CS Uncovered

Our feature article:

The Art of COMPILER CONSTRUCTION

and also

Advanced Data Structures | Puzzles | Insight into smart cities | History of Linux | and more..

Beths COMPUTING SOCIETY

Issue #002

We print **newsletters**, host **competitions** and provide **mentoring**, for students to engage with the rich field of computer science, beginner or advanced. Join us to continue running these initiatives, and make any ideas you have a reality.

We'd love to have you!

Monday 12:50 -14:00 in IT2.

For more info, email Vladimir Filip at 17H-filipv@beths.bexley.sch.uk

Contents

SMART CITIES and how they use technology

Fjoralba Nasto

'Smart city' is a term used for **a city which uses information and communication technology (ICT)** to **improve operational efficiency, share information** with the public and **provide a better quality of government service** and citizen welfare. This is done by effectively utilising a variety of software, user interfaces and communication networks such as the Internet of Things (IoT), to expand their reach and connectivity.

How Smart Cities Work

Smart cities follow four steps to improve the quality of life and enable economic growth through a network of connected IoT devices and other technologies. These steps are as follows:

1. *Collection*: smart sensors gather real-time data
2. *Analysis*: The data is analysed to gain insights into the operation of city services and operations
3. *Communication:* The results of the data analysis are communicated to decision makers
4. *Action:* Action is taken to improve operations, manage assets and improve the quality of city life for the residents

The Internet Of Things and Smart Cities

It wasn't long ago when the only way to access the internet was through a desktop computer, but now pretty much anything can connect to the internet – including from your mobile phone, car, fridge and on-street sensors. This is the Internet of things (IoT).

The internet of Things is a network of connected devices that communicate and exchange data. This data collected from these devices is stored in the cloud or on servers to allow for improvements to be made to both public and private sector in the city.

However, despite the fact that IoT is one of the key factors of a smart city, other technologies might include:

– Application Programming Interfaces (APIs)
– Artificial Intelligence (AI)
– Cloud Computing Services
– Dashboards
– Machine Learning
– Machine-to-Machine Communications
– Mesh Networks

How Artificial Intelligence is used in Smart Cities

Artificial intelligence is defined as the intelligence of a computer to imitate human capabilities and the way that the human brain works in applications of any kind. These include expert systems, natural language processing, speech recognition and machine vision.

According to an article released by AI Magazine, there are 10 ways that AI can be used in smart cities:

1) Environment

Smart cities can use artificial intelligence to see their effect on the local environment, global warming, as well as the pollution level. Using AI and machine learning within pollution control and energy consumption, allows authorities and cities to make well informed decisions that are best for the environment. Smart cities also use AI to detect CO2 which can then lead to decisions around transportation.

2) Energy Tracking

Artificial intelligence can be used within smart cities to analyse and track business and citizen energy usage, with this data it can then be decided where to implicate renewable sources of energy. This can also show cities where energy is being wasted and how it can be saved.

3) Traffic Management

AI technology is being implemented within the transportation industry to reduce traffic and accidents. A traffic management technology known as CIRCLES has the ability to predict and reduce traffic, using deep learning algorithms, this can then reduce the pollution created by traffic too. AI can also be used throughout traffic camera systems to detect road crimes in real time, making them easier to deal with.

4) Waste Management

Smart cities are beginning to use artificial intelligence within their waste management, this type of technology allows cities to track recycling, and identify what can be recycled in the area. Some cities in Sydney take this a step further and use AI powered robots to sort rubbish, as well as clean areas such as lakes and rivers.

5) Public transportation

Public transport has been innovated with the use of AI to be used within smart cities. This technology allows public transport users to receive and access live up dates and tracking, which improves timing and customer satisfaction. Automated busses are also planned to be used within cities, these can reduce emissions, improve routes, and increase the frequency.

6) Parking Systems

Using license plate recognition technology, car parks are able to detect cars that have outstayed hours, this can also enforce payments and tickets. When AI systems are integrated throughout car parking areas, space availability is able to be presented to awaiting users. Some more advanced technology has the ability to recommend spaces depending on the car.

7) Controlling Pollution

Scientists have developed technology which uses AI and machine learning to analyse the current pollutants and predict the pollution levels for the next 2 hours. This type of technology allows authorities to make decisions in advance to reduce their effect on the environment.

8) Predicting future needs

Smart cities that are innovating with AI technology are able to better predict future needs of the people. Using energy tracking technology allows cities to know when new energy sources are needed or when more sustainable methods can be implemented. Some AI technology can also predict and help plan on property developments, this means houses are put onto the market during periods they are needed.

8) Maintenance

A company called RoadBotics has developed a technology using artificial intelligence that has the ability to analyse road imagery to then assess issues and produce cost effective solutions. This allows cities to know when and where repairs need to take place, and deal with them while saving money. This type of technology also improves safety within cities as problems will not go unnoticed.

9) Security

Security camera footage is typically reviewed when a crime has been reported, this doesn't prevent or stop crime. Security cameras that use artificial intelligence have the ability to analyse footage in real time and detect criminal behaviour which can then be instantly reported and dealt with. These cameras can also detect people from their clothes, allowing the technology to find suspects quicker than ever.

Challenges of a Smart City

As smart cities offer plenty of benefits in a rage of different sectors, there is also a big challenge when it comes to security of all the technology being used.

There is a need to ensure smart cities are protected from cyber-attacks, hacking and data theft while also making sure that the data is reported is accurately, so it can be to the benefit of all citizenships and not against them.

In order to manage the security of smart cities there is a need to implement measures such as physical data vaults, resilient authentication management and ID solutions. Some core security objectives involve:

Availability - Data needs to be available in real time with reliable access in order to make sure it performs its function in monitoring the various parts of the smart city infrastructure.

Integrity – The data must not only be readily available, but it must also be accurate. This also means safeguarding against manipulation from outside.

Confidentiality – Sensitive data needs to be kept confidential and safe from unauthorised access. This may mean the use of firewalls or the anonymising of data.

Accountability – System users need to be accountable for their actions and interaction with sensitive data systems. Users logs should record who is accessing the information to ensure accountability should there be any problems.

Conclusion

To control all the potential challenges that the technology might bring into our lives, the government, the private sector, software developers, device manufacturers, energy providers and network service managers need to work together to deliver integrated solutions to guarantee full and reliable security for its citizens.

However, I think we can all agree that the benefits of creating smart connected systems of our urban areas and ensuring sustainability outweigh any potential risks, making technological developments and their impact on our daily lives expand greater than ever.

A Glimpse of the next *CS Uncovered* Issue

On the next issue published, we will look at a greater depth on how London is considered a smart city, and what the government's objectives are on maintaining this title.

References:

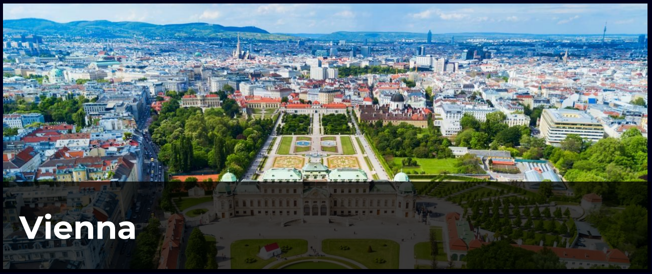
https://www.twi-global.com/technical-knowledge/faqs/what-is-a-smart-city

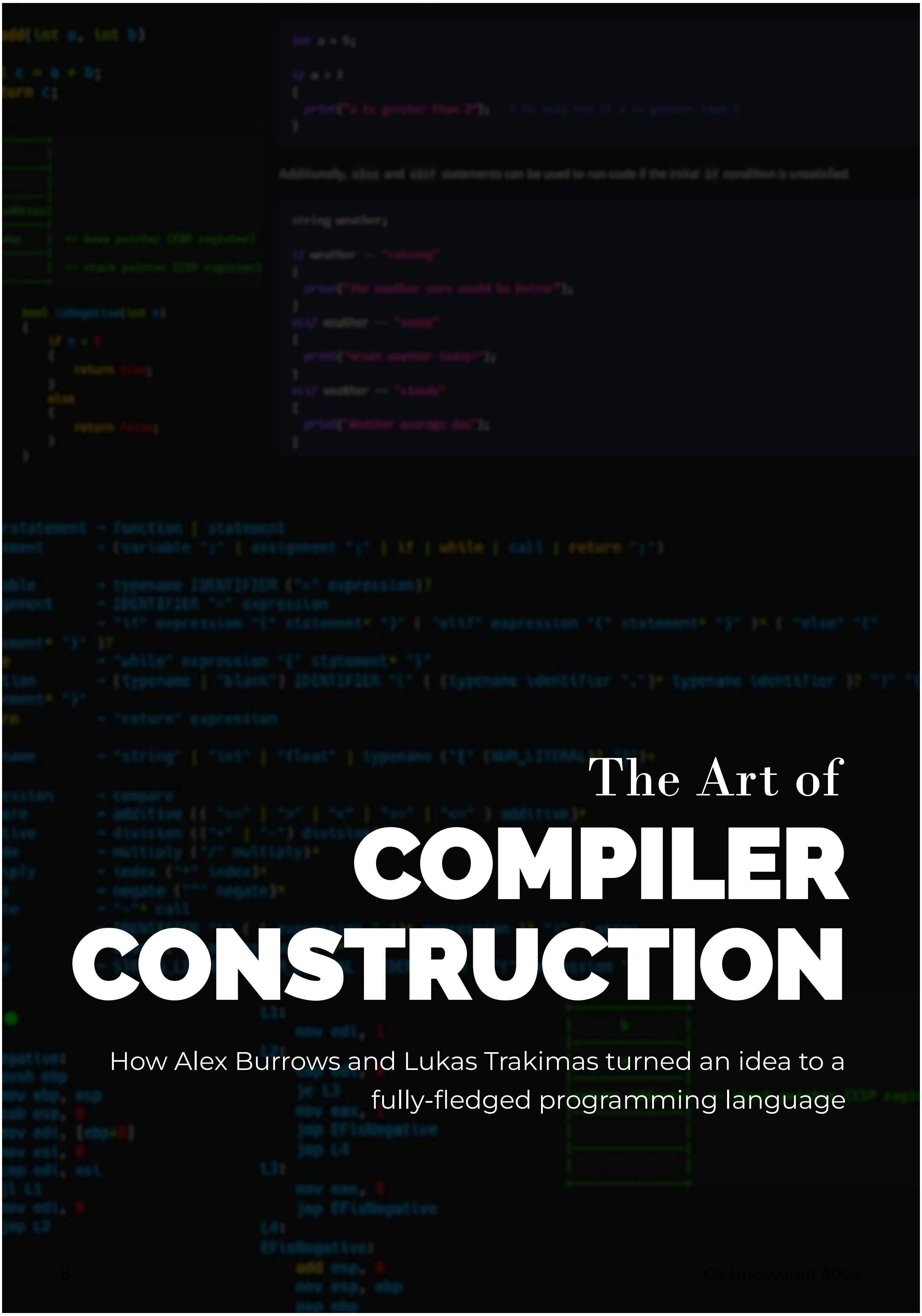
https://aimagazine.com/top10/10-ways-ai-can-be-used-smart-cities

https://www.ukconstructionmedia.co.uk/features/2019-smart-city-infrastructure/

Visiting a Smart City

Visiting a smart city will probably make you acknowledge how quickly the technology is evolving around us and how its every day use affects our ways of living within a city.
Below is a list of leading smart cities around the world:





The Art of COMPILER CONSTRUCTION

How Alex Burrows and Lukas Trakimas turned an idea to a fully-fledged programming language

I still remember how it happened. We'd just finished this barely functioning maths processor in Python, and one of us off-handedly mentioned "ok, so now that this done, I think we're fully qualified to make an entire programming language now!". We laughed about it, "Yeah right, like we'd start such an immense project so close to our exams" and "There's no way we'd be able to pull it off", only start working on it *five minutes later*. This is how our custom-made "written in 1 month" compiler works, to convert a high-level language into x86 assembly. Feel free to follow along with the code yourself at https://github.com/lxkast/Glory.

Syntax & Naming

We started at the point anyone would, to *design the language!* You can't build a compiler without a language to compile. For our language, which we designed rather uninterestingly and within 5 minutes, we settled on syntax very similar to C, with the most glorious name possible: Glory.

You don't truly appreciate how well designed the C syntax is until you make a language around it, we'd find in later stages of development that many features like semicolons, were the right choice.

Lexer

With our syntax in hand, it was time to develop the compiler! All compilers start by taking the **string**, the code coming as text, and **breaking it up** into little pieces for further processing. This is known as lexical analysis; that process of accepting characters and breaking them into "tokens". Every number written in the text, every symbol, every keyword like "int" or "blank" are picked out by the lexer and turned to tokens, with any words that aren't recognised as keywords turned into **identifier** tokens. For example, the code:

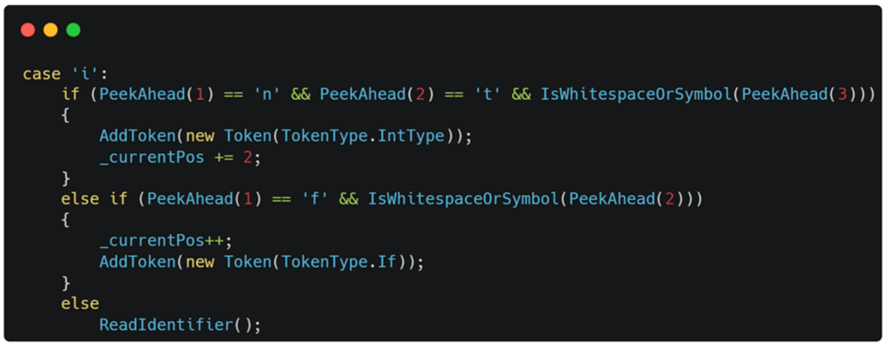
```
int a = 12;
```

Gets tokenised into:

```
[ int , a , = , 12 , ; ]
```

We run this lexing process because the parser (the stage after to *understand* what the code means) gets **extremely** complex and there's quite a bit of decision-making involved in picking out these tokens that we don't want to pack in there. Whitespace, detecting multi-character tokens (like keywords with "int"), understanding numbers, comments, we want all that difficulty gone for the parser and for it to always have simple pieces.

The lexer is quite straight-forward implementation-wise and almost definitely the least interesting part of the compiler. It's just a big for loop and switch, with a few fancy helpers like "PeekAhead". Below, it handles the character "i".



```
case 'i':
    if (PeekAhead(1) == 'n' && PeekAhead(2) == 't' && IsWhitespaceOrSymbol(PeekAhead(3)))
    {
        AddToken(new Token(TokenType.IntType));
        _currentPos += 2;
    }
    else if (PeekAhead(1) == 'f' && IsWhitespaceOrSymbol(PeekAhead(2)))
    {
        _currentPos++;
        AddToken(new Token(TokenType.If));
    }
    else
        ReadIdentifier();
```

A code snippet for lexical analysis of raw text when 'i' is encontered

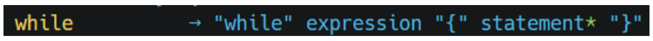
Parser

Once the code is broken into tokens, it's time to understand what it means and form *syntax trees* in memory to perfectly describe each line in a structured way that makes sense. This is known as parsing, and it's quite a beast!

The key component to making a good parser is to create a **grammar** out of it. This is a definitive guide of **exactly** how the syntax of your language is structured. This grammar needs to be very precise, ensuring it flows in such a way that factors in order of operations and all kinds of things, but when you do form it, that's most of the parsing work already done. (Image of grammar for Glory is at the bottom).

It looks quite complex, but if you really look through piece-by-piece, what it's describing is quite simple. Each line in this grammar is a rule, a separate piece, and when understanding the code, the parser starts at the top-most rule and slowly, with a chain of if statements, makes its way down through them.

With the grammar written, we can now use a technique called **recursive descent** to quite literally just turn this grammar *into code*. The idea is quite simple. For every line (rule), we make a new function, and we simply implement each rule sequentially through that. For instance, here's the "while" block:

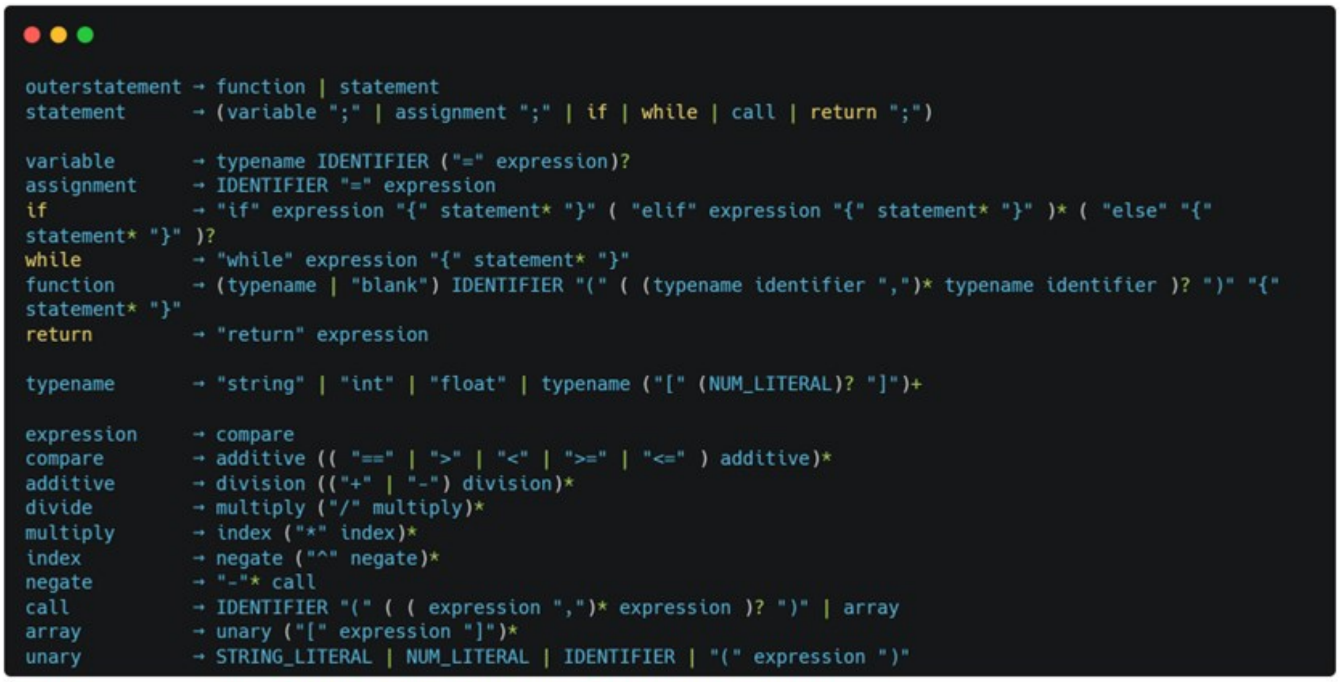


```
while            → "while" expression "{" statement* "}"
```

To implement the "while" rule above into code, we create a "ParseWhile" function, which will:

1. Move past the "while" token (if this function has been called the "ParseStatement" has already checked and established it *is* a while statement, so we can skip it.)
2. Call "ParseExpression" to read through the condition tokens. "ParseExpression" will read as much as it can until it encounters a token that neither it or its sub-rules can handle, which will be the "{" given valid code.
3. Verify there is actually a "{" here, and skip past.
4. Call ParseStatements to continually read statements. Once again, this will return as soon as it's lost on what to do, should be the "}" token.
5. Verify there's a "}" and read it. Our while is done.

The complete grammar defining Glory's syntax



```
outerstatement → function | statement
statement      → (variable ";" | assignment ";" | if | while | call | return ";")

variable       → typename IDENTIFIER ("=" expression)?
assignment     → IDENTIFIER "=" expression
if             → "if" expression "{" statement* "}" ( "elif" expression "{" statement* "}" )* ( "else" "{"
statement* "}" )?
while          → "while" expression "{" statement* "}"
function       → (typename | "blank") IDENTIFIER "(" ( (typename identifier ",")* typename identifier )? ")" "{"
statement* "}"
return         → "return" expression

typename       → "string" | "int" | "float" | typename ("[" (NUM_LITERAL)? "]")+

expression     → compare
compare        → additive (( "==" | ">" | "<" | ">=" | "<=" ) additive)*
additive       → division (("+" | "-") division)*
divide         → multiply ("/" multiply)*
multiply       → index ("*" index)*
index          → negate ("^" negate)*
negate         → "-"* call
call           → IDENTIFIER "(" ( ( expression ",")* expression )? ")" | array
array          → unary ("[" expression "]")*
unary          → STRING_LITERAL | NUM_LITERAL | IDENTIFIER | "(" expression ")"
```

If you visit the project and navigate around, you'll find the "Parser.cs" class with the "ParseWhile" function precisely doing this in the code. Now just imagine that repeated for every single rule, alongside a bunch of type checking and more – and that's the parser! The most impressive function in the parser is probably the function definition handling one, which as you can see from the rule is a very complex function indeed!

We realized that having semicolons in our C-derived language can be quite helpful with error handling. Our parser does not "need" semicolons to understand valid code, all the functions will naturally back out as you hit the end of the line and they no longer know what to do with the next token. However, if code is written incompletely and a statement makes no sense at all, without some form of clean line separating characters like semicolons, the parser has a very difficult job trying to realign itself with what's going on to continue reading to try and find "all errors" instead of just one, and semicolons help them out a lot with it. In the end this didn't impact our parser as it has simple error handling due to time constraints, but we realized this is a very valid use of semicolons in the code. (Doesn't help C++'s errors though...)

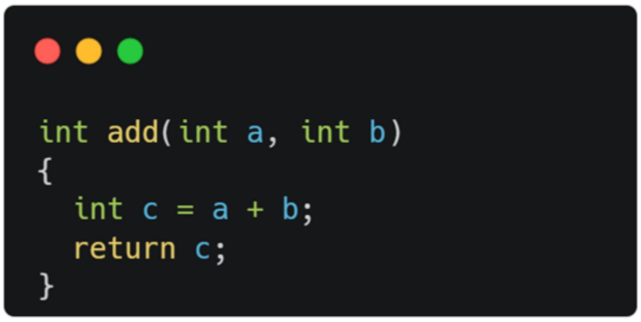
Code Generation

The final stage of the compilation process is to take the statements (tree per line) from the parser and generate assembly code out of it. At the moment, our code generator literally creates *text* assembly code that must be given to an external assembler, though its architecture is intentionally designed so it's easy to swap it to output machine code directly someday.

The Calling Convention

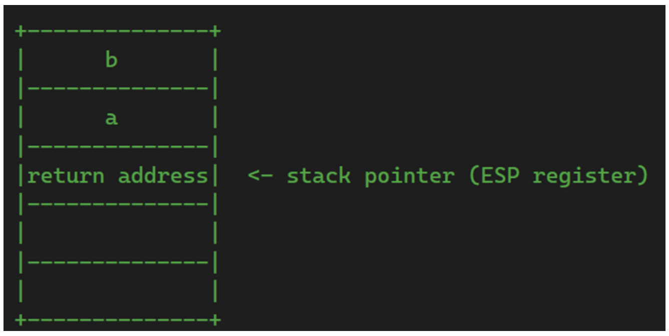
Before we could dive into generating assembly, we had to first set a few standards for exactly *how* the final assembly needs to look. One such standard was the calling conventions. A calling convention describes how memory needs to move around when you call a function. This includes a description of what has to happen in registers and what has to happen to the stack (a piece of memory every thread in a process has for keeping track of local function data).

Consider this function:



```
int add(int a, int b)
{
  int c = a + b;
  return c;
}
```

The first step to calling this function is the "add(...);" line, off in another function. There's quite a lot of complexity to pulling this call off properly, and a lot of additional considerations not described here, such as *backing up in-use registers*. But at its simplest, this is what the stack memory **should** look like as soon as we enter the body of a function. The stack grows downwards, so the top is the stuff pushed first:



```
+--------------+
|      b       |
|--------------|
|      a       |
|--------------|
|return address|  <- stack pointer (ESP register)
|--------------|
|              |
|--------------|
|              |
+--------------+
```

Arguments are pushed onto the stack in reverse order with a return address describing where the CPU should jump to when returning from the function. In practice, these stack operations are done with two "push" instructions for the arguments and one x86 "call" instruction which handles both the return address and function jump automatically.

```
push b
push a
call add
```

After this initial setup from the calling-side, we jump into the function and it's down to the function itself to take control of the stack as necessary, and we do this by kicking the function off with what's known as a **prologue.** The job of the prologue is to set up the function's "stack frame", which is the *region of the stack* the current function is using for its local variables. To do this, the machine has a second register entitled "ebp" (*base pointer*) designed to track the **beginning** of the current stack frame, to allow a sort of "start pointer" and "end pointer" format to be in place. Here's what the prologue looks like in assembly:

```
push ebp
mov ebp, esp
sub esp, N
```

The prologue starts by *backing the base pointer up* so we can return the stack frame back to how it used to be when we return, then configures ebp and esp such that they're showing *room* in place to store **every** local variable the function has in all branches.

Here's how memory looks after the prologue finishes:

```
+--------------+
|      b       |
|--------------|
|      a       |
|--------------|
|return address|
|--------------|
|   old ebp    |  <- base pointer (EBP register)
|--------------|
|      c       |  <- stack pointer (ESP register)
+--------------+
```

Now we have a clear *frame* for our local variables, with the parameters and extra info at a predictable amount behind the unmoving ebp register, and the local variables at a predictable amount ahead. From here, we can execute the function's actual code, which can use offsets around ebp to access relevant things at it pleases.

After the function's body completes, we have to restore all the stack pointers back to how they were before. **Any path** that returns from the function, whether it's from the middle or the end will jump to this *epilogue* logic at the end, and it's given below:

```
add esp, N
mov esp, ebp
pop ebp
ret
```

Once the function returns, the stack looks like this – note that we don't bother to *erase* unused bits of the stack, so all the function's stuff will happen to be sitting around ahead of the stack pointer until it gets overriden.

```
+---------------+
|      b        |
|---------------|
|      a        |  <- stack pointer (ESP register)
|---------------|
|return address |
|---------------|
|   old ebp     |
|---------------|
|      c        |
+---------------+
```

The calling convention also describes how data must be **returned** from the function, which is to use the eax register for any < 8 byte value or, in the case of arrays, for the caller to make some space on the stack above the parameters that the function can *copy its return value* into for the caller.

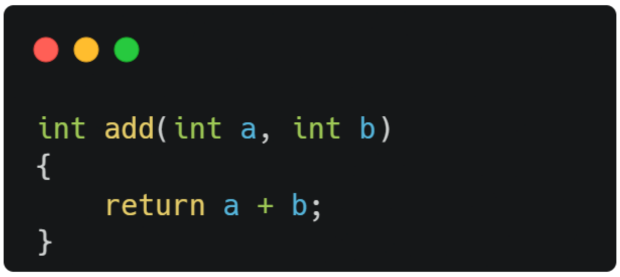
Generating operations

The assembly we generate to perform the actual operations in the code frequently needs to use registers to temporarily hold onto data. For instance, to perform an addition, the CPU **requires** that the two values to be added must be moved to registers first. When you're in the middle of a line with *many nested operations* going on, you'll often need 2, 3, 4 or maybe even 5 registers all "in use" at once. We need a reliable system that allows operations to be *assigned registers* at request, and this is precisely what we do. We have 5 registers free for use, which we dub the "**scratch registers**". These registers have no fixed purpose and a *table* of which are in-use at any time during generation is kept. Then the code for generating all the operations simply asks "Can I have a register for <this length of time>", and it'll be assigned.

Register	EDI	ESI	ECX	EBX	EDX
Available?	Yes	Yes	Yes	Yes	Yes

A significant bulk of the work in the code generator comes from the CompileNode() function, which has the responsibility of generating assembly to represent a node on a tree. This function takes the node to compile, *and* where the result of said operation may go – which could be a register, or place on the stack, whatever the node above the current one said it wants this to go into.

Let's run through the steps for the following function:



```
int add(int a, int b)
{
    return a + b;
}
```

As soon as the prologue finishes generation, the compiler sees the return statement here, and this in turn calls CompileNode with the tree involved (a + b), and eax as the destination.

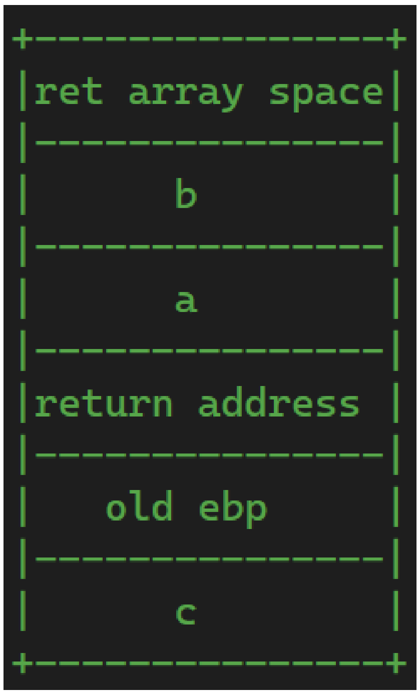
From here, CompileNode will switch on what type of node it's been given, a plus node in this case, and do the following for the addition:

1. Compile the **left-hand** of the addition (accessing a) through a recursive call, outputting the value of that into our destination.
2. Allocate a scratch register temporarily, in this case EDI, and while it's in use:
3. Compile the right-hand node, outputting into the temporary register.
4. Use the add instruction with the destination and right-hand node involved (the x86 instruction outputs directly into the left-hand side, the destination).

A significant amount of the code generation is just about following a *pattern* of instructions for each operation, albeit usually with different registers depending on what the context around the operation was, and a few decisions to vary depending on where the destination is.

Arrays

Arrays were quite the fun thing to implement into Glory because they came with **multiple** challenges. The struggle started with returning arrays, we had to revise our calling convention mid development to, quite literally, make room for them, as they can't just fit in eax and need to go across the stack. Here's a diagram of how the calling convention describes the stack when there's an array return type:



```
+---------------+
|ret array space|
|---------------|
|       b       |
|---------------|
|       a       |
|---------------|
|return address |
|---------------|
|    old ebp    |
|---------------|
|       c       |
+---------------+
```

But returning arrays was only the beginning. The idea of having the "destination" passed into "CompileNode" pulls extremely satisfying results, it allows every node to be written completely independently without needing to pack in logic of what's happening *above it* in the tree. CompileNode is quite simply **told** by the node above "please put the result here" and it does it, every node can be written really modularly like this... Until array indexing came in.

Unfortunately, indexing wasn't kind to our destination system at all because unlike everything else it doesn't simply take the **result** of what it's indexing, it actually needs to reach ***at*** the thing it's indexing.

Consider the code "a[1]". If this wasn't an indexing node, the generator would just say "Please compile this 'a' node I see here, whatever's necessary to do that, to output into <this place>" and then "And now I'll process that", but if we're indexing an array, you don't expect it to be moving "a" around **anywhere**! You expect it to *actually access a memory offset ahead of a*. We *could* have this be really, really dumb, and indeed allocate an entire temporary space for "a" to move into *just* to index that, so it makes a brand-new copy of "a" every single time you index it but that's really horrible, even for Glory's optimisation standards.

There are some potential architectural solutions to this, we could tweak the destination system and try to make it even smarter while still keeping as much modularity, *or* we can let the index node create the pointless copy and have an optimisation layer ***after*** the code generator (discussed later) detect in "hindsight" that there's a redundant copy and remove it. I can't confirm but I suspect this is precisely what the larger compilers do, they likely come up with very stupid things like this initially, but further steps pick up on them because unlike Glory they don't bake their "first draft" straight to assembly. But, because of time restrictions, we went down neither path, and we decided to simply *hard-code* support for different bases into the indexing node. Thankfully, there's only **two things** you can actually index in Glory, and that's *variables* and *function calls*. (Or, so we thought...) So, we simply *hard-coded* a check into the "index node" to look at what node it's indexing and do the relevant thing for each. Variables were easy enough, though indexing function call results was simply disgusting because it effectively requires the entire *call generating system*, which averaged 50 lines of code, to be split in two so for an index variant some extra instructions can be put **right in the middle** of all the call stuff.

Unfortunately, we soon realized, that there's a *third thing* you can index, and that's… An index node. If you have multi-dimensional arrays in your code, which Glory supports, you can very validly do "a [x][y]", which is an index node within an index node. You can also return multi-dimensional arrays, index the returns of said multi-dimensional arrays, and countless more fun things! Technically there really isn't anything special about multi-dimensional arrays, they are quite simply arrays *that contain arrays*, but it was added suffering to make sure we also fully supported all combinations of operations with them in place.

Code Optimisation

The code the compiler generates is extremely inefficient in places. The below code generates this assembly:

```
bool isNegative(int n)
{
    if n < 0
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
FisNegative:
    push ebp
    mov ebp, esp
    sub esp, 0
    mov edi, [ebp+8]
    mov esi, 0
    cmp edi, esi
    jl L1
    mov edi, 0
    jmp L2
L1:
    mov edi, 1
L2:
    cmp edi, 0
    je L3
    mov eax, 1
    jmp EFisNegative
    jmp L4
L3:
    mov eax, 0
    jmp EFisNegative
L4:
EFisNegative:
    add esp, 0
    mov esp, ebp
    pop ebp
```

If you watch the compiler output closely there is very broad range of stupid things happening. Subtracting 0 from a register? Code after a jump instruction? If you were to ask a human to write the isNegative() function in assembly they might give you this instead.

```
FisNegative:
    push ebp
    mov ebp, esp
    mov edi, [ebp+8]
    xor esi, esi
    cmp edi, esi
    jl L1
    xor eax, eax
    jmp EFisNegative
L1:
    mov eax, 1
EFisNegative:
    mov esp, ebp
    pop ebp
    ret
```

Far smaller and faster than the junk out of the compiler. The compiler outputs such terrible code because it has little **optimisation** in it, due to time constraints.

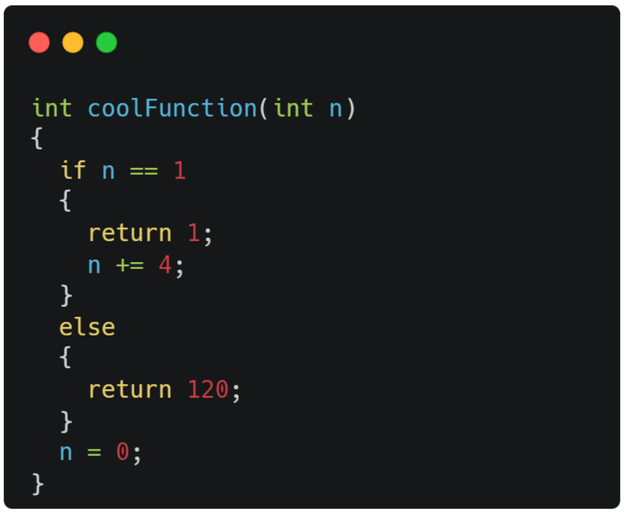
If we were to implement aggressive optimisation, there's two places we can do it. One is to add optimisation, extra intelligence, into the code generator as it currently is. This can certainly go some way and there are indeed some rather "high-level" optimisations we could add to the generator about how to structure/order the ASM (short for assembly) to better flow as a whole. However, simply adding tricks to the code generator alone won't be enough to give us *really clean assembly*. In fact, it's near impossible to do all the optimisations you see above *directly* in the code generator. This is because not only can the code generator not *foresee* what operations are going to happen later (which is very important) but the modularity of the code generator also means making optimisation decisions about how exactly *multiple nodes* are interacting with each other would simply be a mess. But this is very easily fixed.

How it could be made more optimisable

In A-level it's heavily implied that optimisation happens "after the code is generated". This statement is *partially true,* because optimisation is a thing that can, and does, happen at literally *every stage of compilation*, it's not just after. Many compilers have optimisations happen during the initial code generation, like I just described. However, it is correct that most compilers will do a large bulk of their optimisation **after** some kind of initial "code generation phase", the one that we currently end on. To do this, they have the same code generator that we currently have, but tweak it to, instead of outputting ASM directly, generate a *tree*. Unlike the parser's rather abstract tree, this tree describes in **assembly-level detail** *every* little step the machine may need to take to run the program. By doing this, we've now thrown away all the "high-level" niceties of statements and nodes, **all** the code generator leaves us with this massive dump, this singular massive *flow* of operations. Every time need to move in memory, every time we need a temporary value for something, every little offset access or addition/subtraction, it's all broken up in this tree. And what's really nice about this now, is **this** is *much more* optimizable. There's no longer statement and node boundaries in the way, it's all just one big thing, and now we can look ahead to cross-reference, because there's a solid first draft there. From this point it really is just a matter of trying to *brute simplify*, through supports for all kinds of tricks, what's in the tree as much as possible. And once the optimiser is happy with the result, final registers can be assigned, and ASM can be written.

If we took the compiler down this path and changed the code generator to output a tree instead of ASM, we'd likely call it the "lowering phase" instead. "Lowering" is a common term in compiler design that means to take the high-level stuff and nodes and breaks it into *low-level* steps, precisely what the generator would be doing.

As of current, the only notable optimisation our compiler performs is dead code elimination within functions. If given the following code:



```
int coolFunction(int n)
{
  if n == 1
  {
    return 1;
    n += 4;
  }
  else
  {
    return 120;
  }
  n = 0;
}
```

The *parser*, as that's the component that has to perform "flow checking" to make sure the input code always returns, will remove the redundant "n += 4" and "n = 0" lines which is indeed a form of optimisation, performed at a very high-level.

Final words

Our language suffers the most from a lack of features – strings, floats, lists and "for" were all initially planned, but cut to shorten development. While the code generator works fine there is one untested scenario in which it may fail: When it runs out of scratch registers. At the moment, it quite literally explodes here (through an error message), but it is possible to implement unused register backups onto the stack, just like how operating system may use virtual memory to swap pages back and forth, we can use the stack to hold some registers to make some room. However, there are a considerable number of layers to doing this, you have to really careful you don't mis-match the pushes of and pops of that with pushes and pops the operations are laying down and it's the kind of thing that'd be *significantly easier* to do if had more post-lowering stages. Thankfully, in all of our testing and development we've yet to find a case where the 5 scratch registers aren't enough.

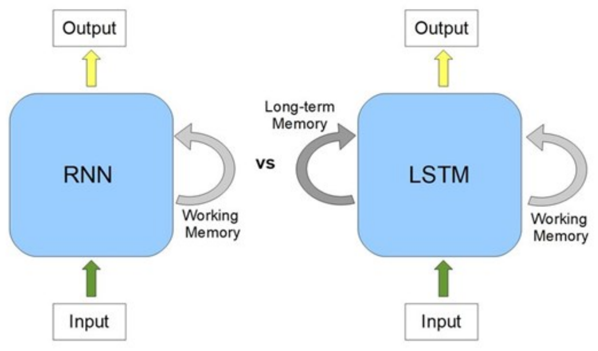
Writing a compiler is a challenging but rewarding task, requiring a deep understanding of low-level computer architecture and hardware. The moral of the story here is: If you just want to make a programming language – make it an interpreted language, it's 1000 times easier.

PREDICTING STOCK PRICES with Long Short-Term Memory

Thanish Senthil

Did you know you can leverage the use of computers to make predictions of the future? As astonishing as it sounds, it is possible with the use of Machine Learning algorithms, which make predictions by finding on patterns across huge amounts of data, but it doesn't mean it is always accurate. In this article we will be exploring a Machine Learning model called Long Short-Term Memory (LSTM) and its applications for predicting the future prices of a stock.

What are Long Short-Term Memory and Recurrent Neural Networks?



Long Short-Term Memory is a type of Recurrent Neural Network (RNN). An RNN is designed to process sequential data - data that has a chronological relationship - such that points in the dataset are dependent on other points in the dataset. Examples of Sequential Data include Time Series, where data is collected at regular time intervals (Stock Price data is what we will be using for the Time Series data), Natural Language Text and much more. An RNN remembers previous time steps and uses that to make predictions.

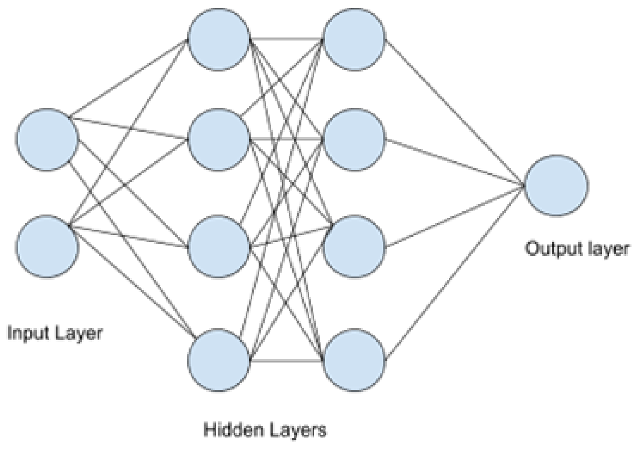
Imagine you are trying to predict the next word in a sentence, if you look at the previous words, you will have a better idea of what the next word is likely to be. This is simply how RNNs work, they use an internal memory to process sequences and use those to make a prediction. These features are not present for a classic neural network such as a feedforward neural network.

Why is LSTM used instead though?

As previously stated, LSTMs are a type of RNN, but the internal memory used in a LSTM network allows them to store long -term dependencies in the data. The memory cells consist of gates which are the input gate, forget gate and output gate and they regulate the flow of data, and they are used to determine whether data should be retained or forgotten. This is why Long Short-Term Memory networks are used instead of Recurrent Neural Networks for Stock Price Prediction.

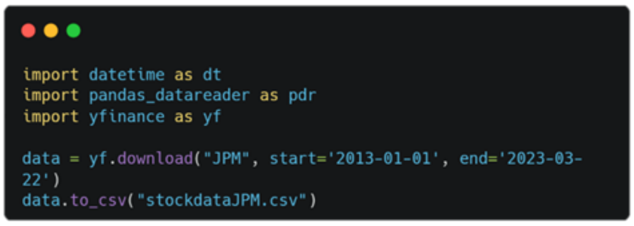
What is a neural network?

Since we've established that Long Short-Term Memory networks are a type of Recurrent Neural Networks but what is a Neural Network? Neural Networks work similarly like the neurons in your brain. Neural Networks consist of multiple layers of interconnected nodes which we refer to as neurons which process and transform data. The layers include an input layer, hidden layers and an output layer. Neural networks are weighted, directed graphs so each node is connected to another node and has a weight. When an input is passed into an input layer, the data is redirected into the hidden layers. As the input is propagated through the network, each layer then performs mathematical operations on the input such as multiplying the input by the weight and adding a bias term. Adding a bias term allows the network to adjust its output based on a constant value regardless of the input data.



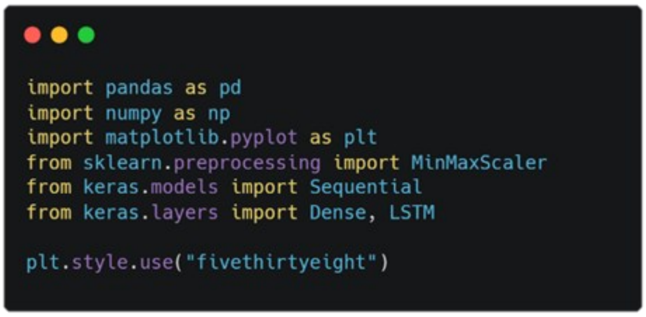
Now that all the theoretical stuff is out the way, lets implement LSTM to predict Stock prices for the next day in Python! We will be predicting J.P. Morgan stock prices.

1. Firstly, we need to gather stock price data of JPM, using these libraries in Python. We simply download the Stock price data from Jan 1st 2013 to 22nd March 2023 and save all the data into a CSV file.



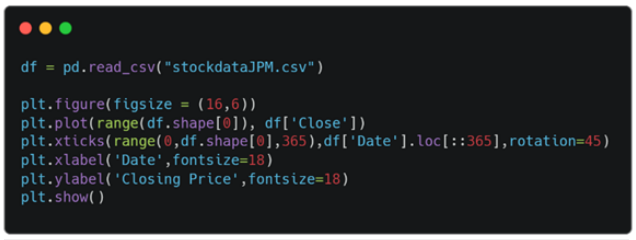
```
import datetime as dt
import pandas_datareader as pdr
import yfinance as yf

data = yf.download("JPM", start='2013-01-01', end='2023-03-
22')
data.to_csv("stockdataJPM.csv")
```

2. Now we need to import the necessary libraries which we will be using for this project. Pandas will be used for reading off the csv file, NumPy for creating NumPy arrays, matplotlib for plotting data, scikit-learn and Keras for creating and training the machine learning model.

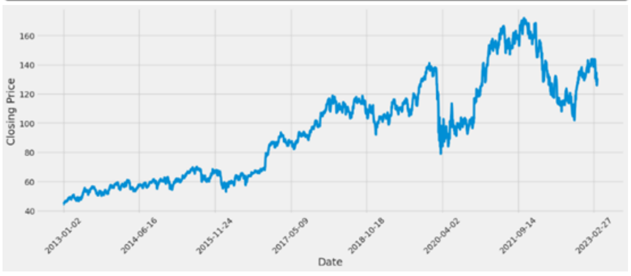


```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM

plt.style.use("fivethirtyeight")
```

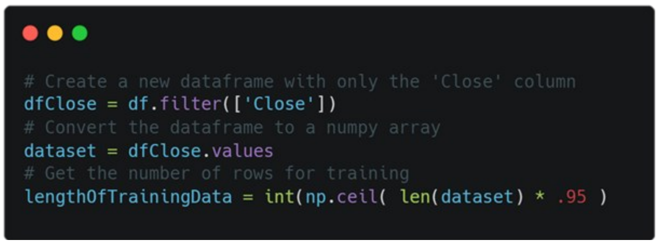
3. This step we add the Stock Price data onto a data frame and plot the closing prices using matplotlib.



```
df = pd.read_csv("stockdataJPM.csv")

plt.figure(figsize = (16,6))
plt.plot(range(df.shape[0]), df['Close'])
plt.xticks(range(0,df.shape[0],365),df['Date'].loc[::365],rotation=45)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Closing Price',fontsize=18)
plt.show()
```

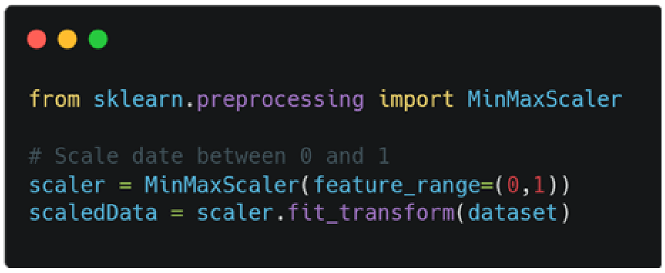


4) Now we need to determine what length of the NumPy array we will be using for training the machine learning algorithm. In this code, we create a new data frame and create a NumPy array based on just Closing prices and take 95% of the length and round it to the nearest integer.



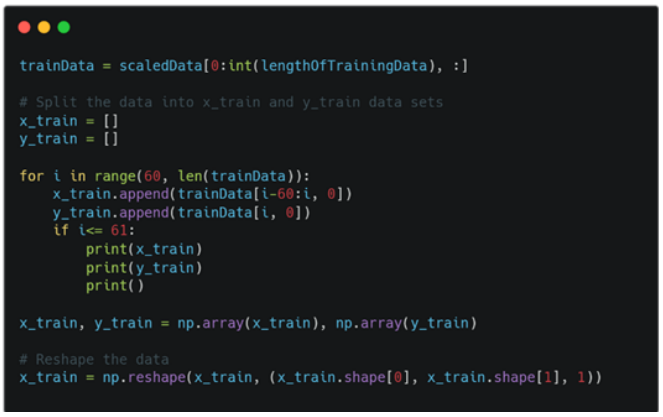
```
# Create a new dataframe with only the 'Close' column
dfClose = df.filter(['Close'])
# Convert the dataframe to a numpy array
dataset = dfClose.values
# Get the number of rows for training
lengthOfTrainingData = int(np.ceil( len(dataset) * .95 )
```

5) The next crucial step is to normalise the data using Min-Max scale. The purpose of normalisation is so that data has a fixed range of values, in this case, 0 and 1. When data is on similar scales, it removes bias as an algorithm may give more weight to one input. Normalisation will reduce numerical problems during training.



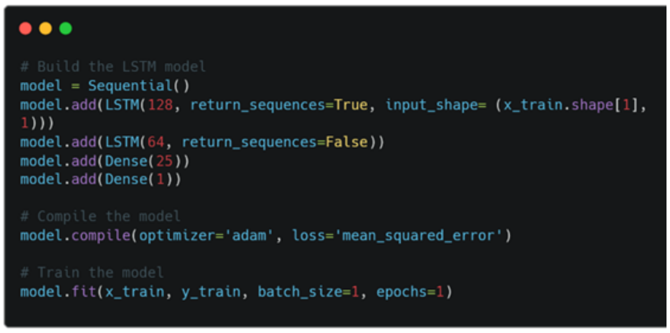
```
from sklearn.preprocessing import MinMaxScaler

# Scale date between 0 and 1
scaler = MinMaxScaler(feature_range=(0,1))
scaledData = scaler.fit_transform(dataset)
```

6) We now specify what the training data will be then we create two empty lists which is used to store input and output sequences for the LSTM. We create a sliding window of 60 time steps for each input sequence. The loop appends the previous 60 rows to 'x_train' and the next row to 'y_train' starting from the 61st row.



```
trainData = scaledData[0:int(lengthOfTrainingData), :]

# Split the data into x_train and y_train data sets
x_train = []
y_train = []

for i in range(60, len(trainData)):
    x_train.append(trainData[i-60:i, 0])
    y_train.append(trainData[i, 0])
    if i<= 61:
        print(x_train)
        print(y_train)
        print()

x_train, y_train = np.array(x_train), np.array(y_train)

# Reshape the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
```

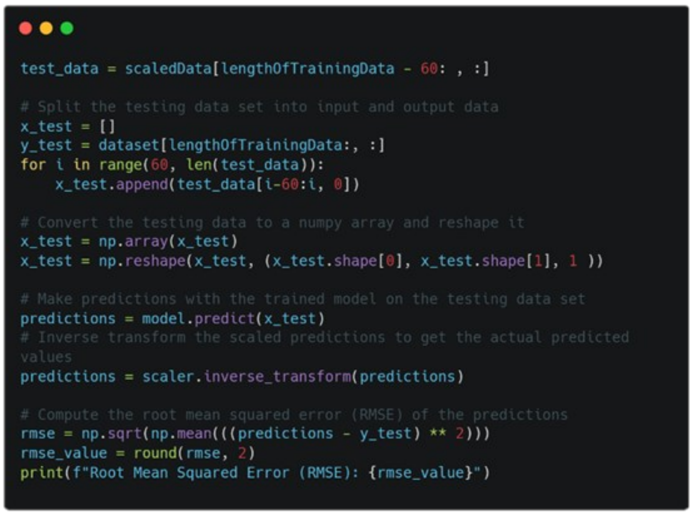
7) This step involves building the LSTM model and training it using Keras. We create two LSTM layers, which are layers consisting of neurons that receive input from all of the neurons in the previous layer. After, we optimise our model with the Adam optimiser and minimise mean-squared error. It is then trained on the training data we provided.



```
# Build the LSTM model
model = Sequential()
model.add(LSTM(128, return_sequences=True, input_shape= (x_train.shape[1],
1)))
model.add(LSTM(64, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(x_train, y_train, batch_size=1, epochs=1)
```

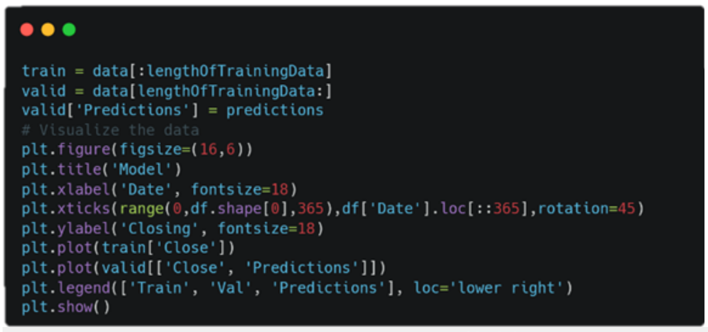
8) Now we prepare the testing data set. We slice the testing data set into an input and output data and make predictions on the input data using a trained model. The predict-

ed values we obtained are transformed into the original scale. A Root Mean Squared Error is also calculated. An RMSE is simply the standard deviation from the actual values and the predictions. We had actually obtained a RMSE of 4.77 which is relatively small as the stock price is around 100-150 dollars

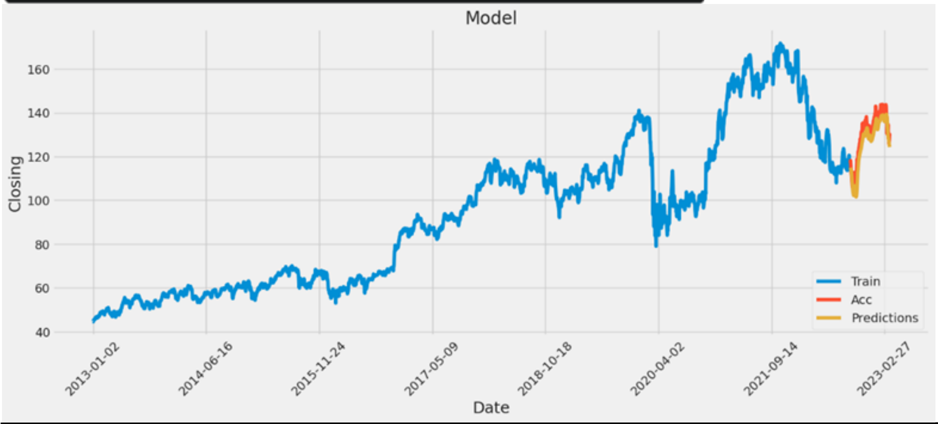


```
test_data = scaledData[lengthOfTrainingData - 60: , :]

# Split the testing data set into input and output data
x_test = []
y_test = dataset[lengthOfTrainingData:, :]
for i in range(60, len(test_data)):
    x_test.append(test_data[i-60:i, 0])

# Convert the testing data to a numpy array and reshape it
x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1 ))

# Make predictions with the trained model on the testing data set
predictions = model.predict(x_test)
# Inverse transform the scaled predictions to get the actual predicted
values
predictions = scaler.inverse_transform(predictions)

# Compute the root mean squared error (RMSE) of the predictions
rmse = np.sqrt(np.mean(((predictions - y_test) ** 2)))
rmse_value = round(rmse, 2)
print(f"Root Mean Squared Error (RMSE): {rmse_value}")
```

9) The very last step involves plotting the data and visualising it to see if our predictions were accurate. This code splits the chart showing which data was used for training, which was used for predictions and the actual value.

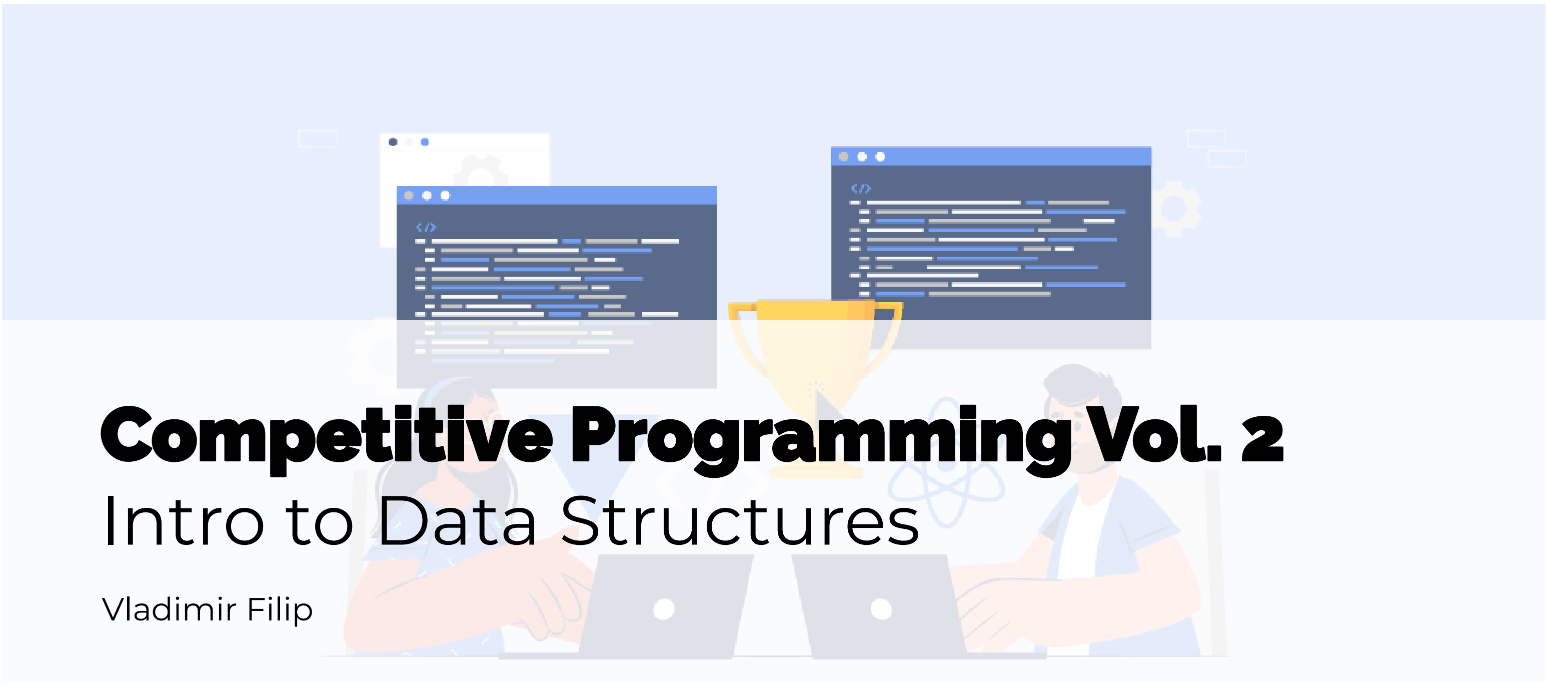


```
train = data[:lengthOfTrainingData]
valid = data[lengthOfTrainingData:]
valid['Predictions'] = predictions
# Visualize the data
plt.figure(figsize=(16,6))
plt.title('Model')
plt.xlabel('Date', fontsize=18)
plt.xticks(range(0,df.shape[0],365),df['Date'].loc[::365],rotation=45)
plt.ylabel('Closing', fontsize=18)
plt.plot(train['Close'])
plt.plot(valid[['Close', 'Predictions']])
plt.legend(['Train', 'Val', 'Predictions'], loc='lower right')
plt.show()
```



Conclusion

It is evident that the program computes a prediction that has a small standard deviation (RMSE of 4.77 from the actual values). This model isn't entirely great for predicting the exact prices, but it produces predictions which follow a trend that is extremely similar to the actual values. It is worth noting that predicting stock prices is notoriously difficult. Stock price movements are stochastic processes which follow Brownian motion. This machine learning model does not take into account any external factors such as news, volatility, interest rates, etc., only historical price data. However, it is interesting how machine learning can be applied in the stock market and helps analyse large amounts of data which can be difficult for human analysts.



Competitive Programming Vol. 2
Intro to Data Structures

Vladimir Filip

In the previous edition we covered the basics of asymptotic analysis, a way to represent the running time and memory usage of an algorithm as a function of its input size. In this article we will cover the design of various data structures and the time complexity of operations applied to them.

What are data structures?

A data structure stores and organises data; it is a way of arranging data so that it can be accessed and updated efficiently.

There are various ways to classify data structures, but all of them can be split into two groups: linear data structures and non-linear data structures. In this edition we shall cover linear data structures, their design and the benefits and drawbacks of using them. In-depth knowledge on this allows you to write faster and more memory-efficient solutions and can even change the way you approach a given problem.

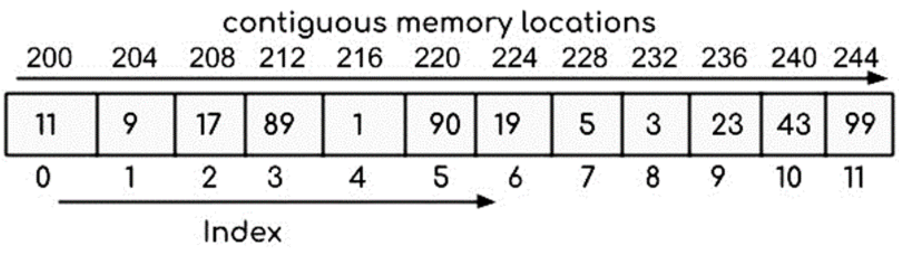
Linear Data Structures

These are data structures that are designed to store its elements in a sequence, with one data element coming after another. There are four linear data structures: arrays, linked lists, stacks, queues and hash tables.

The Static Array

Insertion: O(n), Read: O(1), Deletion: O(n), Creation: O(n)

The static array is the simplest of all data structures, storing a fixed number of elements in memory and allowing them to be accessed by index. The elements are stored in sequence in memory, and so given the memory location of the start of the array and the index, the memory location of any element in the array can be obtained and the element can be read. This makes any read operation on the elements O(1), as the time to read the element does not change with the size of the array. Therefore, you could say that the memory space storing the elements is *contiguous*.



A diagram of how a static array is stored in memory, with all the elements stored in sequence and in one contiguous block.

This contiguous memory space first has to be allocated (marked by the operating system so that data stored cannot be overwritten), making the time complexity of creating a static array O(n), with n being the size of the array, as the amount of memory allocated is directly proportional to the array size.

If an element at index *i* is deleted, or an element is inserted at index *i*, from an array of size *n* then the *n* – *i* elements to the right of that element have to be shifted left in the case of deletion or right in the case of insertion in order to preserve the order of elements without leaving gaps in the array. This makes the time complexity of insertion and deletion O(n – i), and since i can be anywhere between 0 and n – 1, this simplifies to O (n).

Arrays are commonly used due to their fast access times, but due to insertion and deletion happening in linear time, other data structures may be needed if those are the main operations you wish to carry out.

Dynamic arrays are also used in languages such as Python and JavaScript, where the size of the array can be increased and decreased as needed. Insertion and deletion operations are even slower on dynamic arrays (also known as 'lists'), for if there are more elements needed to be stored than the number of elements the allocated memory space can hold, a larger block of memory has to be re-allocated, and all the elements have to be copied to the new memory space.

The Linked List

Push: O(1), Pop: O(1), Creation: O(1), Read: O(n), Insertion: O(n), Deletion: O(n)

A linked list is similar to an array, in which it is used to store multiple elements in one structure, but it does away with contiguous memory location, instead consisting of a number of smaller elements stored in random locations in memory, each containing the data value and a pointer (variable storing a memory address) to the next element. The elements are 'linked' through these pointers, hence the term 'linked list.' The linked list can also store 'head' and 'tail' pointers to store the location of the first and last elements in the linked list to allow for the insertion and deletion of elements on either side.

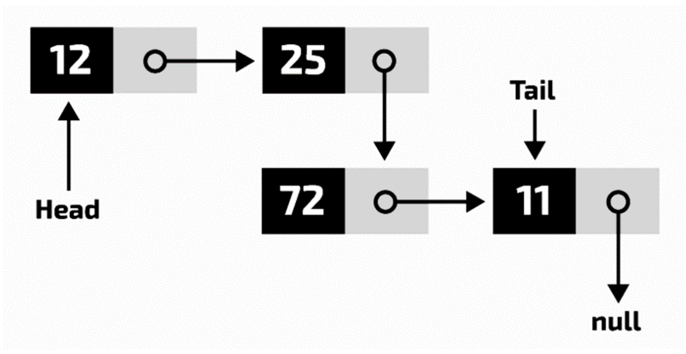


Figure 2: A diagram of how a linked list is structured, some linked lists may contain two pointers in each element, one to the previous element and one to the next element. This is a doubly linked list.

To remove or add elements to the linked list all that is needed is to change the pointers (variables storing memory addresses) on elements either side of the new element, instead of having to shift *n* elements as is the case with an array, making insertion and deletion time complexity in linked lists O(1) relative to the linked list's size. However, this is the case of *push* and *pop* operations where elements are being added or removed either side of the list. Inserting or deleting elements in the middle of the list require traversing an average of *n* elements through the linked list through their pointers until the desired location is reached, making the insertion and deletion of elements at index *i* O(n), with *n* being the size of the linked list. The only allocation required is to allocate enough memory for the new element being inserted and the pointer, which is not affected by the size of the linked list.

A linked list is generally initialised as empty and elements are pushed afterwards, making creation of a linked list a constant-space operation.

Reading an element in the middle of a linked list also requires traversing the linked list from the start of the list to the required element through its pointers. The number of elements required to traverse through grows linearly with the size of the linked list, making the read time complexity O(n), with n being the size of the linked list.

Therefore, a linked list is mainly used if the main operations used are insertions and deletions, and it lays the foundation of other data structures such as stacks and queues.

The Stack

Push & pop: O(1), Creation: O(1), Peek: O (1), Read/insert/delete at index *i* ($0 < i < n$): O(n)

A stack is a linked list where all insertions and retrievals are carried out at the end of the linked list, often referred to as the 'top' of the stack. The stack therefore follows a Last In First Out design, where the last element 'pushed' to the top of the stack is the first element retrieved, or 'popped' from the stack.

Use cases include any scenario where you need to constantly add elements and retrieve the most recently added element, with a stack allowing you to do that in O(1) time.

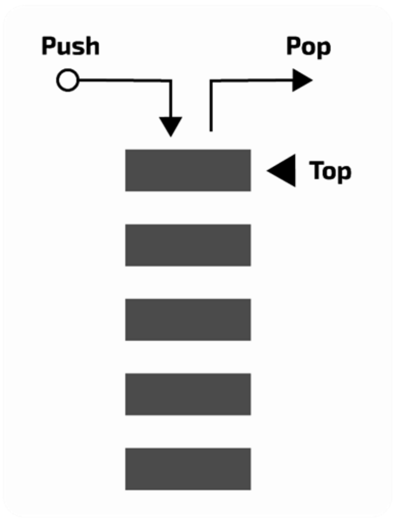


Figure 3: A diagram of a stack. There is often a misconception that stacks are implemented using arrays rather than linked lists, but this is not the case, for if an array was used it would not be possible for push/pop operations to occur in O(1) time and for the size of the stack to be dynamic.

The Queue

Push & pop: O(1), Creation: O(1), Peek: O (1), Read/insert/delete at index *i* ($0 < i < n$): O(n)

A queue is also a linked list, where pushing (also known as enqueuing) an element and popping (also known as dequeuing) an element happens at opposite ends of the linked list. The queue therefore follows a First In First Out (FIFO) design, where elements are popped in the exact order in which they are pushed.

This makes them great for breadth-first traversal of trees, or traversal of permutations where you want permutations of the same recursive depth to be considered consecutively. They can also be used for data buffers due to O(1) push and pop time complexities.

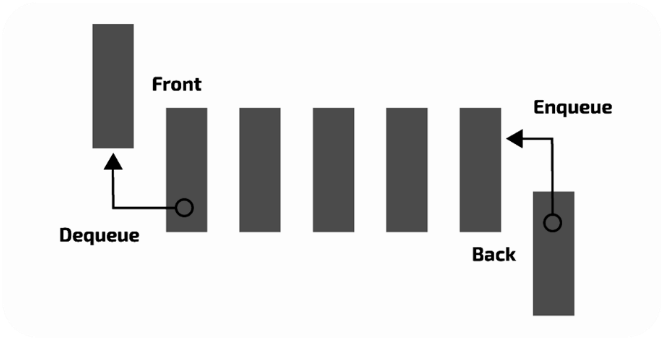


Figure 4: A diagram of a queue, consisting of front and back pointers which are used for enqueuing (adding an element to the queue) and dequeuing (popping an element from the queue) respectively.

The Hash Table

Creation: O(1), Read/Write: O(1)

Also known as a dictionary in Python, the hash table is a data structure that stores data in the form of key-value pairs, allowing values stored in the structure to be identified using the given key.

Internally, hash tables consist of a static array, with each element being referred to as a *bucket*.

Modern hashing functions are made so that they produce unique numbers for as many different keys as possible, assigning each key-value pair a unique bucket. However, collisions can still occur, where different keys are hashed to produce the same number. The bucket then becomes a linked list and both the values in both key-value pairs are stored in there.

If the internal array is very small, and/or if there are a very large number of elements, elements could end up being in stored in very long linked lists, making the read-write time complexity O(n) rather than O(1).

Therefore, while hash tables are very useful as caches (due to quicker access times), they must not be over-used, for if they become large enough to lead to , any performance benefits are lost.

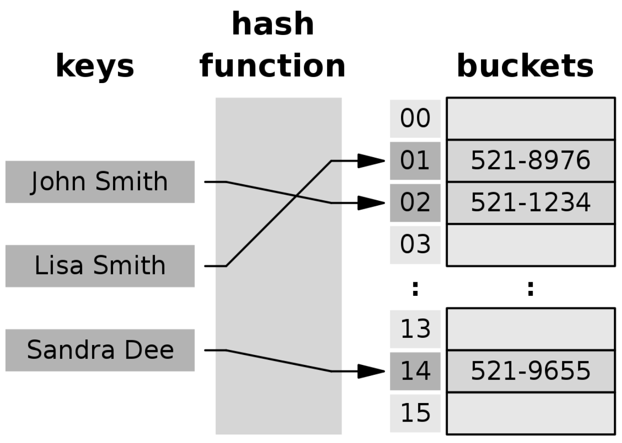


Figure 5: A diagram of a hash table. ‘Buckets’ are often used to refer to the data structure in which values are stored,.

Summary

In this article we have covered four linear data structures, their design and potential use cases. However, the exact way in which you can use them in code varies from language to language, and so I highly encourage you to look up how to use these structures in your chosen language and practise using them.

In the next issue, we will cover the design and characteristics of non-linear data structures, such as graphs, trees and heaps, as well as popular algorithms that use linear and non-linear data structures to their full potential for highly efficient solutions to more abstract problems.

A (Brief) History of Linux

Constantin Filip

With over 600 different active distributions, 32.8 million users (not including the 1.6 billion users of Android), and over 1 500 collaborators working on the Linux kernel, it is believed that it would take thousands of years to rebuild the Linux kernel from scratch. The operating system of choice for the hardcore developer, Linux has cemented its place in the OS landscape as one of the most reliable and refined systems to ever exist, powering everything from the tiny Raspberry Pi to the mighty TOP500 supercomputers. But what are the origins of Linux, and how did it grow into such a vast ecosystem?

In the 1980s, the dominant operating systems were MS-DOS, Unix and other Unix-like systems. The first version of Unix was released in 1969 and already had a strong developer following. In 1974, it was rewritten entirely in C and became the first portable operating system, and in the late 1970s, a number of commercial variants of Unix had been produced by the likes of Microsoft, Sun Microsystems and IBM. However, the main issue with all of these operating systems was that they were proprietary; their source code was kept completely hidden from other developers.

In the formative years of operating systems, this led to several issues. A particular incident, recounted by one Richard Stallman, was when an especially annoying printer could not be fixed because the users had no access to the source code of its drivers. The lack of flexibility introduced by proprietary software in peripherals and system software also prevented collaboration between developers and reduced the control that a user had over their own system.

The rise of proprietary software led Stallman to launch the GNU (Gnu's Not Unix) project in September 1983, with the aim to produce a completely free operating system, which would allow users to study the source code, share the software, modify its behaviour, and publish their own version of the software. In this way, the freeware (later including the term 'open source') movement had begun. The GNU GPL (General Public License) was released soon after and by the 1990s, many elements of what would be the GNU operating system had been created and were all free and open source. However, the project was incomplete; it lacked several low-level features and still did not have a complete kernel. Something was needed to fill that last gap, to make Stallman's vision of an open source operating system finally become a reality. And that something was the creation of a certain 21-year-old student at the University of Helsinki.

Linus Torvalds began work on his own operating system kernel in August 1991.

Linus Torvalds — the creator of Linux

Originally called Freax and later Linux, it was inspired by the Unix-like Minix OS produced by Andrew Tanenbaum.

Torvalds originally began developing Linux on a Minix system using Minix components, and by September, v0.01 of the kernel was posted on ftp.funet.fi, the University of Helsinki's FTP server. A defining feature of Linux which separated it from Minix was its use of a monolithic kernel (i.e. the entire operating system works in the kernel space, a section of the virtual address space used specifically for running the kernel and some device drivers, but most importantly, the kernel space cannot directly access the userspace and vice versa), compared to Minix which used a microkernel. By the end of the year, Torvalds already began to receive code for new features sent to him from other people; this marked the start of Linux as a collaborative project, one which would grow to thousands of developers around the world.

In 1992, Torvalds adopted the GNU GPL license, after originally publishing Linux releases under his own license to restrict commercial activity. This marked a pivotal moment in making Linux one of the most prominent examples of free, open-source software. This meant that anyone could view the source code and modify their version of Linux in any way they wanted. Soon, all Minix components were replaced by ready-made GNU features, and development on Linux began to move away from Minix and continued on existing Linux systems. Developers worked on integrating GNU features to produce a fully-fledged operating system. In March 1994, the first production version of Linux was released (although two years later than Torvalds had anticipated). By then, the OS already included several complex features for the time including multitasking, management of virtual memory and a multi-threaded file system.

A major aspect of Linux that led to its popularity was the ability for developers to create distributions of Linux. A distribution, or distro for short, is an operating system which includes the Linux kernel, GNU tools and libraries together with a package management system.

Linux is typically installed on a machine as a particular distribution. The rise of the Linux distros is a result of the GNU philosophy, but also because of the architecture of Linux itself. The kernel is divided into various self-contained subsystems, each of which has a defined function. They indirectly interact with each other using function calls and shared data structures, with each subsystem further divided into distinct modules. This modularised structure is what made mass collaboration on Linux possible; it allowed many independent developers to contribute to its development without interfering with the work of other developers.

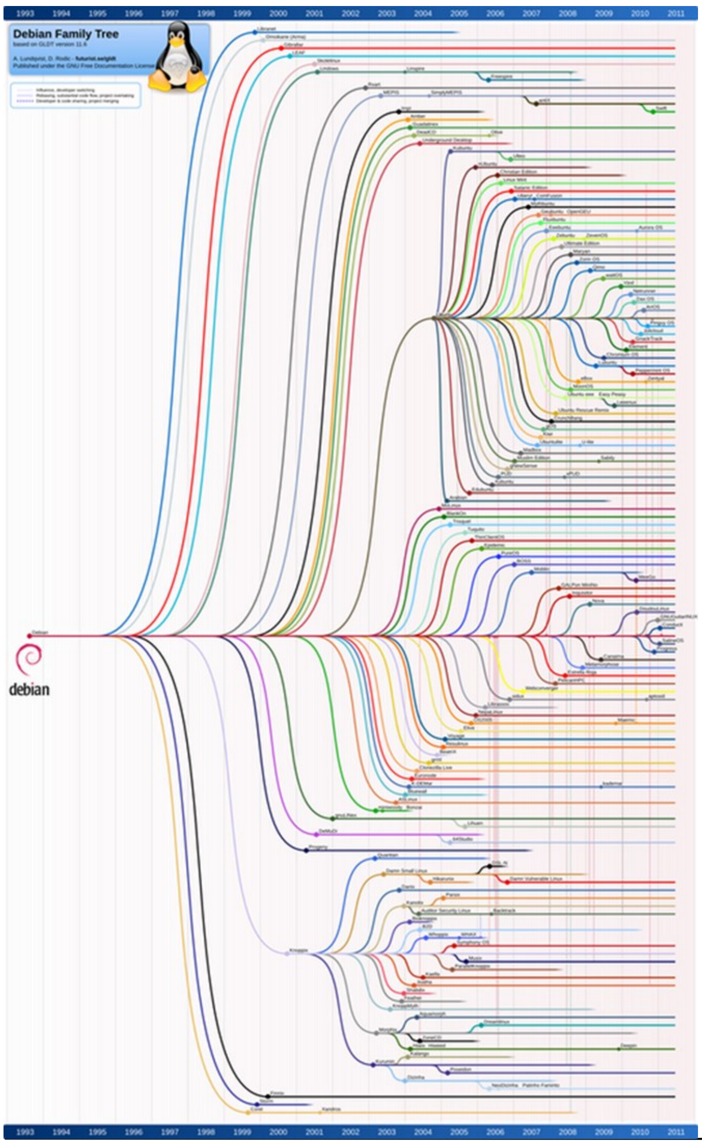
The oldest Linux distro still actively maintained today, Slackware, was first released in July 1993.

Over time, many other popular Linux distributions began to be released. Each distribution was created for a specific need, with a particular design focus. In 1994, Debian was released, noted for its stability and security and ability to run on a wide range of devices. Commercially backed distros maintained by software companies appeared, most notably Ubuntu (based on Debian) and Fedora, while others remained entirely community driven. Distros designed for specific machines, such as servers and embedded devices also appeared, as well as for specific user groups such as scientific communities.

Collaborators continued to add more and more features to Linux, including symmetric multiprocessing, improvement of its graphics stack to use modern GPUs, wireless drivers and advanced peripheral support. In 1997, the first version of GNOME (GNU Network Object Model Environment) was released, a desktop environment for Linux and other Unix-based operating systems, which has become the default environment of Debian, Fedora Linux, Ubuntu, SUSE Linux Enterprise and many other distros. GNOME provided a simple and intuitive UI design which made Linux much more accessible to less experienced developers.

In November 2007, Android, a mobile OS based on Linux, was released. It continues to be developed by a group of developers known as the Open Handset Alliance and sponsored by Google. Since its original release, it has become the most popular operating system, with over three billion active users.

Today, Linux runs on billions of different devices and the kernel has amassed over 27.8 million lines of code as of 2020. It is one of the dominant operating systems on supercomputers, servers and embedded systems, and continues to grow in popularity in the desktop PC market. In the future, the kernel will continue to be updated with all the latest drivers and technologies; Linux is expected to continue to play a major part in the Internet of Things, the motoring industry, cloud infrastructure, personal computing devices, and many other technological innovations.



Here is a (simplified) diagram of the Debian family tree. You'll probably need a magnifying glass to see some of these distros. The chart also only goes up to 2011, so 12 years later at the time of this publication of *CS Uncovered*, it's probably grown quite a bit.

And this is just the distro family tree for *Debian* alone, so it's just a fraction of the full family tree of the whole of Linux.

THE PUZZLE PAGES

BY DINU FILIP

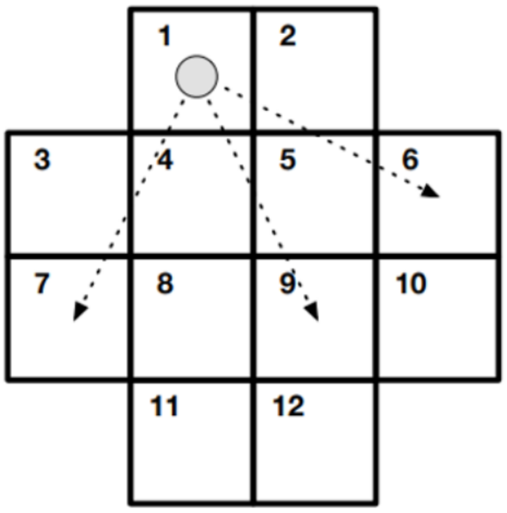
Send your solutions to Dinu at
17h filipc@beths.bexley.sch.uk

Whoever solves the most puzzles by the next issue will win a special prize!

THE KNIGHT'S TOUR

A single chess Knight is able to move on the small cross shaped board below. A Knight can move two spaces in one direction and then one square at a right angle. It jumps to the new square and must always land on a square on the board.

Find a sequence of moves that starts from square 1, visits every square exactly once by making such knight's moves and finishes where it started.



MY BRAIN....

Brainf*ck is an esoteric programming language which consists of the fewest number of distinct commands needed to be Turing complete:

Command	Description
>	Move the pointer to the right
<	Move the pointer to the left
+	Increment the memory cell at the pointer
-	Decrement the memory cell at the pointer
.	Output the character signified by the cell at the pointer
,	Input a character and store it in the cell at the pointer
[	Jump past the matching] if the cell at the pointer is 0
]	Jump back to the matching [if the cell at the pointer is nonzero

Write a program in Brainf*ck that takes a sequence of 5 letters as input and outputs their reverse. What's the smallest number of characters you can use to code this program?

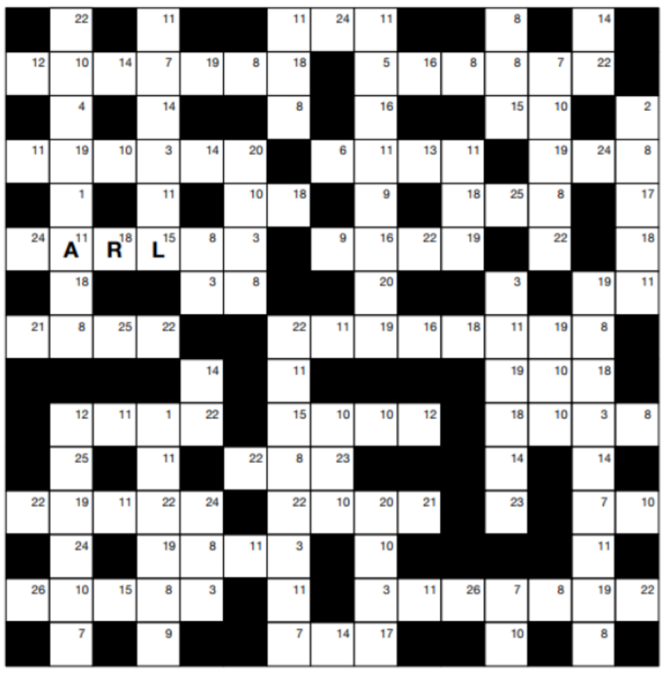
For more info on Brainf*ck check out
https://en.wikipedia.org/wiki/Brainfuck

CYPHER CROSSWORD

Decrypt the cipher at the bottom by completing the crossword. The letters here have been replaced by numbers, and both the key at the top and the cipher contain all letters of the alphabet.

1	2	3	4	5	6	7	8	9	10	11	12	13
										A		

14	15	16	17	18	19	20	21	22	23	24	25	26
	L			R								



14 7 | 22 12 11 20 8 | 7 10 | 10 7 8

20 11 7 | 24 8 11 18 | 25 10 16

22 20 18 8 11 3

TAXICAB NUMBERS

A Taxicab number is defined as the smallest integer that can be expressed as a sum of two positive integer cubes in n distinct ways.

The term comes from the famous conversation between renowned mathematicians Srinivasa Ramanujan and G.H. Hardy, in which Hardy remarked the number 1729 on a taxicab when going to visit Ramanujan in hospital. He said that 1729 seemed to be a rather dull number, to which Ramanujan replied 'No, it is a very interesting number. It is the smallest number expressible as the sum of two positive cubes in two different ways'.

Write a program that calculates the taxicab number which is expressible as the sum of two positive integer cubes in precisely four different ways. **Try extending the program to find all 6 known taxicab numbers (the largest taxicab number is 23 digits long!)**

BASED

Consider a base-62 number system which consists of all of the digits 0 to 9, all the letters of the English alphabet A to Z and all blackboard bold letters of the English alphabet 𝔸 to ℤ so 0 to 9 represent the numbers 0 to 9 in denary, A to Z represent the numbers 10 to 35 in denary and 𝔸 to ℤ represent the numbers 36 to 61 in denary.

Write a program that finds the greatest common factor of the three base-62 numbers below:

$$1\mathbb{E}\mathbb{T}689$$
$$\mathbb{N}ZD2X$$
$$123Z\mathbb{B}\mathbb{C}$$

SCROLLING MESSAGES

We've all seen on the buses and trains the scrolling text banners showing the next station that the train or bus will arrive at. They look something like this:

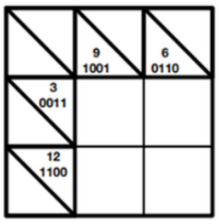
Can you implement this animation using just the developer console in an IDE?

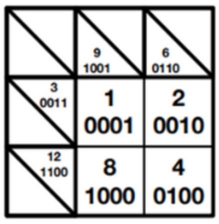
Write a program that will take a message as input and generate a scrolling banner. It should have the following features:

- Text moves from right to left
- The text consists of only dots
- All of the letters should be evenly spaced and perfectly aligned
- Capital letters should be the same height
- The banner should have a width of 60 dots and a height of 18 dots

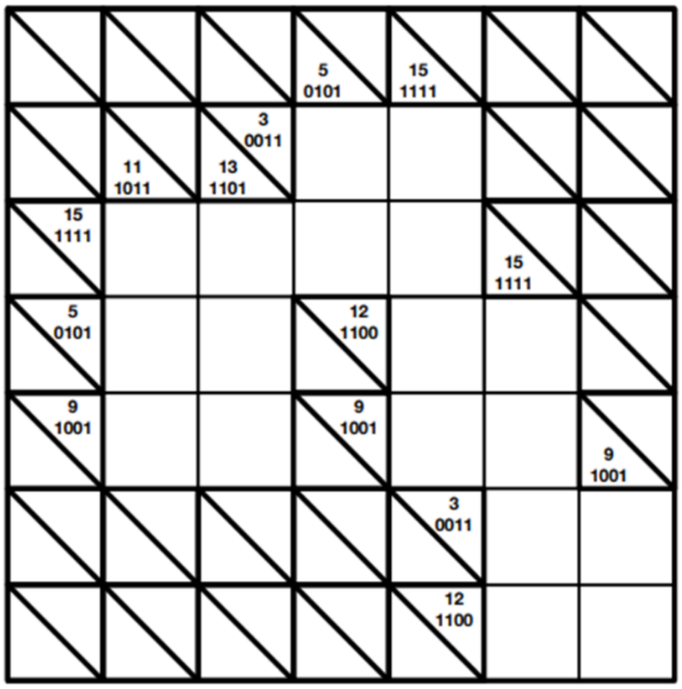
BAKURO

Bakuro is a puzzle in which empty cells of the grid below must be filled with powers of 2 such that the numbers in each block or column or row add up to the number given in the clue above or to the left respectively. No number can be used twice, and each cell must be filled in both binary and denary. For example, the Bakuro board below could be filled as follows:

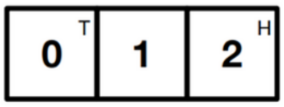


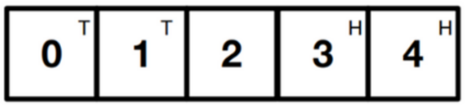


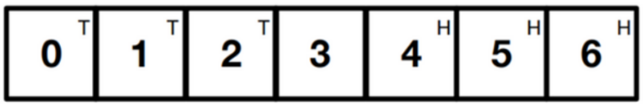
Complete the following Bakuro board



THE SWAP PUZZLE







On the grids to the left, place small coin heads up on the squares marked H and tails up on the tails marked T. Swap the positions of the Heads for the tails in as few moves as possible. You can move a piece left or to the right to an adjacent empty square or jump over a single adjacent piece into an empty square.

For the mathematicians among you, can you find and prove a formula for the smallest number of moves for grids of any length?

Beths
COMPUTING SOCIETY