SKRIV JAVASCRIPT PÅ EN ENS MÅDE

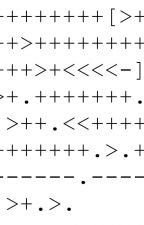
TJEK AT BILLEDERNE SIDDER HVOR DE SKAL

HUSK FIGURTEKSTER

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 positive 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. Jeg ville derfor lave en måde hvor man kunne bruge nogle af funktionerne fra JavaScript biblioteket p5.js, hvilket er et Javascript bibliotek, der gør det muligt at tegne og interagerer med et canvas. For at gøre dette implementerede jeg 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, kort for operation code). 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.

# 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, hvilket er det man på GitHub bruger til at fortælle om sit projekt, hvilket jeg i starten brugte som en notesbog over projektet.

Chrome:

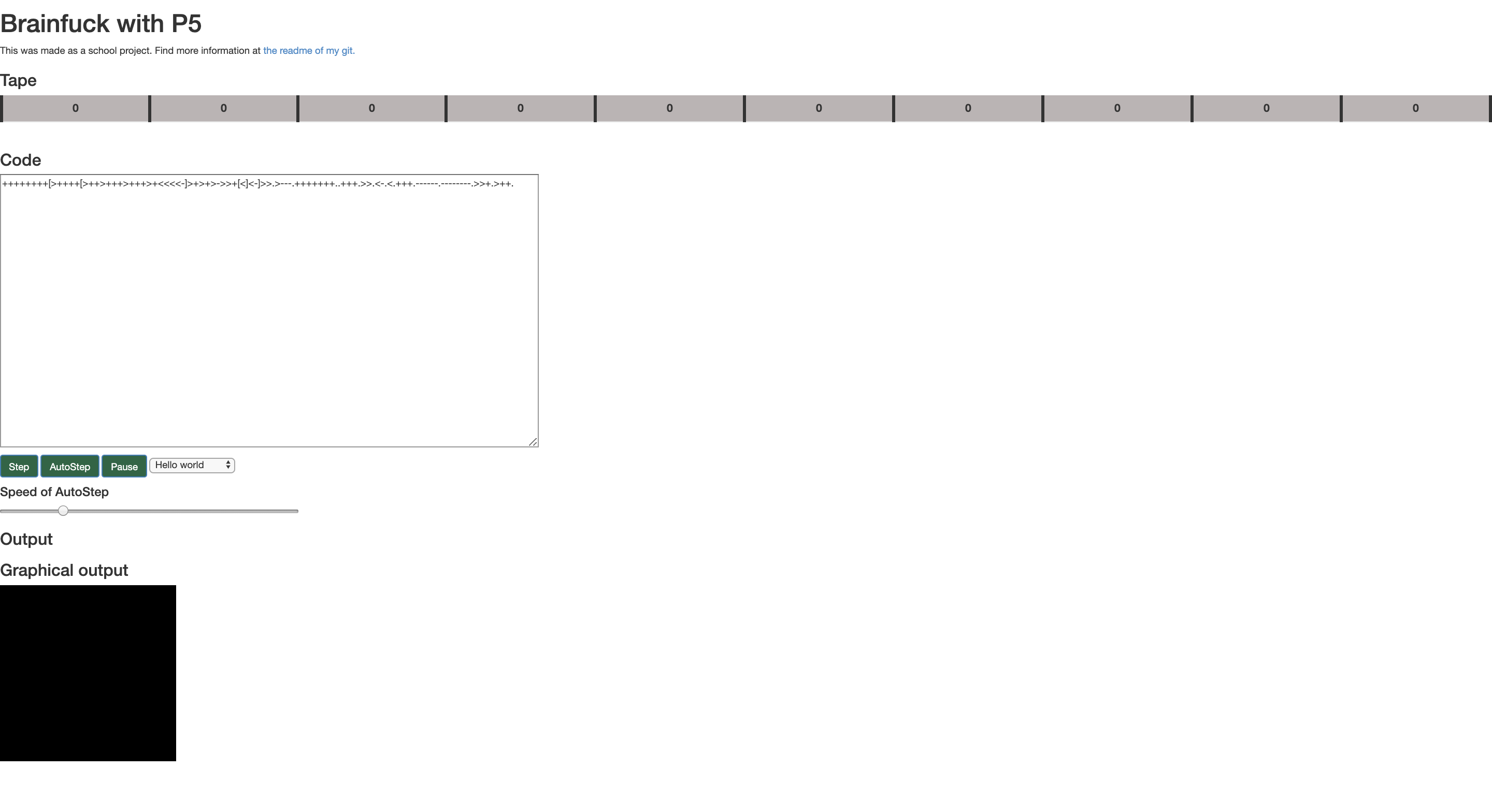
Et billede, der indeholder skærmbillede

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

Nedenunder ses selve brugergrænsefladen til fortolkeren. 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.



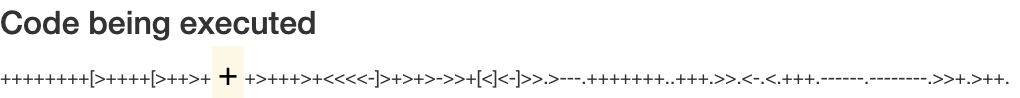


Nedunder tapen er et tekstvindue, hvor brugeren kan redigere koden.

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

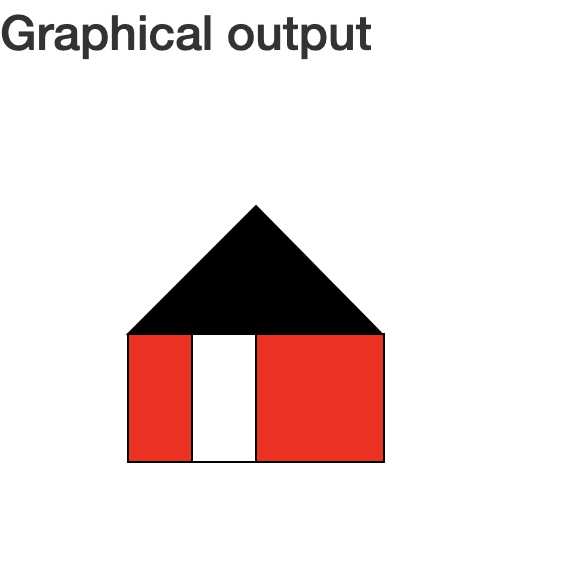
Et billede, der indeholder skærmbillede, elektronik

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.

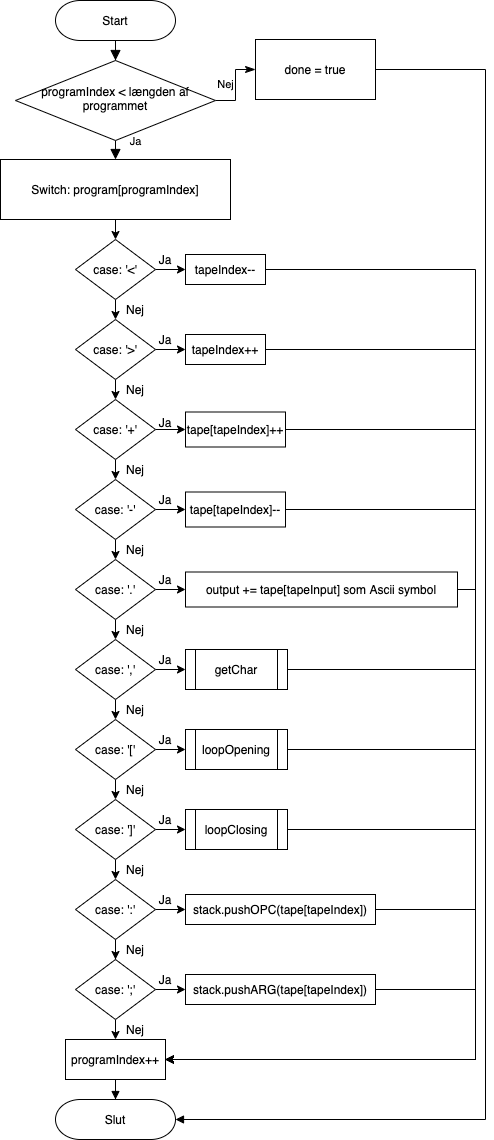
Et billede, der indeholder skærmbillede

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 på 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

Før jeg begyndte at skrive HTML koden fik jeg mig et overblik over hvilke brugergrænseflade elementer jeg behøvede. Herfra var de to største krav, der berørte brugergrænsefladen, henholdsvis kravet om at programmet skulle indeholde et ”Miljø til at kunne eksekvere koden i (med grafisk display)” og ”Miljø til at kunne gå trinvist gennem koden”. Selvom disse krav var relativt åbne, havde jeg en vis forventning om hvad dette indebar. Ved det første krav mente jeg et IDE lignende miljø, hvor man både kunne skrive sin kode, se eksempler, naturlig integration af Brainfucks traditionelle input-output såvel som det grafiske vindue hele mit projekt omspinder. Til det andet krav var det selvfølgelig at der i dette miljø til eksekvering skulle være en måde at gå trinvist igennem koden, men dette var dog ikke alt hvad jeg så nødvendigt. Det man ofte har brug for nå man går trinvist igennem kode er at se hvordan værdierne ændrer sig ved hver operation og se om de gør som man havde forventet. De eneste værdier der er tilgængelige i Brainfuck er tapen, hvilken jeg så valgte skulle indgå i min brugergrænseflade.

For at opnå disse krav og tanker jeg havde til min brugergrænseflade, krævede det at jeg anvendte en lang række HTML elementer. Der var selvfølgelig de klassiske tekstelementer som overskrifter (<hx></hx>) og afsnit (<p></p>), hvor gjorde nogle fra den sidste gruppe interaktive. Udover disse anvendte jeg også et tabelelement til min tape, da der i Brainfuck kun anvendes en endimensionel tape definerede jeg kun en række med ni kolonner, hvilket skulle repræsentere min tape. Indholdet af denne tabel skulle kunne laves om af Javascript koden, der forklare i næste sektion, derfor var det nødvendigt at give hver af kolonnerne systematiske navne (t0-t9).

Et billede, der indeholder telefon, computer, monitor, skærm

Automatisk genereret beskrivelse

Udover tabellen anvendte jeg også HTML elementet textarea som værende hovedinputfeltet hvor brugerens kode skulle stå og kunne redigeres i. Til at kontrollere selve eksekveringen af programmet anvendte jeg knapper, en slider og en dropdownmenu til eksemplerne. Knapperne blev lavet ved <button></button> tagget, hvor deres funktionalitet var beskrevet/kodet i Javascript udenfor kroppen af HTML’en. I HTML er der et input element, indenfor dette input element er der en lang række forskellige slags input elementer, hvor det ønskede element specificeres i en type deklaration inden i tagget. Det specifikke input element jeg ønskede til min slider, er af en type kaldet range, hvor man så specificerer mellem hvilke værdier denne slider skal kunne være og i hvor store trin man kan ændre på dette. Da min slider skulle repræsentere tiden mellem at fortolkerens stepfunktion blev kaldt, hvilket skulle angives i millisekunder, valgte jeg at min slider skulle repræsentere det samme, hvor brugeren kunne vælge alle værdier mellem 0 m/s og 500 m/s med 1 m/s trin. Den resulterende HTML kode så således ud:

Til dropdown menuen med eksempel programmer anvendte jeg select elementet, hvor hver af dropdown menuens muligheder blev specificeret imellem select elementets åbnings og luknings tag med et option tag. Da disse options også skulle indeholde selve koden af eksempel programmerne er dette ikke det pæneste HTML, men herunder ses HTML koden til dropdownmenuen:

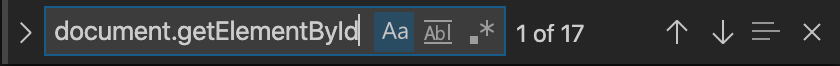
Et billede, der indeholder skærmbillede, monitor, sidder, bord

Automatisk genereret beskrivelse

Det sidste HTML element jeg anvendte, var til specificeringen af hvor p5.js vindue skulle være. Dette blev ved en div klasse, hvilket er en mode at organisere ting i HTML på, med et specifikt id, der blev givet til p5.js instansen.

### Javascript

For at gøre disse elementer interaktive anvendte jeg Javascript. Det jeg anvendte for at Javascript havde adgang til HTML’en er gennem Javascript document objekt. Document objektet i Javascript symboliserer den HTML side Javascripten indgår i, man kan ved dette objekt for adgang til alt i HTML, ved at bruge document objektets metoder, en af de metoder jeg anvendte mest hyppigt var document.getElementById, hvor man kalder metoden med en streng med Id’et på det ønskede HTML element. Derfra kan man så tilgå alle HTML elementets attributter, hvor dem jeg hyppigst tilgik fra HTML elementets value attribut. Jeg anvendte denne metode både til at hente data fra HTML’en, men også for at ændre på den. Denne metode blev anvendt 17 gange i mit program:



Ovenstående brugtes til at få information fra HTML elementerne, men for at få siden til at reagere på klik på knapperne og lign. anvendte jeg HTML elementernes egne onclick eller onchange deklarationer, hvor jeg derfra kaldede en Javascript funktion, der gjorde de ønskede ændringer. Derudover anvendte jeg også en Javascript funktion kaldet setInterval til AutoStepthrough funktionens funktionalitet. setInterval funktionen virker ved at man giver den et interval (tid mellem hvert kald) og en funktion så sørger den for at kalde den funktion i den angivne frekvens.

### CSS

Til CSS delen anvendte jeg et bredt anvendt CSS-framework kaldt Bootstrap. Bootstrap er i bund og grund en lang CSS fil, de har dog lavet et let overskueligt system til at kunne lave hjemmesider, der virker godt på både mobil og computer. Jeg anvendte langt fra Bootstrap til dets fulde potentiale, men udover de basale formateringer anvendte jeg en af deres CSS-klasser til min tape, hvor jeg anvendte div klassen ”table-responsive”. Derudover lavede jeg nogle få udvidelser både i en separat CSS-fil kaldet styles.css, der ligger i data/style folderen sammen med den øvrige CSS, som indeholder følgende:

Et billede, der indeholder skærmbillede, sort, skærm, rød

Automatisk genereret beskrivelse

Derudover skrev jeg også en smule ”inline” CSS nemlig HTML taggets style deklaration. Herunder ses hvordan jeg anvendte dette til min knapper.

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

Automatisk genereret beskrivelse

# Program test

Under hele udviklingen sørgede jeg for at teste hvad jeg var i gang med. Jeg vil kort gennemgå hvilke og hvordan jeg testede forskellige dele af mit program. Til testen af frotolkeren testede jeg først hver af metoderne i klassen selv, derefter da fortolkeren var færdig sørgede jeg for at teste hver af funktionerne igennem og se om det forventede resultat kom ud. For alle undtagen ”[” og ”]” var dette relativt lige til, men for disse prøvede jeg flere af de mulige situationer. Da jeg havde testet om alle delene virkede, kørte jeg en række BF eksempelprogrammer for at se at den også virkede på større programmer. Ift. det grafiske startede jeg med at teste p5.js tegne funktioner direkte i koden. Derefter prøvede jeg at kalde samme funktioner for en af de andre filer, hvilket afslørede en bug, der gjorde jeg skulle lave en smule om på min arkitektur. Da jeg havde testet tegnefunktionerne hver for sig, gjorde det det nemmere at teste stacken da jeg vidste fejlene associeret med de test hovedsageligt ville omhandle Stacken. Til Stacken sørgede jeg for at teste hver af opcodesne først gennem kode direkte og derefter gennem BF fortolkeren. Den sidste del af programmet nemlig brugergrænsefladen testede jeg løbende ved bare at trykke på knapperne og se om det jeg forventede ville ske skete og ellers debugge.

# Konklusion

Da mit program lever op til alle mine krav og kommer igennem mine test, vil jeg sige at projektformuleringen er lykkedes. Det er den fordi mit program gør det muligt at lave BF programmer, der producere interaktiv grafik. Dog er der en lang række forbedringer/udvidelser jeg kunne forestille mig. Disse er følgende: understøtte alle p5.js tegnefunktioner, understøtte tekst i Stacken, mulighed for at definere tegnesekvenser (der kunne få sin egen opcode) og sidst udvide stacken til en mere Forth lignende stack, hvor at resultater ville gemmes på stacken, hvor de så enten kunne poppes af eller blive som argument. Dette ville nok også kræve at man lavede om på argument/opcode rækkefølgen, men ville gøre det langt mere udtryksfuldt. Dog ville mange nok ikke anvende dette da BF ikke er et særlig venligt sprog i det hele taget.

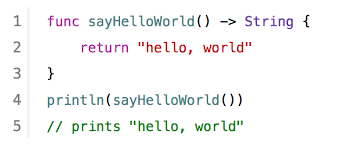
# Bilag ( VIGTIGT - KODEN )

Meget vigtigt

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

Koden

Flowchart predefined processes.



Figur 4: Direkte udvalgt kode, kan ofte være nødvendigt