We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

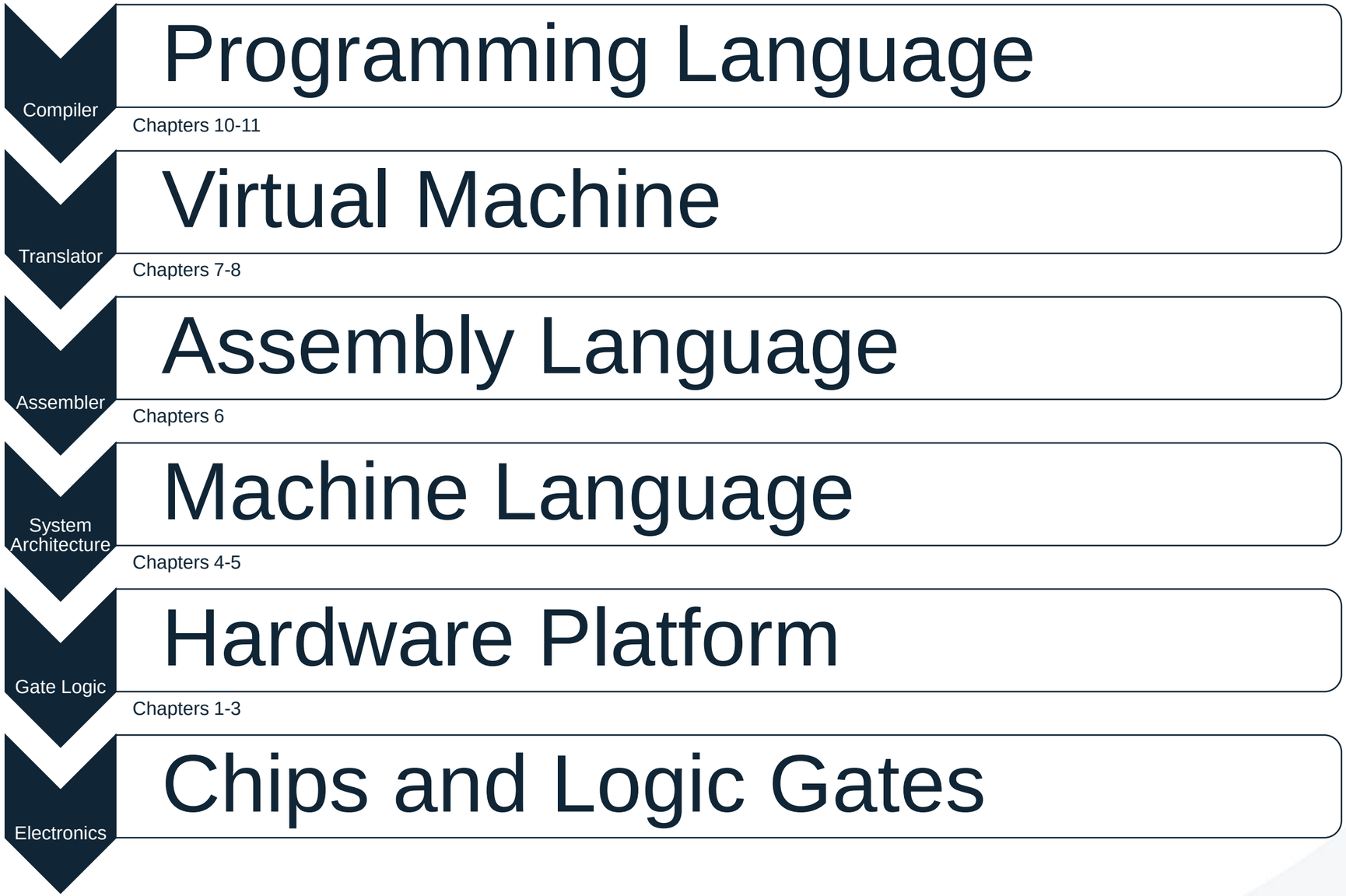
We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

THE UNIVERSITY of ADELAIDE

Computer Systems

Lecture 09/10: Language Parsing and Code Generation

Review: The whole system



THE UNIVERSITY of ADELAIDE

Review: The Jack Language & Compiler Fundamentals

Review: Noteworthy features of the Jack language

- The (cumbersome) `let` keyword, as in `let x = 0;`
- The (cumbersome) `do` keyword, as in `do reduce();`
- No operator priority:

`1 + 2 * 3` yields `9`, if expressions are evaluated left-to-right;

To effect the commonly expected result, use `1 + (2 * 3)`

- Only three primitive data types: `int`, `boolean`, `char`;
In fact, each one of them is treated as a 16-bit value
- No casting; a value of any type can be assigned to a variable of any type
- Array declaration: `Array x;` followed by `x = Array.new();`
- Static methods are called `function`
- Constructor methods are called `constructor`
- Invoking a constructor is done using the syntax *`ClassName`*`.new(`*`argsList`*`)`

All of these design decisions have been taken to make building a compiler easier.

Review: Jack program structure

```
class ClassName
{
    field variable declarations;
    static variable declarations;

    constructor type name ( parameterList )
    {
        local variable declarations;
        statements
    }
    method type name ( parameterList )
    {
        local variable declarations;
        statements
    }
    function type name ( parameterList )
    {
        local variable declarations;
        statements
    }
}
```

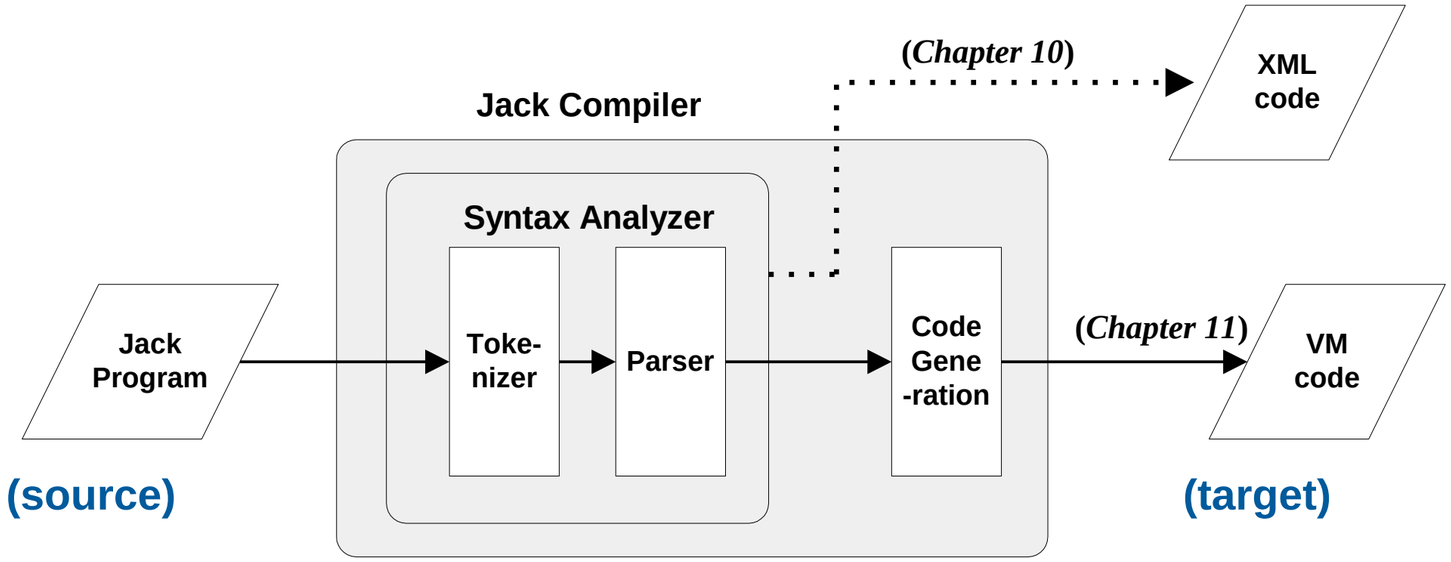
About this spec:

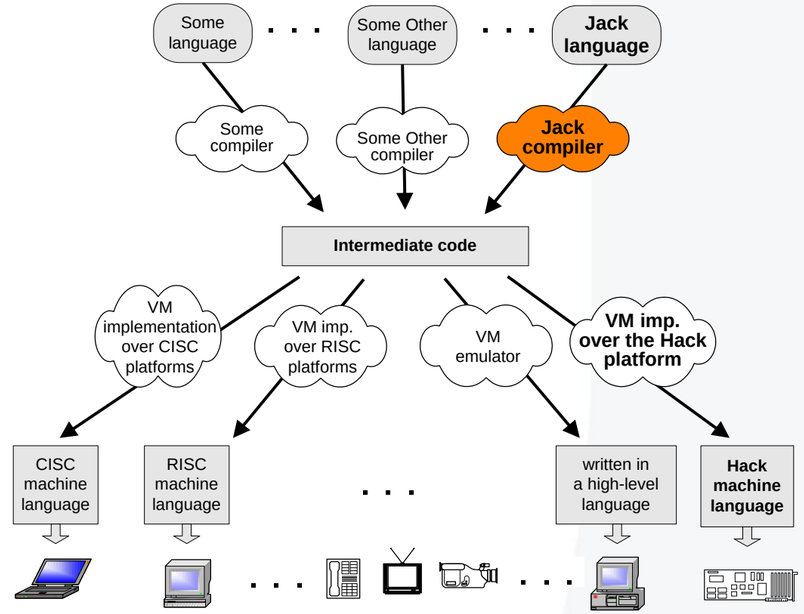
- Every part in this spec can appear 0 or more times
- The order of the field / static declarations is arbitrary
- The order of the subroutine declarations is arbitrary
- Each *type* is either int, boolean, char, or a class name.

A Jack program:

- Each class is written in a separate file (compilation unit)
- Jack program = collection of one or more classes, one of which must be named `Main`
- The `Main` class must contain at least one method, named `main()`

Compiler architecture (front end)





- Syntax analysis: understanding the semantics implied by the source code
  - Tokenizing: creating a stream of “tokens”
  - Parsing: matching the token stream with the language grammar

  XML output = one way to demonstrate that the syntax analyzer works
- Code generation: reconstructing the semantics using the syntax of the target code.

THE UNIVERSITY of ADELAIDE

Tokenizing / Lexical analysis

Code Fragment

```
while ( count < 100 ) /** demonstration code */
{
     let count = count + 1 ;
}
```

Tokens

```
while
(
count
<
100
)
{
let
count
=
count
+
1
;
}
```

Remove white space

Construct a token list (language tokens)

Things to worry about:

- Language specific rules:
  e.g. how to treat “++”
- Language-specific classifications:
  keyword, symbol, identifier, integerConstant, stringConstant,...
- **While we are at it, we can have the tokenizer record not only the token, but also its lexical lassification (as defined by the source language grammar).**

Jack Tokenizer

Char	*XML Entity*
<	<
>	>
'	'
"	"
&	&

Source code

```
if (x < 153) {let city = ”Paris”;}
```

Tokenizer's output

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```

THE UNIVERSITY of ADELAIDE

Syntax Analysis: Parsing

THE UNIVERSITY of ADELAIDE

Parsing

The tokenizer discussed thus far is part of a larger program called a *parser*

Each language is characterized by a *grammar.*
The parser is implemented to recognize this grammar in given texts

The parsing process:

A text is given and tokenized

The parser determines whether or not the text can be generated from the grammar

In the process, the parser performs a complete structural analysis of the text

The text can be an expression in a :

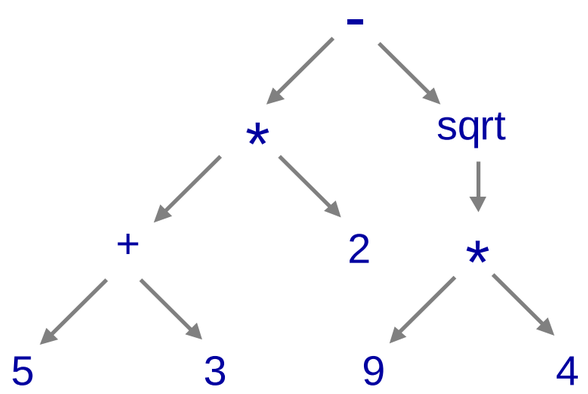
Natural language (English, …)

Programming language (Jack, …).

Parsing examples

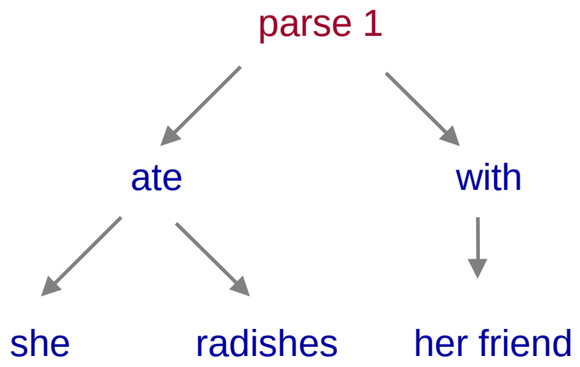
Jack

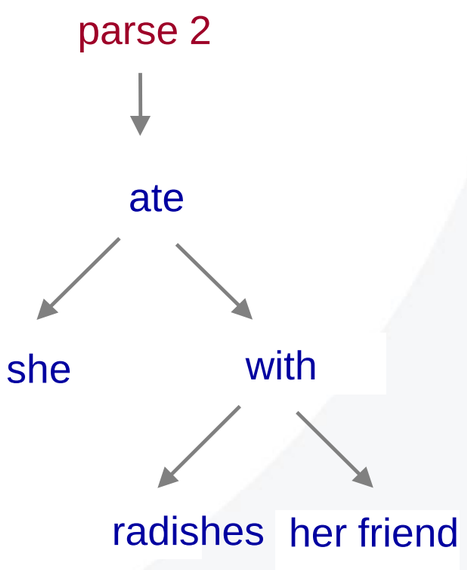
```
((5+3)*2) – sqrt(9*4)
```



English

```
She ate radishes with her friend
```





THE UNIVERSITY of ADELAIDE

More examples of challenging parsing

Time flies like an arrow

Fruit flies like a banana

We gave the monkeys the bananas because they were hungry

We gave the monkeys the bananas because they were over-ripe

I never said she stole my money

I never said she stole my money	Someone else said it
I **never** said she stole my money	I did not say it
I never **said** she stole my money	I implied it
I never said **she** stole my money	Someone did, not necessarily her
I never said she **stole** my money	I considered it borrowed
I never said she stole **my** money	She stole something else of mine
I never said she stole my **money**	She stole something but not money

THE UNIVERSITY of ADELAIDE

A typical grammar of a typical C-like language

Grammar

```
program:         statement

statement:         whileStatement
                 | ifStatement
                 | 'statement' ';'
                 | '{' sequence '}'

whileStatement: 'while' '(' 'expression' ')' statement

ifStatement:     'if' '(' 'expression' ')' statement
                ( 'else' statement )?
sequence:        '' | statement sequence
```

A grammar is a set of rules that describe all legal examples of a language.

It has simple (terminal) forms

It has complex (non-terminal) forms

It is highly recursive.

Code sample

```
while (expression)
{
  if (expression)
     statement;
     while (expression)
     {
        statement;
        if (expression)
           statement;
     }
  while (expression)
  {
     statement;
     statement;
  }

  if (expression)
  {
    statement;
    while (expression)
    {
       statement;
       statement;
    }
    if (expression)
       if (expression)
          statement;
  }
}
```

THE UNIVERSITY of ADELAIDE

Parse tree

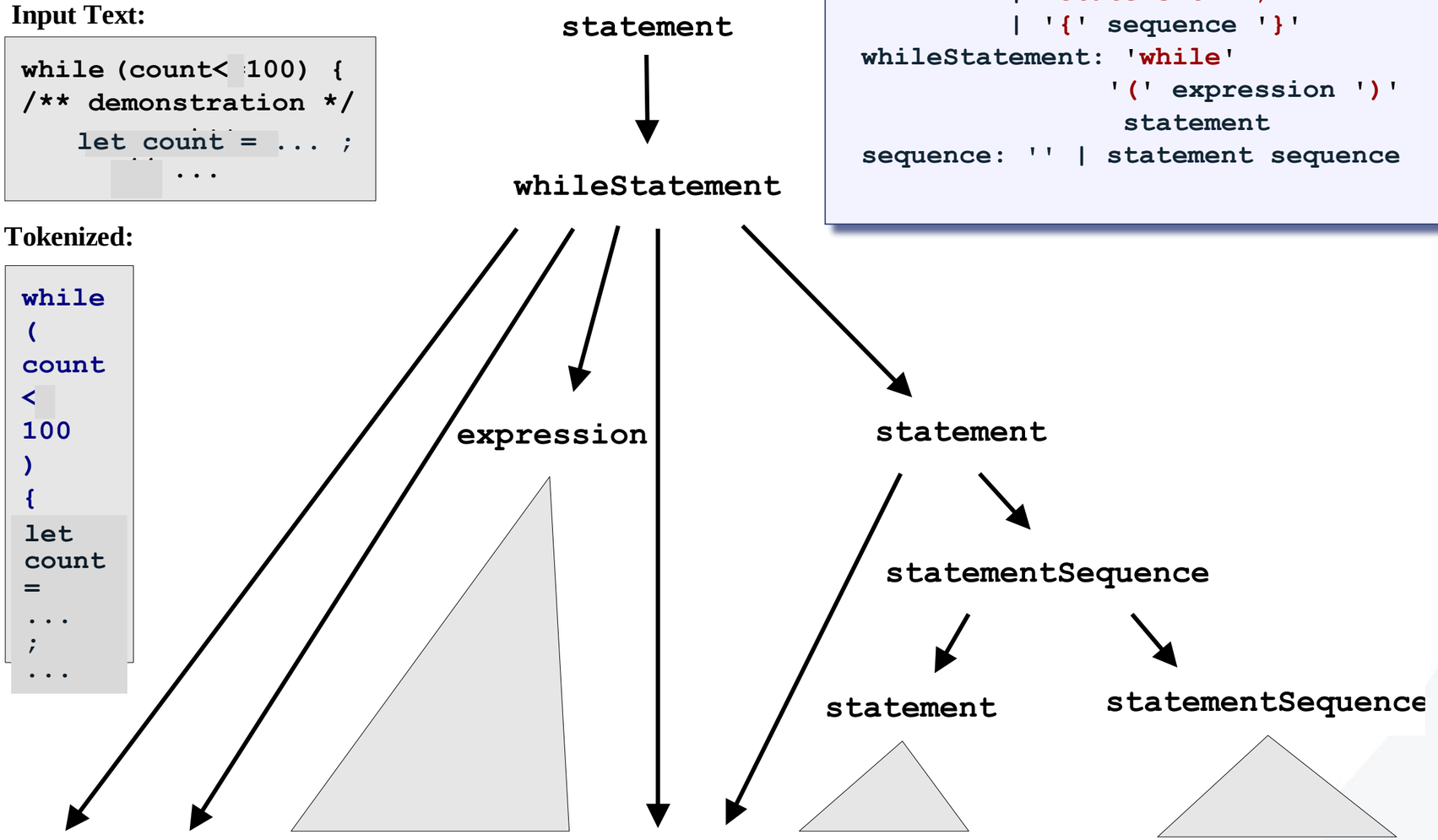
```
program:  statement;

statement: whileStatement
         | ifStatement
         | 'statement' ';'
         | '{' sequence '}'
whileStatement: 'while'
                '(' expression ')'
                 statement
sequence: '' | statement sequence
```

Input Text:

```
while (count< 100) {
/** demonstration */
    let count = ... ;
        ...
```

Tokenized:

```
while
(
count
<
100
)
{
let
count
=
...
;
...
```



THE UNIVERSITY of ADELAIDE

Recursive descent parsing

grammar

```
program:          statement
statement:        whileStatement |
                  ifStatement |
                  'statement' ';' |
                  '{' statements '}'
whileStatement:   'while' '(' 'expression' ')' statement
ifStatement:      'if' '(' 'expression' ')'
                  statement ( 'else' statement )?
statements:       statement*
```

code sample

```
while (expression)
{
   statement;
   statement;
   while (expression)
   {
      while (expression)
         statement;
         statement;
   }
}
```

- LL(1) grammars: the first token determines **the rule**
- In other grammars you have to look ahead more tokens
- Jack is almost LL(1).

<u>Parser implementation:</u> a set of parsing functions one for each rule:

`parseStatement()`

`parseWhileStatement()`

`parseIfStatement()`

`parseStatements()`

The Jack grammar

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	**'class'** | **'constructor'** | **'function'** | **'method'** | **'field'** | **'static'** | **'var'** | **'int'** | **'char'** | **'boolean'** | **'void'** | **'true'** | **'false'** | **'null'** | **'this'** | **'let'** | **'do'** | **'if'** | **'else'** | **'while'** | **'return'**
symbol:	**'{'** | **'}'** | **'('** | **')'** | **'['** | **']'** | '.' | ',' | ';' | '+' | '-' | **'*'** | **'/'** | **'&'** | **'|'** | '<' | '>' | '=' | '~'
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	'"' A sequence of Unicode characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_') not starting with a digit.
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 varName) (',' type varName)*)?
subroutineBody:	**'{'** varDec* statements **'}'**
varDec:	**'var'** type varName (',' varName)* ';'
className:	identifier
subroutineName:	identifier
varName:	Identifier

'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.

THE UNIVERSITY of ADELAIDE

The Jack grammar (cont.)

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? '**;**'

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**'

'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.

THE UNIVERSITY of ADELAIDE

Jack syntax analyser in action

```
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>
 <statement>
  <letstatement>
   <keyword> let </keyword>
   <identifier> temp </identifier>
   <symbol> = </symbol>
   <expression>
     <term>
       <symbol> ( </symbol>
       <expression>
         <term>
           <varName>
            <identifier> xxx </identifier>
           </varName>
         </term>
         <symbol> + </symbol>
         <term>
          <integerConstant> 12 </integerConstant>
         </term>
          ...
```

Syntax analyzer

- Using the language grammar, a programmer can write a syntax analyser program (parser)
- The syntax analyser takes a source text file and attempts to match it on the language grammar
- If successful, it can generate a parse tree in some structured format, e.g. XML.

The syntax analyser's algorithm shown in this slide:

- If xxx is non-terminal, output:

  <xxx>
  Recursive code for the body of **xxx**
  </xxx>

- If **xxx** is terminal (keyword, symbol, constant, or identifier), output:

  <xxx>
  xxx value
  </xxx>

THE UNIVERSITY of ADELAIDE

Jack Tokeniser

We can provide a Jack tokeniser using a variation of the tokenisers developed / used in the workshops

Main interface

next_token()

- token_spelling() // the current token's spelling
- token_kind() // the current token's kind
- token_ivalue() // the current token's integer value

have(TokenKind) // is the current token of the given kind?

mustbe(TokenKind) // current token must be the given kind
// calls next_token() or fatal_error()

Special tokens

tk_eoi and grouping tokens, eg tk_infix_op, tk_term

CompilationEngine:
a recursive top-down parser for Jack

The CompilationEngine effects the actual compilation output.

It gets its input from a Jack Tokeniser and emits its parsed structure into an output file/stream.

The output is generated by a series of parse_xxx() routines, one for every syntactic element xxx of the Jack grammar.

The contract between these routines is that each parse_xxx() routine should read the syntactic construct xxx from the input, read one token exactly beyond xxx, and output the parsing of xxx.
Thus, parse_xxx() must only be called if indeed xxx is the next syntactic element of the input.

In the first version of the compiler, which we now build, this module emits a structured printout of the code, wrapped in XML tags (as in the week 7 and 10 workshops).

Following the workshop approach, the parse_xxx() routines build a parse tree which is turned into XML output by a library routine.

Jack Parser

Parse functions:

ast parse_class() ;
ast parse_class_var_decs() ;
ast parse_static_var_dec() ;
ast parse_field_var_dec() ;
ast parse_type() ;
ast parse_vtype() ;
ast parse_subr_decs() ;
ast parse_constructor() ;
ast parse_function() ;
ast parse_method() ;
ast parse_param_list() ;
ast parse_subr_body() ;
ast parse_var_decs() ;
ast parse_var_dec() ;

ast parse_statements() ;
ast parse_statement() ;
ast parse_let() ;
ast parse_if() ;
ast parse_while() ;
ast parse_do() ;
ast parse_return() ;

ast parse_expr() ;
ast parse_term() ;
ast parse_array_index() ;
ast parse_subr_call() ;
ast parse_expr_list() ;
ast parse_infix_op() ;
ast parse_unary_op() ;
ast parse_keyword_constant() ;

If Statement Example

```
if ::= 'if' '(' expr ')' '{' statements '}' \
                ( 'else' '{' statements '}' )?
```

```
ast parse_if()
{
      mustbe(tk_if) ;
      mustbe(tk_lrb) ;
      ast _expr = parse_expr() ;
      mustbe(tk_rrb) ;

      mustbe(tk_lcb) ;
      ast _then = parse_statements() ;
      mustbe(tk_rcb) ;

      if ( have(tk_else) )
      {
            mustbe(tk_else) ;
            mustbe(tk_lcb) ;
            ast _else = parse_statements() ;
            mustbe(tk_rcb) ;
            return create_if_else(_expr,_then,_else) ;
      }
      return create_if(_expr,_then) ;
}
```

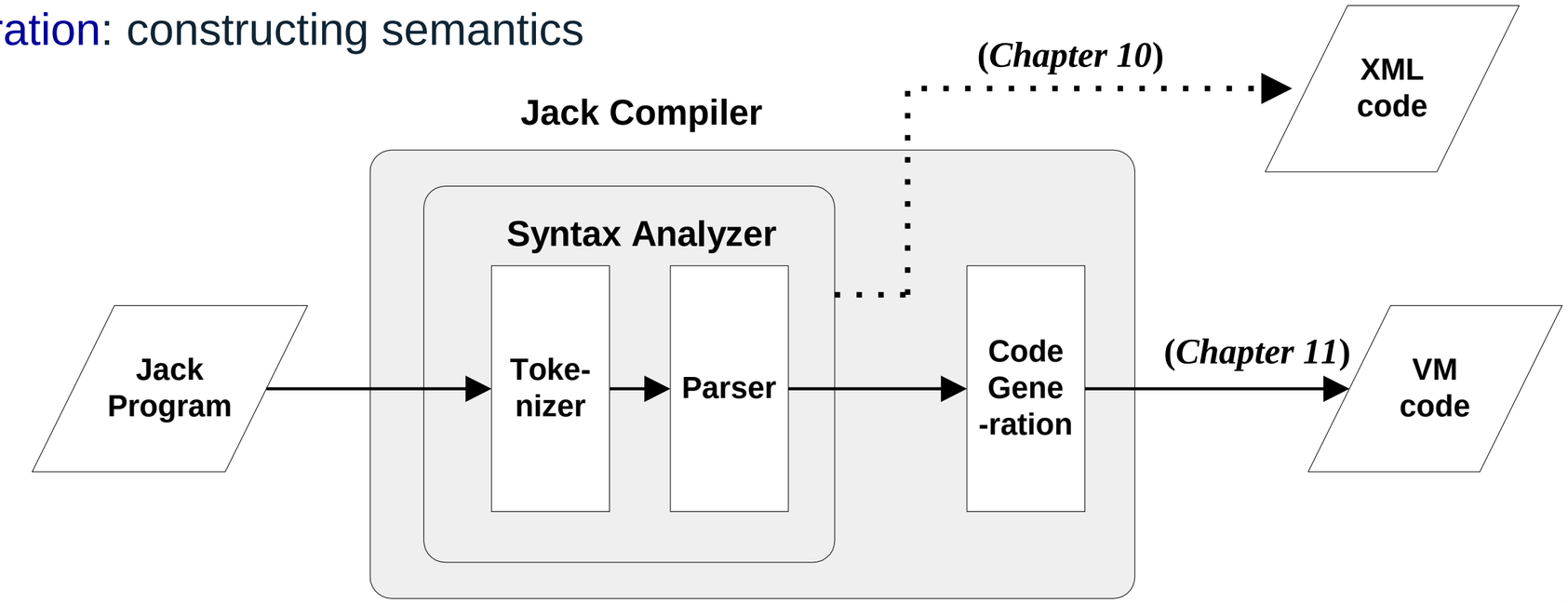
THE UNIVERSITY of ADELAIDE

Syntax Analysis: Parsing

THE UNIVERSITY of ADELAIDE

Summary and next step

- Syntax analysis: understanding syntax
- Code generation: constructing semantics



The code generation challenge:

- Extend the syntax analyser into a full-blown compiler that, instead of generating passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.

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>
  <type><keyword> int </keyword></type>
  <varName><identifier> temp </identifier></varName>
  <symbol> ; </symbol>
</vardec>
<statements>
  <statement>
    <letstatement>
      <keyword> let </keyword>
      <varName><identifier> temp </identifier></varName>
      <symbol> = </symbol>
      <expression>
       <term>
        <symbol> ( </symbol>
        <expression>
         <term>
           <varName><identifier> xxx </identifier></varName>
         </term>
         <op><symbol> + </symbol></op>
         <term>
           <integerConstant> 12 </integerConstant>
        </term>
        </expression>
        <symbol> ) </symbol>
        <op><symbol> * </symbol></op>
        <expression>
           <term>
            <unaryOp><symbol > - </symbol></unaryOp>
                ...
```

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 and handling operations
- Our approach: morph the syntax analyzer into a full-blown compiler: instead of generating XML, we’ll make it generate VM code.

Code Generation

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 memory address to access, typically array elements (silly name, sorry)

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

pointer: holds values this and that so programs can change the segment locations

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>
    <varName><identifier> x </identifier></varName>
  <symbol> = </symbol>
  <expression>
    <term>
      <varName><identifier> x </identifier></varName>
    </term>
    <op><symbol> + </symbol></op>
    <term>
        <integerConstant> 1 </integerConstant>
    </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 indicates 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 onto** `static 0,1`

The object fields `id`, `owner`, `balance` **are mapped onto** `this 0,1,2`

The argument variables `sum`, `bankAccount`, `when` **are mapped onto** `argument 1,2,3`

The local variables `i`, `j`, `due` **are mapped ono** `local 0,1,2`

THE UNIVERSITY of ADELAIDE

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

How the compiler uses symbol tables:

- The compiler builds and maintains a linked list of symbol 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).

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

Symbol Table Interface

At a minimum a symbol table interface should provide:

create()

- to create a new empty symbol table
- **before** parsing a new class, constructor, function or method

delete()

- to delete a symbol table when it is no longer required
- **after** parsing a class, constructor, function or method

insert()

- to insert new entries into the table
- ideally it will report attempts to add duplicate entries ?

lookup()

- to search for an existing entry which may be missing!

Symbol Table Interface

To support a Jack compiler it would be nice to have:

push()

- to create a new empty symbol table at the start of a list of tables
- **before** parsing a constructor, function or method
- insert only affects the first table in the list
- when a lookup fails, the next table in list is automatically searched

pop()

- to remove the first symbol table in a list and delete it
- **after** parsing a constructor, function or method

offset(segment)

- to return the next free offset in a named segment
- each call increments the segment's offset by 1
- the offsets are recorded in the first table in the list
- using push() / pop() ensures that argument and local segment offsets are reset **before** parsing a new constructor, function or method

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: a different copy must be kept for each object, stored in segment `this`

`var` variables: created on subroutine entry, stored in segment `local`

`argument` variables: created during subroutine entry, stored in segment `argument`.

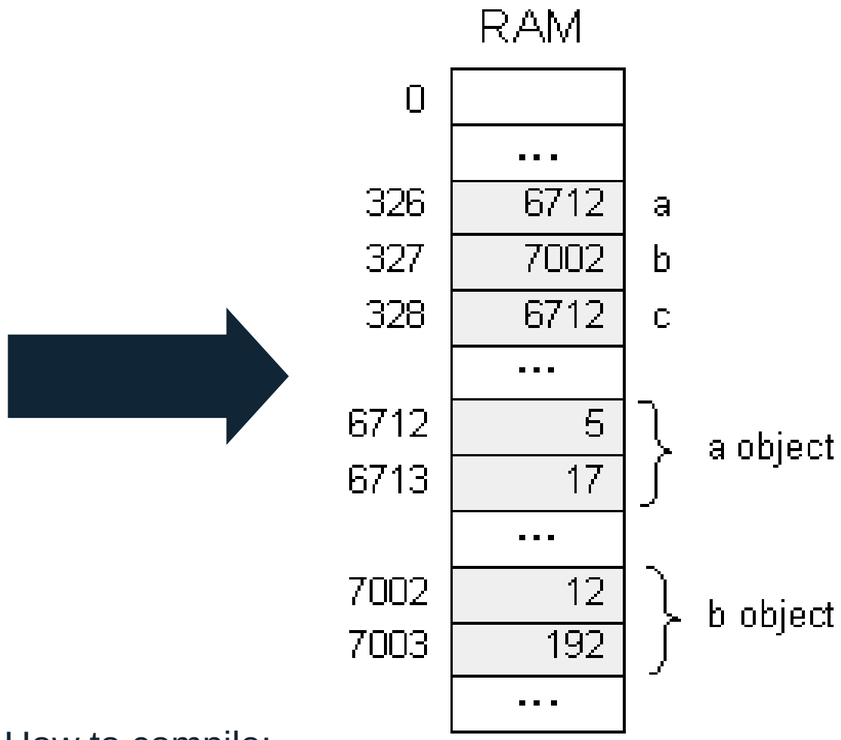
Good news: the VM implementation already handles all these details !

THE UNIVERSITY of ADELAIDE

Handling objects: construction / memory allocation

Jack code

```
class Complex
{
    // Fields (properties):
    field int re;  // Real part
    field int im;  // Imaginary part

    /** Constructs a new Complex number */
    constructor Complex new(int re_, int im_)
    {
        let re = re_;
        let im = im_;
        return this;
    }
}
class Foo
{
    function void bla()
    {
        var Complex a, b, c;

        let a = Complex.new(5,17);
        let b = Complex.new(12,192);
        let c = a; // Only the reference is copied
        return;
    }
}
```



How to compile:

Constructor ClassName new(…) ?

At the start of each constructor, the compiler generates code effecting:

let this = 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

Jack code

```
class Complex
{
    // Properties (fields):
    field int re;  // Real part
    field int im;  // Imaginary part
    /** Constructs a new Complex number */
    constructor Complex new(int re_, int im_)
    {
        let this.re = re_;
        let this.im = im_;
        return this;
    }

    /** Multiplies this Complex number by the
        given scalar, hidden parameter is this */
    method void mult (Complex this,int c)
    {
        let this.re = this.re * c;
        let this.im = this.im * c;
        return;
    }
}
```

How to compile:

```
this.im = this.im * c  ?
```

1. look up the two variables in the symbol table
2. Generate the code:

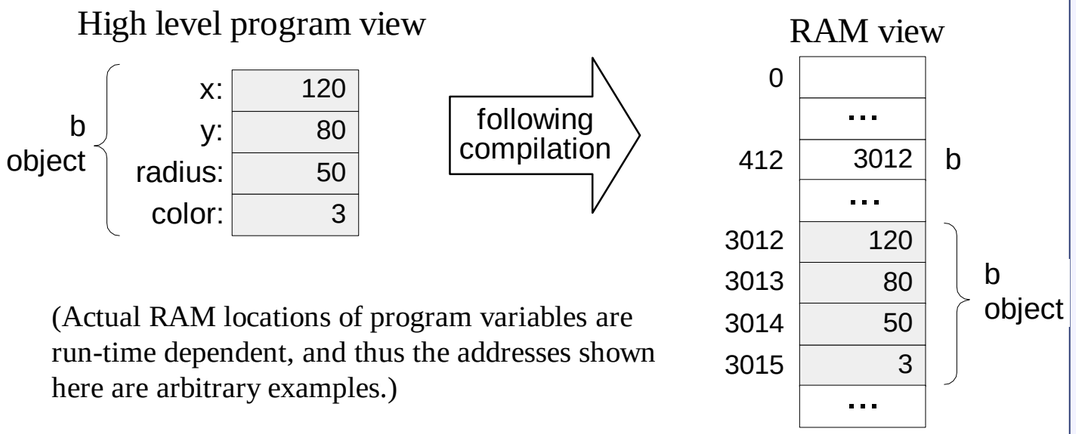
```
this[1] = this[1] * argument[1]
```

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

```
push this 1
push argument 1
call Math.multiply 2
pop this 1
```

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`.

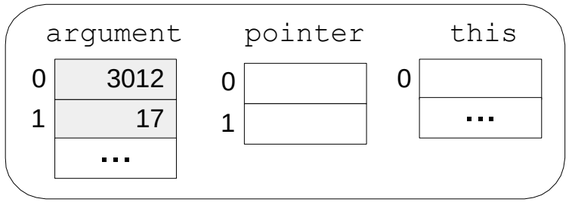


(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

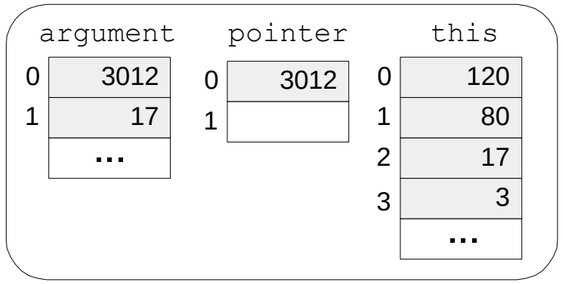
Assume that `b` and `r` were passed to a setR. The object (this) is the hidden first parameter in all method calls.

```
setR(Ball this,int r)
{
        let this.radius = r ;
        return ;
}

// method puts b address into this:
push argument 0
pop pointer 0
// Set b's third field to r:
push argument 1
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]`)

THE UNIVERSITY of ADELAIDE

Handling objects: method calls

Jack code

```
class Complex
{
    // Properties (fields):
    field int re;  // Real part
    field int im;  // Imaginary part
    /** Constructs a new Complex number */
    constructor Complex new(int re_, int im_)
    {
        let this.re = re_;
        let this.im = im_;
        return this;
    }
    /** Multiplies Complex by the scalar */
    method void mult (Complex this,int c)
    {
        let this.re = this.re * c;
        let this.im = this.im * c;
        return;
    }
}
     ...
     var Complex x;
     let x = Complex.new(1,2);
     do x.mult(5);
```

How to compile:

```
do x.mult(5) ;  ?
```

This method call can also be called using:

```
do Complex.mult(x,5) ;
```

Both generate the following code:

```
push x
push 5
call Complex.mult 2
```

General rule: each method call

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

is translated into:

```
push foo
push v1
...
push vn
call bar <n+1>
```

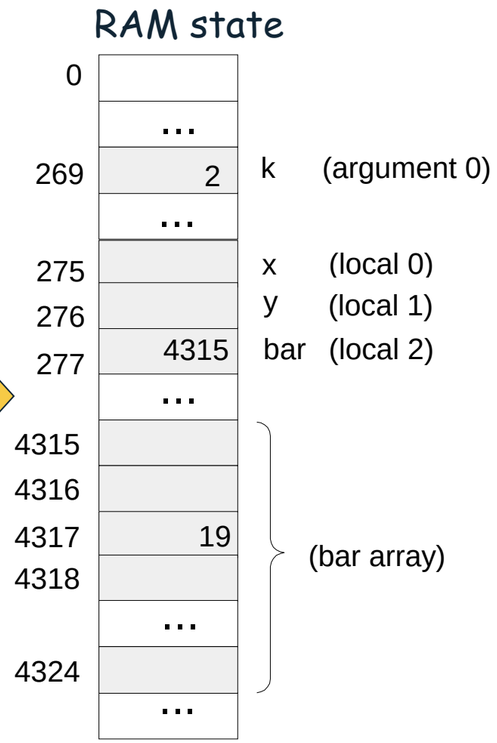
THE UNIVERSITY of ADELAIDE

Handling arrays: declaration / construction

Jack code

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

At run-time:



How to compile:

`let bar = Array.new(10) ; ?`

It is a normal function call:

push constant 10

call Array.new 1

pop local 2

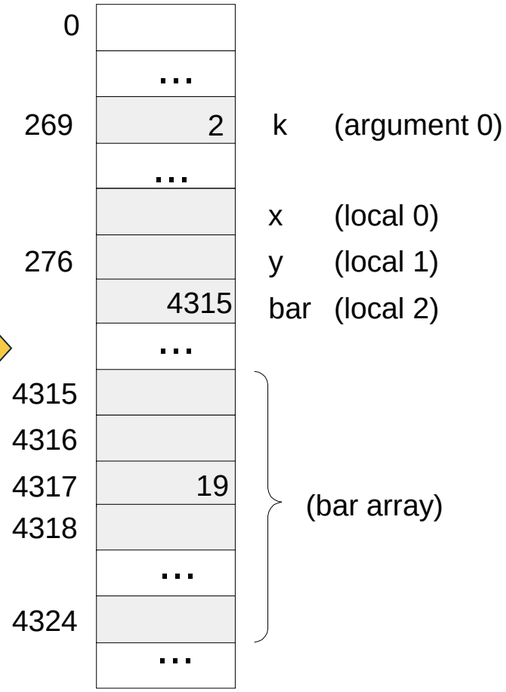
Handling arrays: accessing an array entry by its index

Jack code

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

At run-time:

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



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]
pop addr // addr points to bar[k]
push 19
pop *addr // Set bar[k] to 19
```

VM Code (almost actual)

```
// bar[k]=19, or that=bar+k; that[0]=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
```

THE UNIVERSITY of ADELAIDE

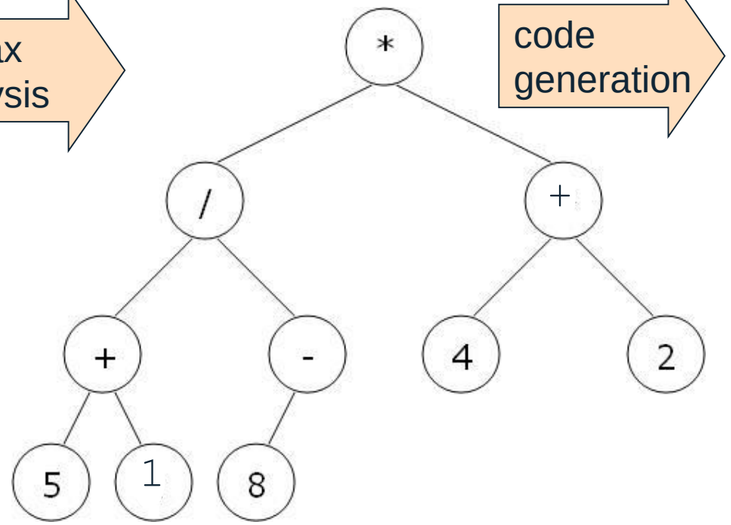
Handling expressions

High-level code

```
((5+1)/-8)*(4+2)
```

syntax analysis

parse tree



code generation

VM code

```
push constant 5
Push constant 1
add
push constant 8
neg
call Math.divide 2
push constant 4
push constant 2
add
call Math.multiply 2
```

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 (*exp*$_1$ *op* *exp*$_2$) then `codeWrite(exp`$_1$`); codeWrite(exp`$_2$`);` output `"op";`

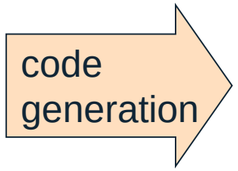
if *exp* is f (*exp*$_1$, ..., *exp*$_n$) then `codeWrite(exp1); ... codeWrite(exp1);` output `"call f";`

THE UNIVERSITY of ADELAIDE

Handling program flow

High-level code

```
if (cond)
{
   s1
} else
{
   s2
}
```

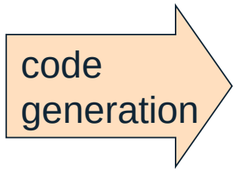


VM code

```
   VM code to compute and push cond
   if-goto IF_TRUE0
   goto IF_FALSE0
label IF_TRUE0
   VM code for executing s1
   goto IF_END0
label IF_FALSE0
   VM code for executing s2
label IF_END0
```

High-level code

```
while (cond)
{
   s
}
```



VM code

```
label WHILE_EXP0
   VM code to compute and push cond
   not
   if-goto WHILE_END0
   VM code for executing s
   goto WHILE_EXP0
label WHILE_END0
```

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
```

THE UNIVERSITY of ADELAIDE

Example Class Bob

```
class Bob

{
	static Array bobs ;

	static int how_many, too_many;
	field string name ;


	constructor Bob baby(string cool_name)

	{

		let name = cool_name ;

		return this ;

	}




	function void setup(int max_bobs)

	{

		let how_many = 0 ;

		let too_many = max_bobs ;

		let bobs = Array.new(max_bobs) ;

		return ;

	}
```

```
function Bob.baby 0
push constant 1
call Memory.alloc 1
pop pointer 0
push argument 0
pop this 0
push pointer 0
return

function Bob.setup 0
push constant 0
pop static 1
push argument 0
pop static 2
push argument 0
call Array.new 1
pop static 0
push constant 0
return
```

THE UNIVERSITY of ADELAIDE

Example Class Bob

```
method string name()
{
    return name ;
}
```

```
function Bob.name 0
push argument 0
pop pointer 0
push this 0
return
```

```
method void rename(string cooler_name)
{
    let name = cooler_name ;
    return ;
}
```

```
function Bob.rename 0
push argument 0
pop pointer 0
push argument 1
pop this 0
push constant 0
return
```

Example Class Bob

```
function void remember(Bob bob)
{
    if ( how_many < too_many )
    {
        let bobs[how_many] = bob.name() ;
        let how_many = how_many + 1 ;
    }
    return ;
}
}
```

```
function Bob.remember 0
push static 1
push static 2
lt
if-goto IF_TRUE0
goto IF_FALSE0
label IF_TRUE0
push static 1
push static 0
add
push argument 0
call Bob.name 1
pop temp 0
pop pointer 1
push temp 0
pop that 0
push static 1
push constant 1
add
pop static 1
label IF_FALSE0
push constant 0
return
```

THE UNIVERSITY of ADELAIDE


THE UNIVERSITY
of ADELAIDE

This Week

- Review Chapters 9 - 11 of the Text Book (if you haven’t already)
- Prac 6 Available Wednesday, due Week 11 (we’ll keep it small)