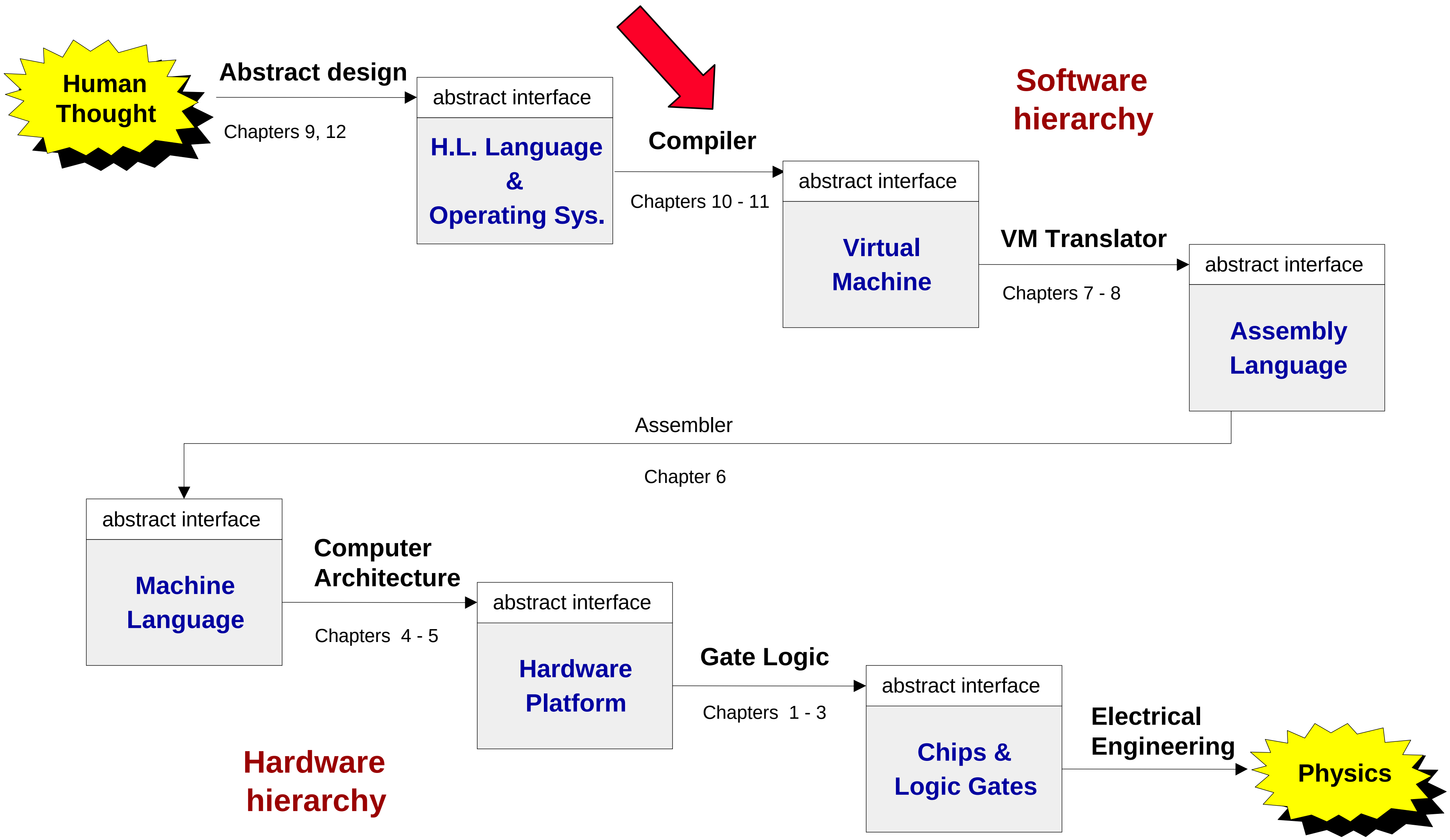
Chapter 11:

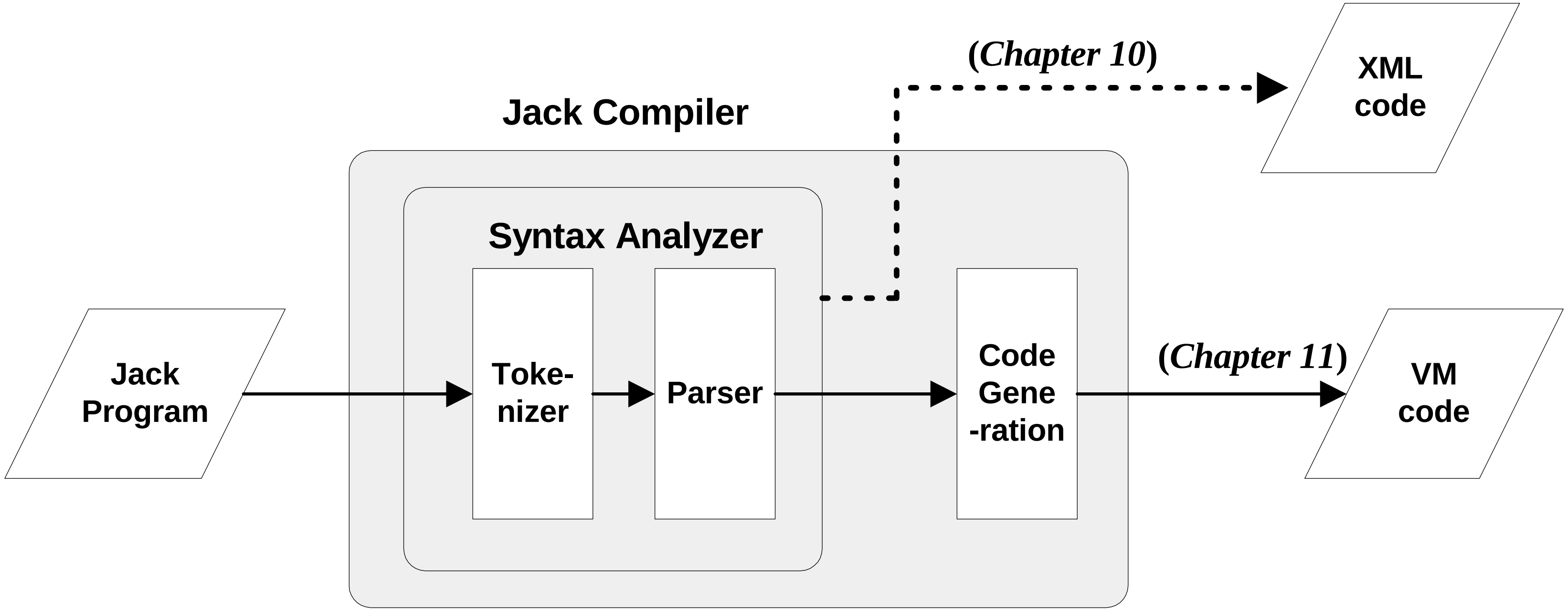
Compiler II: Code Generation

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*: understanding the code
- *Code generation*: constructing semantics



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>

- Extend the syntax analyzer into a full-blown compiler
- Program = a series of operations that manipulate data
- The compiler should convert each "understood" (parsed) source operation and data item into corresponding operations and data items in the target language
- So we have to generate code for
  - handling data
  - handling operations.

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 to it RAM resources)

- *What* ***kind*** *of variable is x?*

  `local`, **`static`**, **`field`**, **`argument`** ?

  (Need to know in order to properly manage its life cycle).

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 list of hash tables, each reflecting a single scope nested within the next one in the list
- The 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 handles all these details !!! Hurray!!!

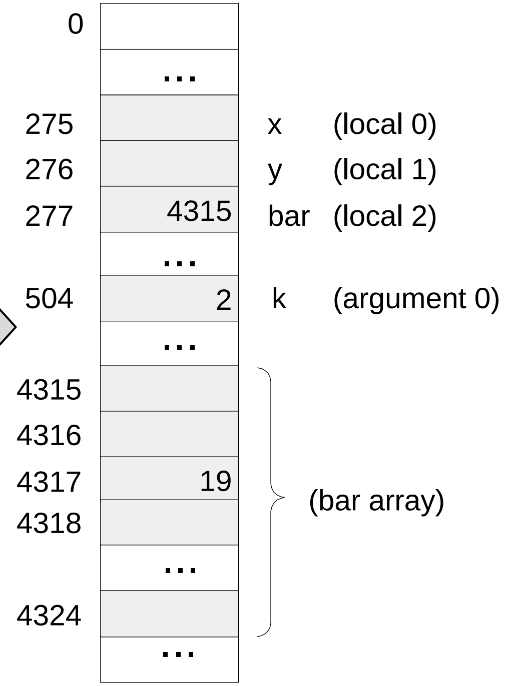
Handling arrays

Java code

```
class Complex {
 ...
 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)`

Is 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 (final)

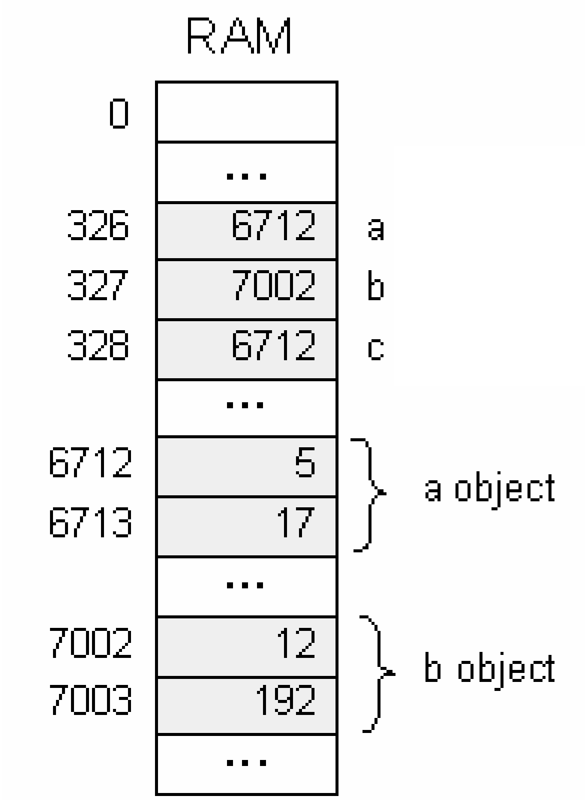
```
// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add
// Use the that segment to access x[k]
pop pointer 1
push constant 19
pop that 0
```

Handling objects: 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
    ...
  }
```

Following compilation:




```
foo = new ClassName(…)
```

Is typically handled by causing the compiler to generate code affecting:

```
foo = Mem.alloc(n)
```

Handling objects: operations

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;
  }
  ...
}
```

<u>Translating</u> `im = im * c` :

- Look up the symbol table
- Resulting semantics:

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

- Of course this should be written in the target language.

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
```

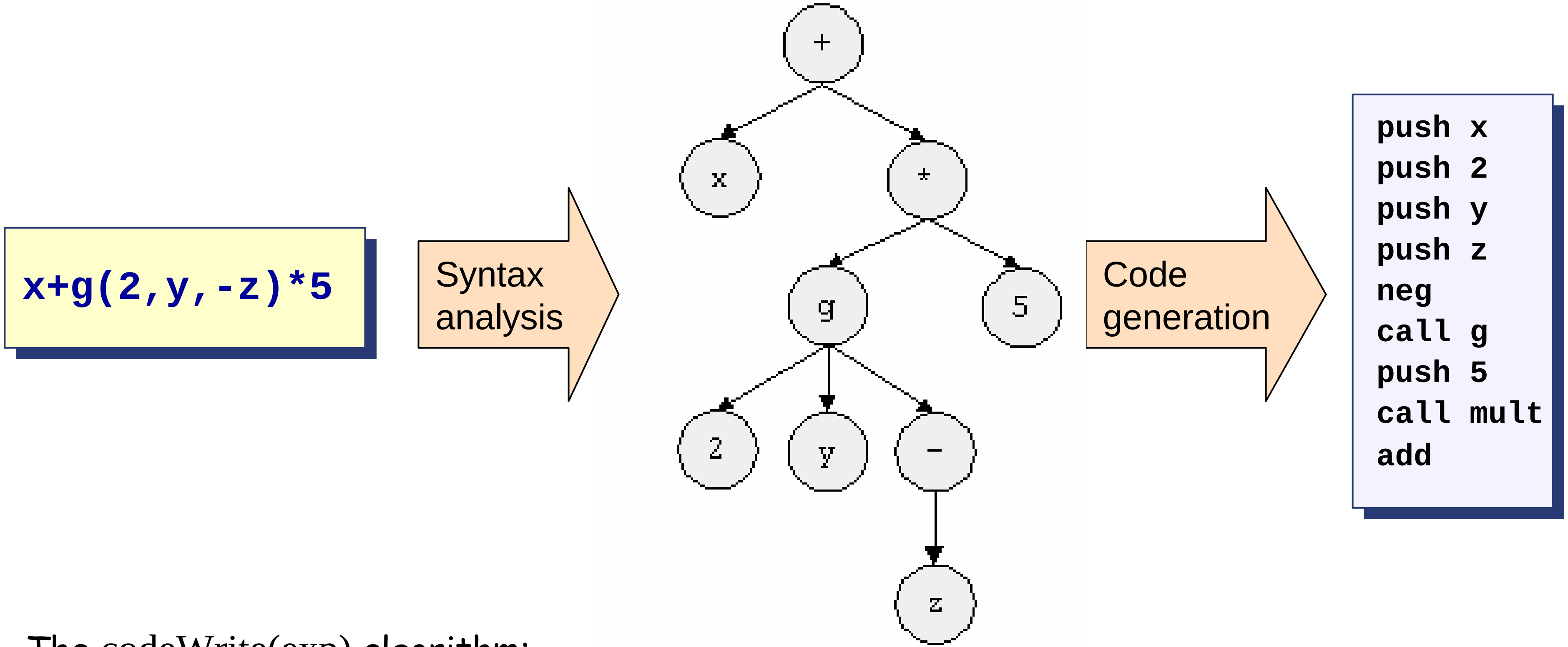
General rule: each method call

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

can be translated into

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

Generating code for expressions



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

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 control flow (e.g. IF, WHILE)

Source code	*Generated code*
`if (cond)`	`  code for computing ~cond`
`   s1`	`  if-goto L1`
`else`	`  code for executing s1`
`   s2`	`  goto L2`
...	`label L1`
	`  code for executing s2`
	`label L2`
	...

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

Program flow

Flow of control structure	*VM pseudo code*

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

```
  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

- "Hard" Jack simplifications:
  - Primitive type system
  - No inheritance
  - No public class fields (e.g. must use `r=c.getRadius()` rather than `r=c.radius`)
- "Soft" Jack simplifications:
  - Limited control structures (no `for`, `switch`, …)
  - Cumbersome handling of char types (cannot use `let x='c'`)
- Optimization
  - For example, `c++` will be translated into `push c`, `push 1`, `add`, `pop c`.
  - Parallel processing
  - Many other examples of possible improvements …