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.

Computer Systems

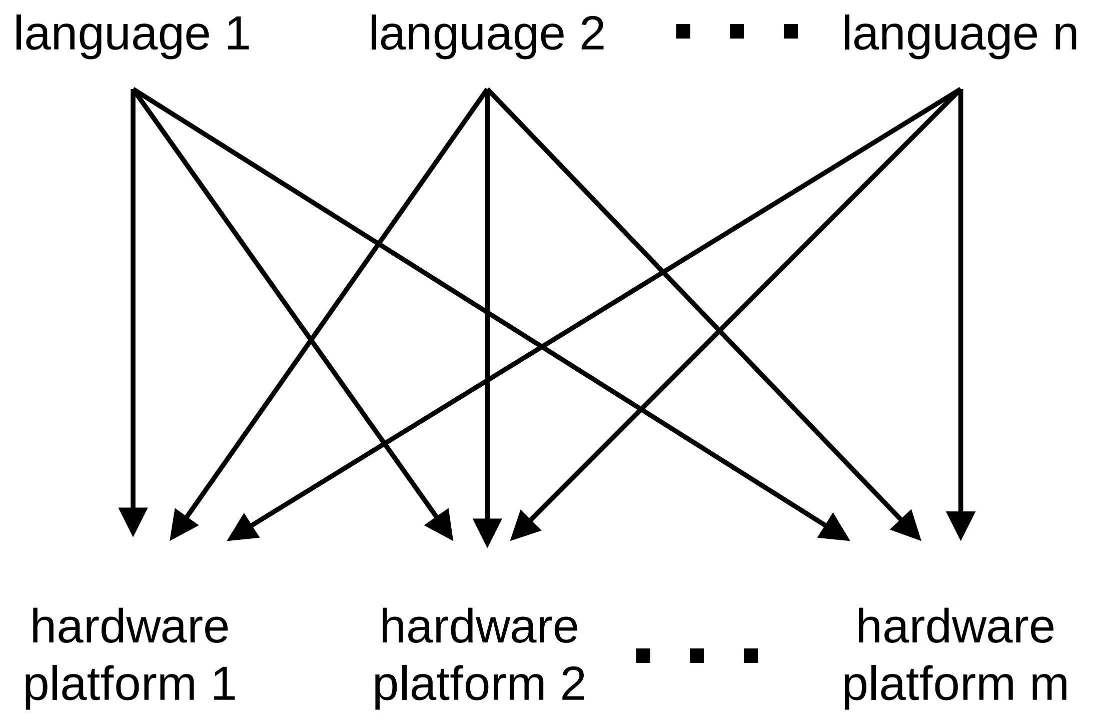
Lecture 08: High Level Language and language parsing

Review: The whole system

- **Programming Language**
  - Compiler
  - Chapters 10-11
- **Virtual Machine**
  - Translator
  - Chapters 7-8
- **Assembly Language**
  - Assembler
  - Chapters 6
- **Machine Language**
  - System Architecture
  - Chapters 4-5
- **Hardware Platform**
  - Gate Logic
  - Chapters 1-3
- **Chips and Logic Gates**
  - Electronics

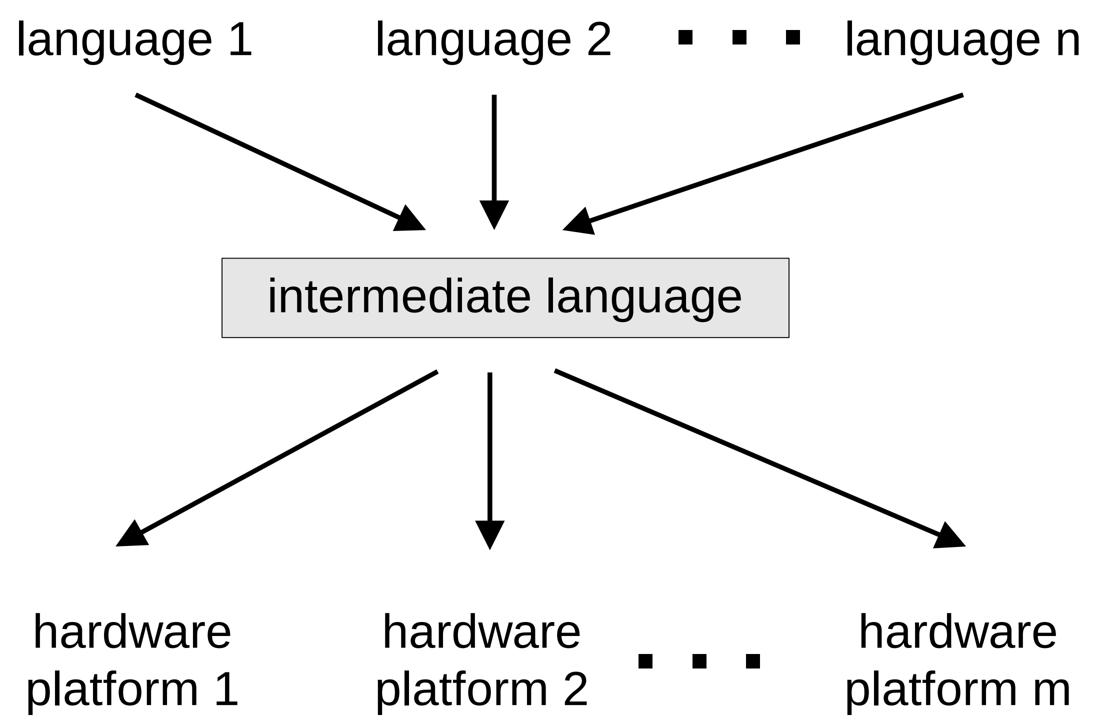
Review: Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:



requires $n + m$ translators

Two-tier compilation:

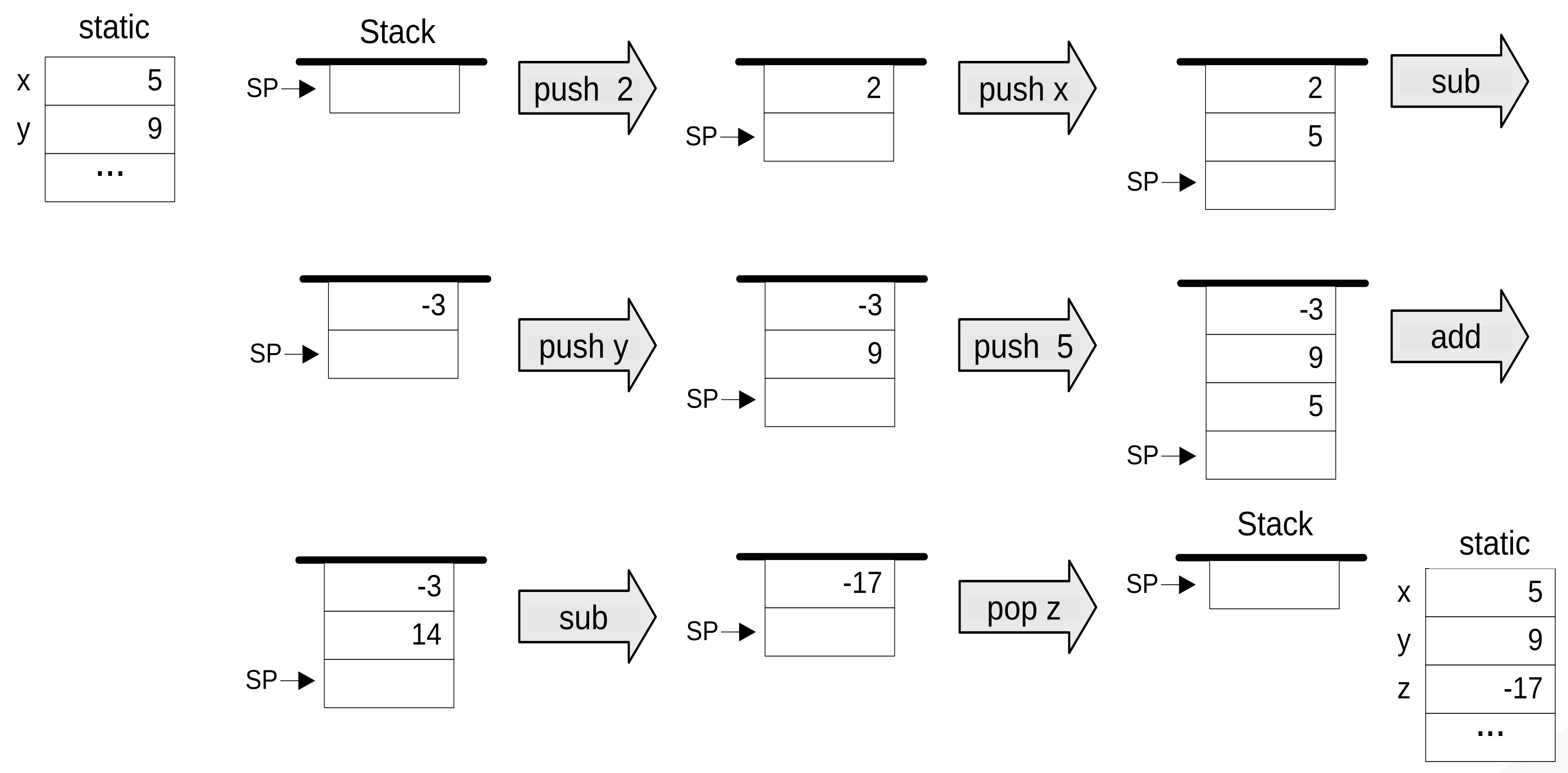
- ❑ First compilation stage: depends only on the details of the source language
- ❑ Second compilation stage: depends only on the details of the target language.

Review: Stack Evaluation of expressions

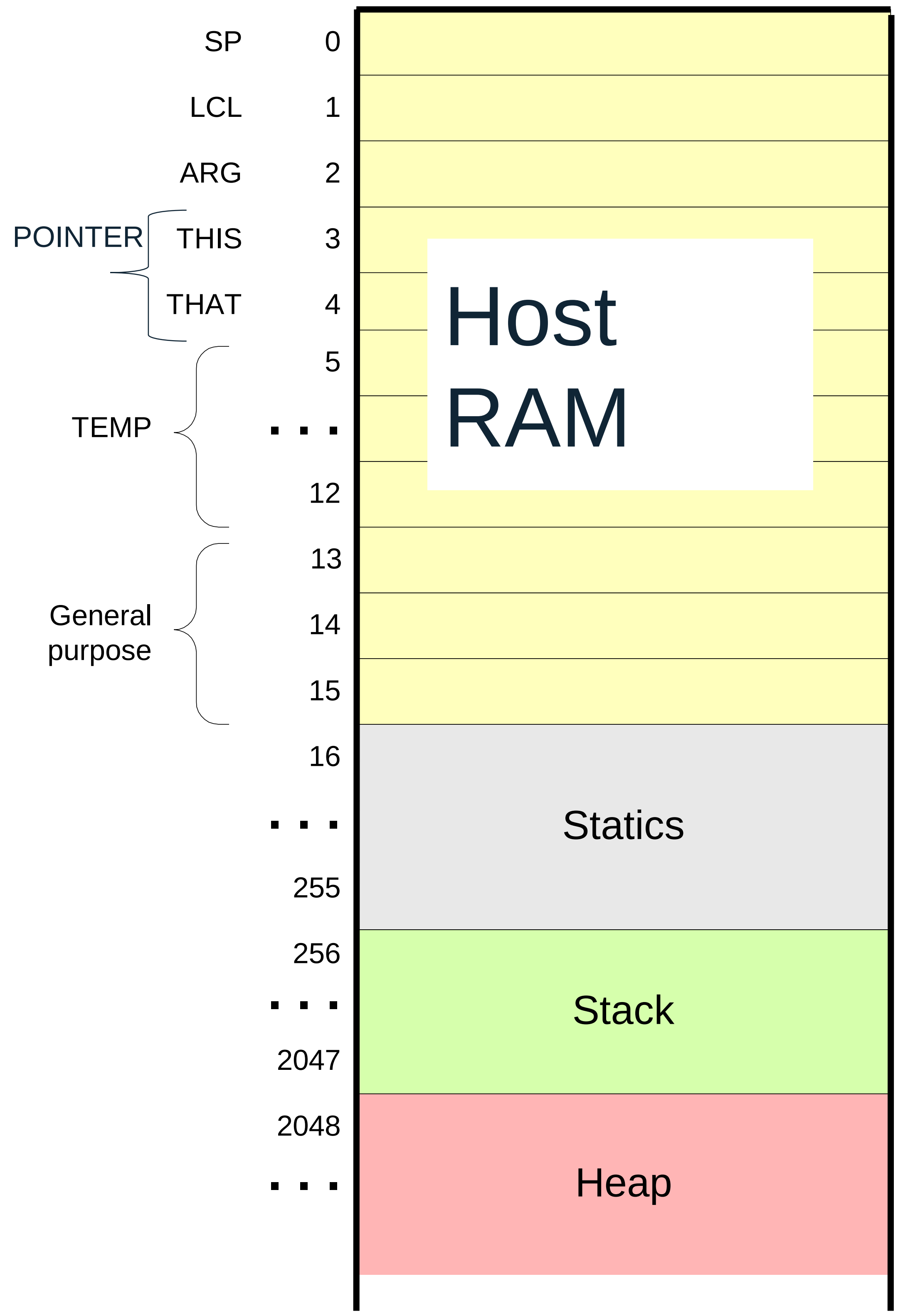
VM code (example)

```
// z=(2-x)-(y+5)
push 2
push x
sub
push y
push 5
add
sub
pop z
```

(suppose that
x refers to static 0,
y refers to static 1, and
z refers to static 2)



Review: VM memory segments & implementation



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)

local,argument,this,that: these method-level segments are mapped somewhere from address 256 onward, on the "stack" or the "heap". The base addresses of these segments are kept in RAM addresses LCL, ARG, THIS, and THAT. Access to the *i*-th entry of any of these segments is implemented by accessing RAM[segmentBase + *i*]

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

pointer: RAM[3..4] to change THIS and THAT.

THE UNIVERSITY of ADELAIDE

Review : Program flow control

In the VM language, the program flow abstraction is delivered using three commands:

```
label c        // label declaration

goto c         // unconditional jump to the
               // VM command following the label c

if-goto c      // pops the topmost stack element;
               // if it's not zero, jumps to the
               // VM command following the label c
```

VM code example:

```
function mult 1
  push constant 0
  pop local 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end
  push argument 0
  push constant 1
  sub
  pop argument 0
  push argument 1
  push local 0
  add
  pop local 0
  goto loop
label end
  push local 0
  return
```

How to translate these three abstractions into assembly?

- ❑ Simple: label declarations and goto directives can be effected directly by assembly commands
- ❑ More to the point: given any one of these three VM commands, the VM Translator must emit one or more assembly commands that effects the same semantics on the Hack platform
- ❑ How to do it? see project 8.

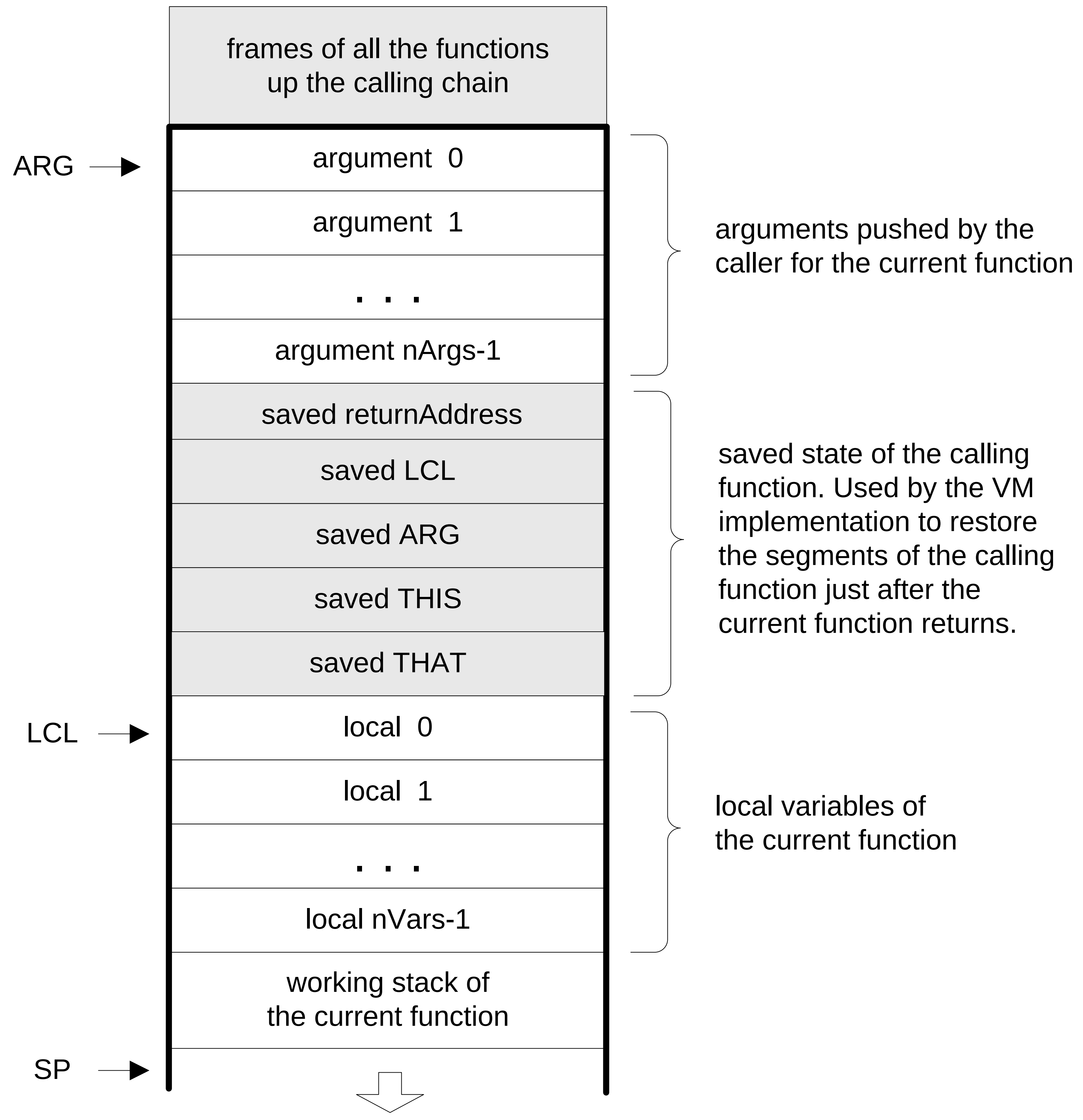
Review: Function commands

```
function g nVars    // here starts a function called g,
                   // which has nVars local variables

call g nArgs        // invoke function g for its effect;
                   // nArgs arguments have already been pushed onto the stack

return              // terminate execution and return control to the caller
```

Q: Why this particular syntax?

A: Because it simplifies the VM implementation (later).

Review: Function call/return implementation


frames of all the functions up the calling chain
ARG
argument 0
argument 1
. . .
argument nArgs-1
saved returnAddress
saved LCL
saved ARG
saved THIS
saved THAT
LCL
local 0
local 1
. . .
local nVars-1
working stack of the current function
SP
arguments pushed by the caller for the current function
saved state of the calling function. Used by the VM implementation to restore the segments of the calling function just after the current function returns.
local variables of the current function

Global stack:

the entire RAM area dedicated for holding the stack

Working stack:

The stack that the current function sees

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.

THE UNIVERSITY of ADELAIDE

The Jack programming language

What is Jack?

Jack is a simple, object-based, high-level language with a Java-like syntax

Although Jack is a real programming language, we don’t view it as an *end*

Rather, we use Jack as a *means* for teaching:

- How to build a compiler
- How the compiler and the language interface with the operating system
- How the topmost piece in the software hierarchy fits into the big pictur

Hello world

```
/** Hello World program. */
class Main
{
    function void main ()
    {
        // Prints some text using the standard library
        do Output.printString("Hello World");
        do Output.println();      // New line
        return;
    }
}
```

Some observations:

- ❑Java-like syntax
- ❑Typical comments format
- ❑Standard library
- ❑Language-specific peculiarities.

Procedural programming example

```
class Main
{
  /** Sums up 1 + 2 + 3 + ... + n */
  function int sum (int n)
  {
    var int sum, i;
    let sum = 0;
    let i = 1;
    while (~(i > n))
    {
      let sum = sum + i;
      let i = i + 1;
    }
    return sum;
  }

  function void main ()
  {
    var int n;
    let n = Keyboard.readInt("Enter n: ");
    do Output.printString("The result is: ");
    do Output.printInt(Main.sum(n));
    return;
  }
}
```

Jack program:

a collection of one or more classes

Jack class:

a collection of one or more subroutines

Execution order:

A Jack program starts by calling, `Main.main()`

Jack subroutine:

- `method`
- `constructor`
- `function` (static method)
- (the example on the left has functions only, as it is “object-less”)

Standard library: (API in the book)

a set of OS services (`methods` and `functions`) organized in 8 supplied classes: `Math, String, Array, Output, Keyboard, Screen, Memory, Sys`

THE UNIVERSITY of ADELAIDE

Object-oriented programming example

The `BankAccount` class

```
/** Represents a bank account.
    A bank account has an owner, an id, and a balance.
    The id values start at 0 and increment by 1 each
    time a new account is created. */

class BankAccount
{
    /** Constructs a new bank account with a 0 balance. */
    constructor BankAccount new(String owner) { ... }

    /** Deposits the given amount in this account. */
    method void deposit(int amount) { ... }

    /** Withdraws the given amount from this account. */
    method void withdraw(int amount) { ... }

    /** Prints the data of this account. */
    method void printInfo() { ... }

    /** Disposes this account. */
    method void dispose() { ... }
}
```

THE UNIVERSITY of ADELAIDE

Object-oriented programming example

```
/** Represents a bank account. */

class BankAccount
{
  // class-level variable
  static int newAcctId;

  // Private variables (aka fields / properties)
  field int id;
  field String owner;
  field int balance;

  /** Constructs a new bank account */
  constructor BankAccount new (String ownr)
  {
      let id = newAcctId;
      let newAcctId = newAcctId + 1;
      let owner = ownr;
      let balance = 0;
      return this;                2
  }

  // More BankAccount methods.

}
```

```
// Code in any other class:
var int x;
var BankAccount b;    1
3 let b = BankAccount.new("joe");
```

Explain `return this`

The constructor returns the RAM base address of the memory block that stores the data of the newly created `BankAccount` object

Explain `b = BankAccount.new("joe")`

Calls the constructor (which creates a new `BankAccount` object), then stores in variable `b`
a pointer to the object's base memory address

Behind the scene (after compilation):

```
push "joe"
call BankAccount.new 1
pop b
```

The calling code pushes an argument and calls the constructor; the constructor's code (not shown above) creates a new object, pushes its base address onto the stack, and returns;

The calling code then pops the base address into a variable that will now point to the new object.

Object-oriented programming example

```
class BankAccount
{
  static int nAccounts;
  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)
  /** Handles deposits */
  method void deposit (int amount)
  {
      let balance = balance + amount;
      return;
  }
  /** Handles withdrawls */
  method void withdraw (int amount)
  {
      if (~(amount > balance))
      {
          let balance = balance - amount;
      }
      return;
  }
  // More BankAccount methods.
}
```

```
...
var BankAccount b1, b2;
...
let b1 = BankAccount.new("joe");
let b2 = BankAccount.new("jane");
do b1.deposit(5000);
do b1.withdraw(1000);
...
```

Explain `do b1.deposit(5000)`

- In Jack, void methods are invoked using the keyword **do** (a compilation artifact)
- The object-oriented method invocation style `b1.deposit(5000)` is a fancy way to express the procedural semantics `deposit(b1,5000)`

Behind the scene **(following compilation):**

```
 // do b1.deposit(5000)

push b1

push 5000

call BankAccount.deposit 2
```

Object-oriented programming example

```
class BankAccount
{
  static int nAccounts;
  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Prints information about this account. */
  method void printInfo ()
  {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
  }

  /** Disposes this account. */
  method void dispose ()
  {
      do Memory.deAlloc(this);
      return;
  }
  // More BankAccount methods.
}
```

```
// Code in any other class:
...
var int x;
var BankAccount b;

let b = BankAccount.new("joe");
// Manipulates b...
do b.printInfo();
do b.dispose();
...
```

Explain `do Memory.deAlloc(this)`

This is a call to an OS function that knows how to recycle the memory block whose base-address is **this**.

Explain `do b.dispose()`

Jack has no garbage collection;
The programmer is responsible for explicitly recycling memory resources of objects that are no longer needed. If you don’t do so, you may run out of memory.

THE UNIVERSITY of ADELAIDE

Abstract data type example

The `Fraction` class API (method signatures)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */

class Fraction
{
    /** Constructs a fraction from the given data */
    constructor Fraction new(int numerator, int denominator) { ... }

    /** Reduces this fraction, e.g. changes 20/100 to 1/5. */
    method void reduce() { ... }

    /** Accessors
    method int getNumerator() { ... }
    method int getDenominator() { ... }

    /** Returns the sum of this fraction and the other one */
    method Fraction plus(Fraction other) { ... }

    /** Returns the product of this fraction and the other one */
    method Fraction product(Fraction other) { ... }

    /** Prints this fraction */
    method void print() { ... }

    /** Disposes this fraction */
    method void dispose() { ... }
}
```

THE UNIVERSITY *of* ADELAIDE

Abstract data type example

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */
class Fraction
{
    field int numerator, denominator;

    constructor Fraction new (int numeratr, int denominatr)
    {
        let numerator = numeratr;
        let denominator = denominatr;
        do reduce() ; // Reduces the new fraction
        return this ;
    }

    /** Reduces this fraction */
    method void reduce ()
    {
        // Code omitted
    }

    // A static method that computes the greatest common denominator of a and b.
    function int gcd (int a, int b)
    {
        // Code omitted
    }

    method int getNumerator () { return numerator; }

    method int getDenominator () { return denominator; }
```

```
// Code in any other class:
...
var Fraction a, b;
let a = Fraction.new(2,5);
let b = Fraction.new(70,210);
do b.print() ; // prints "1/3"
...
// (print method in next slide)
```

THE UNIVERSITY of ADELAIDE

Abstract data type example

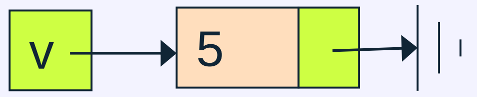
```
    /** Returns the sum of this fraction the other one */
    method Fraction plus (Fraction other)
    {
        var int sum;
        let sum = (numerator * other.getDenominator()) +
                  (other.getNumerator() * denominator()) ;
        return Fraction.new(sum,denominator * other.getDenominator()) ;
    }

    // Similar fraction arithmetic methods follow, code omitted.

    /** Prints this fraction */
    method void print ()
    {
        do Output.printInt(numerator) ;
        do Output.printString("/") ;
        do Output.printInt(denominator) ;
        return ;
    }

}
```

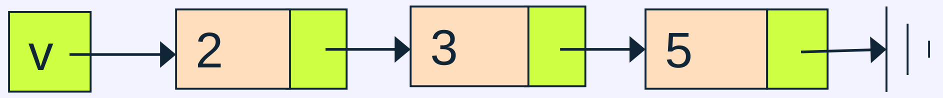
```
// Code in any other class:
var Fraction a, b, c ;
let a = Fraction.new(2,3) ;
let b = Fraction.new(1,5) ;
// computes c = a + b
let c = a.plus(b) ;
do c.print() ; // prints "13/15"
```

THE UNIVERSITY of ADELAIDE

Data structure example

```
/** Represents a sequence of int values, implemented as a linked list.
    The list consists of an int and either a pointer to a list or null. */
class List
{
    field int data;
    field List next;

    /* Creates a new list */
    constructor List new (int car, List cdr)
    {
        let data = car;
        let next = cdr;
        return this;
    }
    /* Disposes this list by recursively disposing its tail. */
    method void dispose()
    {
        if (~(next = null))
        {
            do next.dispose();
        }
        do Memory.deAlloc(this);
        return;
    }
    ...
}  // class List.
```





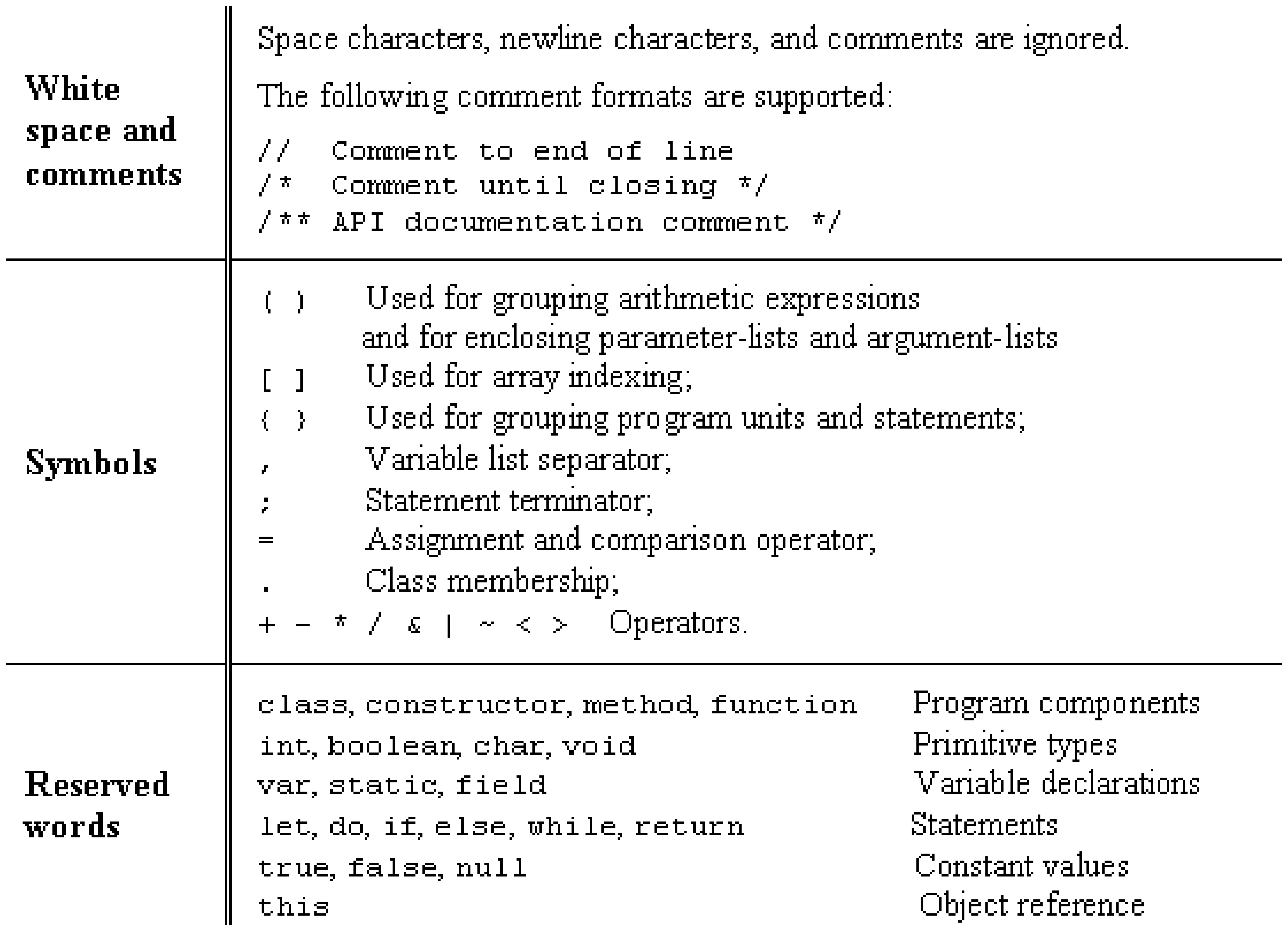
```
// Code in any other class:
...
// Creates a list holding the numbers 2,3, and 5:
var List v;
let v = List.new(5,null);
let v = List.new(2,List.new(3,v));
...
```

Jack language specification

- Syntax
- Data types
- Variable kinds
- Expressions
- Statements
- Subroutine calling
- Program structure
- Standard library

(for complete language specification, see the book).

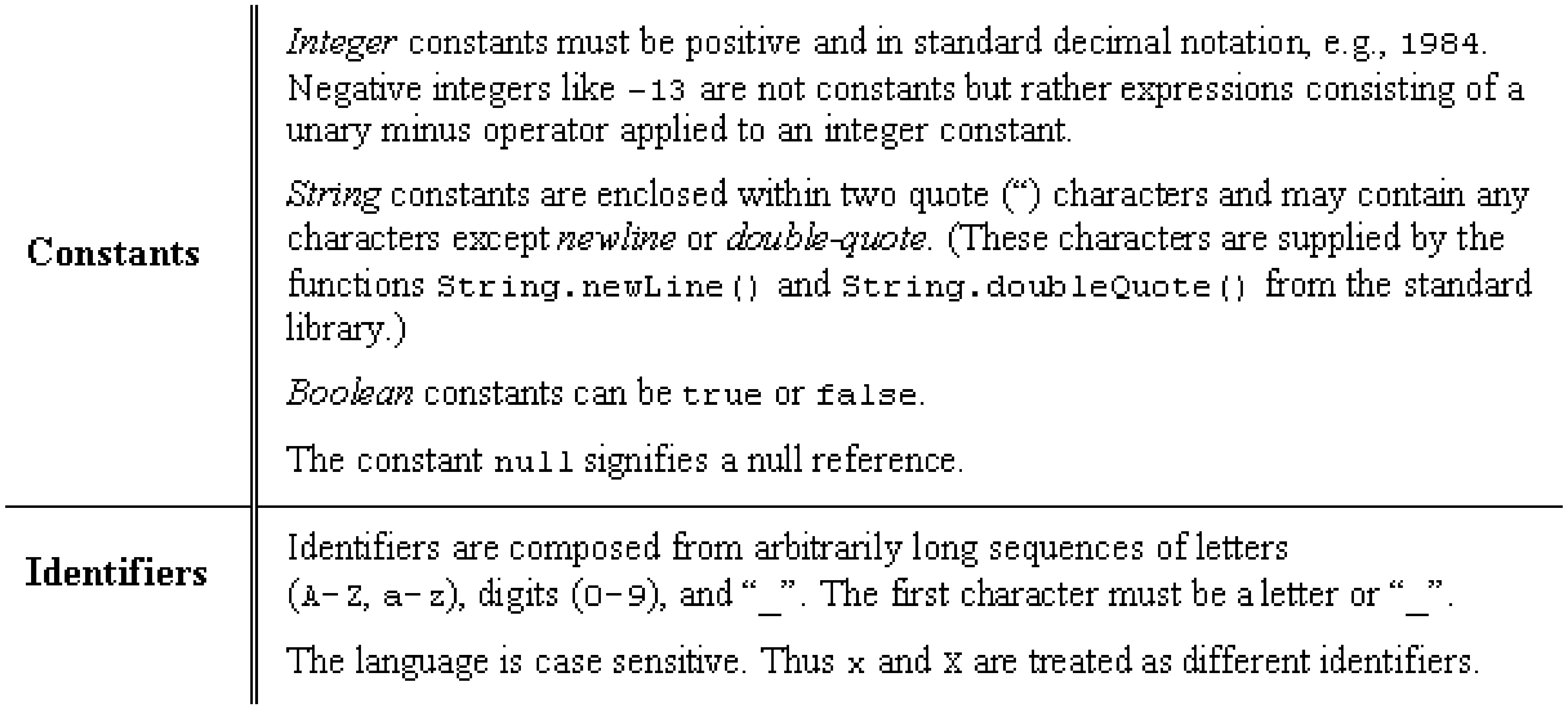
Jack syntax



White space and comments	Space characters, newline characters, and comments are ignored.
	The following comment formats are supported:
	`// Comment to end of line`
	`/* Comment until closing */`
	`/** API documentation comment */`

Symbols	`( )` Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists `[ ]` Used for array indexing; `{ }` Used for grouping program units and statements; `,` Variable list separator; `;` Statement terminator; `=` Assignment and comparison operator; `.` Class membership; `+ - * / & | ~ < >` Operators.
Reserved words	`class`, `constructor`, `method`, `function` — Program components `int`, `boolean`, `char`, `void` — Primitive types `var`, `static`, `field` — Variable declarations `let`, `do`, `if`, `else`, `while`, `return` — Statements `true`, `false`, `null` — Constant values `this` — Object reference

Jack syntax



Constants	*Integer* constants must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. *String* constants are enclosed within two quote (“) characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.) *Boolean* constants can be `true` or `false`. The constant `null` signifies a null reference.
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z`, `a-z`), digits (`0-9`), and “_”. The first character must be a letter or “_”. The language is case sensitive. Thus `x` and `X` are treated as different identifiers.

Jack data types

Primitive types (Part of the language; Realized by the compiler):

int	16-bit 2's complement (from -32768 to 32767)
boolean	-1 and 0, standing for true and false respectively
char	unicode character ('a', 'x', '+', '%', ...)

Abstract data types (Standard language extensions; Realized by the OS / standard library):

- String
- Array

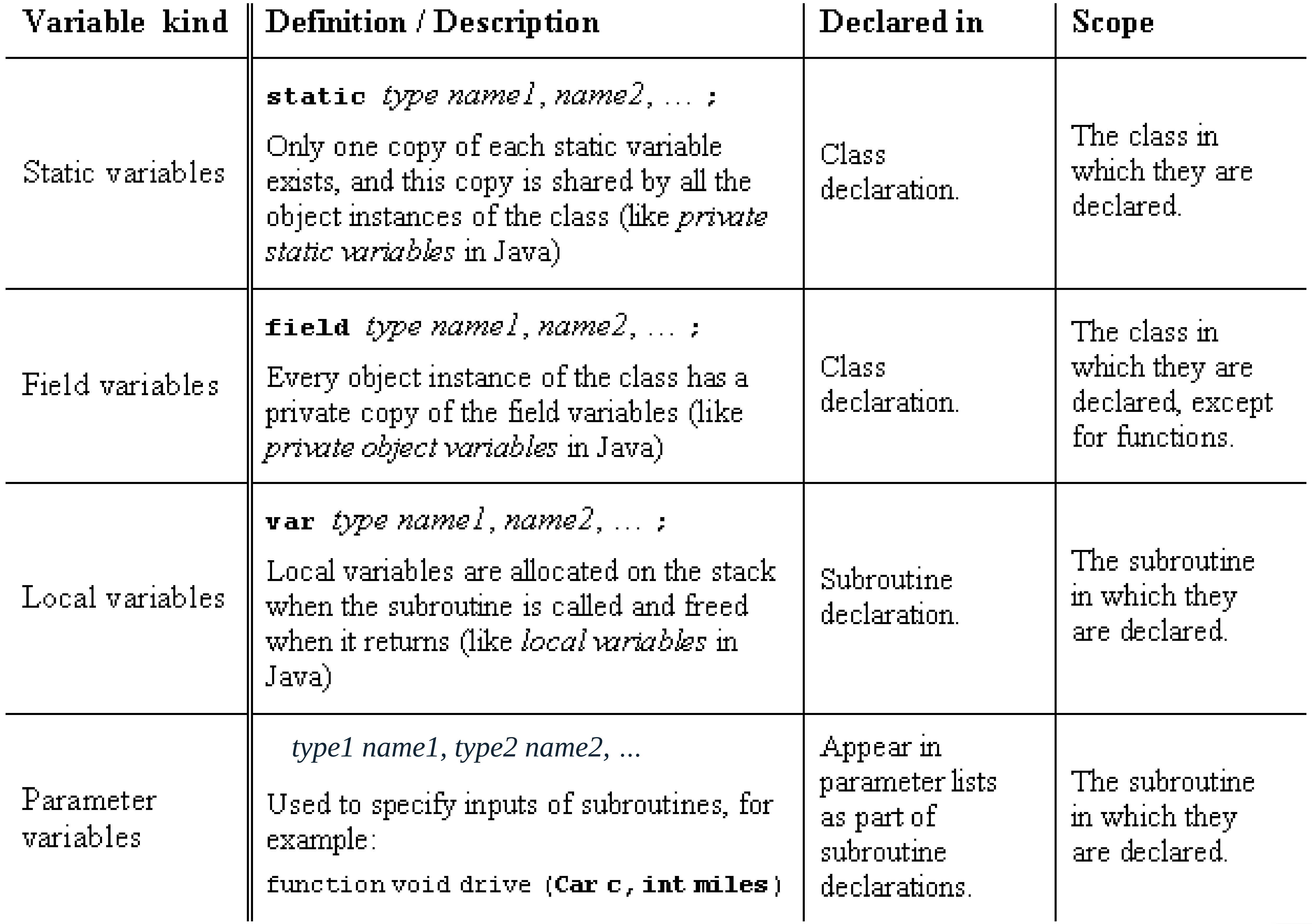
... (extensible)

Application-specific types (User-defined; Realized by user applications):

- BankAccount
- Fraction
- List
- Bat / Ball

. . . (as needed)

Jack variable kinds and scope



Variable kind	Definition / Description	Declared in	Scope
Static variables	`static` *type name1, name2*, ... `;` Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	`field` *type name1, name2*, ... `;` Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	`var` *type name1, name2*, ... `;` Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type1 name1, type2 name2, ...* Used to specify inputs of subroutines, for example: `function void drive (Car c, int miles)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Jack expressions

A Jack *expression* is any one of the following:

- A constant
- A variable name in scope (the variable may be static, field, local, or a parameter)
- The keyword `this`, denoting the current object
- An array element using the syntax `arrayName[expression]`,
  where *arrayNname* is a variable name of type Array in scope
- A subroutine call that returns a non-void type
- An *expression* prefixed by one of the unary operators `-` or `~` :

  `-expression` (arithmetic negation)

  `~expression` (logical negation)

- An expression of the form *expression op expression* where *op* is one of the following:

  `+ - * /` (integer arithmetic operators)

  `& |` (boolean and and or operators, bit-wise)

  `< > =` (comparison operators)

- `( expression )` (an expression within parentheses)

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.

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.

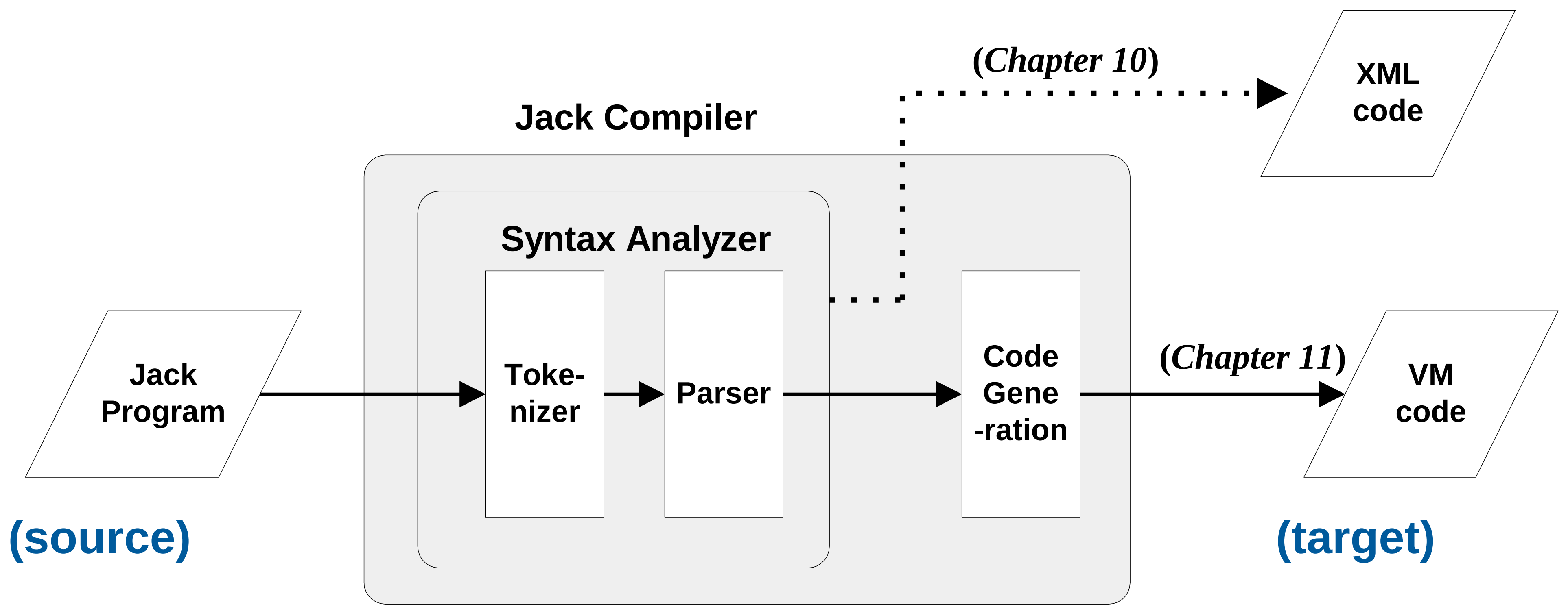
A Jack program:

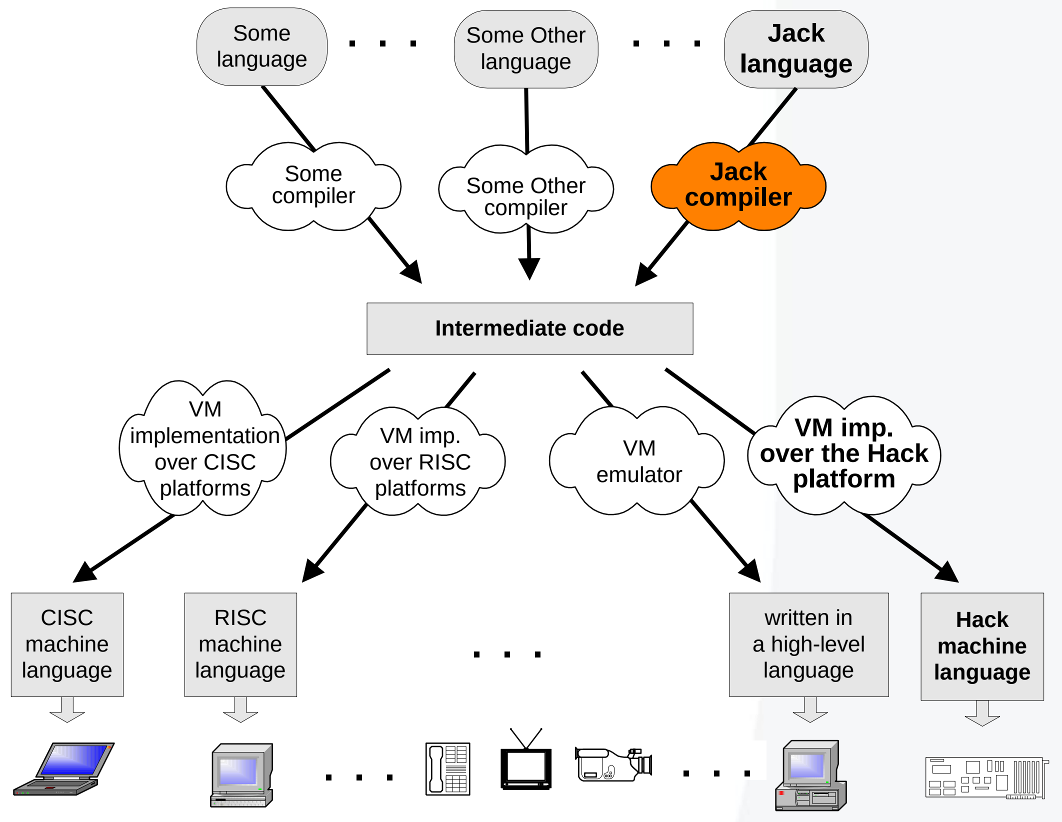
- Each class is written in a separate file (compilation unit)
- Jack program = collection of one or more classes, one of which must be named `Main`
- The `Main` class must contain at least one method, named `main()`

THE UNIVERSITY of ADELAIDE

Compiler Fundamentals

Compiler architecture (front end)





- <u>Syntax analysis:</u> 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

- <u>Code generation:</u> reconstructing the semantics using the syntax of the target code.

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*
<	&lt;
>	&gt;
'	&apos;
"	&quot;
&	&amp;

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>
```

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 :

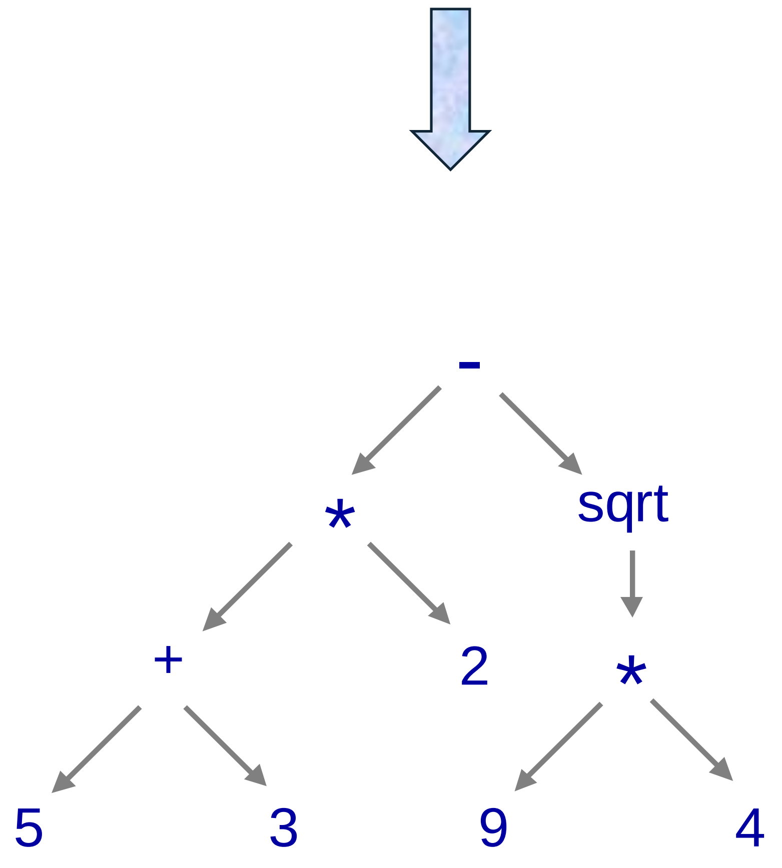
Natural language (English, …)

Programming language (Jack, …).

Parsing examples

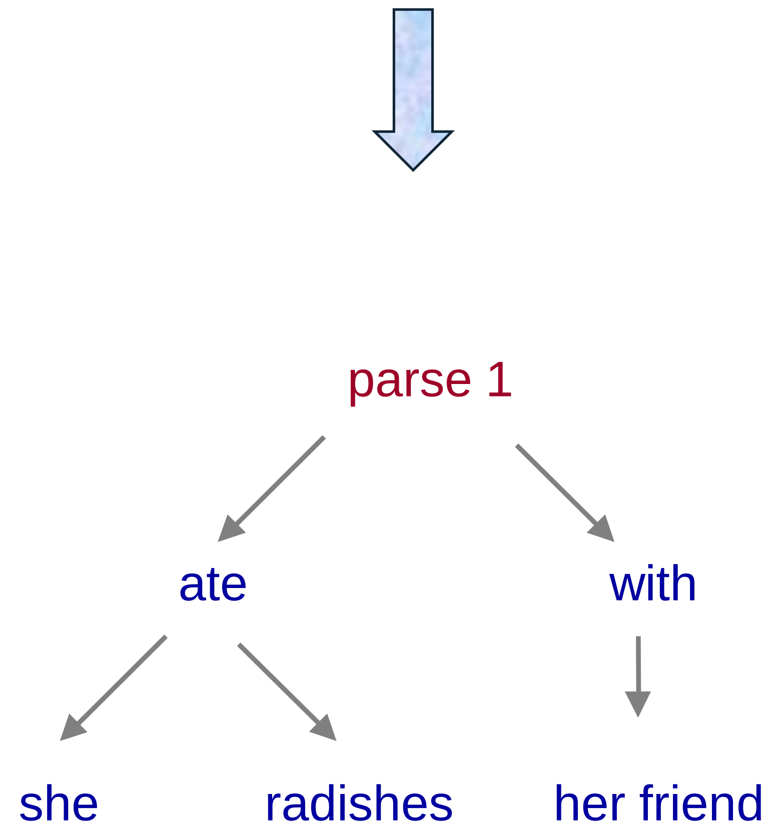
Jack

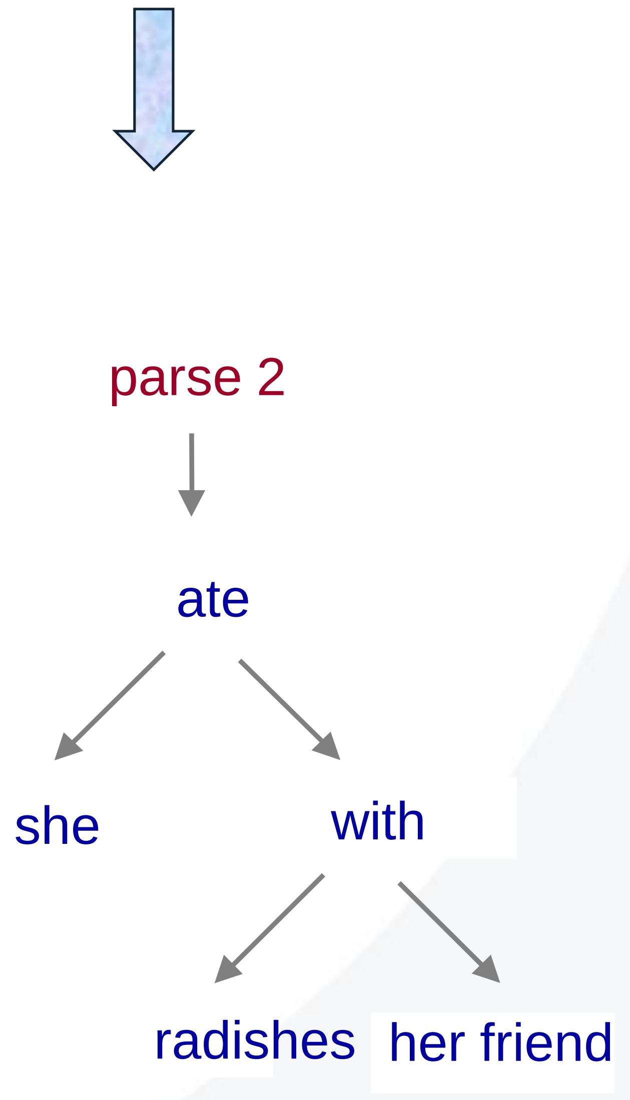
```
((5+3)*2) – sqrt(9*4)
```



English

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





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;
  }
}
```

Parse tree

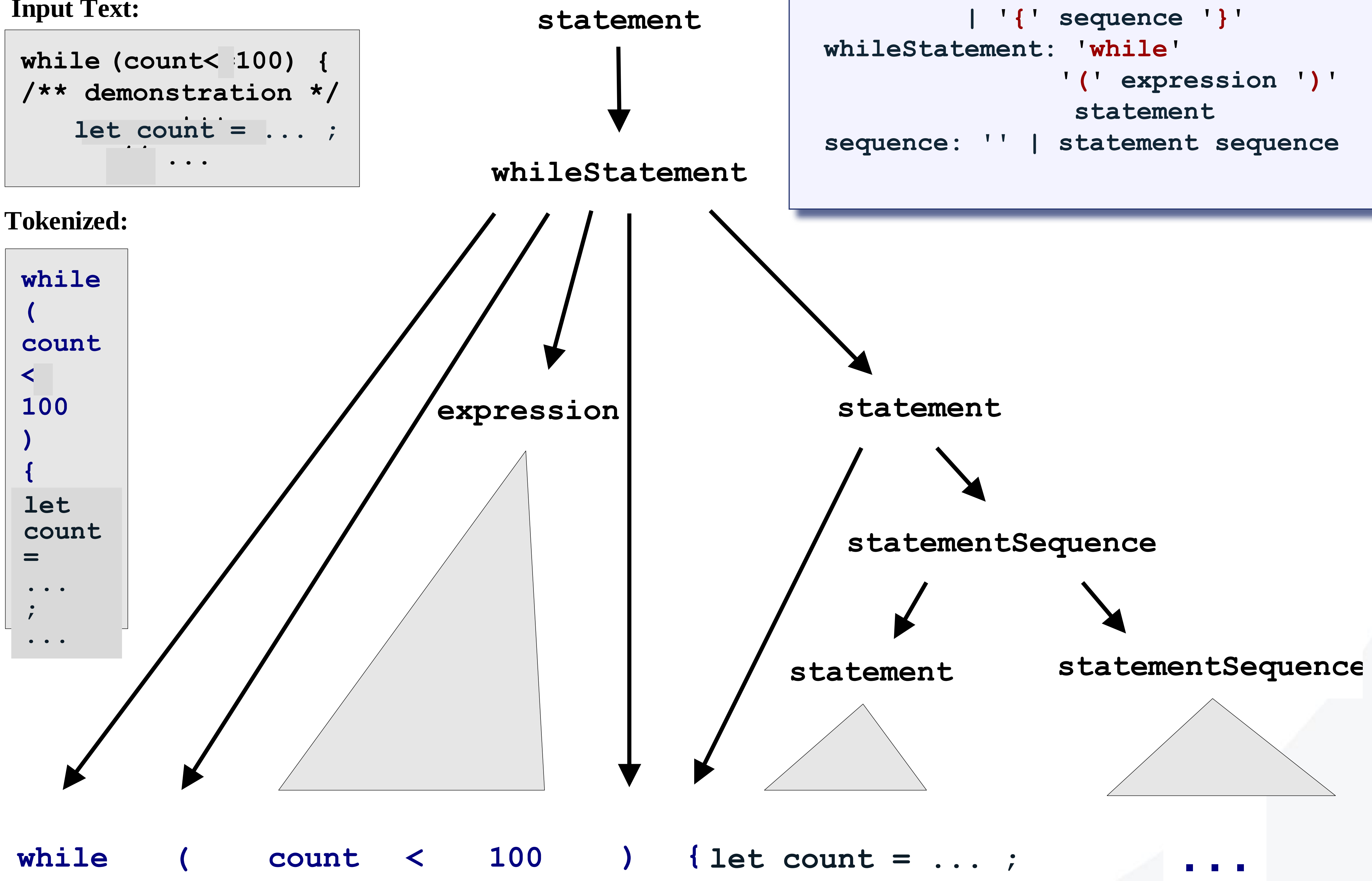
```
program:  statement;

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
=
...
;
...
```



This Week

- Review Chapters 9 & 10 of the Text Book (if you haven’t already)
- Prac Exam During workshops