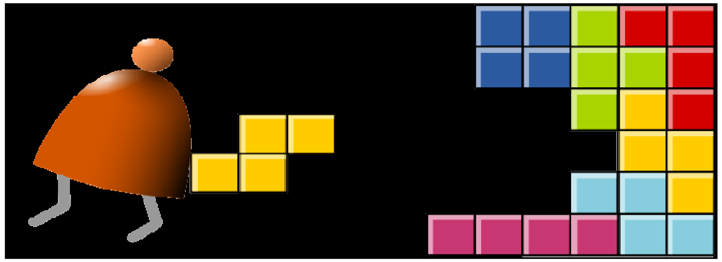
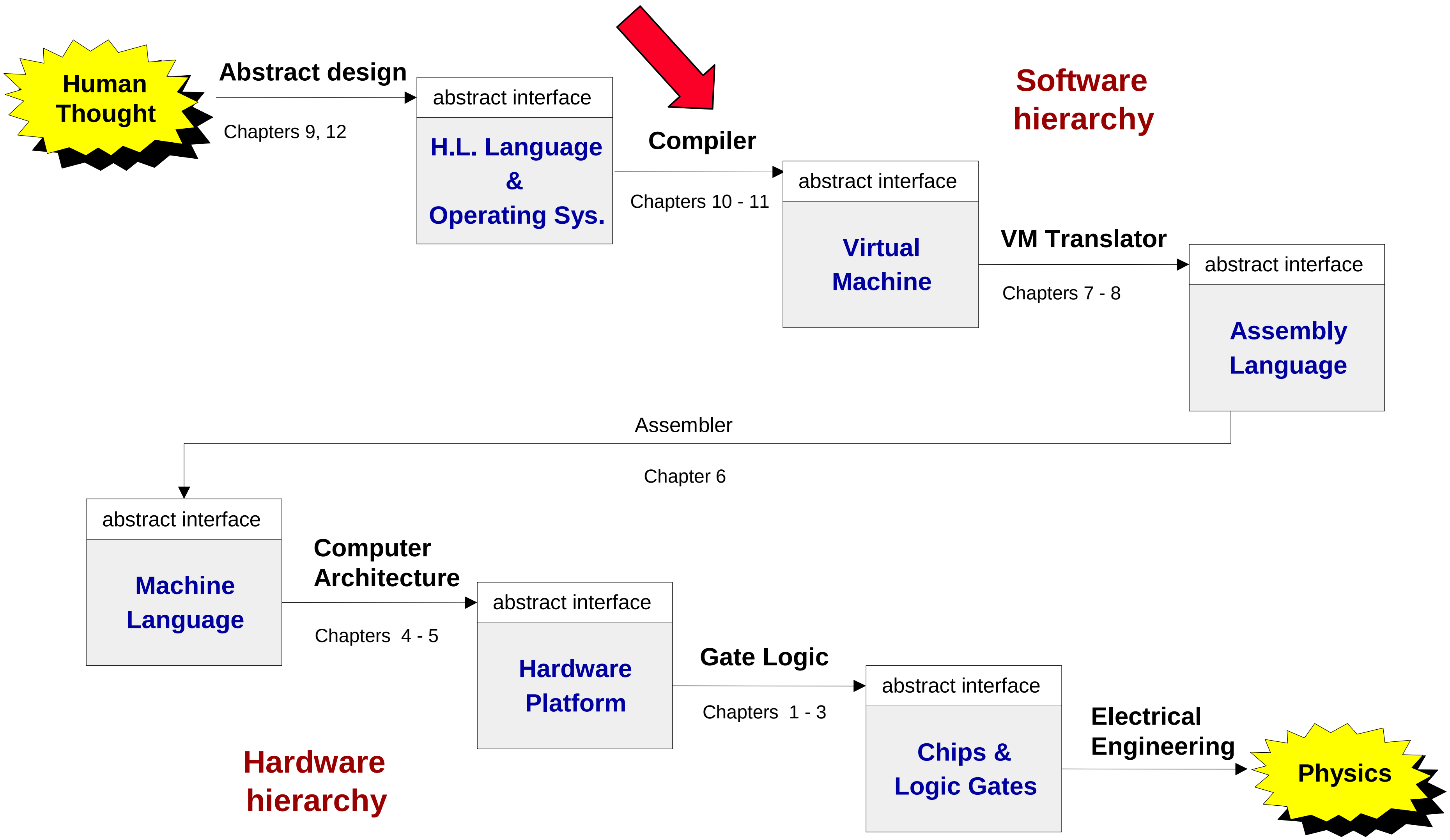
Compiler II: Code Generation



Building a Modern Computer From First Principles

www.nand2tetris.org

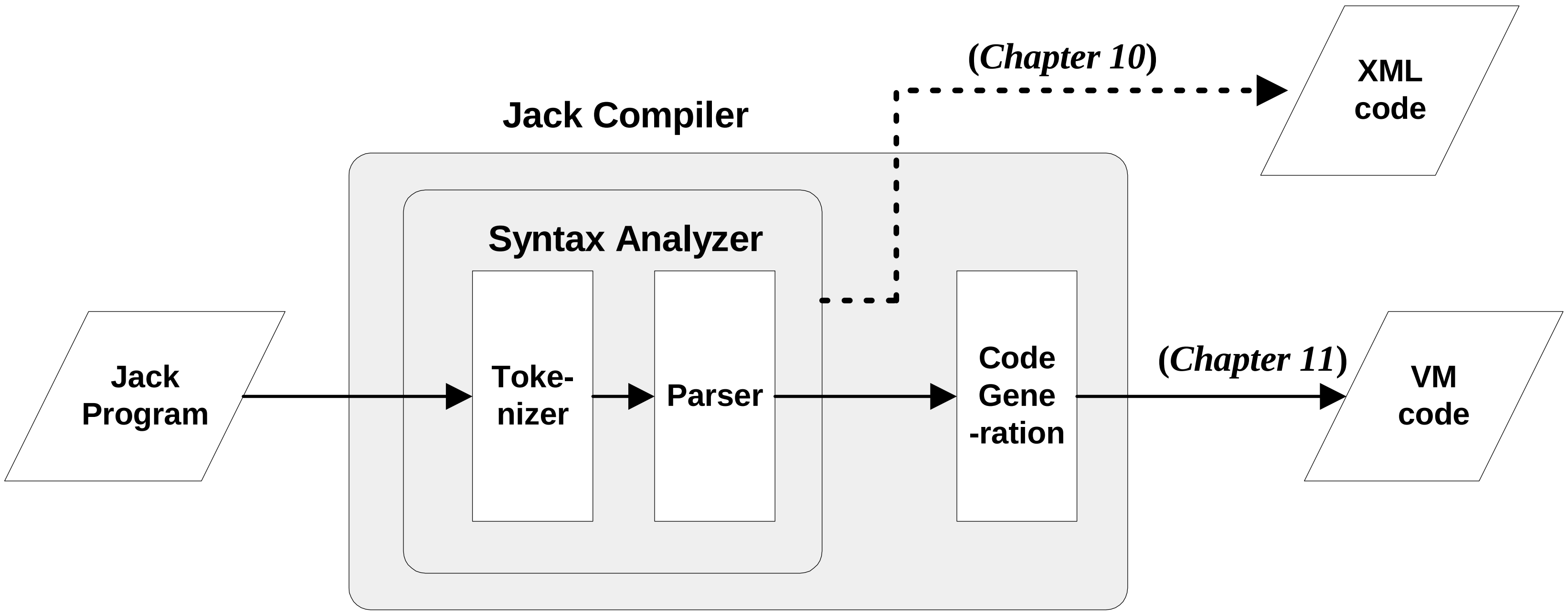
Course map



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

- Syntax analysis: extracting the semantics from the source code
- Code generation: expressing the semantics using the target language



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>
    ...
```

<u>The code generation challenge:</u>

- 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: extend 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 map **this** and **that** on different 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

```
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
```

(assuming that in the source code, x is the first local variable)

Handling data

When dealing with a variable, say x, we have 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`** ?

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

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 ...
  }
```

- Recall that we use 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 we compile this class, we have to create the following mapping:

- **nAccounts** , **bankCommission** are mapped on **static 0,1**
- **id**, **owner**, **balance** of the current object are mapped on **this 0,1,2**
- **sum**, **bankAccount**, **when** are mapped on **arg 0,1,2**
- **i**, **j**, **due** are mapped on **local 0,1,2.**

Symbol table

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // 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 ...
   }
```

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

Classical implementation:

- 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 table upwards.

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

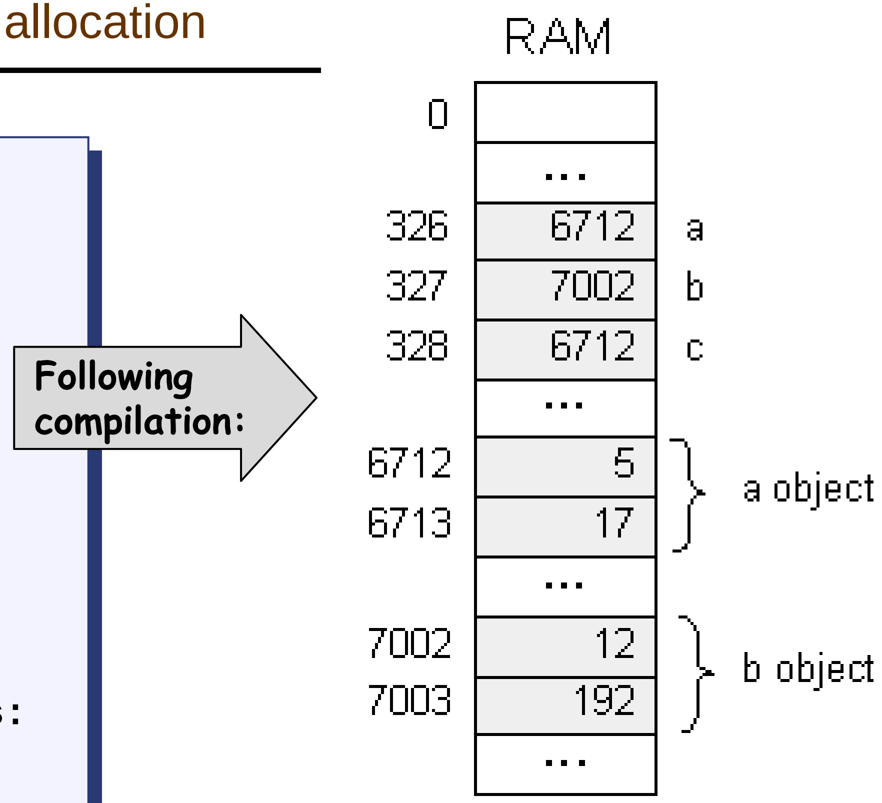
- Static: single copy must be kept alive throughout the program duration
- Field: different copies must be kept for each object
- Local: created on subroutine entry, killed on exit
- Argument: similar to local

Good news: the VM implementation already handles all these details !!!
Hurray !!!

Handling objects: construction / memory allocation

Java code

```
class Complex {
  // Properties (fields):
  int re;  // Real part
  int im;  // Imaginary part
  ...
  /** Constructs a new Complex object. */
  public Complex(int aRe, int aIm) {
    re = aRe;
    im = aIm;
  }
  ...
}
  // The following code can be in any class:
  public void bla() {
    Complex a, b, c;
    ...
    a = new Complex(5,17);
    b = new Complex(12,192);
    ...
    c = a;  // Only the reference is copied
    ...
  }
```



foo = new ClassName(…)

Is handled by causing the compiler to generate code affecting:

foo = Mem.alloc(n)

Where **n** is the number of words necessary to represent the object in the host RAM.

Handling objects: accessing fields

Java code

```
class Complex {
  // Properties (fields):
  int re;  // Real part
  int im;  // Imaginary part
  ...
  /** Constructs a new Complex object. */
  public Complex(int aRe, int aIm) {
    re = aRe;
    im = aIm;
  }
  ...
  // Multiplication:
  public void mult (int c) {
     re = re * c;
     im = im * c;
  }
  ...
}
```

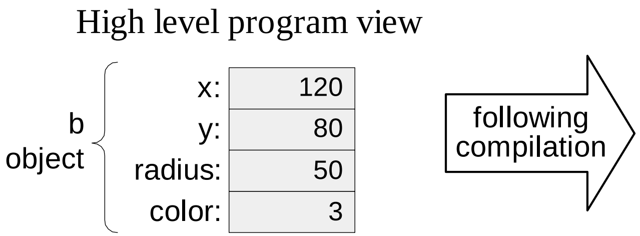
Translating `im = im * c` :

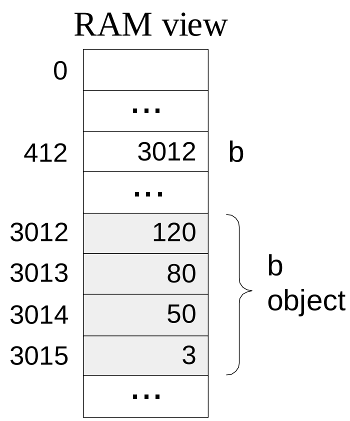
- Look up the symbol table
- Resulting semantics:

```
// im = im * c :
*(this+1) = *(this+1)
            times
            (argument 0)
```

This semantics should be expressed in the target language.

Handling objects: establishing access to the object itself



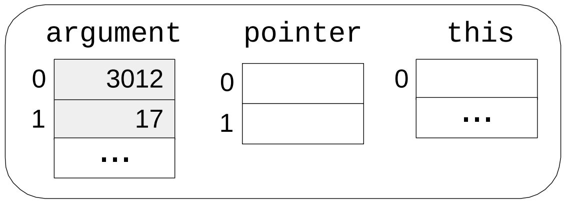


(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

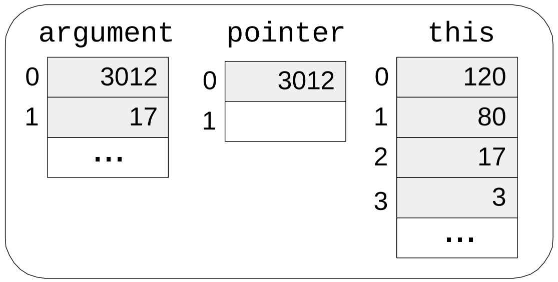
```
/* Assume that b and r were
   passed to the function as
   its first two arguments.
   The following code
   implements the operation
   b.radius=r. */

// Get b's base address:
push argument 0
// Point the this seg. 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 aRe, int aIm) {
    re = aRe;
    im = aIm;
  }
  ...
}

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

Translating `x.mult(5):`

- Can also be viewed as:
  `mult(x,5)`
- Generated code:

```
// x.mult(5):
push x
push 5
call mult
```

General rule: each method call

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

can be translated into

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

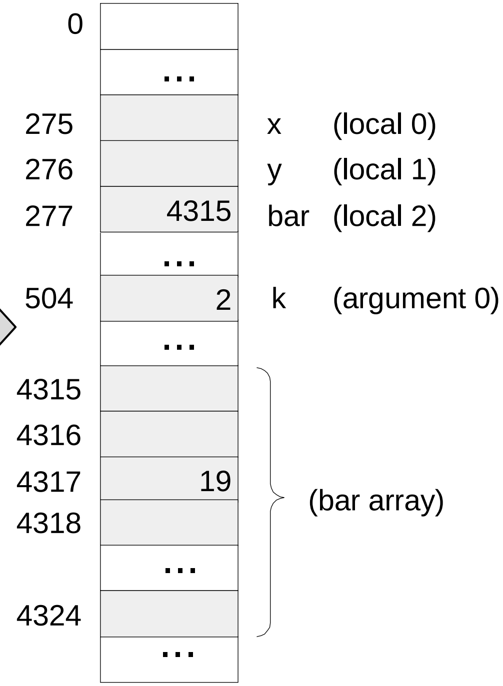
Handling arrays

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`



`bar = new int(n)`

Typically handled by causing the compiler to generate code affecting:

`bar = Mem.alloc(n)`

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
```

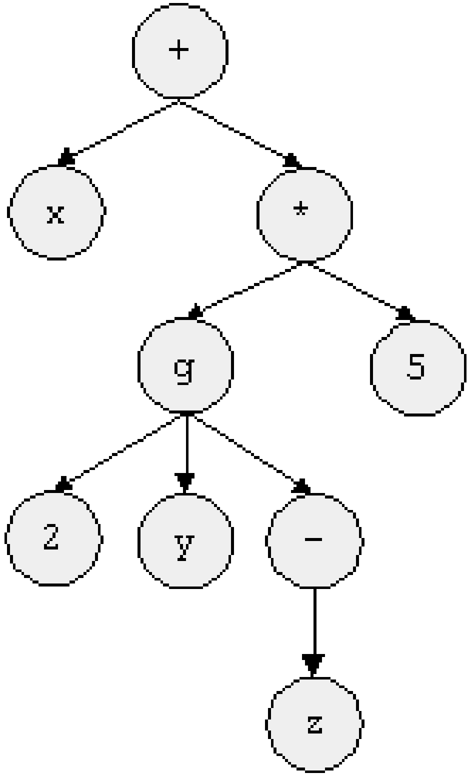
VM Code (actual)

```
// 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

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

Syntax analysis



Code generation

```
push x
push 2
push y
push z
neg
call g
push 5
call mult
add
```

The codeWrite(exp) algorithm:

if *exp* is a number *n*	then	output "`push n`";
if *exp* is a variable *v*	then	output "`push v`";
if *exp* = (*exp1 op exp2*)	then	codeWrite(*exp1*); codeWrite(*exp2*) ; output "`op`";
if *exp* = *op*(*exp1*)	then	codeWrite(*exp1*) ; output "`op`";
if *exp* = *f*(*exp1* … *expN*)	then	codeWrite(*exp1*) … codeWrite(*expN*); output "`call f`".

Handling program flow

Flow of control structure

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

VM pseudo code

```
  VM code for computing ~(cond)
  if-goto L1
  VM code for executing s1
  goto L2
label L1
  VM code for executing s2
label L2
  ...
```

```
while (cond)
   s1
...
```

```
label L1
  VM code for computing ~(cond)
  if-goto L2
  VM code for executing s1
  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 which 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 which are easy to extend: :
  - Limited control structures (no `for`, `switch`, …)
  - Cumbersome handling of char types (cannot use `let x='c'`)

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