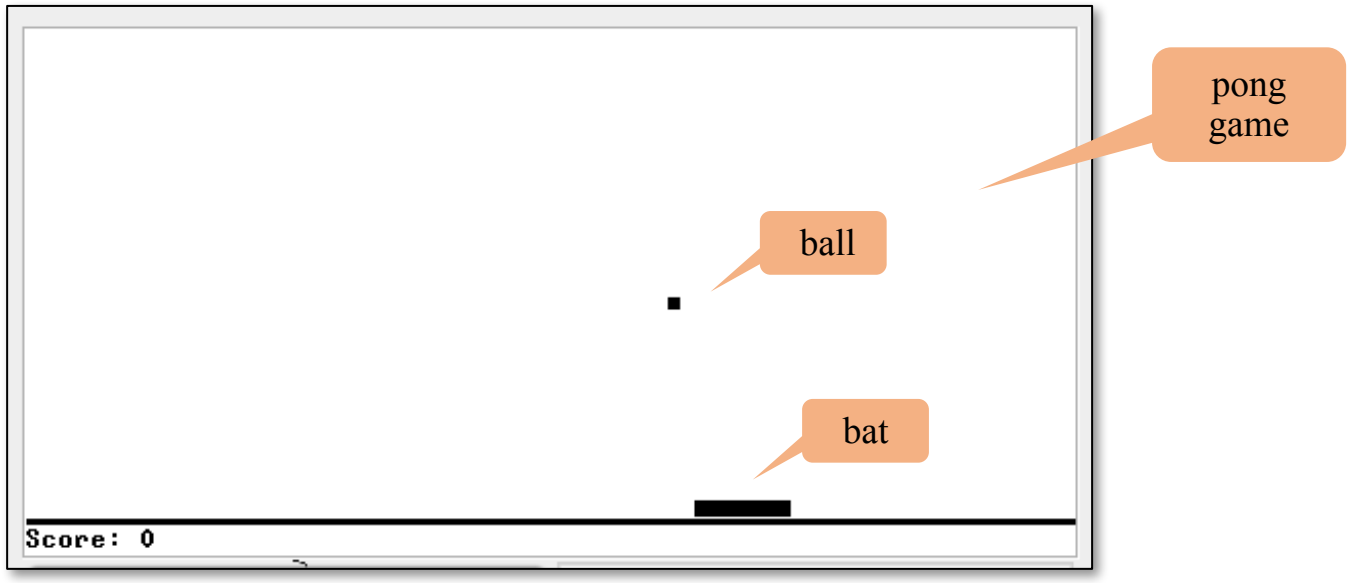
```
/** Computes the average of a sequence of integers */
class Main {
   function void main() {
      var Array a;
      var int length;
      var int i, sum;

      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;
   }
}
```

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

Tests how your compiler handles:

- Arrays
- Strings

# Test app: Pong




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

# Test app: Pong

projects/11/Pong/Ball.jack

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

    // Ball's
    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 (Bat) 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 th
    method void
       var char
       while (~
          // wa
          while
             le
             do
             do
          }

          if (k

          }
          ...
       }
    }
    ...
}
```

Main.jack

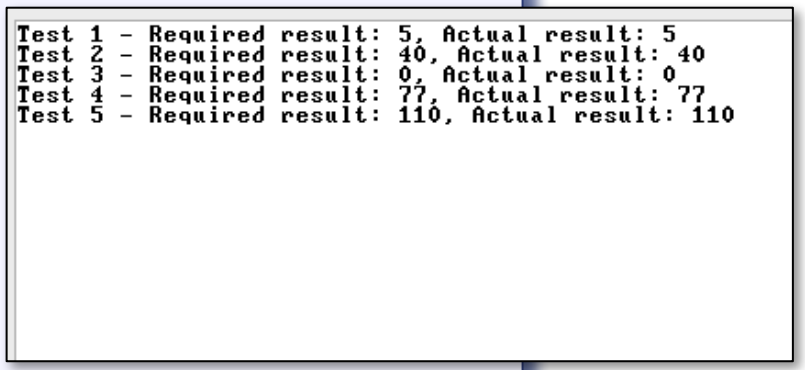
```
/** Main class of the Pong game */
class Main {
   /** Initializes a Pong game and
   starts running it. */
   function void main() {
      var PongGame game;
      do PongGame.newInstance();
      let game = PongGame.getInstance();
      do game.run();
      do game.dispose();
      return;
   }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

# Test program: ComplexArrays

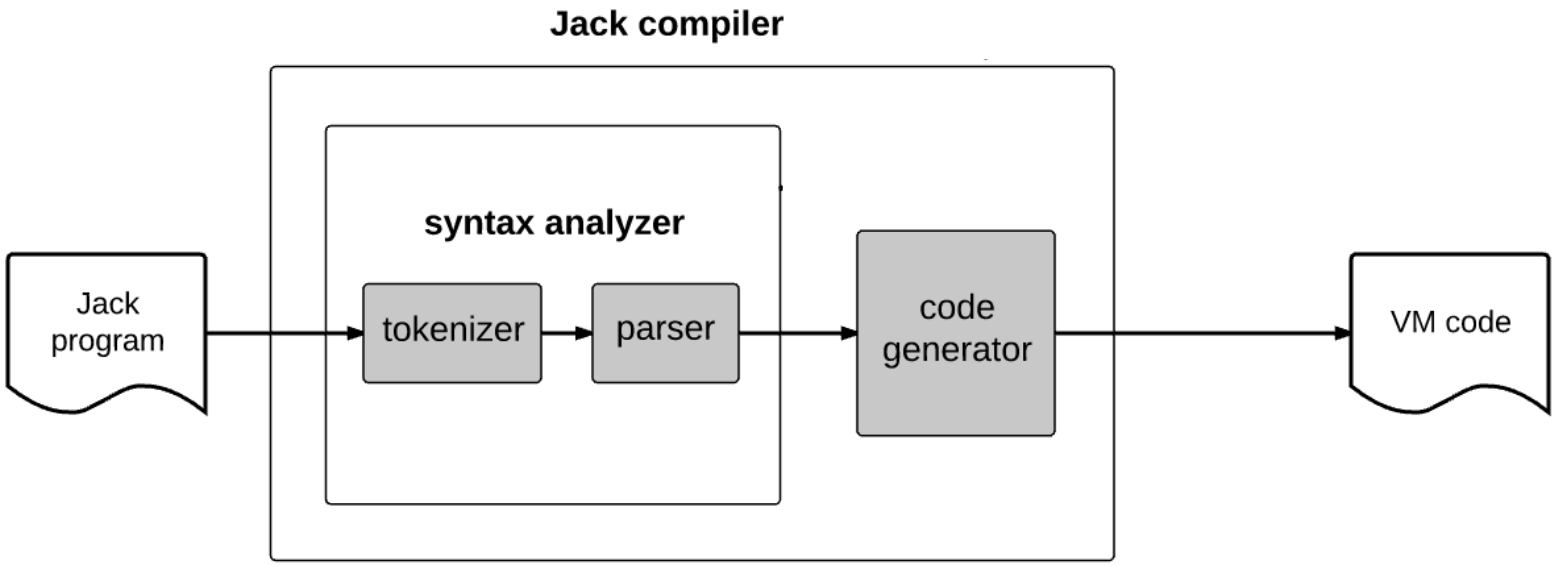
projects/11/ComlpexArrays/Main.jack

```
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



# Recap: project 11




Project 11:

Extend / morph the syntax analyzer into a full-scale compiler

Test programs

- `Seven`
- `ConvertToBin`
- `Square`
- `Average`
- `Pong`
- `ComplexArrays`