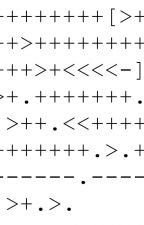
SKRIV JAVASCRIPT PÅ EN ENS MÅDE

Grafik med Brainfuck



Navn: Simon Lykke Andersen



Fag: Programmering C

Lærer: Anders Juul Refslund Petersen

Afleveringsdato: 5/4-2020

Afleveringssted: H.C. Ørsted Lyngby HTX, Lectio

Synopsis 3

Indledning 3

Projektformulering/problemformulering og kravspecifikation 4

Udviklingsmiljø 5

Runtime beskrivelse 7

Implementering 10

Program test 15

Konklusion 16

Bilag ( VIGTIGT - KODEN ) 17

# Synopsis

Brainfuck er et mest kendte esoteriske programmeringssprog. Esoteriske programmeringssprog er noget der er lavet for at være udfordrende, sjovt, interessant og anderledes at programmer i. Brainfuck kode består af 8 symboler (< > + - , . [ ]), derudover er der et bånd med ”uendelig” mange unsigned 8-bit heltal og en pointer til et bestemt sted på dette bånd. Symbolerne giver mulighed for at flytte på pointeren, manipulere båndet, input og output og de firkantede parenteser er en slags loop, der kører hvis værdien pointeren peger på ikke er nul. Mulighederne for hvilke slags programmer man kan lave med Brainfuck er ret begrænsede da man kun kan tage tekst som input og give tekst output, derfor ville jeg lave en måde hvor man kunne bruge nogle af funktionerne fra JavaScript biblioteket p5. For at gøre dette behøvede jeg at have en Brainfuck fortolker, der kunne eksekvere selve Brainfuck koden. Derefter integrerede jeg p5’s funktioner ved at funktionerne var repræsenteret som et tal (jeg har valgt at kalde opcode). Denne opcode skubber man til en stack, der eksisterer separat fra det øvrige Brainfuck økosystem, man skubber også argumenterne til disse funktioner. Jeg har udvidet Brainfucks spartanske symbolsæt med ”:” og ”;”, der henholdsvis er skub opcode og skub argument (i koden pushOPC og pushARG).

# Indledning

Inden jeg begyndte på dette projekt overvejede jeg nøje hvilke biblioteker og hvilket programmeringssprog jeg skulle anvende. Jeg vidste at jeg skulle anvende en eller anden form for bibliotek, der kunne rendere grafik da det var en af grundstenene i mit projekt. Jeg overvejede at anvende processing, da det var det vi havde anvendt i den hidtidige undervisning. Processing ville fungere fint til at udføre selve grafikdelen af mit program, men jeg ville også gerne have en intuitiv GUI til selve eksekveringen af mit program. Der findes biblioteker i processing, der hjælper til at kunne lave flotte GUI’er, men så skulle jeg også kunne lave endnu et vindue fra samme program, hvilket i mine undersøgelser så nært umuligt ud. Næste mulighed jeg overvejede var at lave det i python med pygame som den grafiske del. Jeg har selv arbejdet meget med python og pygame biblioteket, bl.a. til en TuringMaskine, hvor der var et tilhørende kommandolinje interface med mulighed for grafisk eksekvering i pygame. Dog ville dette kræve mere for at kunne køre da den mulige bruger skulle have python og pygame installeret. Sidste mulighed, hvilket var den jeg endte med at vælge, var at bruge p5.js biblioteket til Brainfucks grafiske del og almindelig Javascript til selve Brainfuck fortolkeren. Denne mulighed gjorde det muligt at bruge HTML til at lave GUI’en til selve eksekveringen.

Hvad vil jeg skrive her:

HVORFOR JAVAscript (fordi p5) og html, hvorfor ikke Processing? Fordi at det er nemmere at lave et brugergrænseflade.

Det skulle være blandede paradigmer objektorienteret (til selve interpreteren), men resten skulle stortset være imperativt

P5 skulle køres i instancemode, for man kunne integrerer det i programmet som en del man kunne fjerne.

OOP fordi det var noget af det vi havde lært om i skolen og det var passende da jeg havde en række data (tape, symbol m.m.) der skulle fungere sammen med nogle funktioner. Hvor jeg gerne ville kunne rykke hele molivitten på en gang (hvilket man så kan gøre ved bare at have en instans af objektet).

h

# Projektformulering/problemformulering og kravspecifikation

Projektformulering:

Lav en version af Brainfuck, der gør det muligt at lave programmer, der kan producere grafiske output, helst med mulighed for brugerinteraktion, i Brainfuck.

For at være sikker på at mit program i slutningen af dette projekt opnåede dette havde jeg defineret en række krav, der videre specificerede hvad det rent faktisk skulle til for at opnå målet.

Kravspecifikation:

* Fungerende Brainfuck fortolker
* Mulighed for at kalde tegne funktioner i Brainfuck
* Mulighed for at få bruger input i Brainfuck (keyboard og mus)
* Miljø til at kunne eksekvere koden i (med grafisk display)
* Miljø til at kunne gå trinvist gennem koden

# Udviklingsmiljø

I dette projekt har jeg anvendt Visual Studio Code som editor og hvor jeg løbende har testet min kode i Chrome, hvor jeg har anvendt deres udviklerværktøjer til at undersøge sidens elementer.

Visual Studio Code:

Et billede, der indeholder skærmbillede, monitor

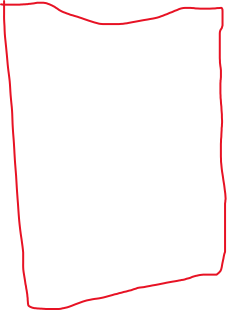
Automatisk genereret beskrivelse

Visual Studio Code er en kode editor udviklet af Microsoft, hvor det er muligt at downloade en lang række udvidelser, selvfølgelig syntax highlighting af en lang række sprog, men også mere komplekse ting. I dette projekt anvendte jeg det bl.a. til se en renderet version af min README.md.

Chrome:

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

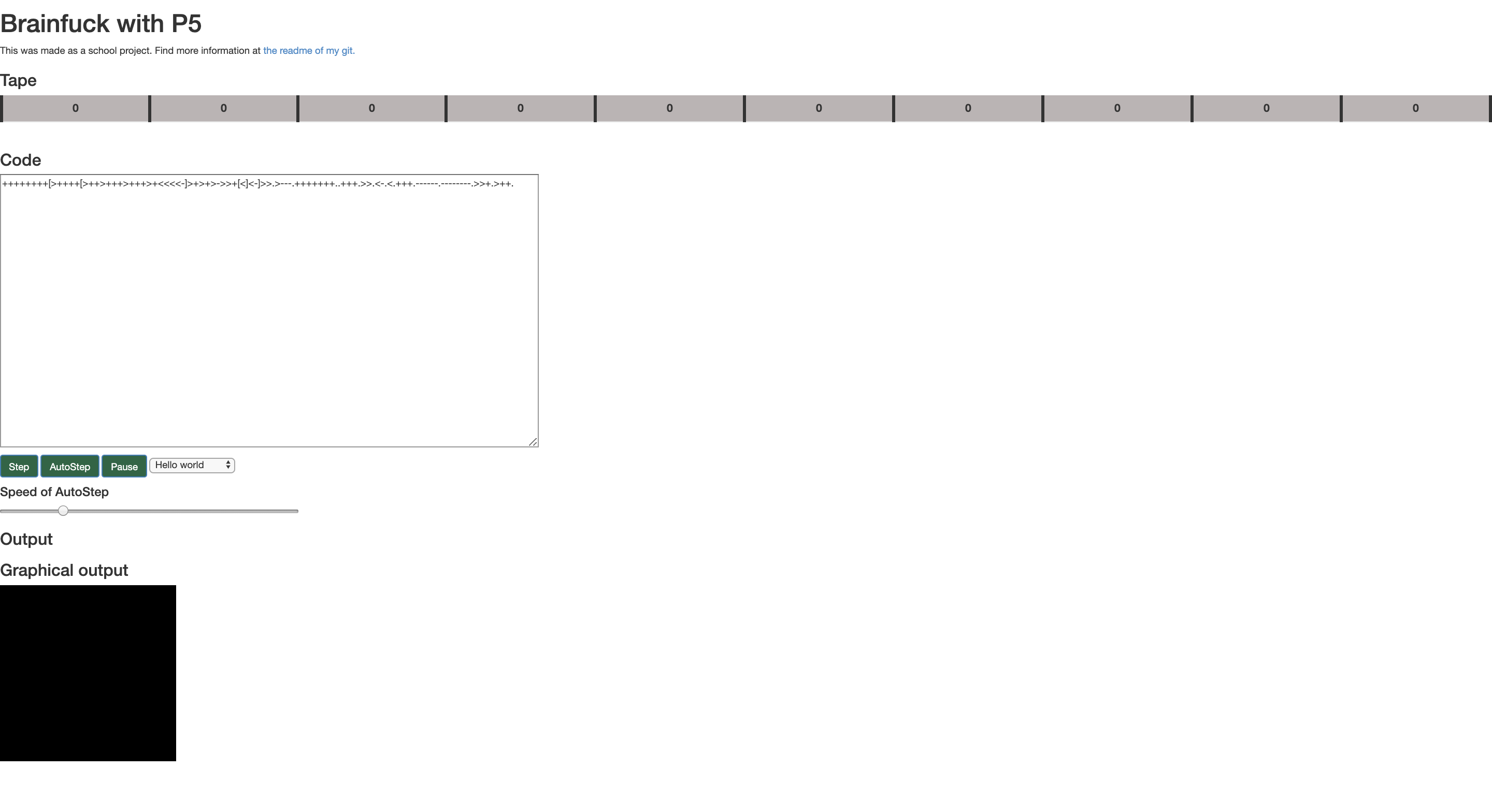


Chrome er en browser udviklet af Google og er en af de mest anvendte browsere i verdenen. Chrome er også en af de browsere med de bedste udviklingsværktøjer. Disse udviklingsværktøjer er bl.a. et værktøj, der gør det muligt at kunne finde den specifikke HTML kode, der står bag et element på en hjemmeside, ved bare at klikke/holde musen over dette element. Derudover er der en konsol, hvor JavaScript fejlmeddelelser kommer op. Denne konsol har jeg blandt andet også anvendt til traditionel debugging ved at udskrive værdier forskellige steder i koden ved JavaScript kommandoen ”console.log(…)”.

# Runtime beskrivelse

ET AFSNIT OM HVORDAN JEG SPECIFIKT HAR UDVIDET BF

Nedenunder ses selve hele siden. Jeg har prøvet at holde siden simpel for at gøre den så overskuelig som mulig. Jeg vil nedenunder gå igennem de forskellige delelementer.



Øverst på siden er der en lang bar, hvor der står en række tal. Dette er Brainfucks memory den såkaldte det jeg i synopsen kaldte båndet, hvilket jeg herfra vil kalde tape. Når man kører et program, bliver pladsen (mere specifikt den grå baggrund) på tapen man befinder sig på rød. Som ses på billederne nedenunder. Hvis pladsen der peges på, er udenfor de ti (der er lavet) sættes pladsen som værende i midten, hvor tallene så bare skifter omkring pladsen.



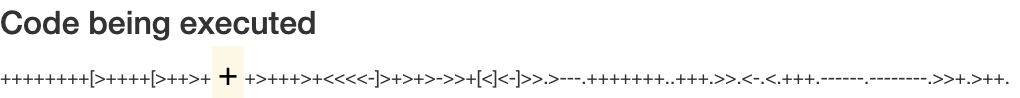


Nedunder tapen er et tekstvindue, hvor brugeren kan redigere koden.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

Under tekstvinduet er et afsnit, der under eksekvering, får samme tekst som i tekstvinduet ovenover, men det symbol, der eksekveres i det øjeblik, bliver highlightet og gjort større.



Under afsnittet med det highligtede symbol er en række knapper, en dropdown menu og en slider. Den første knap ”Step” kører et enkelt step, den eksekverer altså et symbol. Den anden knap ”AutoStep” gør sådan at Step funktionen kaldes ved et fast interval som bestemmes ved værdien af slideren. Den tredje knap ”Pause” sætter programmet på pause, dette er kun nødvendigt ved AutoStep. Herefter har vi en dropdownmenu med en række eksempel programmer, hvor ”Hello World” programmet er valgt som default. Det er kun programmet ”Cirkler ved mus” og ”Hus” som anvender det grafiske output. Slideren bestemmer tiden mellem hver gang AutoStep bliver kaldt, hvor tallet er ms mellem Step funktionen bliver kaldt.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

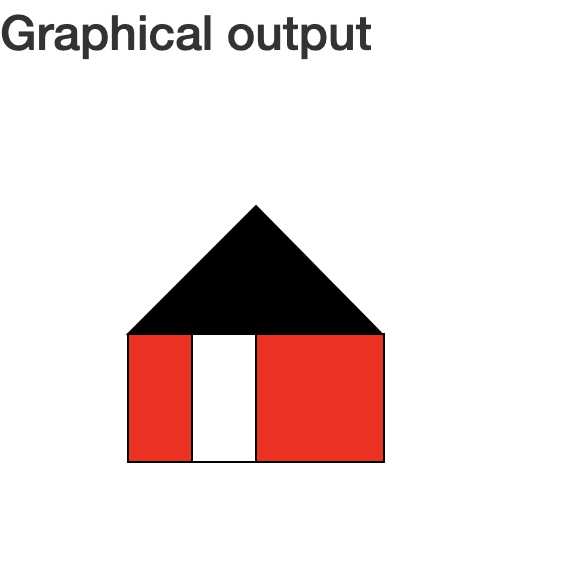
Et billede, der indeholder skærmbillede, elektronik

Automatisk genereret beskrivelse

Under knapperne m.m. er tekst outputtet som et afsnit. Her er hvordan outputtet ser ud efter ”Hello World” programmet har kørt.



Under tekst outputtet er det grafiske output. Dette er vinduet, hvor udfaldet af de grafiske funktioner kaldt fra Brainfuck vil kunne ses. Her er vinduet efter programmet ”Hus” har kørt.



# Implementering

Overordnet set består mit program af to dele. Den ene del er det man ser på selve siden altså GUI’en/brugerfladen til eksekveringen af Brainfuck koden, den anden del er det den bagvedliggende del, der gør den første del muligt altså selve fortolkeren. Jeg vil inkludere det grafiske tillæg til Brainfuck i den anden del selvom det i koden står ved den første del. Jeg vil starte med at beskrive den anden del.

## Bagvedliggende del

### Overblik over klasserne

Delen hvor selve fortolkeren og den grafiske del af Brainfuck består af en række klasser, objekter og hvordan disse interagerer. Dette har jeg beskrevet ved klasse diagram. Som ses nedenunder.

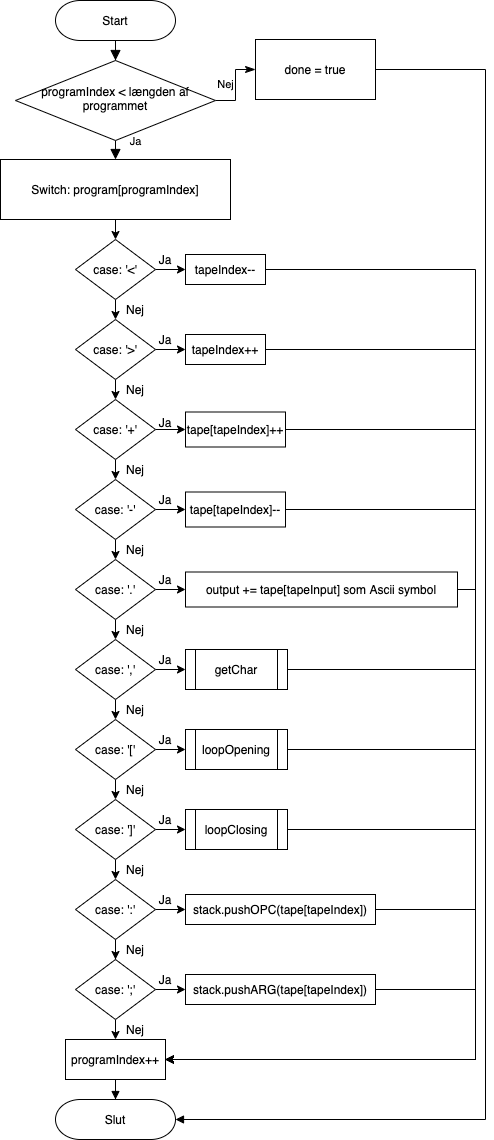
Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

Dette giver et overblik over hvordan de forskellige bestanddele interagerer med hinanden. Jeg har forsøgt at anvende UML’s konventioner omkring pilenes form. Men jeg vil lige gennemgå hvad diagrammet rent faktisk betyder. Vi har en klasse Interpreter som er selve Brainfuck fortolkeren, men uden muligheden for at anvende de grafiske funktioner. Denne klasse har to metoder, step og eval, step funktionen kører kun et step (altså håndtere et symbol), hvor eval kører til programmet er færdigt. Så er der en klasse InterpreterWithStack som er nedarvet fra vores Interpreter klasse. Denne klasse har endnu et medlem med navnet stack som er en instans af Stack klassen (som jeg vil forklare mere om senere). Den har også to metoder (med samme navn som superklassen), men som er redefineret for at kunne håndtere to ekstra symboler nemlig ”:” og ”;”, hvilket er dem der fra Brainfuck interagerer med det grafiske (altså stacken). Så er der så Stack klassen, stack klassen er den som står for den grafiske del, den kalder nemlig p5 funktionerne når den nøvendige information er der. Stack klassen indeholder en instans af den interpreter, den selv er en instans hos. Det gør den fordi der er nogle opcodes, der ”returnerer” en værdi som bare bliver lagt på den plads på tapen fortolkeren er lige nu. Derudover har den et andet medlem med navn stack som er det, der indeholder de kaldte opcodes og de tilhørende argumenter. Så er der de to metoder pushOPC og pushARG, der står for at sætte de nødvendige informationer på stacken og at eksekvere tegnefunktionerne når det nødvendige antal argumenter er givet. Til sidst er der klassen p5, hvor der anvendes en instans af p5 til at kunne tegne med. Da p5 klassen har super mange medlemsvariabler og metoder har jeg valgt ikke at specificere dem yderligere, vi anvender kun dem, der kaldes direkte fra Stack klassen.

### Step metode

Med dette generelle overblik over hvordan de forskellige klasser interagerer vil jeg nu vise en af de mest anvendte metoder, nemlig InterpreterWithStack’ens step metode. Jeg har lavet et flowchart over Step metoden. Step metoden eksekverer et symbol af programmet. I dette flowchart anvender jeg en del prædefinerede processer, disses flowcharts kan alle findes i bilag[HVILKE BILAG]. I step metoden anvendte jeg et switch case, hvilket var en smule udfordrende at afbildede i flowchartet, da switch caset i virkeligheden laver et hashtable over de mulige værdier, i stedet for at gå dem slavisk igennem som var det en række if-else if… statements som jeg har afbilledet det som.



### Stack

I ovenstående flowchart er det eneste, der ikke laver om klassens egen værdier, når der skubbes ting på stacken. Jeg vil nu forklare hvad denne stack er og hvordan den fungerer. Som også sås i UML diagrammet har jeg lavet en Stack klasse. Jeg vil først påpege at denne Stack klasse er ikke det samme som forstås ved en normal stack, der er en veletableret datatype, men mere en specifik anvendelse af sådan en. I moderne højniveau programmeringssprog er der ikke en separat stack datatype, men derimod metoderne push og pop, man kan anvende på lister for at få samme resultat. Dette er sådan en liste, der er gemt i Stackens stack medlemsvariabel. I denne liste skubber vi opCodesne med en tilhørende liste til argumenter, når der skubbes en opCode. Hvis der skubbes et argument sættes det ind i det øverste elements argumentliste. Hver gang der skubbes en opCode eller et argument tjekkes der om det nødvendige antal argumenter er tilstede. Hvis det nødvendige antal argumenter er tilstede eksekverer den opCodens tilhørende funktion med de skubbede argumenter. Derefter popper den opCoden med argumentlisten af stacken. Nedenunder ses en repræsentation af hvad der blev beskrevet ovenover.

Et billede, der indeholder skærmbillede

Automatisk genereret beskrivelse

## Brugergrænseflade

Nu er vi nået til den del af programmet, der rent faktisk gør det muligt at køre sit program osv. nemlig brugergrænsefladen. Da denne brugergrænseflade er en hjemmeside i HTML, kan vi inddele brugergrænsefladekoden i tre dele. Den første del er det der står for de elementer man ser på siden, det der identificeres som traditionel HTML, den anden del er den Javascript kode, der står for en stor del af funktionaliteten bag disse elementer og den tredje del er den, der sørger for udseendet af de elementer man ser på siden, hvilket er beskrevet i noget kaldet Stylesheets hvor internettets Stylesheet standard er CSS.

### HTML

UDFRA MINE KRAV SPECIFIKATIONER VIDSTE JEG HVILKE GUI ELEMENTER SIDEN SKULLE HAVE

Hvilke elementer anvendte jeg, tekst (overskrifter, afsnit…),tabel, textarea, knapper, dropdown menu, slider… og så en div klasse hvor mit p5 vindue var placeret.

### Javascript

Forklar hvordan man interagerer med HTML elementerne gennem javascript (document.gelElementByID) hvordan det stod for at opdatere både tabellen, og <p> output tingen. Og hvordan du brugte button triggere m.m. til at opdatere og sådan (INNERHTML….). Derudover hvordan man anvendte set interval til at kalde en funktion som både ville kører et step i fortolkeren og så opdaterer alle elementerne.

### CSS

Bootstrap plus lidt af mit eget (i form af de to ekstra ting jeg fik skrevet.)

For at gøre det mere overskueligt valgte jeg også at skrive noget af CSS’en direkte i html elementerne

Et billede, der indeholder tekst, skilt, sort, grøn

Automatisk genereret beskrivelse

Inden jeg begynder at forklare disse i højere grad vil jeg nævne en tredjedel som jeg vælger ikke at gå mere ind i. Denne tredjedel er det der er kendt som Stylesheet (CSS), hvilket er det der specificere hvordan de enkelte bruger elementer skal se ud. Grunden til jeg ikke vælger at fortælle i længere drag om Stylesheet er fordi jeg anvendte et meget bredt anvendt CSS-Framework kaldt Bootstrap. Jeg anvendte den mindste version af Bootstrap-der

For det første hvordan: Noget javascript og noget html (hvor der er noget integreret javascript i html (både stacken og p5.js), men så også en masse ting der bliver kaldt på siden og står for at gøre hjemmesiden responsiv, der kan jeg vise et eksempel (nok step funktionen fordi jeg gennemgår den dybere senere hen))

Selve løsningens implementation:

* UML over klasserne
* Flowchart over step funktionen

Hvordan det kaldes fra html:

* Step (hvordan det bruger step)

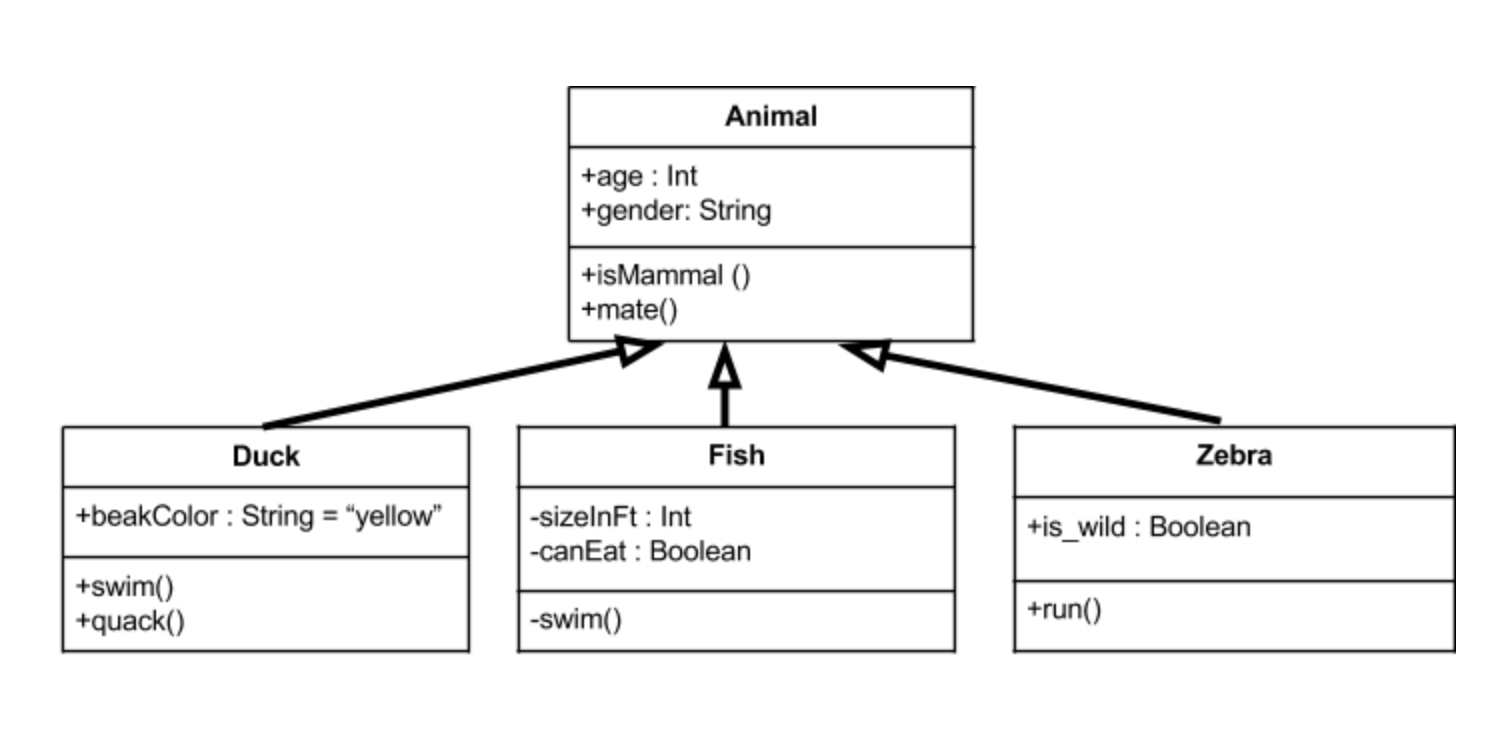
Overordnet struktur af programmet (flowchart el lignende pseudokode)   
Udvalgt dokumentation af selve koden i programmet   
Detaljeret dokumentation af dele af programmet som er særlig interessant.

FORKLAR OM HVORDAN P5 og stakken taler sammen

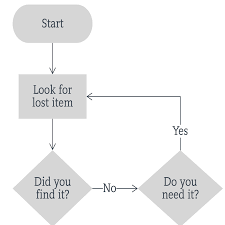
JEG VED GODT DER FINDES NOGET DER HEDDER EN STACK.

F.eks:

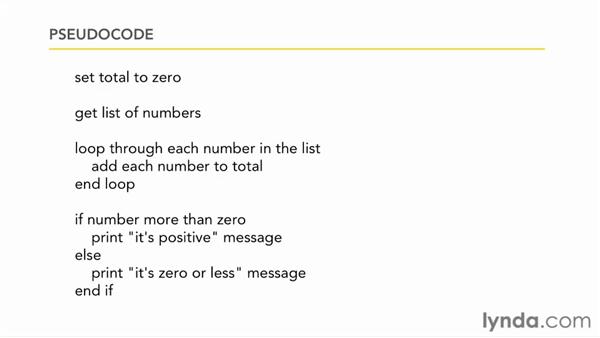
Du kan f.eks. vælge nogle af følgende dokumentations metoder afhængig af jeres projekt:



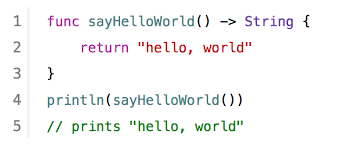
Figur 1: Klassediagram ( kan kun bruge til oo arkitektur )



Figur 2: Flowdiagram ( kan bruges til det meste )



Figur 3: Pseudo kode , godt til at illustrere programmerings-tankegang og forenkling af problemstilling



Figur 4: Direkte udvalgt kode, kan ofte være nødvendigt

# Program test

Mit program består af en mange dele, der hver især har forskellige måder at teste på.

TEST AF BF

Hver af dem:

+ - < > . ,

[] (forklar hvordan de skal testes individuelt)

[-] meget anvendt feature

Flere loops indeni hinanden

TEST AF P5 i intancemode

TEST AF STACK

Prøv hver af opcodesne

Prøv hver af opcodesne i Bf

TEST AF GUI hver af knapperne også videre. (Om funktionerne der kaldes af knapperne gør det vi forventer)

Hvad vil jeg skrive her.

Til brainfuck: Der er eksempel programmer ude på internettet. Hello World, og test af de individuelle browsere

Test af programmet, dvs. kørsler af programmet, hvor du tester (væsentlige) dele af programmet. Skriv gerne hvad du vil teste og hvordan testen gik.

F.eks:

Lav f.eks. det man i industrien kalder en UA test (slå det op hvis du er interesseret) Det går ud på du tester om programmet kan det der står i kravspecefikationen, ganske enkelt. Test hvert enkelt krav for sig.

# Perspektivering

Hvad programmet ellers skulle have kunnet. Muligvis hvor man kunne poppe elemeneter fra stacken så at det var muligt at få mere komplekse operationer til at ske. Derudover en måde at kunne lave tekst, på stacken og en måde at kunne lave prædefinerede metoder. Så det blev mere i stil af Forth (selvom det så ville være omvendt notation af det jeg ellers havde anvendt, men brugbarheden af det kunne også overvejes, da det ville være nemmere da man så bare kunne loade en masse værdier på)

# Konklusion

+ perspektivering af program (hvis jeg arbejdede videre på mit program ville jeg…)

# Bilag ( VIGTIGT - KODEN )

Meget vigtigt

Bilag – udskrift af kildekoden. Sørg for at der er kommentarer i kildekoden.

Koden

Flowchart predefined processes.