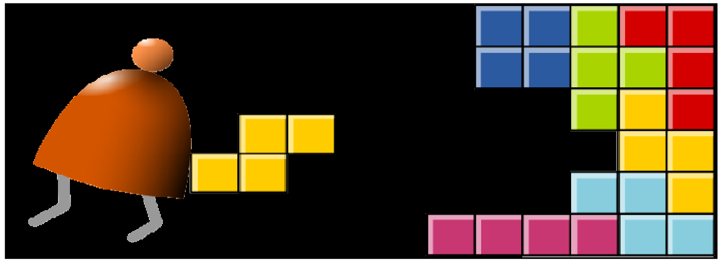
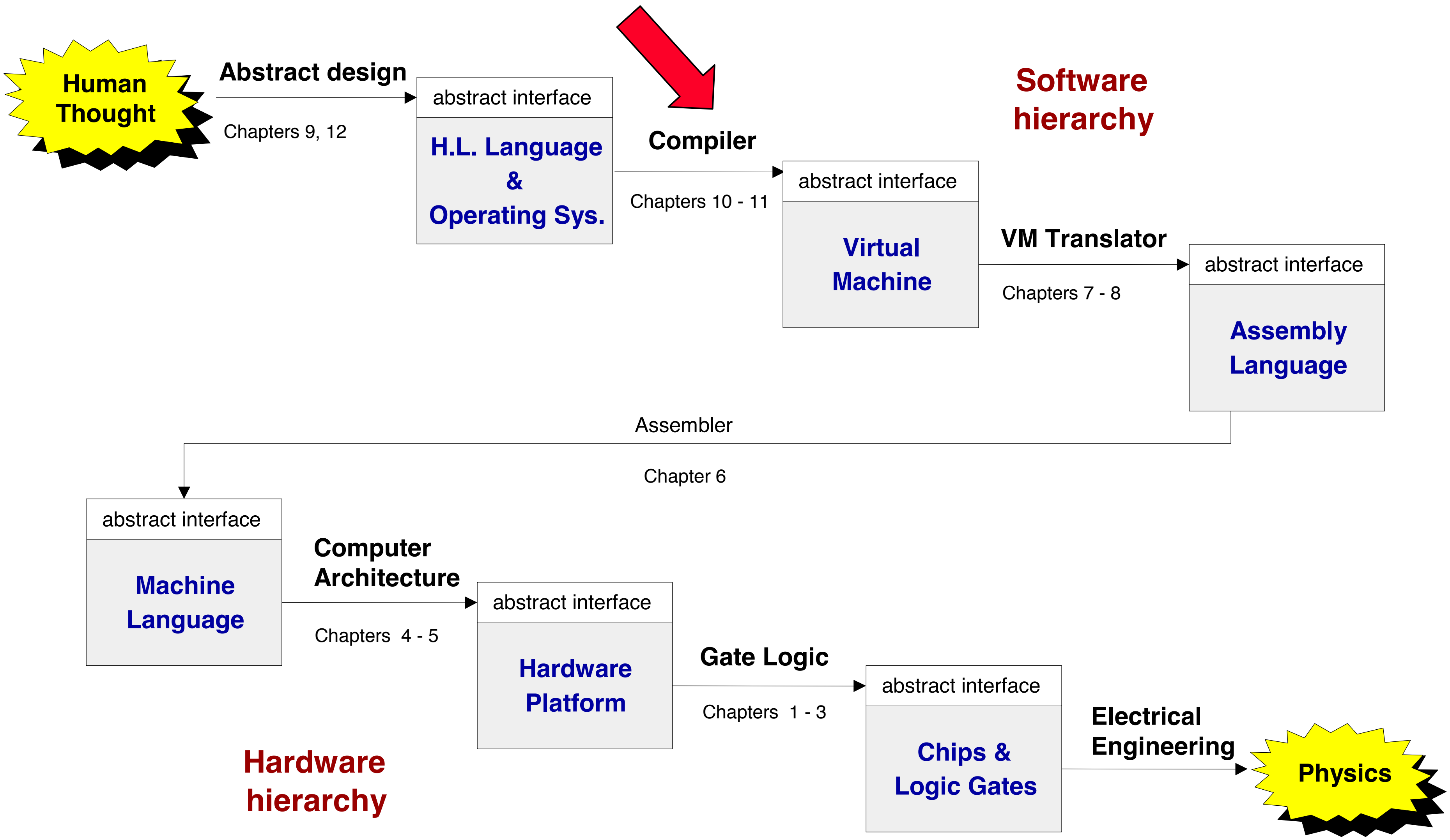
Compiler II: Code Generation



Building a Modern Computer From First Principles

www.nand2tetris.org

Course map

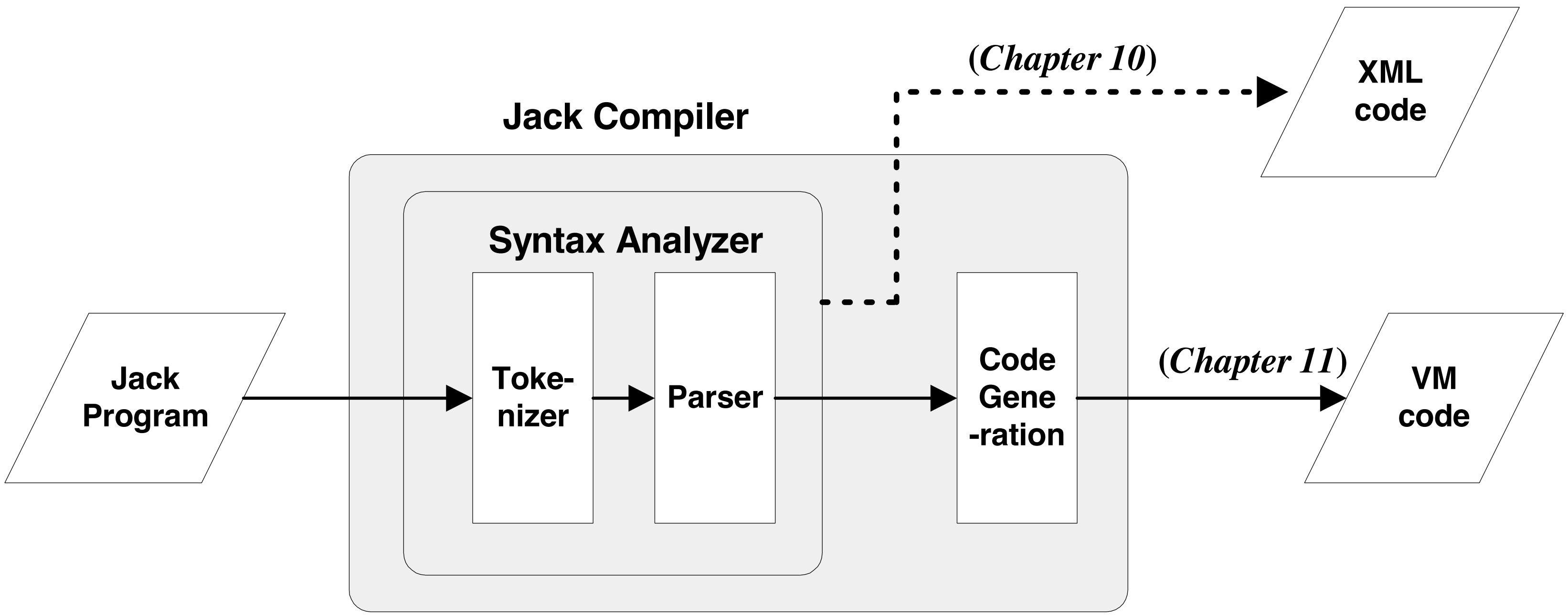


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

The big picture

1. Syntax analysis: extracting the semantics from the source code — previous lecture

2. Code generation: expressing the semantics using the target language — this lecture



Syntax analysis (review)

```
Class Bar {
    method Fraction foo(int y) {
       var int temp; // a variable
       let temp = (xxx+12)*-63;
       ...
    ...
```

Syntax analyzer

```
<varDec>
  <keyword> var </keyword>
  <keyword> int </keyword>
  <identifier> temp </identifier>
  <symbol> ; </symbol>
</varDec>
<statements>
  <letStatement>
    <keyword> let </keyword>
    <identifier> temp </identifier>
    <symbol> = </symbol>
    <expression>
       <term>
         <symbol> ( </symbol>
         <expression>
           <term>
             <identifier> xxx </identifier>
           </term>
           <symbol> + </symbol>
           <term>
             <int.Const.> 12 </int.Const.>
            </term>
    </expression>
    ...
```

The code generation challenge:

- Program = a series of operations that manipulate data
- Compiler: converts each "understood" (parsed) source operation and data item into corresponding operations and data items in the target language
- Thus, we have to generate code for
  - handling data
  - handling operations
- Our approach: morph the syntax analyzer (project 10) into a full-blown compiler: instead of generating XML, we'll make it generate VM code.

Memory segments (review)

VM memory Commands:

```
pop segment i
push segment i
```

Where *i* is a non-negative integer and *segment* is one of the following:

`static`: holds values of global variables, shared by all functions in the same class

`argument`: holds values of the argument variables of the current function

`local`: holds values of the local variables of the current function

`this`: holds values of the private ("object") variables of the current object

`that`: holds array values (silly name, sorry)

`constant`: holds all the constants in the range `0` … `32767` (pseudo memory segment)

`pointer`: used to anchor `this` and `that` to various areas in the heap

`temp`: fixed 8-entry segment that holds temporary variables for general use;
Shared by all VM functions in the program.

Code generation example

```
method int foo() {
  var int x;
  let x = x + 1;
  ...
```

Syntax analysis

```
<letStatement>
  <keyword> let </keyword>
  <identifier> x </identifier>
  <symbol> = </symbol>
  <expression>
    <term>
      <identifier> x </identifier>
      </term>
        <symbol> + </symbol>
      <term>
        <constant> 1 </constant>
       </term>
  </expression>
</letStatement>
```

Code generation

```
push local 0
push constant 1
add
pop local 0
```

(note that x is the first local variable declared in the method)

Handling variables

When the compiler encounters a variable, say x, in the source code, it has to know:

What is x's *data type*?

Primitive, or ADT (class name) ?

(Need to know in order to properly allocate RAM resources for its representation)

What *kind* of variable is x?

`local, static, field, argument` ?

(We need to know in order to properly allocate it to the right memory segment; this also implies the variable's life cycle).

Handling variables: mapping them on memory segments (example)

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;    // Some local variables
      var Date due;    // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

- The target language uses 8 memory segments
- Each memory segment, e.g. `static`, is an indexed sequence of 16-bit values that can be referred to as `static 0`, `static 1`, `static 2`, etc.

When compiling this class, we have to create the following mappings:

The class variables	`nAccounts` , `bankCommission`	are mapped on	`static 0,1`
The object fields	`id`, `owner`, `balance`	are mapped on	`this 0,1,2`
The argument variables	`sum`, `bankAccount`, `when`	are mapped on	`arg 0,1,2`
The local variables	`i`, `j`, `due`	are mapped on	`local 0,1,2`.

Handling variables: symbol tables

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;    // Some local variables
      var Date due;    // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

How the compiler uses symbol tables:

- The compiler builds and maintains a linked list of hash tables, each reflecting a single scope nested within the next one in the list
- Identifier lookup works from the current symbol table back to the list's head

  (a classical implementation).

Handling variables: managing their life cycle

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

<u>Variables life cycle</u>

`static` variables: single copy must be kept alive throughout the program duration

`field` variables: different copies must be kept for each object

`local` variables: created on subroutine entry, killed on exit

`argument` variables: similar to `local` variables.

<u>Good news:</u> the VM implementation already handles all these details !

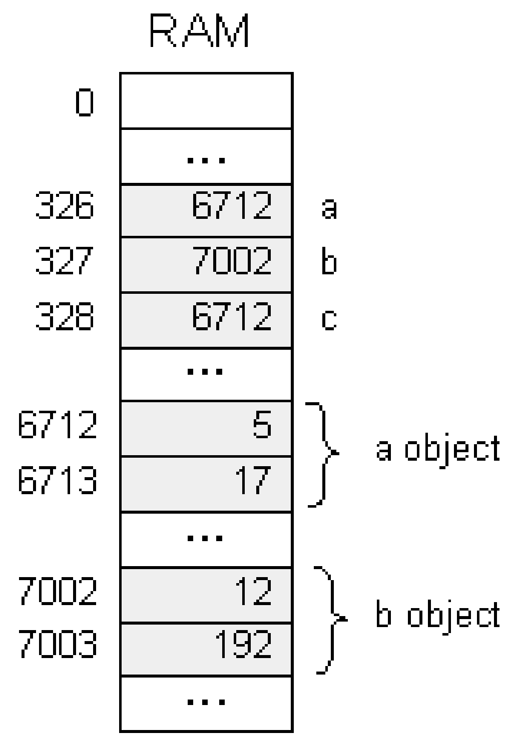
Handling objects: construction / memory allocation

Java code

```
class Complex {
    // Fields (properties):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex (int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    public void bla() {
        Complex a, b, c;
        ...
        a = new Complex(5,17);
        b = new Complex(12,192);
        ...
        c = a; // Only the reference is copied
        ...
  }
```

Following compilation:



How to compile:

`foo = new ClassName(…)` ?

The compiler generates code affecting:

`foo = Memory.alloc(n)`

Where `n` is the number of words necessary to represent the object in question, and `Memory.alloc` is an OS method that returns the base address of a free memory block of size `n` words.

Handling objects: accessing fields

Java code

```
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
    /** Multiplies this Complex number
        by the given scalar */
    public void mult (int c) {
        re = re * c;
        im = im * c;
    }
    ...
}
```

How to compile:

`im = im * c` **?**

1. look up the two variables in the symbol table

2. Generate the code:

```
*(this + 1) = *(this + 1)
              times
              (argument 0)
```

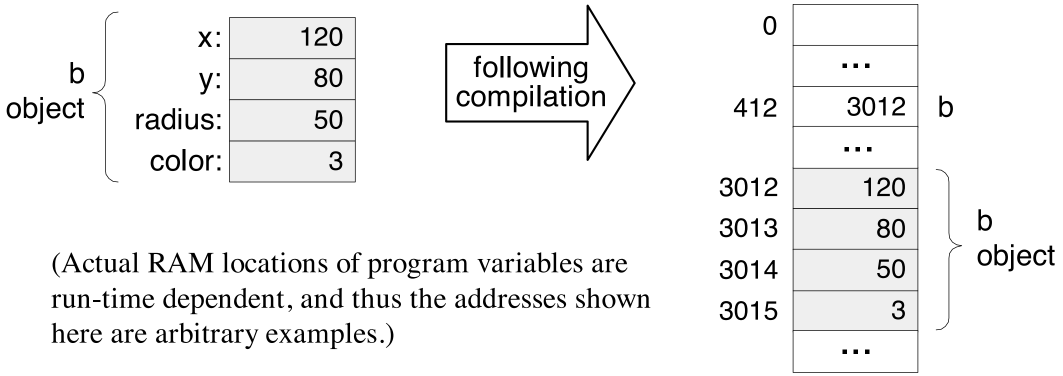
This pseudo-code should be expressed in the target language.

Handling objects: establishing access to the object's fields

<u>Background:</u> Suppose we have an object named `b` of type `Ball`. A `Ball` has `x`,`y` coordinates, a `radius`, and a `color`.

High level program view

RAM view



(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

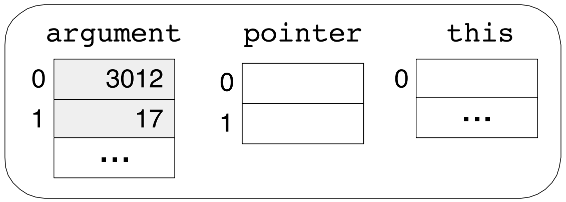
Assume that `b` and `r` were passed to the function as its first two arguments.

<u>How to compile (in Java):</u>

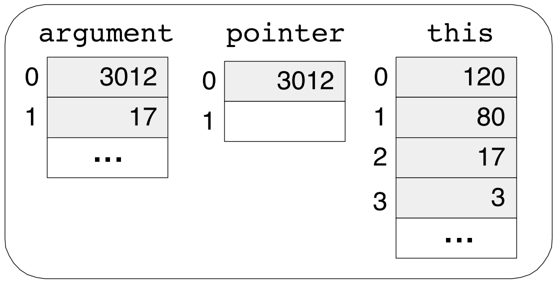
```
b.radius = r  ?

// Get b's base address:
push argument 0
// Point the this segment to b:
pop pointer 0
// Get r's value
push argument 1
// Set b's third field to r:
pop this 2
```

Virtual memory segments just before the operation `b.radius=17`:



Virtual memory segments just after the operation `b.radius=17`:



(`this 0` is now alligned with `RAM[3012]`)

Handling objects: method calls

Java code

```
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    ...
    public void bla() {
        Complex x;
        ...
        x = new Complex(1,2);
        x.mult(5);
        ...
   }
}
```

How to compile:

`x.mult(5)` ?

This method call can also be viewed as:

`mult(x,5)`

Generate the following code:

```
push x
push 5
call mult
```

General rule: each method call

`foo.bar(v1,v2,...)`

is translated into:

```
push foo
push v1
push v2
...
call bar
```

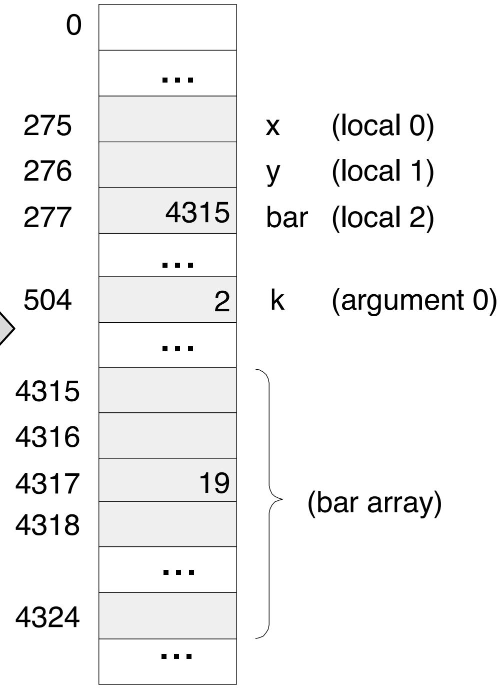
Handling arrays: declaration / construction

Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

Following compilation:

RAM state



<u>How to compile:</u>

`bar = new int(n)` ?

Generate code affecting:

`bar = Memory.alloc(n)`

Handling arrays: accessing an array entry by its index

Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

Following compilation:

RAM state, just after executing `bar[k] = 19`

0			
	...		
275		x	(local 0)
276		y	(local 1)
277	4315	bar	(local 2)
	...		
504	2	k	(argument 0)
	...		
4315			(bar array)
4316			
4317	19		
4318			
	...		
4324			
	...		

<u>How to compile:</u> `bar[k] = 19` ?

VM Code (pseudo)

```
// bar[k]=19, or *(bar+k)=19
push bar
push k
add
// Use a pointer to access x[k]
pop addr // addr points to bar[k]
push 19
pop *addr // Set bar[k] to 19
```

VM Code (actual)

```
// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add
// Use the that segment to access x[k]
pop pointer 1
push constant 19
pop that 0
```

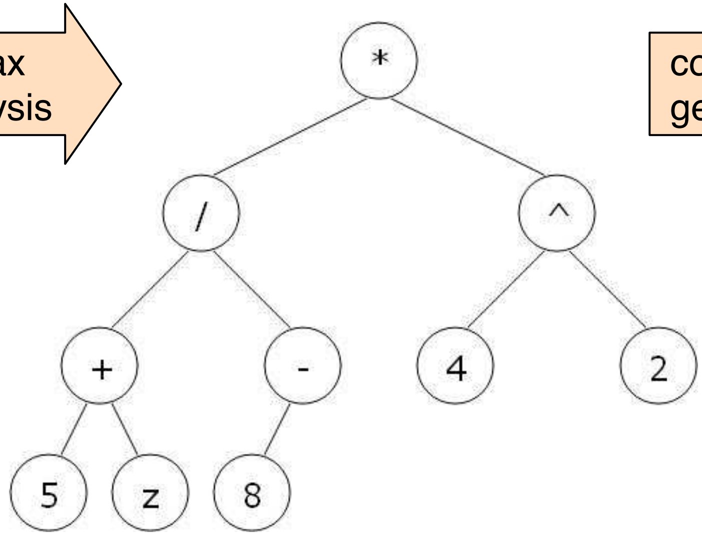
Handling expressions

High-level code

```
((5+z)/-8)*(4^2)
```

syntax analysis

parse tree



code generation

VM code

```
push 5
push z
add
push 8
neg
call div
push 4
push 2
call power
call mult
```

To generate VM code from a parse tree *exp*, use the following logic:

<u>The `codeWrite(`*exp*`)` algorithm:</u>

if *exp* is a constant *n* then output `"push n"`

if *exp* is a variable *v* then output `"push v"`

if *exp* is *op*(exp_1) then `codeWrite(`exp_1`);` output `"op";`

if *exp* is (exp_1 *op* exp_2) then `codeWrite(`exp_1`);` `codeWrite(`exp_2`);` output `"op";`

if *exp* is f (exp_1, ..., exp_n) then `codeWrite(exp1); ... codeWrite(exp1);` output `"call f";`

Handling program flow

High-level code

```
if (cond)
   s1
else
   s2
...
```

code generation

VM code

```
   VM code to compute and push !(cond)
   if-goto L1
   VM code for executing s1
   goto L2
label L1
   VM code for executing s2
label L2
   ...
```

High-level code

```
while (cond)
   s
...
```

code generation

VM code

```
label L1
   VM code to compute and push !(cond)
   if-goto L2
   VM code for executing s
   goto L1
label L2
   ...
```

Final example

High level code (`BankAccount.jack` class file)

```
/* Some common sense was sacrificed in this banking example in order
   to create a non trivial and easy-to-follow compilation example. */
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;  // As a percentage, e.g., 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) { /* Code omitted */ }

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;   // Some local variables
      var Date due;   // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
      return;
   }
   // More methods ...
}
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

Pseudo VM code

```
function BankAccount.commission
  // Code omitted
function BankAccount.trasnfer
  // Code for setting "this" to point
  // to the passed object (omitted)
  push balance
  push sum
  add
  push this
  push sum
  push 5
  call multiply
  call commission
  sub
  pop balance
  // More code ...
  push 0
  return
```

Final VM code

```
function BankAccount.commission 0
  // Code omitted
function BankAccount.trasnfer 3
  push argument 0
  pop pointer 0
  push this 2
  push argument 1
  add
  push argument 0
  push argument 1
  push constant 5
  call Math.multiply 2
  call BankAccount.commission 2
  sub
  pop this 2
  // More code ...
  push 0
  return
```

Perspective

Jack simplifications that are challenging to extend:

- Limited primitive type system
- No inheritance
- No public class fields, e.g. must use `r = c.getRadius()`
  rather than `r = c.radius`

Jack simplifications that are easy to extend: :

- Limited control structures, e.g. no `for`, `switch`, …
- Cumbersome handling of `char` types, e.g. cannot use `let x='c'`

Optimization

- For example, `c=c+1` is translated inefficiently into `push c`, `push 1`, add, `pop c`.
- Parallel processing
- Many other examples of possible improvements …