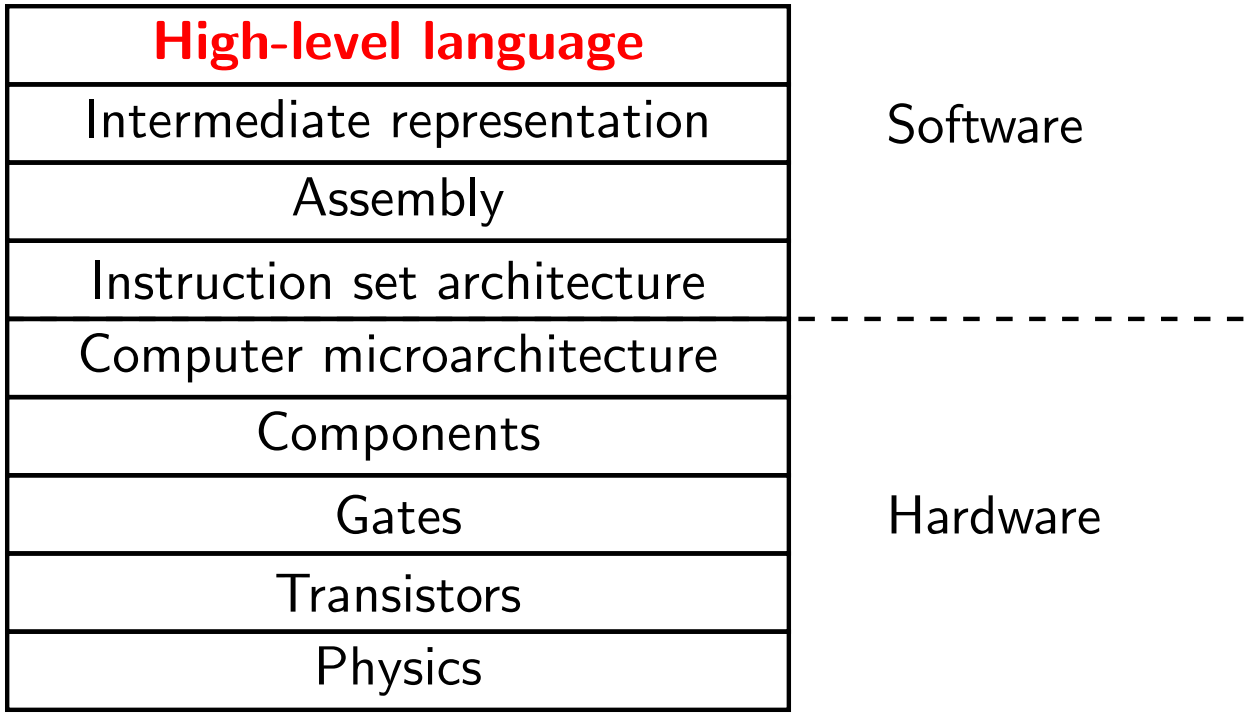
The Jack language

John Lapinskas, University of Bristol

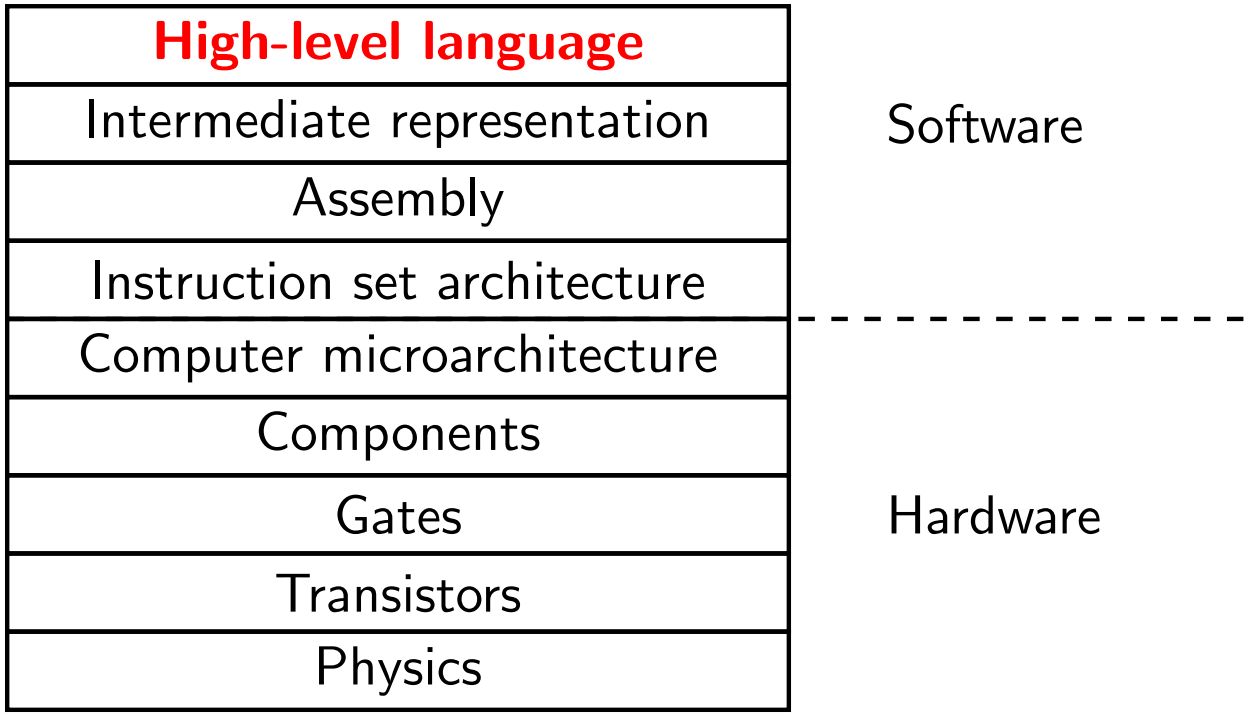
Our goals for this week



This week, we'll at last be dealing with a high-level language — **Jack**.

Our goal will be to compile Jack to Hack VM — together with our VM translator and our Hack assembler, this will give us a full compiler.

Our goals for this week



The details of Jack are non-examinable — in this unit we don't care about Jack itself, or about coding in Jack, but about *compiling* Jack.

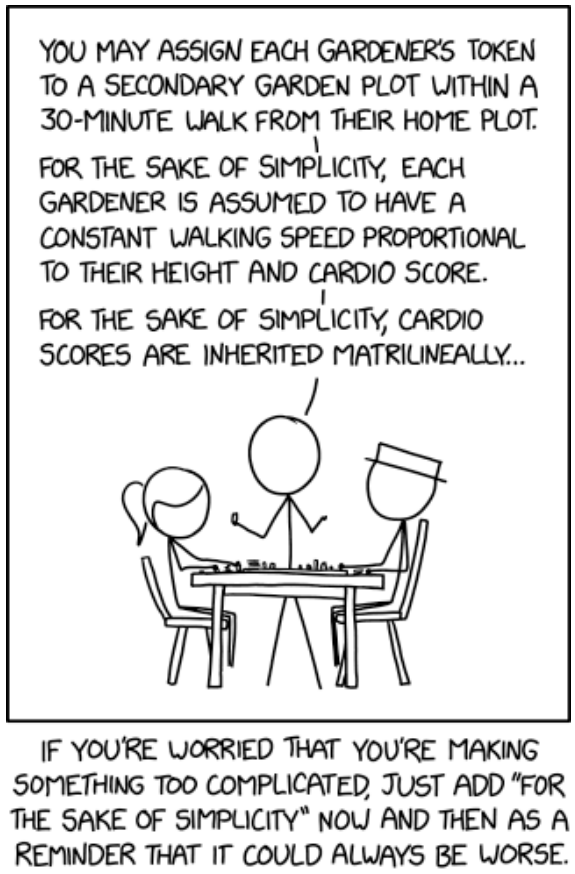
So take this video as a high-level overview, but don't bother memorising it, and use the grammar described next lecture as your main reference.

If in doubt, you can use the nand2tetris Jack compiler (downloadable from the unit page) to test the validity of any piece of Jack code.

The ethos of Jack

The two most common phrases in discussing Jack are:

- "Just like in C, but..."
- "For the sake of simplicity [in compiling]..."



Source: xkcd 2587 (here). Alt text: For the sake of simplicity, gardeners are assumed to move through Euclidean space — neglecting the distortion from general relativity — unless they are in the vicinity of a Schwarzschild Orchid.

Variables, statements and comments

Variables are declared like in C, but with a `var` keyword at the start, e.g. `var int x;` or `var char a, b, c;`.

- The built-in types are `char`, `int` and `boolean`.
- Libraries provide `Array` and `String` types.
- Unlike C, variables can't be initialised on declaration, so e.g. `var int x = 0;` is not valid.

For simplicity, all variables must be defined at the start of a function.

Variables, statements and comments

Variables are declared like in C, but with a `var` keyword at the start, e.g. `var int x;` or `var char a, b, c;`.

- The built-in types are `char`, `int` and `boolean`.
- Libraries provide `Array` and `String` types.
- Unlike C, variables can't be initialised on declaration, so e.g. `var int x = 0;` is not valid.

For simplicity, all variables must be defined at the start of a function.

Statements assigning values to variables work like C, but with a `let` keyword at the start, e.g. `let x = 5;` or `let c = "z";`. For simplicity, assignment operators like += and *= are not supported.

These extra keywords are purely to make parsing and compiling a little easier, and were a common feature of early languages like BASIC.

Variables, statements and comments

Variables are declared like in C, but with a `var` keyword at the start, e.g. `var int x;` or `var char a, b, c;`.

- The built-in types are `char`, `int` and `boolean`.
- Libraries provide `Array` and `String` types.
- Unlike C, variables can't be initialised on declaration, so e.g. `var int x = 0;` is not valid.

For simplicity, all variables must be defined at the start of a function.

Statements assigning values to variables work like C, but with a `let` keyword at the start, e.g. `let x = 5;` or `let c = "z";`. For simplicity, assignment operators like += and *= are not supported.

These extra keywords are purely to make parsing and compiling a little easier, and were a common feature of early languages like BASIC.

Like in C, comments are denoted with // or /*...*/, and newlines and whitespace are ignored (except for spaces separating tokens).

Functions

Functions are declared like in C, but with a `function` keyword at the start. For example,

```
function int Main.max (int x, int y) {
    // Function body here
}
```

defines `Main.max` to have two `int` arguments `x` and `y` and return an `int`.

Like in C, a function which returns no value is defined with `void`, e.g. `function void Main.print (string toPrint)`.

Functions

Functions are declared like in C, but with a `function` keyword at the start. For example,

```
function int Main.max (int x, int y) {
    // Function body here
}
```

defines `Main.max` to have two `int` arguments `x` and `y` and return an `int`.

Like in C, a function which returns no value is defined with `void`, e.g. `function void Main.print (string toPrint)`.

Like in C, functions return values with the syntax e.g. `return 42;` or `return;`. Unlike C, every function must end with a `return`.

Functions

Functions are declared like in C, but with a `function` keyword at the start. For example,

```
function int Main.max (int x, int y) {
    // Function body here
}
```

defines `Main.max` to have two `int` arguments `x` and `y` and return an `int`.

Like in C, a function which returns no value is defined with `void`, e.g. `function void Main.print (string toPrint)`.

Like in C, functions return values with the syntax e.g. `return 42;` or `return;`. Unlike C, every function must end with a `return`.

Function calls are also like in C, with one exception: a line which only calls a function and throws the return value away must start with the `do` keyword. For example, `Main.print("Hello, world!");` is not valid, but `do Main.print("Hello, world!");` is valid.

Expressions

In all programming languages, an **expression** is part of a statement that returns a value.

A literal (like 5 or `true` or `"Hello, world!"`) is an expression — the value is just the literal itself. Variable names are expressions, too.

In C, anywhere you could write a literal, you can also write a more complicated expression, such as:

- `(z+x)/7`.
- `fibonacci(x+1)`.
- `max(fibonacci(*x), fibonacci(fibonacci(y++)))`.

The same is true in Jack. `do`, `let` and `return` statements can all be followed by expressions, and function calls can use expressions as arguments. There's no limit on how long or complex an expression can be.

Supported expressions

The following common C expression components are supported:

- Arithmetic: `+`, `-`, `*`, `/`.
- Logic: `&`, `|`, and `~` (which means NOT). These act as both bitwise and logic operators.
- Comparison: `=` (not `==`), `>` and `<`.
- Literals: Integers, strings, `true`, `false`, and `null` (which is the same as `false`).
- Array subscripts `[]` (see later).
- Variables.
- Function calls (which can have expressions as arguments).
- Parentheses `()`.

For simplicity, the following common C expression components are not supported:

- The modulo operation: `%`.
- Bit shifts: `<<` and `>>`.
- Referencing/dereferencing: Unary `*` and `&`.
- Increment/decrement: `++` and `--`.
- Comparison: `!=`, `<=` and `>=`.
- Assignment as part of an expression returning `true` on success.
- The ternary operator `?`.
- **In Jack, there is no operator precedence without `()`s. For example, `1+2*3` may evaluate to either `7` or `9`.**

Control flow: Loops and conditionals

Jack supports if-else statements with expressions just like C, except that `else if` is not supported. For example:

```
if (x > 5 & ~(y + f(x) = 7)) {
    // Code here
} else if (z = 2) {
    // Code here
}
```

is not valid Jack, due to the `else if`, but...

Control flow: Loops and conditionals

Jack supports if-else statements with expressions just like C, except that `else if` is not supported. For example:

```
if (x > 5 & ~(y + f(x) = 7)) {
    // Code here
} else { if (z = 2) {
    // Code here
}}
```

is valid Jack.

Jack also supports C-like `while` loops:

```
while (x < 5 | Main.fibonacci(x) = 13) {
   // Code here
}
```

For simplicity, it doesn't support `do...while` or `for` loops.

Variable types

There is no casting in Jack. Implicit type conversions are supported between `int`, `char` and `boolean` variables, treating:

- `char` variables as `int`s according to the Hack character set (Appendix C of Nisan and Schocken, also see week 5);
- `boolean` values of `true` as `-1` and `false` as `0`.

For example,

```
var int x; var boolean y; var char z;
let x = -1; let y = x; let z = x + 71;
```

is valid Hack and will set `y` to `true` and `z` to `"F"`.

We will get this "for free" by representing `char` and `boolean` variables as `int`s this way in memory, then completely ignoring their type information.

Handling types properly can get *extremely* complicated for reasons that will become clear in OOP next TB.

Variable types

There is no casting in Jack. Implicit type conversions are supported between `int`, `char` and `boolean` variables, treating:

- `char` variables as `int`s according to the Hack character set (Appendix C of Nisan and Schocken, also see week 5);
- `boolean` values of `true` as `-1` and `false` as `0`.

For example,

```
var int x; var boolean y; var char z;
let x = -1; let y = x; let z = x + 71;
```

is valid Hack and will set `y` to `true` and `z` to `"F"`.

We will get this "for free" by representing `char` and `boolean` variables as `int`s this way in memory, then completely ignoring their type information.

Handling types properly can get *extremely* complicated for reasons that will become clear in OOP next TB. See e.g. this paper which shows that accurate type checking in Java is literally impossible unless you're willing to accept your type checker failing to terminate on some inputs!

The one new-ish idea: Classes

In place of C's structs, Jack has something *slightly* harder: **classes**.

Classes are a core feature of object-oriented programming (**OOP**) that you'll see next term. But thankfully, Jack lacks all the (many!) features of classes that make them any more complicated than structs.

The one new-ish idea: Classes

In place of C's structs, Jack has something *slightly* harder: **classes**.

Classes are a core feature of object-oriented programming (**OOP**) that you'll see next term. But thankfully, Jack lacks all the (many!) features of classes that make them any more complicated than structs.

The only real difference between Jack's classes and C's structs is:

- In C, the functions associated with a struct (e.g. `read_token`, `malloc_token` etc.) are separate from the `struct` statement that defines it. They could even be in different files!
- In Jack, the functions associated with a class are *contained in* the `class` statement that defines it.

This leads to some natural syntax (which is common in OOP more generally) which is nice for the Jack programmer, but which we have to deal with in compiling.

Class variables: `fields` and `statics`

A **`field`** variable in Jack is the exact equivalent of a normal struct variable in C.

C code

```
struct Foo {
    int x;
    int y;
}
```

Corresponding Jack code

```
class Foo {
    field int x;
    field int y;
}
```

Unlike C, structs can also have **`static`** variables declared with syntax `static int x`. A `static` variable is shared between all members of the class.

Class variables can be declared with the normal syntax: `var Foo myFoo;`.
All class variables must be defined at the start of the class.

For simplicity, Jack doesn't support the C syntax `myFoo.x` to access or update the value of the `x` field of the `Foo`-type variable `myFoo`. Instead, we use methods.

Methods

A **method** is a special sort of function which "belongs" to a class. It can only be called from a specific instance of that class.

From inside a method definition, the fields of the class variable act as local variables. (This replaces C-style `myFoo.x` syntax.) The class variable itself can also be accessed via the `this` keyword, and `methodCall()` is interpreted as `this.methodCall()`.

C code

```
void twiddleFoo(struct Foo abc) {
    abc.x = abc.x + abc.y;
    abc.y = abc.y - 1;
}
twiddleFoo(myFoo);
twiddleFoo(myOtherFoo);
```

Corresponding Jack code

```
class Foo() {
    // [Rest of definition omitted]
    method void twiddle() {
        let x = x + y;
        let y = y - 1;
    }
}
myFoo.twiddle();
myOtherFoo.twiddle();
```

In the code on the right, the value of `this` will be `myFoo` in the first call to `twiddle` and `myOtherFoo` in the second call, just like `abc` in the code on the left.

Constructors

Classes can also contain `constructors`, a special type of function intended to create a new class variable.

At runtime, a `constructor` function will automatically create a new class variable on the heap at the start of the function call.

The `this` keyword is set to this new class variable, and the constructor should end with a `return this;` statement. Like with methods, fields of `this` can be accessed as pre-defined local variables.

C code

```
struct Foo* newFoo(int a) {
    Foo* new = malloc(sizeof(foo));
    new->x = 5;
    new->y = a;
    return new;
}
myFoo = newFoo(42);
```

Corresponding Jack code

```
class Foo() {
// [Rest of definition omitted]
    constructor Foo newFoo(int a) {
        let x = 5;
        let y = a;
        return this;
    }
}
myFoo = Foo.newFoo(42);
```

Constructors

Classes can also contain `constructors`, a special type of function intended to create a new class variable.

At runtime, a `constructor` function will automatically create a new class variable on the heap at the start of the function call.

The `this` keyword is set to this new class variable, and the constructor should end with a `return this;` statement. Like with methods, fields of `this` can be accessed as pre-defined local variables.

Classes can contain `functions` as well, which work as described earlier.

When called, both `constructors` and `functions` must start with the name of the class. So e.g. `let myFoo = Foo.makeFoo();` is valid Jack code, but `let myFoo = makeFoo();` is not.

Finally, note that both code examples above leak memory in the same way. As classes in Jack are allocated on the heap, they must be freed from memory just like `malloc`'d pointers to structs in C. Normally this would be done via a dedicated method (often called `dispose`), which would in turn call `Memory.deAlloc(this)` after any other cleanup.

Classes vs structs: The summary

C code with structs

```
struct Foo {
    int x;
    int y;
}

struct Foo* newFoo(int a) {
    Foo* new = malloc(sizeof(foo));
    new->x = 5;
    new->y = a;
    return new;
}

void twiddleFoo(struct Foo* abc) {
    abc->x = abc->x + abc->y;
    abc->y = abc->y - 1;
    free(abc);
}

// [A bunch of code omitted]
struct Foo* my_foo = newFoo(42);
twiddleFoo(my_foo);
```

Corresponding Jack code with classes

```
class Foo {
    field int x;
    field int y;

    constructor Foo newFoo(int a) {
        let x = 5;
        let y = a;
        return this;
    }

    method twiddleFoo() {
        let x = x + y;
        let y = y - 1;
        Memory.deAlloc(this);
    }
}

// [A bunch of code omitted]
var Foo my_foo = Foo.newFoo(42);
my_foo.twiddleFoo();
```

[ ] syntax for arrays and strings

In Jack, all class-type variables are stored on the heap — only `ints`, `chars` and `booleans` are stored on the stack.

Effectively, all of these variables are stored as pointers — treating a variable of type `Foo` as an `int` will reveal it to be the heap address at which the fields of `Foo` are stored. (See video 3...)

The expression `my_object[i]` means: go to the address of `my_object`, add `i` to it, and return the result.

The implementation of the `Array` and `String` classes then ensures that the address of `my_object` will be the first entry of the array/string, so the syntax does what you would expect.

(This is all very much a Jack idea — C stores structs on the stack!)

This also gives us a backdoor into memory. For example, the code `var int x; let x = 16384; let x[0] = 0;` will set RAM[0x4000] to zero.

Compiling multiple files

In Jack, each file contains one class declaration. The file `Foo.jack` should contain the class `Foo` (along with all its fields, methods, and so on).

Everything must be inside a class. Those parts of your program which (in C) wouldn't be associated with a struct should be placed in the `Main` class.

Every Jack program will start by calling the function `Main.main()`, making it the analogue of the `main` function in C.

Every Jack file is compiled to Hack VM *separately*. Afterwards, multiple `.vm` files can then be combined into a single Hack assembly file by the VM translator, and from there compiled to Hack machine code by the assembler.

However, Jack files can still use classes defined in other Jack files. The compiler simply assumes any such classes will be available, and that e.g. any methods used will be present. This sidesteps the need for e.g. C's makefiles, at the cost of worse error handling.

Putting it all together: An example program

```
// Inputs some numbers and computes their average
class Main {
    function void main() {
        var Array a;
        var int length;
        var int i, sum;

        let length = Keyboard.readInt("How many numbers?  ");
        let a = Array.new(length); // constructs the array

        let i = 0;
        while (i < length) {
            let a[i] = Keyboard.readInt("Enter a number:  ");
            let sum = sum + a[i];
            let i = i + 1;
        }

        do Output.printString("The average is ");
        do Output.printInt(sum / length);
        return;
    }
}
```

Source: Nisan and Schocken example program Average.

Here the `Keyboard.readInt`, `Array.new`, `Output.printString` and `Output.printInt` functions are all part of the standard libraries (see Nisan and Schocken appendix 6).

We won't need to care about these for writing a compiler, though!