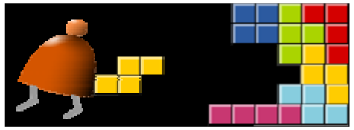
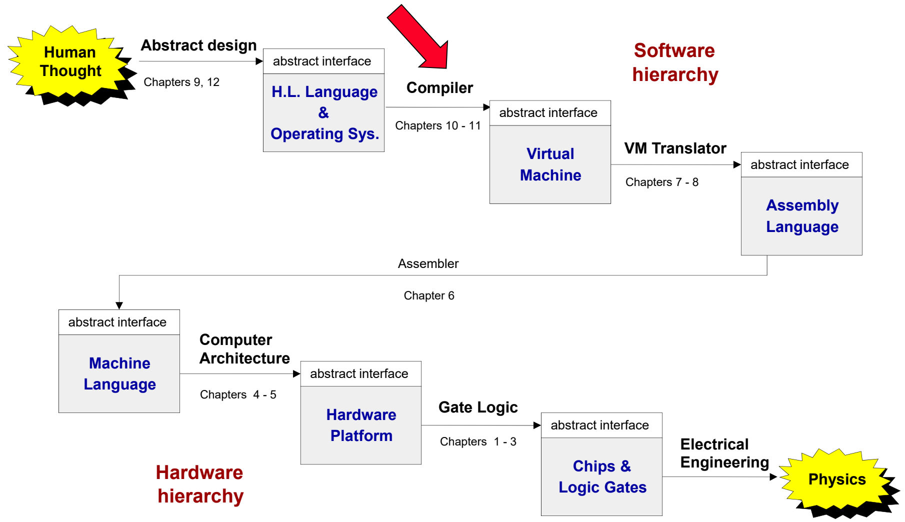
Compiler II: Code Generation



Building a Modern Computer From First Principles

www.nand2tetris.org

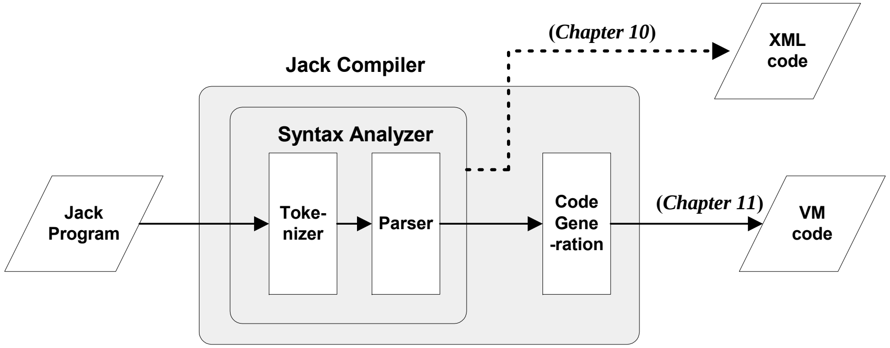
Course map



The big picture

1. Syntax analysis: extracting the semantics from the source code (previous chapter)

2. Code generation: expressing the semantics using the target language (This chapter)



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

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:

- 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.

Memory segments (review)

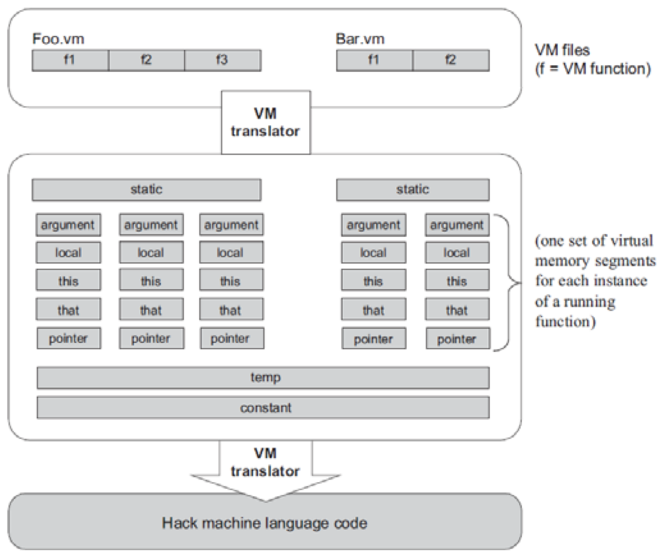
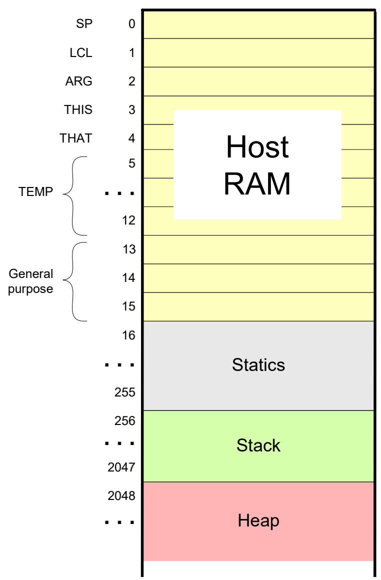


Figure 7.7 The virtual memory segments are maintained by the VM implementation.

VM implementation on the Hack platform (review)

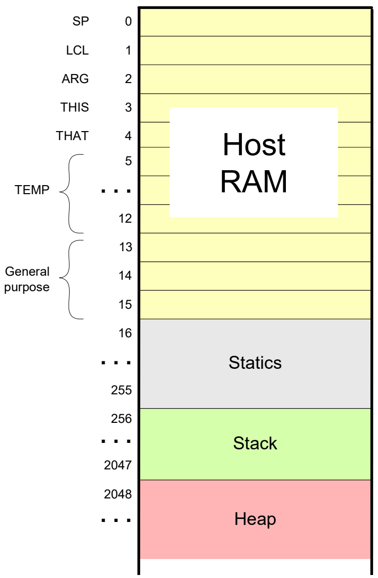


Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

The stack: mapped on `RAM[256 .. 2047]`; The stack pointer is kept in RAM address `SP`

`static`: mapped on `RAM[16 ... 255]`; each segment reference `static` *i* appearing in a VM file named `f` is compiled to the assembly language symbol `f.i` (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

VM implementation on the Hack platform (review)



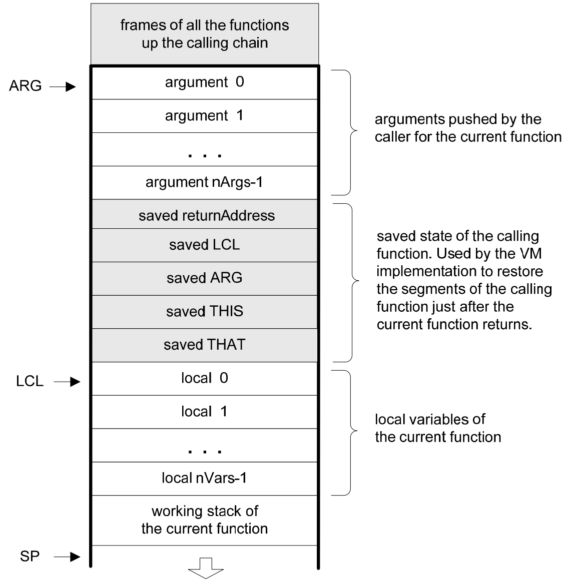
`local`, `argument`: these method-level segments are stored in the stack, The base addresses of these segments are kept in RAM addresses `LCL` and `ARG`. Access to the *i*-th entry of any of these segments is implemented by accessing `RAM[segmentBase +` *i*`]`

`this`, `that`: these dynamically allocated segments are mapped somewhere from address `2048` onward, in an area called "heap". The base addresses of these segments are kept in RAM addresses `THIS`, and `THAT`.

`constant`: a truly a virtual segment: access to `constant` *i* is implemented by supplying the constant *i*.

`pointer`: contains `this` and `that`.

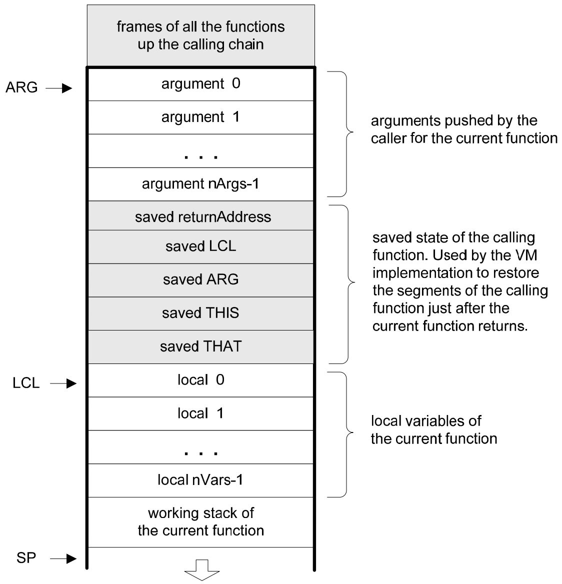
VM implementation on the Hack platform (review)



Global stack: the entire RAM area dedicated for holding the stack

Working stack: The stack that the current function sees

VM implementation on the Hack platform (review)



- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, and has access only to its memory segments
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.

Code generation example

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

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

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
```

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?

static, field, local, 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 propetrties
  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-define 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.

Handling variables: mapping them on memory segments (example)

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

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

Handling variables: mapping them on memory segments (example)

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

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 argument 0,1,2

The local variables i, j, due
are mapped on local 0,1,2.

Handling variables: symbol tables

```
class BankAccount {
  static int nAccounts;
  static int bankCommission;
  field int id;
  field String owner;
  field int balance;
  method void transfer(int sum, BankAccount from, Date when){
    var int i, j;
    var Date due;
    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

Variables life cycle

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

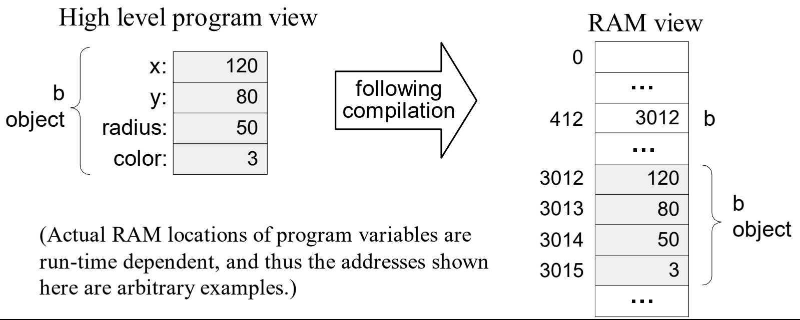
argument variables: similar to local variables.

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

Handling objects: establishing access to the object's fields

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

```
Class Ball {
  field int x, y, radius, color;
  method void SetR(int r) { radius = r; }
}
...
Ball b; b=Ball.new();
b.SetR(17);
```



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

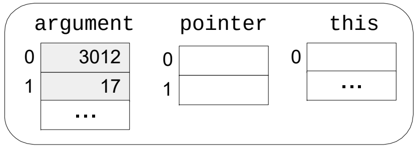
Handling objects: establishing access to the object's fields

```
Class Ball {
  ...
  void SetR(int r) { radius = r; }
}
...
Ball b;
b.SetR(17);
```

Handling objects: establishing access to the object's fields

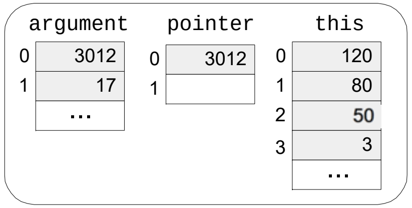
```
Class Ball {      need to know which
                  instance it is working on
  ...
  void SetR(int r) { radius = r; }
}
...               need to pass the object
Ball b;           into the function
b.SetR(17); => Ball.SetR(b, 17)

// 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:



this 0 is now aligned with RAM[3012]

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



Handling objects: construction / memory allocation

Java code

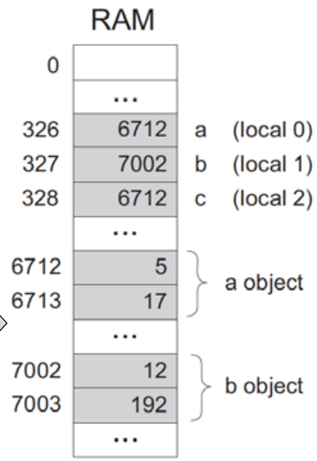
```
class Complex {
    // Fields (properties):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex (int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}
```


Handling objects: construction / memory allocation

Java code

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

Following execution:



Handling objects: construction / memory allocation

Java code

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

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 {
    // 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;
    }
    /** 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: method calls

Java code

```
class Complex {
  ...
  public void mult (int c) {
    re = re * c;
    im = im * c;
  }
  ...
}
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
```


Handling objects: method calls

Java code

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

General rule: each method call

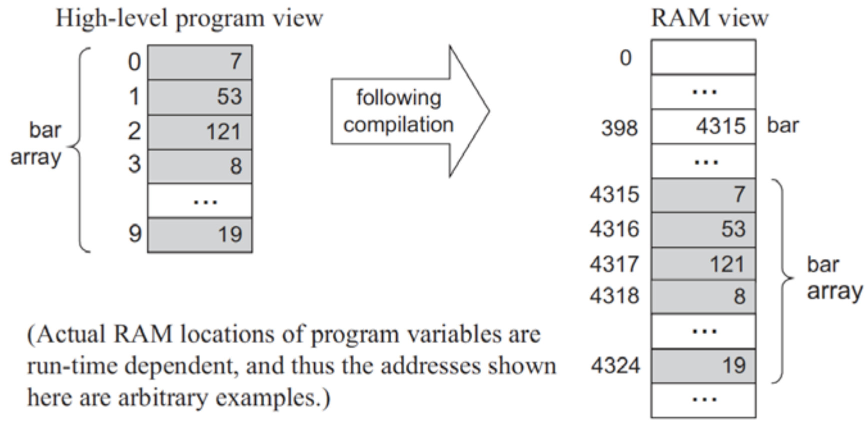
`foo.bar(v1,v2,...)`

is translated into:

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


Handling array

```
int foo() {  // some language, not Jack
    int bar[10];
    ...
    bar[2] = 19;
}
```



Handling array

VM code

```
/* Assume that the bar array is the first local variable declared in the
   high-level program. The following VM code implements the operation
   bar[2]=19,  i.e., *(bar+2)=19. */
 push local 0      // Get bar's base address
 push constant 2
 add
 pop  pointer 1    // Set that's base to (bar+2)
 push constant 19
 pop  that 0       // *(bar+2)=19
 ...
```

Virtual memory segments just before the bar[2]=19 operation:

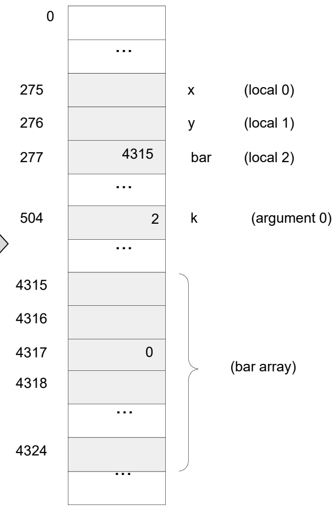
Virtual memory segments just after the bar[2]=19 operation:

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:



How to compile:

`bar = new int(n)` ?

Generate code affecting:

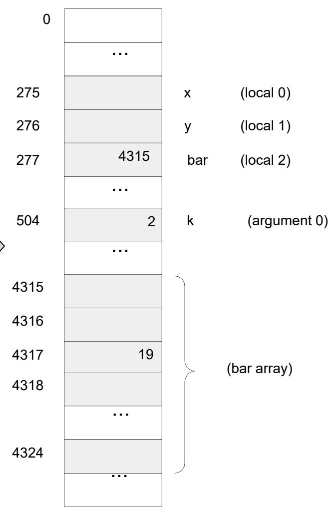
`bar = Memory.alloc(n)`

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`

How to compile: `bar[k] = 19` ?

Handling arrays: accessing an array entry by its index

How to compile: `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]

// addr points to bar[k]
pop addr

push 19

// Set bar[k] to 19

pop *addr
```

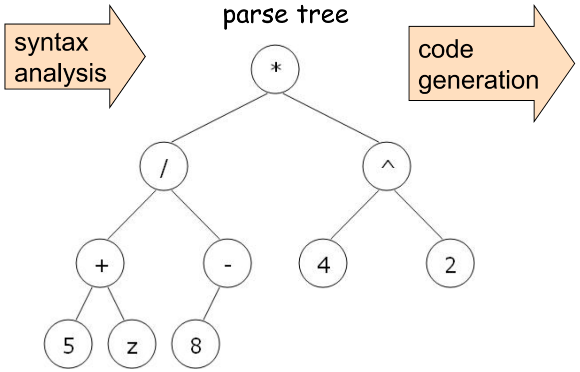
VM Code (actual)

```
// bar[k]=19,
// or *(bar+k)=19
push local 2
push argument 0
add
// Use a pointer to
// access x[k]


pop pointer 1

push constant 19



pop that 0
```

Handling expressions

High-level code

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



VM code

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


Handling expressions (Jack grammar)

Expressions: (annotations: term, binary term)

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'

'x': x appears verbatim
x: x is a language construct
x?: x appears 0 or 1 times
x*: x appears 0 or more times
x|y: either x or y appears
(x,y): x appears, then y.

Handling expressions (Jack grammar)

Expressions: (annotations: term; constant; variable; function; unary op)

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'

'x': x appears verbatim
x: x is a language construct
x?: x appears 0 or 1 times
x*: x appears 0 or more times
x|y: either x or y appears
(x,y): x appears, then y.

Handling expressions

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

The codeWrite(*exp*) algorithm:

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 $f(exp_1, ..., exp_n)$ then codeWrite(exp1); ... codeWrite(expn); output "call f";

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

The Jack grammar (Expression)

Expressions:

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'

From parsing to code generation (simplified expression)

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

From parsing to code generation

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

```
EXP() :
   TERM();
   while (next()==OP)
      OP();
      TERM();
```


From parsing to code generation

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

```
EXP() :
   TERM();
   while (next()==OP)
      OP();
      TERM();
```

```
TERM():
   switch (next())
     case INT:
                 eat(INT);
     case VAR:
                 eat(VAR);
```

From parsing to code generation

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

```
OP():
  switch (next())
    case +: eat(ADD);

    case -: eat(SUB);

    case *: eat(MUL);

    case /: eat(DIV);
```

```
EXP() :
  TERM();
  while (next()==OP)
    OP();
    TERM();

TERM():
  switch (next())
    case INT:
          eat(INT);
    case VAR:
          eat(VAR);
```


From parsing to code generation

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

```
OP():
  switch (next())
    case +: eat(ADD);

    case -: eat(SUB);

    case *: eat(MUL);

    case /: eat(DIV);
```

```
EXP() :
  TERM();
  while (next()==OP)
    OP();
    TERM();

TERM():
  switch (next())
    case INT:
          eat(INT);
    case VAR:
          eat(VAR);
```


From parsing to code generation

- EXP → TERM (OP TERM)*
- TERM → integer | variable
- OP → + | - | * | /

```
OP():
  switch (next())
    case +: eat(ADD);
          return 'add';
    case -: eat(SUB);
          return 'sub';
    case *: eat(MUL);
          return 'call Math.mul';
    case /: eat(DIV);
          return 'call Math.div';
```

```
EXP() :
  TERM();
  while (next()==OP)
    op=OP();
    TERM();
    write(op);

TERM():
  switch (next())
    case INT: write('push constant '
                  +next());
          eat(INT);
    case VAR: write('push '
                  +lookup(next()));
          eat(VAR);
```


The Jack grammar (Expression)

Expressions:

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'

The Jack grammar (statement)

Statements:

statements: statement*

statement: letStatement | ifStatement | whileStatement | doStatement | returnStatement

letStatement: **'let'** varName (**'['** expression **']'**)? **'='** expression **';'**

ifStatement: **'if'** **'('** expression **')'** **'{'** statements **'}'** (**'else'** **'{'** statements **'}'**)?

whileStatement: **'while'** **'('** expression **')'** **'{'** statements **'}'**

doStatement: **'do'** subroutineCall **';'**

ReturnStatement **'return'** expression? **';'**

```
STATEMENTS() :
   while (next() in {let, if, while, do, return})
      STATEMENT();
```


The Jack grammar (statement)

Statements:

statements: statement*

statement: letStatement | ifStatement | whileStatement | doStatement | returnStatement

letStatement: **'let'** varName (**'['** expression **']'**)? **'='** expression **';'**

ifStatement: **'if'** **'('** expression **')'** **'{'** statements **'}'** (**'else'** **'{'** statements **'}'**)?

whileStatement: **'while'** **'('** expression **')'** **'{'** statements **'}'**

doStatement: **'do'** subroutineCall **';'**

Retur

```
STATEMENT() :
   switch (next())
      case LET:       LET_STAT();
      case IF:        IF_STAT();
      case WHILE:     WHILE_STAT();
      case DO:        DO_STAT();
      case RETURN:    RETURN_STAT();
```


let statement

letStatement: **'let'** varName (**'['** expression **']'**)? **'='** expression **';'**

Parsing

```
LET_STAT():
   eat(LET);
   eat(VAR);
   eat(EQ);
   EXP();
   eat(SEMI);
```

Parsing with code generation

```
LET_STAT():
   eat(LET);
   variable=lookup(next());
   eat(VAR);
   eat(EQ);
   EXP();
   eat(SEMI);
   write('pop ' + variable)
```


Handling program flow

ifStatement: **'if'** **'('** expression **')'** **'{'** statements **'}'** (**'else'** **'{'** statements **'}'**)?

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

High-level code

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
   ...
```

Handling program flow

whileStatement: **'while'** '(' expression ')' '{' statements '}'

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

High-level code

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
   ...
```


The Jack grammar (class)

Program structure: A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:

class: **'class'** className '{' classVarDec* subroutineDec* '}'

classVarDec: (**'static'** | **'field'**) type varName (',' varName)* ';'

type: **'int'** | **'char'** | **'boolean'** | className

subroutineDec: (**'constructor'** | **'function'** | **'method'**) (**'void'** | type) subroutineName '(' parameterList ')' subroutineBody

parameterList: ((type ... ',' type varName)*)?

subroutineBody: '{' varDec ... '}'

varDec: **'var'** type ... (',' varName)* ';'

className: identifier

subroutineName: identifier

varName: identifier

```
CLASS() :
  eat(CLASS);
  eat(ID);
  eat('{');
  while (next() in {static, field})
     CLASSVARDEC();
  while (next() in {constructor, function, method})
     SUBROUTINEDEC();
  eat('}');
```


The Jack grammar (class)

Program structure: A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:

class: **'class'** className '{' classVarDec* subroutineDec* '}'

classVarDec: (**'static'** | **'field'**) type varName (',' varName)* ';'

type: **'int'** | **'char'** | **'boolean'** | className

subroutineDec: (**'constructor'** | **'function'** | **'method'**) (**'void'** | type) subroutineName '(' parameterList ')' subroutineBody

parameterList: ((type ... ',' type varName)*)?

subroutineBody: '{' varDec ... '}'

varDec: **'var'** type ... (',' varName)* ';'

className: identifier

subroutineName: identifier

varName: identifier

```
CLASS() :
  eat(CLASS); class=registerClass(next());
  eat(ID);
  eat('{');
  while (next() in {static, field})
     CLASSVARDEC(class);
  while (next() in {constructor, function, method})
     SUBROUTINEDEC(class);
  eat('}');
```


The Jack grammar (class)

classVarDec: (**'static'** | **'field'**) type varName (',' varName)* ';'

type: **'int'** | **'char'** | **'boolean'** | className

```
CLASSVARDEC(class) :
  switch (next())
    case static: eat(STATIC);  kind=STATIC;
    case field: eat(FIELD);  kind=FIELD;
  switch (next())
    case int:      type=INT; eat(INT);
    case char:     type=CHAR; eat(CHAR);
    case boolean:  type=BOOLEAN; eat(BOOLEAN);
    case ID:       type=lookup(next()); eat(ID);
  registerClassVar(class, next(), kind, type);
  eat(ID);
  while (next()=COMMA)
     registerClassVar(class, next(), kind, type);
     eat(ID);
```

Put them together

```
class BankAccount {
  static int nAccounts;
  static int bankCommission;
  field int id;
  field String owner;
  field int balance;
  method void transfer(int sum, BankAccount from, Date when){
    var int i, j;
    var Date due;
    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


```
...
let balance = (balance + sum) – commission(sum * 5)
```

Pseudo VM code

```
function BankAccount.commission
  // Code omitted
function BankAccount.transfer
  // Code for setting "this" to
  // to the passed object (omit
  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.transfer 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 …