
Visual Minecraft Programming



27. MAJ 2015

Gruppe SW408F15

Aalborg Universitet



TITEL:
Visual Minecraft Programming

PROJEKTPERIODE:
02-02-15 til 27-05-15

PROJEKTGRUPPE:
SW408F15

GRUPPEMEDLEMMER:
Alexander Kaleta Jensen
Andi Rosengreen Kjærsig Aaes
Niklas Ulstrup Larsen
Kaare Bak Toxværd Madsen
Malthe Dahl Jensen
Thomas Bjelbo Thomsen

SEMESTER:
4. Semester

VEJLEDER:
Claus Skaaning

Oplagstal: 8
Sideantal: 102
Bilagsantal og -art: 23 sider
1 stk. DVD
Afsluttet den: 27-05-2015

SYNOPSIS

Denne rapport omhandler udviklingen af et nyt programmeringssprog, til computerspillet Minecraft. Programmeringssproget skal gøre det lettere at bygge store og sjove konstruktioner i Minecraft, og er specifikt målrettet til børn, unge og personer uden erfaring med programmering. Rapporten analyserer andre programmeringssprog designet specifikt til børn, samt beskriver computerspillet Minecraft og det benyttede plugin ScriptCraft.

Rapporten beskriver nogle af de designvalg, der er blevet truffet for at udvikle dette sprog. Den indeholder desuden en overordnet beskrivelse af en compilers forskellige dele, funktioner og formål. Derudover bliver implementeringen grundigt beskrevet, med de forskellige dele. Herunder valg af en compiler generator, scope rules, type checker og code generator.

Endelig bliver der udført kvalitetssikring af sproget og den tilhørende compiler, i form af fejltest.

Link til compiler:
<https://www.dropbox.com/sh/lgkmnop1ee140j/AAA6BDxDsUWKjKzdW6EJ4h-ta?dl=0>

Underskrifter

Alexander Kaleta Jensen

Andi Rosengreen Kjærsig Aaes

Kaare Bak Toxværd Madsen

Malthe Dahl Jensen

Niklas Ulstrup Larsen

Thomas Bjelbo Thomsen

Forord

Denne rapport er resultatet af et 4. semesters projekt, udarbejdet af 6 studerende på Softwarestudiet på Aalborg Universitet. Projektet startede den 2. februar 2015 og blev afsluttet den 27. maj 2015. Projektet er udarbejdet ud fra emnet "Design, definition og implementation af programmeringssprog", hvor der er arbejdet videre med en specifik problemstilling. I forbindelse med projektet er der udviklet et programmeringssprog på baggrund af ScriptCraft. Projektet er udarbejdet under vejledning af Claus Skaaning.

I rapporten vil der blive brugt `monospace` til visualisering af funktioner, metoder og variabler. Til kilder vil der blive brugt Vancouver metoden, hvor kilder angives med et nummer i klammer (eksempelvis [2]), og kommaseparation, hvis der er flere kilder i samme klammer. Kilder tilhørende en linje sættes indenfor punktum, og kilder hørende til et helt afsnit skrives efter sidste punktum.

Dette projekts sprog har vi valgt at kalde for *Visual Minecraft Programming*, hvilket i løbet af rapporten vil blive refereret til som VMP.

Projektet benytter et plugin til Minecraft ved navn ScriptCraft, som kan køre JavaScript-kode. Når noget omtales som ScriptCraft-kode vil det betyde at det er JavaScript-kode med ScriptCraft-udvidelsen. I løbet af rapporten vil udtrykkene blok og block begge blive benyttet. Blok refererer til blokke i Minecraft, mens at block refererer til typen i VMP.

Der kan i bilag C.0.4 findes en tutorial til det udviklede programmeringssprog VMP, der gennemgår opsætningen af compileren samt hvordan sproget kan benyttes.

Denne projektgruppe er ikke associeret med Mojang, som er skaberne af Minecraft.

Indholdsfortegnelse

Kapitel 1 Problemanalyse	1
1.1 Indledning	1
1.2 Minecraft	1
1.3 Børn skal lære at programmere	2
1.4 Børn og programmering	3
1.5 Problemdefinering	7
Kapitel 2 Sprogdesign	8
2.1 Sprodkriterier	8
2.2 ScriptCraft	10
2.3 Designvalg	13
Kapitel 3 Compilerdesign	20
3.1 Scanner og Parser	22
3.2 Type checker	37
3.3 Opsamling af compilerdesign	47
Kapitel 4 Implementering	48
4.1 Compiler generatorer	48
4.2 SableCC implementering	54
4.3 Scope	63
4.4 Type checker	66
4.5 Code Generator	68
4.6 Optimering	76
4.7 Grafisk brugerflade	86
Kapitel 5 Test	88
5.1 Test	88
5.2 Fejlhåndtering	93
5.3 Opsamling af test	96
Kapitel 6 Opsamling	98
6.1 Konklusion	98
6.2 Fremtidigt arbejde	99
Bibliografi	100
Bilag A Bottom up, rightmost derivation	103

Bilag B SableCC grammar	106
Bilag C Vejledning i at bruge VMP	109
Bilag D maltheCastle() figures	122

Problemanalyse 1

1.1 Indledning

Minecraft er et computerspil, hvor spilleren kan bygge alt hvad han eller hun har lyst til, men det kan ofte være tidskrævende at bygge konstruktioner som spilleren har forestillet sig. I dette projekt ønsker vi derfor at designe og udvikle et nyt programmeringssprog, som gør det muligt at konstruere og bygge nye ting i Minecraft, både hurtigere og lettere end hvis spilleren selv skulle bygge konstruktionen blok for blok.

Programmeringssproget vil samtidig introducere børn til programering, så de lærer hvordan programmeringslogik fungerer. Programmering er oftest anset som værende et kedeligt værktøj, men ved at sammenkoble programmering med et spil som Minecraft, er der mulighed for at børnene bliver motiverede til at lære programmeringslogik.

Der vil derfor blive analyseret andre programmeringssprog, henvendt specifikt til børn, samt beskrevet plugin til Minecraft der gør det muligt at bygge vha. kode skrevet i ScriptCraft.

1.2 Minecraft

Minecraft er et spil, som blev skabt af svenskeren Markus Persson i 2009, skrevet i programmeringssproget Java, og nu ejet af Microsoft. Spillet kan spilles på to måder; enten som et overlevelsesspil, hvor det handler om at finde materialer for at kunne bygge huse og forsvarssystemer, for så at kunne overleve natten fra monstre. Eller som et åbent spil, hvor det blot handler om at skabe og bygge hvad man lyster. Minecraft kan spilles i multiplayer, hvilket betyder at det er muligt at spille med andre spillere, såsom sine venner, hvor man kan vælge at samarbejde om at bygge en stor konstruktion, eller konkurrere mod hinanden. For mange spillere er hovedformålet mere at bygge og skabe, end at overleve. Spillet er af genren sandbox, hvilket vil sige at spilleren får nogle muligheder stillet til rådighed, og så er det op til spilleren at bruge mulighederne til hvad end spilleren vil. I den tid Minecraft har været udgivet, har spillere konstrueret alt lige fra træhuse, til jernbanesystemer, til en fuld funktionel 16 bit computer. Geodatastyrelsen har endda bygget Danmark i størrelsesforhold 1:1 i Minecraft, som ligger frit tilgængeligt på deres hjemmeside, og som fungerer som et godt eksempel for at vise at der ikke er nogen grænser for hvad man kan bygge i Minecraft [1]. Dog kan det tage rigtigt lang tid, når man skal

bygge store konstruktioner som en by eller lignende, eftersom man normalt kun kan bygge en blok af gangen. Derudover er der muligheden for at installere mods, som kan give spilleren endnu flere muligheder.[2]

Spillets design er meget unikt, da verdenen er bygget op ad blokke, hvor hver blok er samme størrelse, og en blok har størrelsen én kubikmeter. Måden man bygger i spillet er at sætte blokkene sammen, som minder meget om måden man bygger med LEGO klodser. Maksimum størrelsen af en Minecraft verden er omkring 921.600×10^{12} blokke og har et overfladeareal som er 9.258.235 gange større end Jordens[3].

Minecraft har en brugerbase på næsten 20 millioner spillere og kan spilles på PC/Mac, Xbox 360, Xbox One, Playstation 3, Playstation 4, Playstation Vita, iOS, Android, Windows Phone og Amazon Kindle Fire. Minecraft er også blevet meget populært blandt børn i den grad, at der bliver produceret meget legetøj her i blandt Minecraft LEGO[4].

1.3 Børn skal lære at programmere

I dette afsnit vil der blive beskrevet hvorfor det er relevant at lære børn at kode i en tidlig alder, og hvilke lande som allerede er begyndt at lære deres børn at programmere. Der vil yderligere blive diskuteret, hvad børn får ud af at lære at kode i skolen, og om det også skal på skemaet i den danske folkeskole.

Programmering i skolen i forskellige lande

I nogle lande er man begyndt at lære børn at skrive kode, allerede i folkeskolen. Det er blandt andet USA, Storbritannien og Estland, som er blandt de første lande til at introducere børn for programmering.

USA: I USA, har de benyttet sig af et system kaldet Code.org. Code.org er en non-profit organisation, som hjælper folk, især skoleelever, til at komme i gang med forskellige dele inden for datalogi. Organisationen er blandt andet blevet støttet af Bill Gates og Facebooks grundlægger Mark Zuckerberg. Code.org holder en 'Computer Science Education Week' på nogle skoler i USA for børn og unge, og i forbindelse med denne uge, afholdes der også kampanjen "Hour of Code Challenge", hvor brugerne bliver opfordret til at gennemføre små programmeringsøvelser. Øvelserne indebærer at skulle skrive nogle linjer kode, for at opnå et bestemt mål.[5] Kampagnen blev støttet af USA's præsident Barack Obama, som også under en tale opfordrede alle børn til at lære at kode:

"We've got to have our kids in math and science, and it can't just be a handful of kids. It's got to be everybody. Everybody's got to learn how to code early." - Barack Obama[6]

Storbritannien: I Storbritannien har man også sat programmering på skoleskemaet i alle folkeskoler. Forløbet med at lære børn at kode, starter allerede i første klasse (5-6 år), hvor de får en basal introduktion til hvad en algoritme er, som kan forklares som "et sæt af instruktioner", hvilket kan illustreres som f.eks. en bageopskrift.

I de senere klasser (7-11 år), skal de skrive og debugge nogle enkle programmer, med specifikke mål, og blive introducerede til begreber som variabler, sekvenser, udvælgelse og gentagelser.

Til sidst i folkeskolen, ved alderen 11-14 år, skal de kunne benytte to eller flere programmeringssprog til at skrive deres egne programmer. Eleverne skal desuden lære boolean logik (f.eks. AND, OR og NOT operatorer), og lære om hvordan software og hardware arbejder sammen.[7]

Danmark: Til trods for at enkelte folkeskoler i Danmark, har deltaget i kampagner som f.eks. "Hour of Code Challenge" er det først egentligt muligt at lære programmering som et valgfag i gymnasiet. Det er dog blevet vedtaget at introducere programmering som en del af matematik- og fysik/kemiundervisningen efter sommeren 2015. I undervisningsvejledningen til matematik er Scratch, som beskrevet i afsnit 1.4, nævnt, men det er op til den enkelte kommune hvorvidt de vil investere i IT-efteruddannelse af lærerne. I hvor høj grad programmering bliver prioriteret på de enkelte skoler, afhænger derfor af den pågældende kommunens indstilling.[8]

Programmering er fremtidens sprog

Computere er alle steder, og de fleste folkeskoleelever har også deres egen computer, smartphone, spillekonsol osv. Med det stigende behov for software til alverdens computere, er der også behov for personer til at udvikle softwaren. Det er derfor relevant at introducere børn til softwareudvikling i en tidlig alder, og give dem en basal forståelse for hvordan computere og programmer fungerer. Det er altså ikke længere nok bare at lære hvordan man benytter software, men også hvordan man udvikler ny software.[9]

1.4 Børn og programmering

Børn i dag har mange muligheder for at lære at programmere, da der findes mange programmeringssprog og -kurser, som specifikt henvender sig til børn og begyndere.[10] Nogle af disse programmeringssprog vil blive analyseret, for at finde frem til hvad der er vigtigt, at tage hensyn til, i udviklingen af et programmeringssprog henvendt til børn.

Inden for udvikling af programmeringssprog til børn, er der tre principper programmeringssprog designes ud fra. Den første mulighed er at lave et programmeringssprog bestående primært af kode som skrives af programmøren, og denne kode skal udformes med en bestemt syntaks bestående af bogstaver og symboler. En anden mulighed er spilinteraktion, hvor programmering sker gennem et spil, hvor formålet er at nå et mål i spillet gennem kodning. En tredje mulighed er at anvende kommandoblokke, som kan sættes sammen og udføre arbejde, bestemt ud fra sammensætningen af disse kommandoblokke, i et visuelt udviklingsmiljø. Programmeringssprogene, som er henvendt til børn, er designet ud fra disse tre principper.

- **Kommandoblok programmering** ved brug at grafiske elementer som sammenstættes i blokke.
- **Tekstprogrammering** hvor en bestemt syntaks kompileres til et program.
- **Spilinteraktion** som kan gøre programmeringen lettere og sjovere.

I dette afsnit vil der blive analyseret og beskrevet nogle børnevenlige programmerings-sprog, hvorefter der vil blive konkluderet på de essentielle elementer i de forskellige sprog, som kan være relevante for udviklingen af projektets programmeringssprog.

1.4.1 CodeMonkey

CodeMonkey er et browser-baseret spil, designet til at lære børn at programmere. For at gøre det sjovt for børnene, går spillet ud på at komme igennem nogle baner, hvor man skal programmere en abe til at finde alle bananerne i en bane[11], med enkle kommandoer der lærer børn at bruge programmeringsprincipper, som loops, objekter, variabler, array, boolean udtryk m.m. [12]. Spillet giver en score efter hver bane, så spilleren kan se hvor godt personen har gjort det, og dermed eventuelt afprøve banen igen for at få en højere score.

Spillet er lavet til børn i 9-års alderen og ældre, men kan bruges af alle aldre, inklusiv yngre børn [12]. Sproget er designet til at skulle bruges som undervisningsmateriale, og derfor er der en version af spillet, hvor en eventuel lærer får en profil med forskellige funktioner, som f.eks. overvågning af score og fremskridt for eleverne. Det er ikke vigtigt om lærere eller børn har erfaring med programmering, da man i spillet bliver oplært hen ad vejen. Læreren får både et step-by-step pensum, med 13 lektioner af én time, så læreren ikke selv behøver at bruge tid på at planlægge alle undervisningerne. Læreren får også et cheat-sheet til alle banerne, og muligheden for at kontakte support.

Når man laver et program i spillet, skrives koden i CoffeeScript, som er et sprog med en syntaks, der minder forholdsvis meget om syntaksen i det engelske sprog. Med CoffeeScript er der både mulighed for at programmere imperativt og objektorienteret[13]. CoffeeScript bliver kompileret til JavaScript, som er muligt at bruge på hjemmesider.

Formålet med spillet er, at programmere en abe til at løbe igennem en bane og samle bananer. Når aben har fået alle bananer har man vundet banen. Når man starter spillet ser man hurtigt at spillet er opdelt i to dele, en side med en tekstboks lavet til at programmere i, og en side med en grafisk flade hvor man kan se sin kode blive udført i realtid. Det gælder så om at få aben til at samle bananerne op, på trods af en række forhindringer på banen. Man skal altså bruge funktioner for at få aben til at dreje, og gå. Man kan enten selv skrive koden manuelt, eller trykke på en række knapper, som så udfylder de forskellige funktioners korrekte navne, så man ikke behøver at stave sig igennem det. Når spilleren har samlet alle bananerne, kommer der et pop-up vindue, hvor brugeren får et antal stjerner der viser hvor godt man har klaret sig.

1.4.2 Scratch

Scratch er et af de programmeringssprog, som anvender grafiske kommandoblokke til at programmere interaktive historier og spil. Sproget er lavet med fokus på, at skulle kunne bruges til mange forskellige projekter, samtidig med at det skal være muligt at gøre projekterne personlige. Udtryk er delt op i forskellige former, som kan indkapsle hinanden. Det ses f.eks. på figur 1.1, hvor "Hello World!" laves i dette sprog.[14]

Blokkene, som sproget er bygget op af, har forskellige farver og former. Kontrolstrukturer



Figur 1.1. Scratch - Hello World

såsom "forever" og "repeat" er C-formet, for at illustrere at andre blokke kan sættes ind i. Et andet eksempel er if-udtryk som har et specifikt hulrum i deres form, hvori kun boolean klodser kan sammenkobles. Samt repeat, som er en løkke der gentager klodserne mellem de to arme, et vilkårligt antal gange. Se figur 1.2.



Figur 1.2. Scratch - If-statement og repeat

Ved at give klodserne forskellige former, har Scratch sikret at klodser kun kan sættes syntaktisk rigtigt sammen[15]. Derved er det ikke nødvendigt for personer at lære en syntaks, for at kunne programmere. Dette kan gøre det nemmere for personer, der ikke har kendskab til programmering i forvejen, at få en god opstart. Metoden som Scratch anvender til at udforme scripts uden syntaks, vil være det mest optimale for børn at anvende, da det er vigtigere for børnene, at de lærer strukturen og udførmning af kommandoblokke, end at de lærer syntaksen for koden bag kommandoblokkene[16].

Spillet er udviklet for 8- til 16-årige, men personer i alle aldre bruger Scratch[14]. Unge brugere lærer, ved brug af Scratch, omkring matematiske og beregningsmæssige idéer og får en forståelse for forskellige matematiske begreber såsom koordinater og variabler.[17]

Udover at kunne programmere med byggeklodserne, er det også muligt i Scratch at tilføje 2D grafik, ved at tegne i programmet eller importere billeder.

1.4.3 Small Basic

Small Basic er et imperativt programmeringssprog, lavet i 2011 af Microsoft[18], til at være sjovt at bruge samt let at lære og forstå. Sproget er meget småt, så det indeholder kun de helt basale funktioner og er på mange måder en nedkogt version af Visual Basic. Sproget indeholder 14 forskellige keywords, som brugeren kan bruge til at programmere med. Efter intern test af Small Basic, viser det sig at børn mellem 10 og 16 har haft stor

succes med programmeringssproget. Både børn og voksne fandt sproget brugbart som begyndersprog.[18, 19]

Small Basic er gratis, med muligheden for både at lave meget enkle programmer, og forholdsvis komplekse programmer. Small Basic har lavet deres sprog meget småt og imperativt, men udover det har Small Basic andre kendtegns [20]:

- **Variable typer:** Variablers type deklarereres ikke i Small Basic. Når variablen bliver brugt, vil programmet bruge den efter dens indhold.
- **Variabler** er altid globale og skal altid initialiseres.

Et udtryk der er forsimpleret i forhold til sprog såsom C# og Java, er for-løkken som Small Basic bruger. Den tager bare en start parameter og en slut parameter, hvor den kører igennem løkken samme antal gange som forskellen mellem start og slut parameteren. Denne for-løkke kræver ikke nogen opdateringsparameter, som den eksempelvis gør i C#. Et eksempel på denne for-løkke kan ses i kodeeksempel 1.1. Denne for-løkke vil udskrive 1 til 24 i output.

```

1 For i = 1 To 24
2   TextWindow.WriteLine(i)
3 EndFor

```

Kodeeksempel 1.1. Smallbasic for-løkke

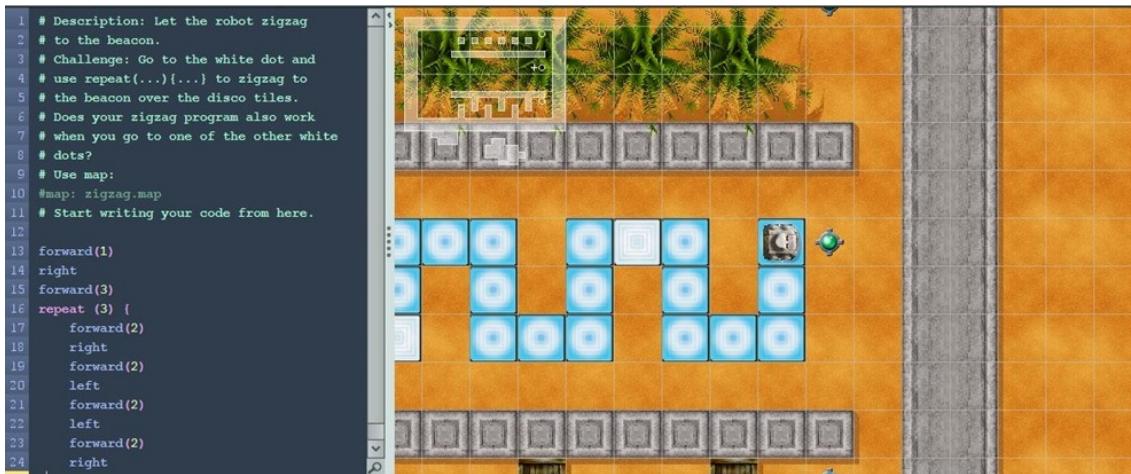
1.4.4 Robomind

RoboMind er et af de programmeringsmiljøer, hvor der er fokus på at lære det basale inden for programmering, for at lære begyndere inden for programmeringsverdenen, hvordan kode kan kontrollere en simuleret robot, i et programmeringsmiljø hvor der skal kodes med en bestemt syntaks. RoboMind anvender det første princip med programmering gennem syntaks, samt det andet princip med kodning gennem spilinteraktion.

RoboMind har et script-sprog hvor løkker og betinget forgrening, med if-else kæder, anvendes til at kontrollere den simulerede robot[21, 22]. Dette programmeringssprog er et af de sværere sprog for nybegyndere, da der kræves kendskab til syntaksen, for at et script skal virke, og nye programmører skal derfor bruge tid på at sætte sig ind i syntaksen før programmeringen kan begynde. Se figur 1.3 for eksempel på syntaks i RoboMind.

1.4.5 Opsamling

Sprogene der er blevet undersøgt og beskrevet i dette afsnit, indeholder alle elementer som gør dem gode som introduktion til programmering. Tre af sprogene indeholder et grafisk spilelement eller et værktøj til at skabe egne historier med, hvilket er med til at gøre det mere underholdende. Samtidig er syntaksen, som bliver benyttet af sprogene, gjort enkel. Dette ses f.eks. ved Scratch, hvor en løkke bliver kaldet repeat og kun kan bruge en heltalsparameter der angiver hvor mange gange løkken skal løbes igennem. I Small Basic er der mulighed for at lave en for-løkke der går fra et tal til et andet. I modsætning til en for-løkke i f.eks. C#, skal der i Small Basic ikke beskrives en opdatering i for-løkken. Herved



Figur 1.3. Screenshot fra RoboMind miljøet[23]

har Small Basic gjort for-løkken mere enkel, som kan formindske mulige fejl. Samtidig har Scratch forsøgt at gøre det nemmere at arbejde med deres programmeringssprog, ved at lave udtryk til blokke, der kan sættes sammen. Blokkene har forskellige farver og former alt efter hvad de repræsenterer, og kan kun sættes syntaktisk rigtigt sammen, hvilket sikrer at der ikke kan laves syntaksfejl.

1.5 Problemdefinering

Programmering er blevet kaldt for fremtidens sprog, og mange mener derfor, at det er noget børn skal introduceres til allerede i de yngre aldre. Det essentielle ved at introducere børn til programmering, er ikke nødvendigvis at lære dem at blive dygtige til at kode i bestemte sprog, men at lære dem tankegangen bag at programmere; hvordan man kommunikerer med computere og hvilke muligheder der er inden for softwareudvikling. Der eksisterer allerede flere programmeringssprog og -kurser udviklet specifikt til børn og unge, som f.eks. CodeMonkey, Scratch og SmallBasic. Disse sprog adskiller sig fra andre programmeringssprog ved at have særlig vægt på læsbarhed, og ofte tilføjet et grafisk element, eller en form for udfordring eller spil til at øge motivationen for at fortsætte.

Et område hvor man kunne fange mange mennesker, er i et spil som Minecraft med omkring 20 millioner brugere, hvor der bliver brugt rigtigt mange timer. Minecraft er et såkaldt sandbox spil, som giver spilleren mulighed for at bygge hvad som helst, i et stort åbent univers. Dog kan byggeprocesserne være både ensformige og langsommelige. Det ville derfor være en mulighed at skrive programmer til at udføre disse byggeprocesser hurtigere, og dermed gøre det muligt at programmere sjove konstruktioner til Minecraft universet. Ud fra disse iagttagelser, er følgende problemformulering beskrevet:

Hvordan skal et børnevenligt programmeringssprog designes, så det er let anvendeligt til at konstruere både enkle og komplekse konstruktioner i spillet Minecraft?

Sprogdesign 2

I dette kapitel vil designet af projektets sprog, VMP, blive beskrevet. Der vil blive dannet kriterier for sprogets læsbarhed, skrivbarhed og pålidelighed, samt hvor simpelt og ortogonalt sproget skal være. Derudover vil der være en beskrivelse af sprogets strukturer og keywords, samt tankerne der er blevet lagt bag dem.

2.1 Sprogkriterier

I dette afsnit vil der blive diskuteret og opsat nogle kriterier for VMP. Der vil blive beskrevet nogle nøgleegenskaber for hvordan et programmeringssprog kan vurderes, og der vil blive diskuteret hvad der er vigtigt at lægge vægt på under dette projekt. Sebestas beskrivelser[24, s.27-38] af evalueringkskriterier er blevet brugt som kilde og inspiration til dette afsnit.

Evalueringkskriterier

Når et programmeringssprog skal evalueres, bliver det ofte gjort ud fra tre grundlæggende principper; læsbarhed (readability), skrivbarhed (writability) og pålidelighed (reliability). Nogle gange nævnes også pris (cost) som et kriterium, hvilket der ikke vil blive taget højde for i dette projekt, da det ikke har nogen særlig betydning.

Læsbarhed beskriver hvor let et program, skrevet i et bestemt sprog, er at *læse*. Dette er vigtigt, fordi et program sjældent kun bliver udviklet og vedligeholdt af en enkelt person, og det er derfor vigtigt at andre også kan læse og forstå koden bag programmerne, og hurtigt sætte sig ind i forskellige funktioners betydning. Det er derudover også relevant selv når en person arbejder på et projekt, således at man kan forstå sin egen kode efter en pause. Et sprogs læsbarhed bliver påvirket af flere forskellige ting; naturligvis hvor *simpelt* sprogets syntaks og opbygning er, da et stort sprog ofte også vil være mere besværligt at læse og forstå, end et lille og simpelt sprog. En anden faktor der kan være med til at forværre et sprogs læsbarhed er *mangfoldighed* (multiplicity). Dette er når den samme ting kan gøres på flere forskellige måder. F.eks. i C hvor `count = count + 1;` er det samme som at sige `count++;`. Hvor dette måske kan være smart og en fordel i nogle situationer, er det med til at gøre sproget sværere at læse. Mangfoldighed hænger lidt sammen med en tredje vigtig faktor, som er *operator overloading*, hvor den samme operator kan bruges til flere forskellige ting, alt efter hvor og hvordan den bliver benyttet.

Til sidst er der *ortogonalitet*, hvilket er graden af restriktioner og undtagelser, der er i et sprog. F.eks. begrænsninger på at nogle datatyper kan returneres, og andre ikke kan. I et sprog med høj ortogonalitet er det tilladt at gøre stort set hvad som helst, og der er få eller ingen undtagelser. Høj ortogonalitet kan øge både læsbarheden og skrivbarheden af et sprog, fordi man ikke konstant skal tage højde for specifikke undtagelser. Et sprog kan dog også blive for ortogonal, og dermed mindske læs- og skrivbarheden, hvis alt er tilladt, selvom det ikke nødvendigvis giver mening. Det er derfor vigtigt at finde et passende niveau.

Skrivbarhed beskriver hvor nemt det er at skrive et program i et givent sprog. Skrivbarhed kan blandt andet beskrives ved at se på følgende eksempel i C:

```
count++;  
count = count + 1;
```

De to linjer udfører den samme funktionalitet, og bliver endda oversat til nøjagtigt det samme kode af en compiler. At skrive `count++` er blot hurtigere og mere bekvemt, til trods for at det kan gøre det sværere at læse og dermed mindske læsbarheden, fordi det er en ny ting man som læser skal lære og forstå. Skrivbarheden påvirkes desuden også af - ligesom læsbarheden - hvor simpelt et sprog er, og hvor høj ortogonalitet der er. Et simpelt sprog er således ikke kun lettere at læse, det er også nemmere at skrive. I hvert fald til en vis grad.

Pålidelighed bliver brugt til at beskrive hvor pålideligt et sprog er. Et sprog kan siges at være pålideligt, hvis det gør hvad det skal under alle omstændigheder. En måde at sikre pålidelighed, er ved *type checking*, som kontrollerer at programmet ikke indeholder typefejl, f.eks. hvis en funktion forventer at modtage en float, men får en integer. Selvom dette virker som en naturlighed, er det et vigtigt element for at et programmeringssprog bliver pålideligt. Type checking kan enten ske på compiletime (et statisk tjek) eller runtime (et dynamisk tjek)[25]. Uover type checking, kan man også bruge *exception handling*, som blandt andet bliver brugt i C#. Dette er en måde at håndtere fejl i programmet mens det kører, og give en passende fejlmeldelse.

Overvejelser og prioritering

I dette projekt vil der især blive lagt vægt på høj læsbarhed. Dette er valgt fordi VMP hovedsageligt henvender sig til børn og begyndere, og det derfor vigtigt at det er let at forstå. Det skal være et sprog, som er nemt at læse for enhver person, om det er en meget øvet og erfaren programmør, eller om det er en nybegynder på 11 år. Skrivbarheden er ikke lige så højt prioriteret, da VMP hovedsageligt vil blive benyttet til forholdsvis små programmer, og det er derfor mere acceptabelt, at skulle skrive længere sætninger, hvis det er med til at øge læsbarheden. Dog kan det på visse steder muligvis betale sig at gå på kompromis, og tilføje funktioner eller egenskaber, hvis det i høj grad er med til at øge skrivbarheden.

Simpelhed og ortogonalitet er blevet prioriteret adskilt fra de andre kriterier, da det er egenskaber som påvirker læsbarhed, skrivbarhed og pålidelighed. For at opnå den høje læsbarhed, vil det blive forsøgt at holde VMP så simpelt som muligt. Dette er især vigtigt fordi sproget henvender sig til børn, og det er derfor VMP ikke må være kompliceret.

Egenskab	Prioritering
Læsbarhed	1
Skrivbarhed	2
Pålidelighed	3

Egenskab	Prioritering
Simpelt	1
Ortogonal	2

Tabel 2.1. Prioritering af de nævnte sprogkriterier

At lave et sprog simpelt kan f.eks. gøres ved at begrænse antallet af datatyper, og holde mængden af specialtegn (? ; ! m.m.) på et fornuftigt niveau. At lave et simpelt sprog er med til at øge både læs- og skrivbarheden. Derudover vil det blive forsøgt at undgå for meget mangfoldighed, da det kan gøre det forvirrende, at der er flere måder at gøre det samme på.

2.2 ScriptCraft

I dette afsnit vil der blive set på et plugin til Minecraft, som giver mulighed for, inde og uden for spillet, at benytte JavaScript til at skabe konstruktioner og modificere spillet. Der vil blive beskrevet, hvordan ScriptCraft skaber byggeklodserne i spillet ud fra JavaScript. Der bliver set på ScriptCraft, da ved at udnytte dets allerede eksisterende plugin til Minecraft, ikke er nødvendigt i dette projekt at skabe et helt nyt plugin.

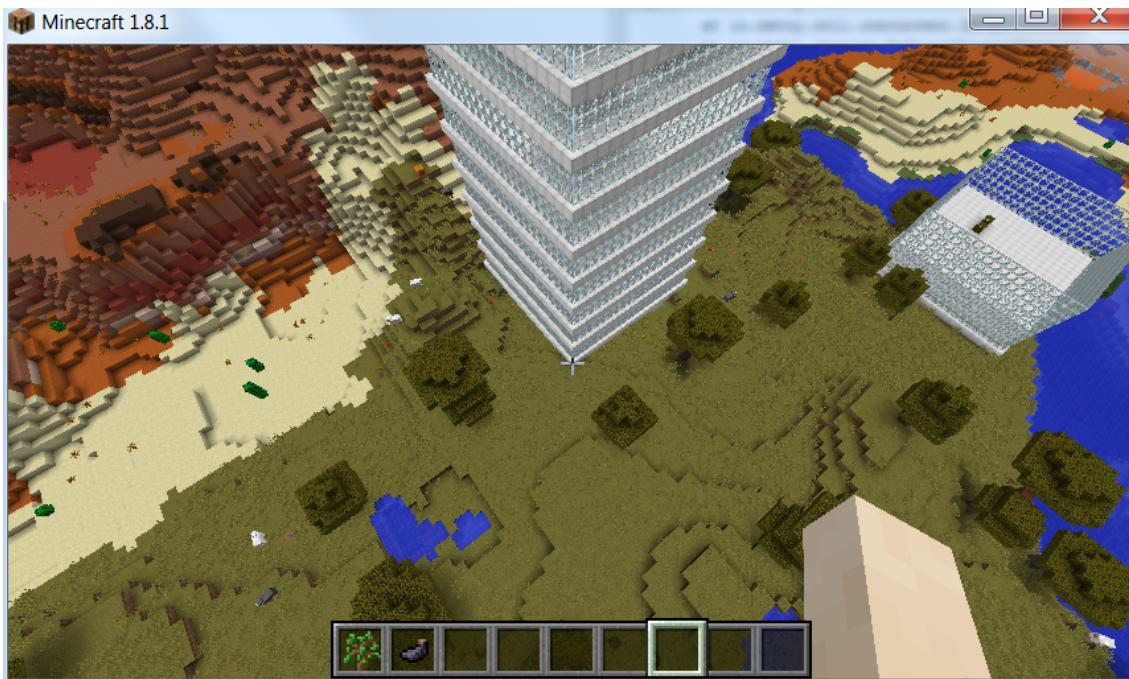
ScriptCraft er et open-source Minecraft plugin, der åbner op for muligheder til at lave kode inde i spillet.[26] Ved hjælp af ScriptCraft kan man lave både enkelte og komplekse ændringer til spillets regler, og det kan samtidig også bruges til hurtigt at bygge meget store konstruktioner, der ellers ville tage meget lang tid at bygge. Et eksempel kan ses på figur 2.1, hvor der er bygget et højhus ved hjælp af et kort script, som kan ses i kodeeksempel 2.1. Hvis man ville bygge det samme uden at bruge et script, kunne det tage flere timer at konstruerer.

```

1 function myskyscraper( floors )
2 {
3     var i ;
4     if ( typeof floors == 'undefined' )
5     {
6         floors = 10;
7     }
8     this.chkpt('myskyscraper');
9
10    for ( i = 0; i < floors; i++ )
11    {
12        this
13            .box(blocks.iron,20,1,20)
14            .up()
15            .box0(blocks.glass_pane,20,3,20)
16            .up(3);
17    }
18    this.move('myskyscraper');
19 }
```

Kodeeksempel 2.1. ScriptCraft - kode til bygning af højhus

En anden måde man kan bruge ScriptCraft på er, at lave JavaScript filer som man gemmer i en bestemt mappe. Ved at lave funktioner i disse JavaScript filer, kan man genbruge



Figur 2.1. Screenshot fra Minecraft, resultat af kodeeksempel 2.1 med parameteren 12

kode mange gange. Et eksempel kunne være, at man har skrevet et script, der automatisk bygger et højhus. Hvis scriptet gemmes som en funktion, kan det nu kaldes et vilkårligt antal gange, uden at skulle skrive koden igen.[27] Dette vil igen kunne spare tid, hvis man nu f.eks. ville bygge en hel by, og dermed skulle bruge mange højhuse.

Udover at ScriptCraft giver mulighed for at bruge JavaScript, er der også lavet tilføjelser til sproget, som gør det nemmere at arbejde med i forbindelse med Minecraft. Der er eksempelvis lavet en kommando til at lave en kasse, samt andre primitive figurer, på den position hvor spilleren peger. Et vigtig element i ScriptCraft er en "drone", der giver muligheder for at bygge større og mere komplekse konstruktioner. Dette gøres f.eks. ved at man i koden programmerer dronen til først at bygge noget et sted, og derefter bygge noget et andet sted, uden at skulle skrive flere stykker kode[27].

2.2.1 Drone

Dronen er et essentielt mod, som følger med ScriptCraft. Dronen bruges hver gang der benyttes nogle af ScriptCrafts funktioner, i forbindelse med byggeri eller bevægelse. Dronen bygger blokkene for brugeren, og gør at det ikke er nødvendigt, at skulle ændre position i Minecraft. Det er muligt at bevæge dronen ved hjælp af forskellige funktioner, som kan sættes i forlængelse af hinanden, og bruges sammen med funktioner til at bygge forskellige elementer i Minecraft. ScriptCrafts dokumentation af deres API[28], er brugt som kilde i dette afsnit.

Dronen benytter flydende interface, som betyder at alle dronens metoder returnerer **this**, og dette gør at de kan kædes sammen. Samtidig med det flydende interface, er alle dronens metoder globale funktioner, som hvis ikke tildeles en drone, skaber en drone. Dette gør det muligt at skrive kode på følgende måde:

```
1 fwd(3).left(2).box(blocks.grass).up().box(blocks.grass).down()
```

Kodeeksempel 2.2. ScriptCraft - Udnyttelse af flydende interface og drone metoder er globale

i stedet for at skrive det som i kodeeksempel 2.3:

```
1 var theDrone = new Drone(self);
2 theDrone.up().fwd(3).left(2).box(blocks.grass).up().box(blocks.grass).down();
```

Kodeeksempel 2.3. ScriptCraft - Instanstiering og brug af dronen

Det er derfor ikke nødvendigt at tage hensyn til dronen, når der bliver programmeret konstruktioner, da ScriptCraft selv sørger for, at drone objektet bliver oprettet. Ønskes det at have mere kontrol og kun arbejde med én instans af dronen, kan dette stadig gøres.

Funktioner

Som det også ses fra kodeeksempel 2.2, er der flere metoder til at kontrollere dronen på. Alle tilgængelige metoder ses her:

- up()
- down()
- left()
- right()
- fwd()
- back()
- turn()

Alle disse metoder tager en parameter, der angiver hvor mange blokke dronen flytter sig i den angivne retning. Metoden **turn()** roterer dronen, og parameteren angiver hvor mange gange dronen skal rotere 90° med uret.

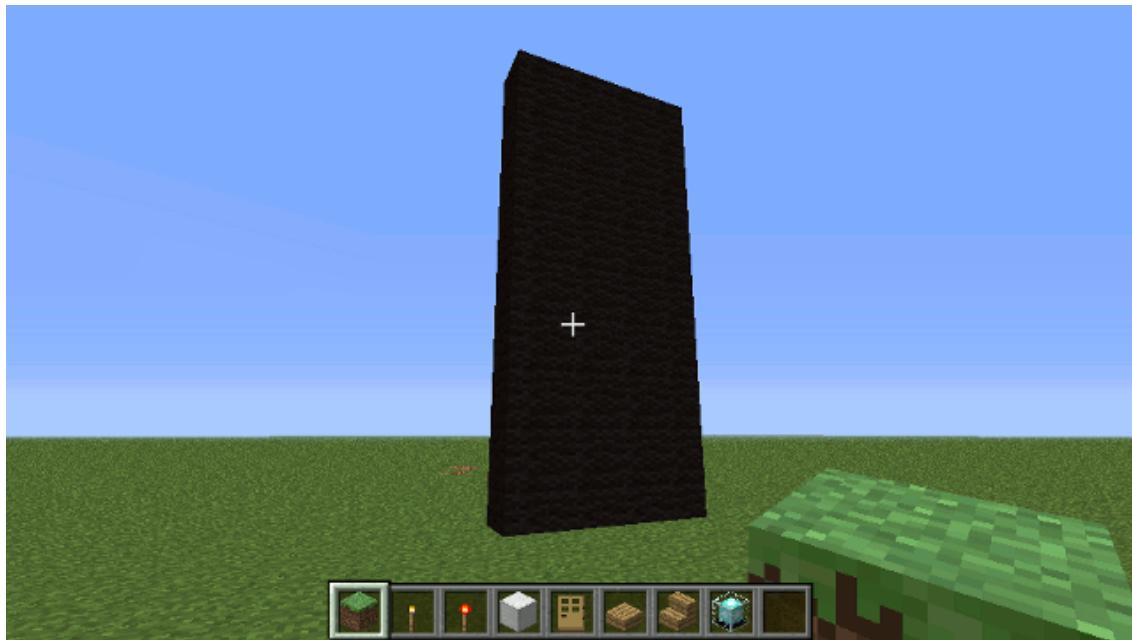
Udover at kunne kontrollere dronen, bliver den også, som nævnt tidligere, brugt til at placere blokke. Den simpleste af disse er **box()**, som tager 4 parametre: **b**, som er et "block-id", som angiver hvilken slags blok i Minecraft, der skal bruges. Derudover er der parametrene **w**, **h** og **d** for henholdsvis bredde, højde og dybde. Et eksempel på **box()** metoden ses på henholdsvis kodeeksempel 2.4 og figur 2.2

```
1 box(blocks.wool.black, 4, 9, 1);
```

Kodeeksempel 2.4. ScriptCraft - Box() metoden i brug

Udover metoden **box()** har dronen også flere metoder til at skabe forskellige konstruktioner, nogle af disse er:

- **arc()** - til at konstruere buer med



Figur 2.2. Visuelle resultat af Kodeeksempel 2.4. [28]

- `blocktype()` - til at konstruere bogstaver med
- `cylinder()` - til at konstruere cylindere
- `prism()` - til at konstruere prismaer med

Da dronen er det element af ScriptCraft som benyttes til at bygge klodserne med, vil dronen også blive udnyttet i dette projekt. Det er dog valgt at benytte den implicit i VMP, på en sådan måde, at programmøren ikke initialisere dronen, men udnytter dets flydende interface og globale funktioner. Dronens position i Minecraft vil blive brugt til at holde styr på konstruktionernes start og slut, hvilket kan gøre det nemmere at fortsætte og bygge videre på konstruktioner.

2.3 Designvalg

I dette afsnit vil der blive redegjort for, hvordan forskellige strukturer og keywords ser ud i VMP, samt beskrevet hvorfor der er foretaget valget om at designe dem således, med henblik på afsnit 2.1. Denne redegørelse er med til at vise tankerne bag de specifikke udformningsvalg, og øge forståelsen for hvorfor VMP ser ud som det gør.

2.3.1 Datatyper

For at VMP skal være enkelt og hurtigt at forstå, er det valgt at sproget kun skal indeholde to primitive datatyper:

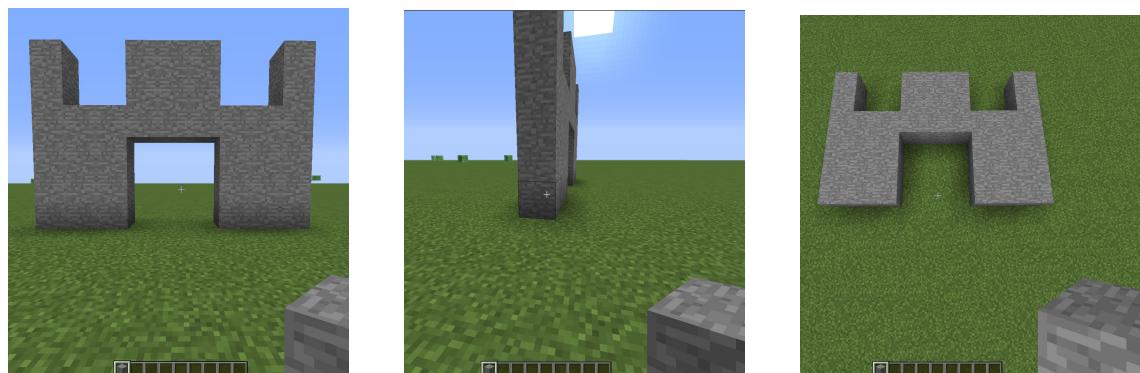
- **Integer** - Variabler består af to eller flere karakterer langt variabelnavn.
- **Block** - Variabler består af ét karakter-langt variabelnavn, hvor værdier repræsenterer Minecraft blokkenavne.

Disse datatyper kan kun deklarereres i funktioner, og ikke inde i kontrolstrukturer. Dette er valgt, for at holde det simpelt, og det sikrer at der ikke opstår problemer ved, at der oprettes en block-variabel inde i en kontrolstruktur, som programmøren prøver at benytte uden for kontrolstrukturen. Når variablerne deklarereres, skal dette gøres statisk. Dette skyldes at koden bliver kørt af et plugin, som vil blive yderligere beskrevet i afsnit 2.2.1, hvilket betyder at der skal type-kontrolleres ved compiletime, og ikke ved runtime. Derfor er det nødvendigt, at der angives hvilken type variabler har, når de deklarereres.

Da der kun benyttes disse datatyper i VMP, som ikke har nogle relationer til hinanden, er der ingen type konvertering i dette programmeringssprog. Der er altså ingen måde, hvorpå integers kan konverteres til block typen, eller omvendt. Derfor er der heller ikke mulighed for, at benytte "mixed-mode" udtryk, altså udtryk hvor der indgår både integer og block-variabler som operander.

2.3.2 build-udtryk

Det er i `build` at blockvariabler bliver brugt til at skabe konstruktioner med, ved at benytte variabelnavnene som repræsentationer af blokkene i Minecraft, og placere dem i et plan efter hvordan det ønskes repræsenteret i Minecraft. For at holde styr på orienteringen i forbindelse med 3D rummet der konstrueres i, er det valgt at `build`-udtrykket skal have en orienteringsparameter, som enten er `front`, `top` eller `side`.



Figur 2.3. Kodeeksempel 2.5 med orienteringsparametrene `front`, `side` og `top`

Grunden til at det er valgt at variabelnavne, til typen `block`, kun skal kunne være én karakter lang, er måden de bruges til at konstruere med i koden. Hvis variabelnavnet kan være mere end en karakter, vil det gøre koden, og overblikket over den endelige konstruktion i Minecraft uoverskuelig. Se kodeeksempel 2.5. Derudover er det sikret i VMP, at der ikke vil opstå problemer hvad angår identiske funktions- og variabelnavne, da en funktions navn afsluttes med parenteser.

```

1 function tower()
2
3     block s = STONE
4     block & = STONE
5
6     build(front)
7         s - - s s s - - s
8         s - - s s s - - s
9         s s s s s s s s
10        s s s - - - s s s

```

```

11      s s s - - - s s s
12      & s s - - - s s s
13  end build
14 end function

```

Kodeeksempel 2.5. Konstruktion af tårn i VMP

Som det ses af kodeeksemplet 2.5, er det valgt at det skal være muligt i VMP, at bruge variabelnavne, med typen **block**, direkte som en repræsentation af en blok fra Minecraft. Dette giver i højere grad et overblik over en konstruktion, samtidig med at den bliver programmeret. Idéen bag at "bygge" et lag ad gangen, i ét plan, kommer fra 3D-printere hvor der printes et lag ad gangen, og fra afsnit 1.4, hvor der beskrives hvordan programmeringssprog benytter kommandoblokke til at programmere med. Disse kommandoblokke sammensættes grafisk for at udforme den ønskede funktionalitet, og princippet er det samme i **build**-udtrykket.

For at øge kontrollen over konstruktionerne, og sørge for at man kan fortsætte fra en konstruktion, eller et plan, til en anden fra en bestemt position, er der tilføjet tre karakterer som kan styre dette. De tre tegn vil blive omtalt som control-blocks. Disse er **!**, som angiver startpunktet, **?** som angiver slutpunktet og **&**, som angiver både start- og slutpunktet. Disse sikrer at programmøren nemmere kan styre hvordan der ønskes at bygge videre på konstruktioner, hvis der f.eks. benyttes flere funktioner. Hvis konstruktioner skal placeres forskudt for hinanden, kan denne funktionalitet i VMP gøre det nemmere.

Grunden til at der benyttes **!** og **?** til denne funktion i VMP er, at karakterene oftest bliver benyttet i nogle computerspil af genren MMORPG (Massively Multiplayer Online Role Playing Game), til at illustrere hvor en opgave i spillet starter og slutter. Samtidig optages der ikke nogle navne til blokvariable, da disse kun kan være fra a-z og A-Z.

```

1 function creation1()
2   block ! = STONE
3   block ? = STONE
4   block & = WOOD
5
6   block s = STONE
7   block w = WOOD
8
9   build(front)
10    s s ?
11    s s s
12    ! s s
13  end build
14
15  build(top)
16    w w w
17    & - w
18    w w w
19  end build
20 end function

```

Kodeeksempel 2.6. Brug af start- stop- og både-og-punkt karakterer

Her ses et eksempel på, hvordan start-, slut- og både-og-punktet kan bruges. For ikke at skulle dele disse to konstruktioner op i to funktioner, og bevæge sig i Minecraft og specificere inde i spillet, hvor hver konstruktion skal starte, kan det derimod gøres, som

det ses i kodeeksempel 2.6. Ved at placere slut-punkt karakteren, '?', øverst til højre i det første **build**-udtryk, kan man sikre det næste **build**-udtryk vil starte fra denne blok, hvilket også er illustreret på figur 2.4.



Figur 2.4. Illustration af kodeeksempel 2.6 inde i Minecraft. Billede 1 er efter første **build**-udtryk, og billede 2 efteranden **build**

2.3.3 Operatorer

I forbindelse med tildelingsoperatoren, er det valgt ikke at tillade tildeling af flere variabler på samme tid. Dette er valgt, for at have en god læsbarhed i VMP.

Logiske operatorer

Det er blevet valgt at bruge engelsk tekst med store bogstaver til at symbolisere de logiske operatorer **AND** og **OR**. Dette vil forhåbentligt gøre det lettere, at huske og få udtryk til at være lettere forståelige. VMP er udformet med fokus på høj læsbarhed, og det er derfor oplagt at benytte ord, der så vidt muligt beskriver funktionaliteten.

I forbindelse med logiske udtryk er det valgt at benytte "short-circuit"-evaluering. Dette betyder, at det ikke er nødvendigt at evaluere begge operander ved **AND**, hvis den første operand er *false*, vil udtrykket evalueres *false* og ved **OR**, hvis den første operand er *true*, vil udtrykket evalueres til *true*. Dette gøres, da det er en hurtig og billig måde at evaluere et logisk udtryk. Et muligt problem ved at benytte "short-circuit"-evaluering er, hvis udtrykket på højre side af f.eks. **AND** er en funktion, som frigør plads i systemet, og derfor kan være vigtig, ikke bliver eksekveret og evalueret, hvis det første udtryk er *false*. Dog ses dette som et meget specifikt tilfælde, som højst sandsynligvis ikke vil have nogen påvirkning i VMP.

Operatorprioritering

Det er valgt i VMP at holde antallet af operatorer lavt, samtidig med at holde sig til den standard, såsom infix binære operatorer, der bruges inden for matematik og flere programmeringssprog. Dette er valgt, for at holde programmeringssproget enkelt og let at gå til, selv for personer med begrænset eller ingen erfaring. Prioriteringen af operatorerne i VMP kan ses i tabel 2.2, hvor prioriteringen og associativiteten også ses.

Udover disse, er det også muligt at benytte parenteser til at indkapsle udtryk, og derved sikre at udregningen af udtryk sker på den ønskede måde. Dette kan f.eks. være et udtryk,

Prioritet	Operator	Beskrivelse	Associativitet
1	()	Funktions kald	Venstre mod højre
2	* / %	Multiplikation, division og modulus (rest)	
3	+ -	Addition og subtraktion	
4	< <=	Mindre end og mindre end eller lig med	Venstre mod højre
	> >=	Større end og større end eller lig med	
5	= /=	Relation, lig med og ikke lige med	
6	AND	Logisk og	
7	OR	Logisk eller	
8	=	Assignment	Højre mod venstre

Tabel 2.2. Operator prioritet

såsom $a + b * c$, hvor $b * c$ først vil blive udført, hvorefter der adderes med a ifølge operatorprioriteringen. Ved brug af parenteser, er det dog muligt at udføre additionen først, hvis dette ønskes. Dette gøres ved, at indkapsle udtrykket $a + b$ i parenteser, således: $(a + b)$.

Det er fravalgt at tage unary operatorer som post- og præfix increment og decrement med i VMP, for som nævnt tidligere, at holde sproget simpelt. Samtidig ses de heller ikke som en nødvendighed i sproget, men kan evt. tilføjes senere i processen.

Overloading

Overloading af operatorer ses ikke nødvendigt i VMP, og er derfor valgt fra. Dette skyldes at der ikke er et stort antal typer at operere på, og det derfor ikke ses nødvendigt at give brugeren mulighed for, at overlaade de operatorer der allerede kan operere med disse typer. Samtidig kan operator overloading forringe læsbarheden i programmeringssproget, hvilket ikke ønskes.

2.3.4 Kontrolstrukturer

Det er valgt kun at have to kontrolstrukturer i VMP, for at holde det enkelt og let overskueligt, dog uden at miste for meget funktionalitet. Kontrolstrukturerne består af **if-sætninger** og løkker i form af **repeat**. **repeat** fungerer som en for-løkke, dog med det forbehold, at det kun er nødvendigt at skrive ét tal som parameter, f.eks. **repeat(5)**. Det er altså ikke nødvendigt at skrive initialiseringsudtryk, fortsættelsesbetingelse eller opdateringsudtryk. Der er desuden også mulighed for at bruge **if-else** strukturer samt **else-if** kæder. Dette er valgt, fordi det mindske mængden af data, som skal indtastes og håndteres af programmøren, samt at gøre det mere overskueligt. Idéen bag at bruge **repeat()** kommer fra afsnit 1.4, hvor der ses på programmeringssproget *Scratch*, som benytter denne form for for-løkke.

For at give et godt overblik over koden og have en god læsbarhed i VMP er det valgt, at kontrolstrukturer skal afsluttes enten med **end if** eller **end repeat**, alt efter hvilken kontrolstruktur der bruges.

Da der er fokus på, at holde VMP enkelt, er det valgt kun at benytte **if-sætninger**

og `else-if` kæder i sproget, og ikke tillade yderligere betingede udtryk, som det ses i programmeringssproget C, se kodeeksempel 2.3.4.

```
1   betingelse ? evalueret-hvis-true : evalueret-hvis-false
```

2.3.5 Funktioner

Da målet for VMP er, at eksekvere funktioner til at lave konstruktioner i Minecraft med, er det valgt at funktioner ikke skal have nogen returværdi, altså fungere som procedurer. Det ses ikke nødvendigt i VMP at kunne benytte returværdier, da en funktions fokus er på, at eksekvere kommandoer til at konstruere konstruktioner, og ikke til udregninger osv.

Ved ikke at benytte returværdier i funktioner, undgår man samtidig funktions-bivirkninger. Funktions-bivirkninger er når en funktion ændrer enten en af dens parametre eller en global værdi. Et eksempel kan være udtrykket "`a + fun(a)`". Hvis funktionen `fun()` ikke har en bivirkning som ændrer værdien `a` vil rækkefølgen af evalueringen af udtrykket ikke have nogen betydning. Men hvis `fun()` ændrer på værdien `a` er der en bivirkning.

```
1   a = 10
2   b = a + fun(a)
```

Kodeeksempel 2.7. Udtryk til forklaring af funktions-bivirkning

Antages der i kodeeksempel 2.7, at funktionen `fun()` returnerer værdien 10, og ændrer værdien af dens parameter til 20, vil vi kunne læse værdien af variablen `b` til to forskellige resultater, alt efter hvordan programmeringssproget evaluerer udtrykket. Hvis værdien af `a`, altså den første operand, læses først vil udtrykket blive 20. Men evalueres funktions operanden først, vil den returnere 10, som før, men den vil samtidig ændre værdi af `a` til 20 og udtrykket bliver derfor i stedet 30.

Det er muligt i VMP at benytte funktioner. Dette er også begrænset af ScriptCraft, da dette er måden de gør det muligt at programmere i Minecraft med. Det er muligt at sende én eller flere parametre med i funktioner, enten blockvariabler, som gør det muligt at genbruge funktioner, til at bygge konstruktioner med forskellige bloktyper, eller integers som f.eks. kan anvendes i kontrolstrukturer.

Funktioner gør det muligt for programmøren at genbruge kode, og funktioner kan kaldes i andre funktioner. Herved kan dele af en konstruktion deles op i flere funktioner, genbruges og samles i en samlet funktion til sidst. I VMP er der ikke en start (`main()`) funktion, som der f.eks. er i C, men funktioner kaldes derimod direkte i Minecraft. Dette er begrænset af ScriptCraft, som gør det på denne måde. Der er dog enighed om, at dette en god måde at gøre det på, da det som nævnt, gør det muligt inde i Minecraft hurtigt at genbruge og kalde funktioner til forskellige konstruktioner, uden at skulle ud og evt. oprette en ny `main`-funktion.

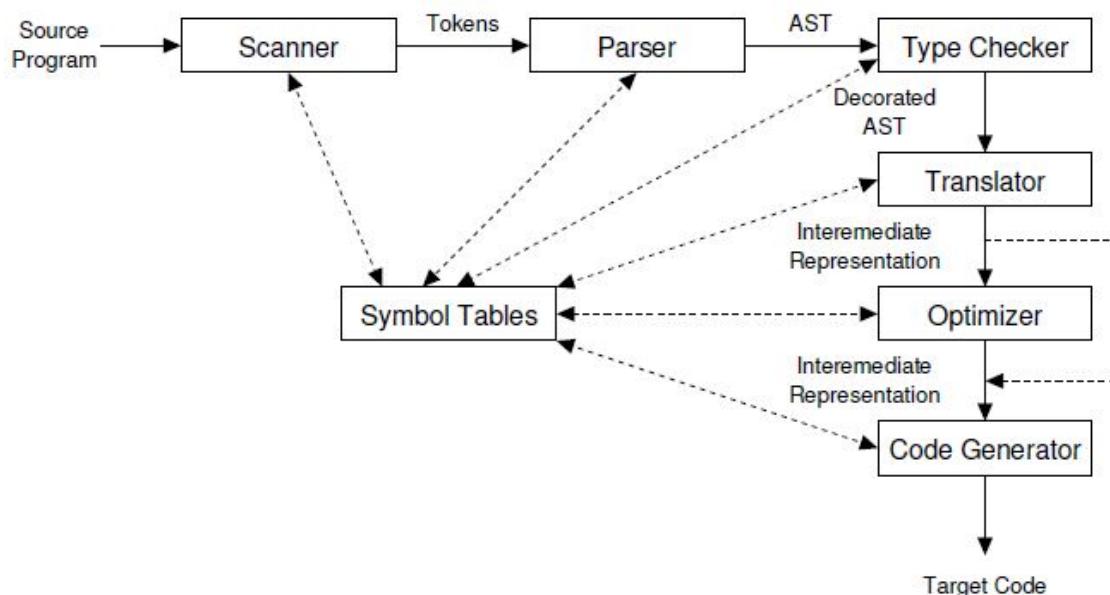
Det er ikke tilladt at benytte funktionsoverloading i VMP. Funktionsoverloading gør det muligt at have funktioner med det samme navn, men med forskellige parametre eller returtyper. Dette er valgt fra, da der er fokus på at sproget skal være enkelt og overskueligt, mens funktionsoverloading kan skabe forvirring.

Parameter passing

Parameter passing reglen for VMP, vil følge den som gælder for JavaScript, da VMP er baseret på ScriptCraft som blot er en udvidelse af JavaScript. I JavaScript er funktionsparametre *pass by value*[29], hvilket betyder at funktionen kun har adgang til værdien af den aktuelle parameter, og ikke dens placering. Hvis værdien ændres i funktionen ændres den ikke i den oprindelige parameter.

Compilerdesign 3

Der vil i dette kapitel blive beskrevet, og analyseret de forskellige faser, som indgår i en compiler, som det ses på figur 3.1. Derefter vil der blive beskrevet, hvordan nogle af disse faser, og deres forskellige elementer, bliver udført i dette projekt. Til at beskrive de forskellige faser i en compiler er Fisher, Cytron og Leblancs bog[30, Kap. 1] benyttet.



Figur 3.1. compilerens faser

3.0.1 Scanner

Scanneren begynder analysen af kildeprogrammet, ved at indlæse input og dele karakterer ind i grupper af tokens (identifiers, heltal, osv.), se afsnit 3.1.1. Disse tokens sendes videre til parseren. Udover dette udfører scanneren også opgaver, som at fjerne unødvendig information fra kildeprogrammet, såsom kommentarer. Den behandler compiler kontrol "directives", som fortæller om der skal inkluderes kildetekst fra filer.

En scanner kan genereres vha. en scanner generator, hvilket er et program der returnerer en scanner, ved input af de tokens den skal kunne genkende.

3.0.2 Parser

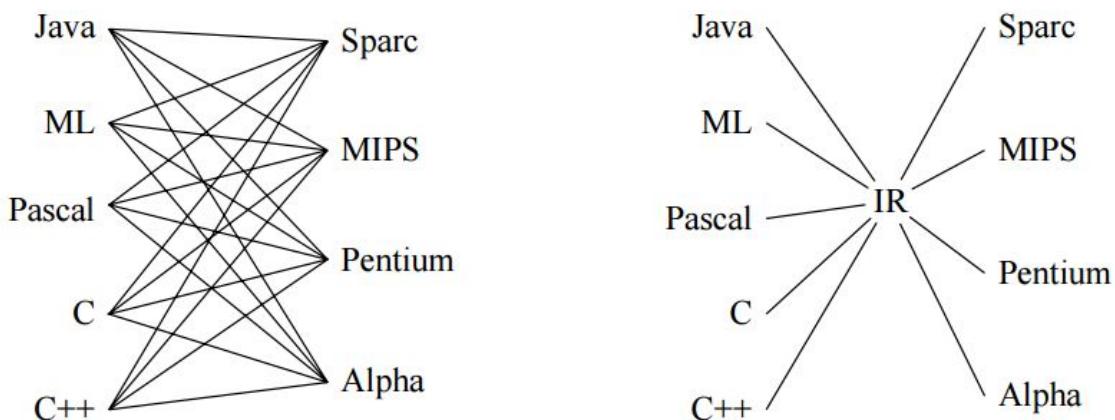
Parseren er baseret på en formel syntaksspecifikation, såsom en Kontekst Fri Grammatik (KFG), se afsnit 3.1.2. Den indlæser de tokens, som scanneren sender videre, og grupperer dem i "sætninger" i forhold til syntaksspecifikationen, der verificerer om syntaksen er korrekt. Er den ikke korrekt, udstedes der en fejlbeskæft, der beskriver den fejl parseren er stødt på. Løbende som en syntaktisk struktur bliver genkendt, vil parseren bygge et Abstrakt SyntaksTræ (AST), se afsnit 3.1.3, som en repræsentation af programmets struktur. Dette AST bliver derefter brugt som grundlag i semantisk forarbejdning.

3.0.3 Type checker (Semantisk analyse)

Type checkerens undersøger den statiske semantik for hver enkel node i ASTet. Den sikrer at det, som noderne repræsenterer, er lovligt og muligt i sproget, såsom at kontrollere om alle "identifiers" er blevet deklareret og at typer er korrekte, se afsnittene 3.2.1 og 3.2.3. Hvis opbygningen af AST-noden er korrekt, i forhold til semantikken, dekoreres noden ved at tilføje typeinformation.

3.0.4 Oversætter (Translator)

Er en AST-node semantisk korrekt, er det muligt at oversætte noden til en IR (Intermediate Representation). IR er abstrakt maskinkode, som ligger mellem kildekoden og målkoden. Ved at benytte IR er det muligt, at benytte færre compilere til at oversætte fra n kildesprog til m målsprog. Det kan ses på figur 3.2, at benyttes der IR skal der kompileres fra et kildesprog, til en IR og derfra til ét af de ønskede målsprog. Benyttes derimod ikke en IR, skal der kompileres fra et kildesprog til hvert enkelt målsprog. Der går derfor fra at være $n * m$ compilere til $n + m$ ved brug af IR. Det er dog muligt for simple compilere, der ikke optimerer, at oversætte direkte til målsproget, uden at benytte en IR. Dette gør compilerens design mere simpelt, ved at fjerne en hel fase, imod at compileren ikke er nær så "retargetable".



Figur 3.2. Compilere for fem sprog, og fire målmaskiner. Venstre uden IR, højre med IR.[31]

3.0.5 Symboltabel

Symboltabeller gør det muligt, at holde på information omkring identifiers, og gør dem tilgængelige i compilerens forskellige faser. Hver gang en identifier bliver deklareret, eller brugt, vil en symboltabel give adgang til alt den information, der opbevares omkring den. Informationen i en symboltabel, omhandler oftest om den identifier man tilgår er blevet deklareret, og hvilken type den er af. Symboltabeller bliver oftest brugt under type checking, men kan også blive brugt i andre faser. Se afsnit 3.2.2.

3.0.6 Optimizer

Den IR kode som bliver genereret af oversætteren, bliver analyseret og transformert til kode der er funktionelt identisk med det fra oversætteren, dog forbedret af optimizeren. Fasen fra det genererede kode fra oversætteren, til den forbedrede IR kode, er kompleks, og sker oftest i flere underfaser, hvor enkelte gentages flere gange. For at gøre oversættelsen hurtigere, gøres det oftest muligt for compileren at slå optimeringen fra.

3.0.7 Kodegenerator

Den IR kode som enten kommer direkte fra oversætteren, eller fra optimeringsfasen, bliver til målmaskinkode i kodegeneratorfasen. Denne fase kræver detaljeret information omkring målmaskinen, såsom "register allocation" og "code scheduling". En måde en del af denne proces gøres på, er ved at prøve at matche "low-level" IR til målinstruktion skabeloner, hvor kodegeneratoren selv vælger de instruktioner, der matcher IR instruktionerne.

3.1 Scanner og Parser

Dette afsnit vil beskrive projektets scanner- og parserfaser, ved at beskrive syntaksen, som VMP skal benytte. Syntaksen bliver beskrevet af et sæt regler, der definerer de ord, som er en sekvens af karakterer fra et alfabet, der benyttes til at udforme et program i VMP. Syntaksen for sproget kan beskrives ved brug af Backus-Naur Form (BNF), som frembringer programmeringssprogets Kontekstfri Grammatik (KFG). Denne rapport anvender ligeledes en udvidet form for BNF, kaldet Extended Backus-Naur Form (EBNF), som gør det muligt at beskrive regelsættet i en mere kompakt form, hvilket øger læs- og skrivbarheden af regelsættet.

3.1.1 Syntaks teori

I et programmeringssprog er symboler/ord defineret som terminaler. Tokens består af én eller flere terminaler. F.eks. vil en token kunne være et *ID* (p), et aritmetisk symbol (=, +, -) eller et tal (5). Et andet udtryk der bliver brugt er non-terminaler som er samlinger af tokens og andre non-terminaler.

Syntaksen bliver beskrevet med en liste af tokens, samt en KFG. Terminalerne beskrives med regulære udtryk.

3.1.2 Format af EBNF

Programmeringssprogets KFG er en liste af produktionsregler, hvor non-terminaler beskrives med deres produktion af terminaler og non-terminaler, som vist i eksemplet 3.1. Se afsnit 3.1.4 for uddybelse af KFG. I dette projekt bliver EBNF metasyntaks standarden ISO/IEC 14977 anvendt, som er blevet adopteret af The International Organisation for Standardization[32].

Anvendte symboler i den pågældende EBNF standard:

Brug	Notation
Definition	=
Konkatenering	,
Konkatenering	Mellemrum
Terminering	;
Terminering	.
Disjunktion	
Valg	[...]
Repetition	{ ... }
Gruppering	(...)
Terminal	"..."
Terminal	', ... ,'
Kommentar	(* ... *)
Undtagelse	-

```
1 build = "build" "(" orientation ")" blocks "end build"
```

Kodeeksempel 3.1. EBNF format

I kodeeksempel 3.1 ses non-terminal **build** på venstre side af lighedstegnet (=), og beskrives af den højre side af lighedstegnet som værende terminal "**build**" efterfulgt af terminal "(" , nonterminal **orientation**, terminal ")" og nonterminal **blocks**. Til sidst afsluttet af terminalen "**end build**".

3.1.3 Grammatik

Med EBNF kan man beskrive en grammatik, men der er forskellige måder at opbygge en grammatik på. I dette afsnit vil der blive beskrevet to forskellige opbygninger af grammatik, henholdsvis LL og LALR grammatikker, da de to compiler generatorer der er fokuseret på, bruger disse grammatikker, og da de er to af de mest brugte sprog inden for compiler generering. Dette afsnit er baseret på kilderne Fisher, Cytron og Leblancs bog[30, Kap. 4-6] og Sebesta[24, Kap. 4].

LL og LALR grammatikker hører begge under kontekstfrie grammatikker(KFG). Ud fra en grammatik er det muligt at lave et kontekstfrit sprog, og ud fra sproget er det muligt at lave en pushdown-automat(PDA) eller en non-deterministisk(ND) pushdown-automat(PDA). En PDA er en metode til logisk, at vise hvilke input og tilstande det kræver, at parse den kontekstfrie grammatik. I dette projekt vil der ikke direkte blive lavet en PDA, men en compiler generator vil bruge en PDA når den laver parseren. Derfor fokuseres der på grammatikken, som compiler generator skal bruge, for at generere en parser og en scanner.

Når der skal laves en grammatik, er det vigtigt, at grammatikken ikke er tvetydig. For et hvilket som helst gyldigt input, er det vigtigt at der kun er én mulighed for hvordan det kan parses. En parser genererer et parsetræ, som den derefter kan lave om til et Abstrakt Syntaks Træ (AST), ud fra grammatikken. Der må ikke være mulighed for at parseren kan generere to forskellige parsetræer, da dette vil betyde at det samme program vil kunne parses på mere end en måde, og derfor kan den samme kode ende ud med at have to forskellige betydninger.

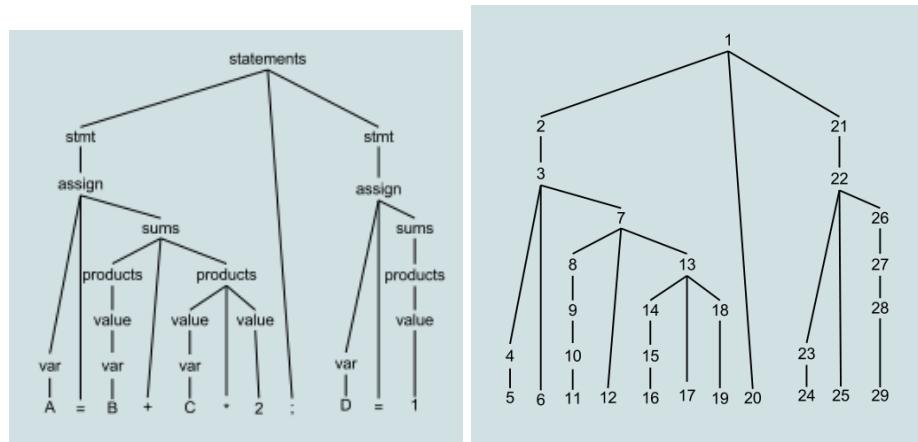
Det er muligt at tilføje lookahead til en parser, hvilket gør det muligt at se, hvad der kommer senere i input, hvilket bruges til at finde ud af hvilken produktion der skal vælges. Med et lookahead på 0 og en mulighed for, at vælge to produktioner der starter med samme token, vil give tvetydighed. Med et lookahead på 1, ville det være muligt at have to produktioner med samme start token, hvis et eventuelt token, der kommer derefter er forskellig. På den måde er det muligt undgå mange tvetydigheder. Dog er der i nogle tilfælde teoretisk set, brug for at have uendelig lookahead, og i disse tilfælde vil problemet kun kunne løses ved at skrive grammatikken om. Lookahead bliver beskrevet i parentesen efter navnet på de forskellige grammatikopbygninger, f.eks. LL(k) og LALR(k) hvor k er et heltal svarende til antallet af lookahead. Der er forskel på hvordan de forskellige grammatikopbygninger fungerer, en LL grammatik fungerer med top-down parsing og LALR bruger en bottom-up tilgang.

Top-down

Top-down parsing er en parsing metode, hvor man starter ved roden af parsetræet, og udvider det indtil man har inputstreng. Top-down parsing vil ske ved at kigge på første token i strengen, mens der bliver kigget på roden af træet, se figur 3.3. Parseren skal så finde ud af hvordan roden bliver lavet om til den første token på strengen. I eksemplet starter den ved *statements* og kigger på tokenen *A*, med den information er det muligt at lave *statements* til *stmt*. For at færdigøre parsing af første token, skal der parses indtil vi kommer helt ned til tokenen, i eksemplet figur 3.3 skal *stmt* laves til *assign* og videre til *var*, inden den finder muligheden for bare at skrive *A*. Efter at *A* er fundet, går parseren tilbage og finder sidste rod, som stadig har tomme grene. I eksemplet vil parseren gå tilbage til *assign*, som har en tom gren, den tomme gren skal indeholde '=' tegnet, og den bliver læst ind som det næste.

Hvis man tilføjer lookahead til en topdown parser, kan parseren kigge på den næste del af inputstrengen. Dette kan hjælpe hvis parseren står i roden af parsetræet, og ikke har mulighed for at vide, hvilken gren den skal udvide med, da der er to eller flere muligheder åbne for inputstrengen. Dette er grunden til at en LL parser, ikke kan tage left-recursion grammatikker, da lookahead skal bestemmes inden der skal laves et parsetræ, og det ikke vides hvor meget lookahead der skal bruges inden inputstrengen er givet, mere om dette i afsnit 3.1.3 som omhandler LL.

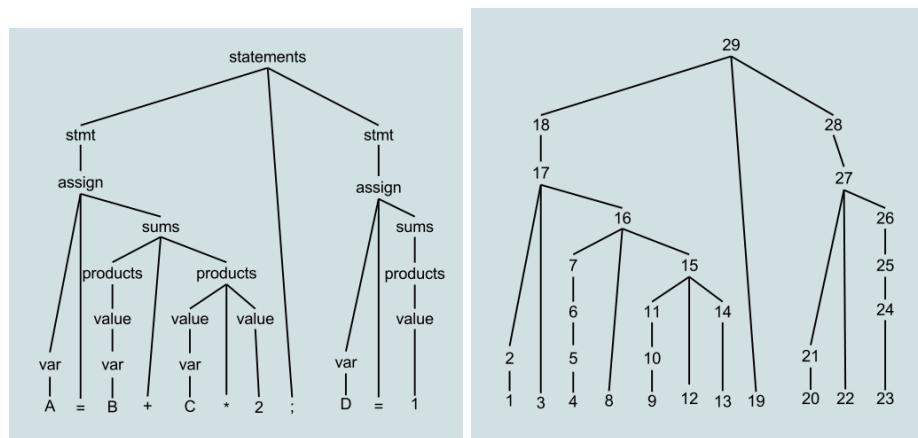
Top-down parsing er mere ligetil at forstå, da det er en logisk opsætning og gennemgang af en streng. Men det har også nogle begrænsninger, såsom left recursion. En top-down parser er i de fleste tilfælde simpelere, end en bottom-up parser, at implementere i hånden.



Figur 3.3. Top-down parsing, billede fra wikipedia[33].

bottom-up

En bottom-up parser kan være mere kompliceret at forstå end en top-down parser. Den måde LALR laver sin bottom-up parsing, er ved hjælp af shifting/reduce (skiftning/reduktion). Den starter med strengen og en tom stak, når den så læser den første del fra strengen, skal den vælge om den vil skifte eller reducere. Da reduktionen sker på stakken, med de øverste elementer, startes der altid med et skift. Første del af strengen bliver så læst over på stakken, og nu skal parseren vurdere, om det som ligger på stakken kan reduceres. Hvis ikke den kan reduceres, skal den skifte noget mere fra strengen over på stakken. Parseren skal fortsætte med dette, indtil strengen er tom, og der kun er én ting i stakken. Hvis man ser på billedet 3.4, kan man se, at efter første del af sætningen er læst over på stakken, kan den reduceres, fra A til var . Efter første reduktion skal den skifte to gange inden der kan reduceres noget igen. Denne reduktion omhandler dog kun øverste del af stakken, og den samme del, tredje del, på strengen kan reduceres tre gange. Det er først ved del nummer syv på strengen (14. skridt i genereringen af parsetræet) at den får tilstrækkeligt over på stakken til at den kan reducere flere dele fra stakken sammen til én.

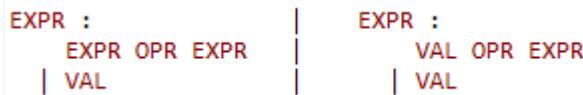


Figur 3.4. Bottom-up parsing, billede fra wikipedia[33].

Denne parser skal ikke bruge lookahead for at lave left recursion, derudover er det heller ikke et problem med right recursion. Hvis der er right recursion skal hele strengen dog læses over på stakken og reduceres bagefter, dette er ikke en effektiv metode at parse på, men det er muligt for en bottom-up parser. Derudover er der ikke problemer med, at flere produktioner har samme token i venstre side af deres definering, da den skal bruge hele strengen der passer ind i produktionen, inden den reducerer det. Det er muligt at lave bottom-up parsing på andre måder, LR bruger en mindre effektiv metode i forhold til størrelse på parse tabellen, men en stærkere måde i forhold til mængden af grammatikker den kan parse. Generelt har bottom-up shift/reduce mange flere grammatikker den kan parse, end top-down parseren.

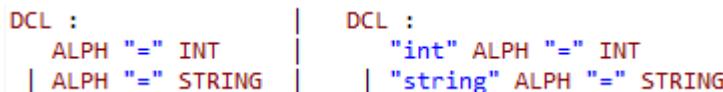
LL

En LL parser bruger en PDA. Benyttes der ikke lookahead bruges en ND PDA, og en deterministisk PDA, hvis lookahead benyttes. LL betyder "Left to right, Leftmost derivation", og i en LL parser vil der blive brugt top-down parsing. Når der bliver brugt top-down parsing, er det umuligt at parse noget med left-recursion, se figur 3.5, til venstre, for fejlen. Parseren vil ikke være i stand til at finde ud af, hvor mange gange produktionen skal kaldes lige efter hinanden, uden et uendeligt antal lookaheads. Det er ikke muligt at bruge et uendeligt lookahead, så grammatikken skal skrives om, som det ses på figur 3.5 til højre.



Figur 3.5. Til venstre: Left-recursion, til højre: en mulig løsning

En LL parser læser fra venstre mod højre, og for at undgå tvetydighed må der ikke være mulighed for, at en inputstrengh kan have to muligheder i grammatikken. Dette kan forekomme hvis der er to produktioner, der begge har samme token eller produktion i venstre side, se eksempel på figur 3.6 til venstre. Dette kan, som tidligere beskrevet, løses med lookahead. Lookaheadelet skal være lige så langt, som antallet af tokens eller produktioner efter hinanden. F.eks. som set på figur 3.6 til venstre er der to identiske tokens i træk, og dette kan løses ved et lookahead. Det er også muligt at løse tvetydighed, ved at ændre grammatikken, f.eks. ved at tilføje en token foran de to identiske tokens, som kan ses på figur 3.6 til højre. Dette er muligt da grammatikken skal læses fra venstre mod højre.



Figur 3.6. Til venstre: tvetydighed, til højre: en mulig løsning

LALR

LALR står for LookAhead Left to right Rightmost derivation. En LALR parser bruger bottom-up princippet. LALR parseren er en simplificeret udgave af en LR parser, som

er udviklet til at være mere hukommelseseffektiv, i forhold til LR (Left to right, Right most derivation) grammatikker. Den har fordelen af ikke at skulle "backtracke" når den parser, fordi LALR er bygget til at lave lookahead. LR(1) parseren er en meget "stærk" parser, der kan tage grammatikker som LALR(1) ikke kan, til gengæld så er LALR(1) hurtigere og kræver mindre hukommelse. På trods af at en LR parser generelt kan tage flere grammatikker, kan en LR(0) parser ikke tage lige så mange grammatikker som LALR(1).

Det er svært at sammenligne LALR og LL parsere, da det ikke er alle grammatikker der ligger i både LL og LALR. En LL grammatik kan være mere intuitiv at skrive, da leftmost derivation er mere logisk end rightmost derivation. Når LALR grammatik skal parses, bruges en version af PDA hvor man har indsæt sæt fra en deterministisk finit automat(DFA) imellem stak elementerne.

LALR(1) parseren er den mest brugte parser, af den grund at en LALR(1) parser har et stort segment af grammatikker den kan benytte, samtidig med at den er hurtig og kræver en forholdsvis lille mængde hukommelse.

3.1.4 Kontekstfri grammatik

Programmeringssprog er bygget op af en endelig mængde af strenge, som er lavet ud fra et endelig alfabet. Til at beskrive dette kan en kontekstfri grammatik benyttes. I afsnit 3.1.2 blev der beskrevet, at en KFG består af en række produktionsregler for et programmeringssprog, og disse regler bruges til at definere hvilke sekvenser af strenge, som det er tilladt at anvende i sproget.

I kodeksempel 3.2 ses en simplificeret udgave af den kontekstfrie grammatik for VMP. Grammatikken er lavet med standarden EBNF (Extended Backus-Naur Form). Da det er vigtigt at grammatikken ikke er tvetydig, så der ikke kan dannes to forskellige parsetræer ud fra samme grammatik, har vi undersøgt grammatikken for tvetydighed ved hjælp af værktøjet dk.brics.grammar[34]. Den egentlige grammatik der bruges sammen med den valgte compiler generator kan ses i bilag B. Grammatikken som bruges videre er LALR(1), hvilket vil sige, at den kan parses med en LALR parser, som parser input fra venstre til højre og konstruerer en rightmost derivation. Forskellen på grammatikken i dette afsnit, samt den der bruges med compiler generatoren uddybes i afsnit 4.1.2 og 4.2.1.

I projektets kontekstfrie grammatik er den initierende non-terminal `program`, og terminaler er indkapslet med anførselstegn (""). For at gøre grammatikken mere kompakt og forståelig, vil der eksempelvis blive skrevet ("a" – "z") eller ("0" – "9"), som betyder et bogstav mellem a og z eller et tal mellem 0 og 9.

```

1 program = {function};
2
3     function = "function" word "(" [formalparameters] ")" {statementsdcl} "end"
4         function";
5
6     statementsdcl = (declaration|control|build|move|functioncall|assignment);
7
8     statements = (control|build|move|functioncall|assignment);
9
10    functioncall = word "(" [actualparameters] ")";
11
12    formalparameters = formalparameter

```

```

12     | formalparameter "," formalparameters;
13
14 formalparameter = (intid|blockid);
15
16 actualparameters = actualparameter
17     | actualparameter "," actualparameters;
18
19 actualparameter = (value|blocktype|blockid);
20
21 declaration = "int" intid "=" expressions
22     | "block" blockid "=" blocktype;
23
24 assignment = intid "=" expressions
25     | blockid "=" blocktype;
26
27 operator = ("+" | "*" | "/" | "-" | "%");
28
29 expressions = expression [operator expressions];
30
31 expression = value
32     | "(" expressions ")";
33
34 move = direction "(" expressions ")";
35
36 direction = ("forward" | "back" | "up" | "down" | "left" | "right");
37
38 control = "if" "(" logcalexpressions ")" {statements} "end if" {elseif} [else]
39     | "repeat" "(" expressions ")" {statements} "end repeat";
40
41 elseif = "else-if" "(" logcalexpressions ")" {statements} "end if"
42
43 else = "else" {statements} "end else";
44
45 value = (number|intid);
46
47 intid = word;
48
49 blockid =(letter|"!"|"?|"&");
50
51 logcalexpressions = logcalexpression [("AND"|"OR") logcalexpressions];
52
53 logicaloperator = ("<="|">="|"<"|">"|"="|"/=");
54
55 logcalexpression = expression logicaloperator expression
56     | "(" logcalexpressions ")";
57
58 orientation = ("top"|"front"|"side");
59
60 build = "build" "(" orientation ")" {block} "end build"
61     | "build" letter;
62
63 block = ("_"|"!"|"?"|"&"|letter|newline);
64
65 letter = (( "a" - "z") | ( "A" - "Z"));
66
67 number = (( "1" - "9") {("0" - "9") } | "0");
68
69 word = letter {letter};
70
71 blocktype = ("AIR" | "STONE" | "GLASS" | "GRASS" | "DIRT" | "BEDROCK" | "WATER"
72     | "LAVA" | "SAND" | "GRAVEL" | "GOLDORE" | "IRONORE" | "COALORE"
73     | "LOG" | "LEAVES" | "LAPISORE" | "SANDSTONE" | "DIAMONDRE"
74     | "REDSTONEORE" | "ICE" | "SNOW" | "CACTUS" | "CLAY" | "PUMPKIN"
75     | "BROWNMUSHROOMBLOCK" | "REDMUSHROOMBLOCK" | "MELONBLOCK" | "
76         MYCELIUM"
77     | "COCOA" | "EMERALDORE" | "HARDCLAY" | "LEAVES2" | "LOG2"

```

```

77      | "STAINEDCLAY" | "PACKEDICE" | "COBBLESTONE" | "PLANKS" | "SPONGE"
78      | "DISPENSER" | "STICKYPISTON" | "WOOL" | "GOLDBLOCK" | "TNT"
79      | "BOOKSHELF" | "MOSSYCOBBLESTONE" | "TORCH" | "MOBSPAWNER"
80      | "OAKSTAIRS" | "CHEST" | "REDSTONEWIRE" | "CRAFTINGTABLE" | "
81          FARMLAND"
82      | "FURNACE" | "STONESTAIRS" | "FENCE" | "MONSTEREGG" | "STONEBRICK"
83      | "IRONBARS" | "GLASSPANE" | "CAULDRON" | "SANDSTONESTAIRS" | "
84          FLOWERPOT"
85      | "PRISMARINE" | "SEALANTERN" | "NETHERRACK" | "SOULSAND" | "
86          GLOWSTONE"
87      | "QUARTZORE" | "OBSIDIAN");

```

Kodeeksempel 3.2. Kontekstfri grammatik

Den kontekstfrie grammatik starter ud med non-terminalen `program`, der gør det muligt at oprette flere funktioner, og laver hovedindkapslingen af programmet. Funktioner startes med keyword `"function"`, efterfulgt af navnet på den funktion man vil lave. Derefter er der to parenteser, med mulighed for at placere parametre i mellem, til at sende med ind i funktionen. Dette efterfølges af `{statementdcl}` som giver mulighed for gentagelse af non-terminalen `statementdcl`, som danner alt indholdet af funktionen, i form af deklareringer, opgaver, kontrolstrukturer, `build`-udtryk, kald til andre brugerdefinerede funktioner og `move`-kommandoen. Til sidst afsluttes hovedindkapslingen af programmet med keyword `"end function"`.

Whitespace bliver hovedsageligt ignoreret i VMP, hvilket vil sige at der efter alle terminaler kan være et vilkårligt antal whitespaces, uden at det ændrer meningen af programmet. Der er dog et par undtagelser; den første er hvis der ikke er nogen whitespace imellem to terminaler, så kan det ske, at de to terminaler vil blive betragtet som én. Den anden undtagelse forekommer når der bruges `end repeat`, `end if`, `end build` eller `end function`. Disse skal allesammen have præcist ét whitespace i mellem `end` og det efterfølgende ord, da deres terminaler er defineret sådan.

Linjeskift bliver også generelt ignoreret, dog så gør det ikke når det bliver brugt inde i et `build`-udtryk, da det her bruges til at definere lag af en konstruktion. Disse linjeskift bliver her opfanget af `newline`, der kan findes som en del af non-terminalen `block`.

I kodeeksempel 3.3 ses et eksempel på et kort program skrevet ud fra grammatikken i kodeeksempel 3.2.

```

1 function box()
2     block b = STONE
3     block & = STONE
4     repeat(2)
5         build(top)
6             b b
7             & b
8         end build
9         up(1)
10    end repeat
11 end function

```

Kodeeksempel 3.3. Angivelse af simpel 2x2 kasse

I bilag A kan der ses en bottom-up rightmost derivation af programmet fra kodeksempel 3.3.

3.1.5 Typer og keywords

I VMP er der forskellige typer tilgængelige for programmøren, og disse typer kan have forskellige værdier. I dette afsnit vil typerne samt deres værdier blive gennemgået, og kodeksempler for syntaksen krævet for anvendelse af disse typer vil blive fremvist. Der er to kontrolstrukturer tilgængelige i VMP, som gør det muligt at strukturere og yderligere arbejde med koden. Disse kontrolstrukturer er **if** og **repeat**, som er henholdsvis betinget selektiv og iterativ.

VMP indeholder to datatyper, **integer** og **block**, og disse to typer anses for at være de to grundlæggende typer for programmeringssproget.

Integer datatypen indeholder, som navnet af datatypen angiver, en numerisk værdi inden for grænserne af integer, beskrevet i afsnit 2.3.1. Når programmøren vil anvende integer datatypen, skal programmøren foretage en eksplisit typeerklæring, på den variabel som programmøren vil anvende som en integer type. Integer datatypen kræver, at der bliver angivet en variabel, der er to eller flere karakterer langt, som ikke allerede er angivet som en anden type; en tildelingsoperator (=) og en numerisk værdi. Samtidig kræves det, at variablen bliver initialiseret ved deklarering. Se kodeksempel 3.4 for angivelse af integer variabel.

```
1 int variabelnavn = 17
```

Kodeksempel 3.4. Angivelse af integer variabel

Programmører kan i VMP anvende aritmetiske udtryk på forskellige variabler af datatypen integer, og reglerne samt syntaksen for disse udtryk vil nu blive listet og gennemgået ud fra to integer variabler ”*x*”, ”*y*”.

- **Addition** angives ved + mellem værdier eller variabler, ”*x* + *y*”.
- **Subtraktion** angives ved – mellem værdier eller variabler, ”*x* – *y*”.
- **Multiplikation** angives ved * mellem værdier eller variabler, ”*x* * *y*”.
- **Division** angives ved / mellem værdier eller variabler, ”*x/y*”. Division vil altid afrunde til laveste heltal.
- **Modulus** angives ved % mellem værdier eller variabler, ”*x%y*”. Modulus vil returnere resten fundet efter division.
- **Parenteser** angives ved (før og efter) værdier eller variabler, ”(*x+y*)**x*”. Parenteser gør det muligt for programmøren at ændre prioritering af operatører i udregningen af et udtryk.

Block datatypen er VMPs datatype, der modellerer de forskellige blokke i Minecraft. **Block** datatypen indeholder værdier i form af blokke fra Minecraft (f.eks. kan værdien være

LÖG, som er en træblok i Minecraft). Når programmøren vil anvende **block** datatypen, skal programmøren foretage en eksplisit typeerklæring, på den variabel som programmøren vil anvende som en **block** type. **Block** datatypen kræver, at der bliver angivet en variabel, der ikke allerede er genkendt som værende deklareret, og variablerne må kun være én karakter i længde; en tildelingsoperator (=) og en **block** værdi. **Block**-variabler skal også initialiseres ved deklarering. Se kodeeksempel 3.5 for angivelse af **block** variabel.

```
1  block b = LOG
```

Kodeeksempel 3.5. Angivelse af **block** variabel

Disse to datatyper er de eneste elementære datatyper i VMP, som indeholder værdier. Ud over disse datatyper, kan programmører anvende prædefinerede udtryk og funktioner.

build keywordet bruges til at bygge blokkene inde i Minecraft for programmøren. Når **build** keywordet anvendes, er det muligt at sende én blokvariabel med, som skal bygges, ved at anvende **build** keywordet, efterfulgt af ét mellemrum samt ID for den ønskede blok der skal bygges, se evt. kodeeksempel 3.6. **Build** vil i dette tilfælde kun bygge én blok af typen tildelt dette ID. Keywordet **build** kan også bruges på en anden måde, hvis man eventuelt vil bygge flere blokke ad gangen, se kodeeksempel 3.7. **Build** kræver i disse tilfælde en orienteringsparameter (front, top, bottom).

Når programmøren vil definere et bygningsmønster i **build**, skal det udføres som vist i kodeeksempel 3.7.

```
1  function front_mur()
2    block s = STONE
3    build s
4    forward(1)
5    build s
6  end function
```

Kodeeksempel 3.6. Angivelse af **build**-keywordet hvor der kun skal bygges en blok

```
1  function front_mur()
2    block s = STONE
3
4    build(front)
5      s - s - s - s - s - s
6      s s s s s s s s s s
7      s s s s s s s s s s // - betyder at blokken springes over
8      s s s s - - - s s s s
9      s s s s - - - s s s s
10     s s s s - - - s s s s
11   end build
12 end function
```

Kodeeksempel 3.7. Angivelse af **build**-keywordet hvor der bygges flere blokke

Når funktioner defineres, skal dette angives med keywordet **function**; navn, som er et vilkårligt antal karakterer langt, efterfulgt af en startparentes "(", mulige parametre, og en slutparentes ")". Se kodeeksemplerne 3.7 og 3.8 for angivelse af en funktion.

```
1  function Trappe(trin)
2    block s = STONE
3    block ! = STONE
4    block ? = STONE
```

```

5   repeat(trin)
6     build(top)
7       s s
8       s s
9       !
10    end build
11    up(1)
12  end repeat
13 end function

```

Kodeeksempel 3.8. Angivelse af funktion

Kontrolstrukturer er til for at hjælpe med at styre forskellige elementer i funktionerne. Der eksisterer to kontrolstrukturer, som kan bruges i VMP: `if()` og `repeat()`. De kan henholdsvis sikre, at en specifik stump kode kun bliver kørt, hvis et forudbestemt parameter er opfyldt, eller gentage en specifik stump kode et bestemt antal gange. Kontrolstrukturerne benyttes ved først at kalde henholdsvis `if()` eller `repeat()`, hvorefter den stump kode der ønskes kørt inden for kontrolstrukturen skrives. Derefter afsluttes kontrolstrukturen med at gentage den valgte kontrolstruktur med et `end` præfix, som vist i kodeeksempel 3.9

```

1 block b = STONE
2 if(1 = 1)
3   build(b)
4   up(1)
5   build(b)
6 end if

```

Kodeeksempel 3.9. If kontrolstruktur

Både `if()` og `repeat()` kontrolstrukturerne tager én eller flere parametre. Ved `if()` skal parameteren være et boolsk udtryk, som returnerer enten værdien *true* eller *false*. Evalueres udtrykket til *true* udføres den stump kode som kontrolstrukturen indeholder. `repeat()` tager kun én parameter, som angiver hvor mange gange kodestumpen skal udføres. Parameteren i `repeat()` kan kun være af datatypen `integer`, i form af en enkelt variabel eller et aritmetisk udtryk. Se kodeeksempel 3.8, hvor variablen `trin` overføres gennem en funktionsparameter.

3.1.6 Reserverede ord

Opsummering af de reserverede ord og tegn som VMP udnytter. Alle tegn og ord, som er integreret i programmeringssproget er opdelt i nedenstående tabel.

- **Identifier** gruppen indeholder de reserverede ord og tegn, som kan bruges i programmet til at opsætte variable af forskellige typer. Denne gruppering indeholder reglerne for tegn der kan bruges i navne på variable af enten typen `int` eller `block`, samt tegnene der anvendes som control blocks i `build`-udtrykket.
- **Keyword** gruppen indeholder reserverede ord, som bruges til variabel deklaration, kontrolstrukturer, funktioner samt de ord der anvendes til at bevæge dronen i Minecraft.

- **Value(orientation)** gruppen indeholder de ord der kan anvendes som værdier, der accepteres af `build`-udtrykket, til at bestemme orienteringen af en konstruktion.
- **Assignment operator** gruppen indeholder tegnet for tildeling.
- **Arithmetic operator** gruppen indeholder tegnene der bruges i forbindelse med aritmetiske udtryk.
- **Logical operator** gruppen indeholder de ord og tegn der bruges i forbindelse med boolske udtryk.
- **Value(blocktypes)** gruppen indeholder de ord der bruges til at beskrive værdier af `block` datatypen.

Reserverede ord/symboler	Type	Beskrivelse
(['a'-'z'] [‘A’-‘Z’]) ([‘a’-‘z’] [‘A’-‘Z’])+	Identifier(int)	Navn på int variabler
['a'-'z'] [‘A’-‘Z’]	Identifier(block)	Navn på block variabler
'!'	Identifier(block start)	Tegnet markerer start positionen for bygning af blocks i en build konstruktion
'?'	Identifier(block end)	Tegnet markerer slut positionen for bygning af blocks i en build konstruktion
'&'	Identifier(block start/end)	Tegnet markerer både start og slut positionen for bygning af blocks i en build konstruktion
"int"	Keyword	Benyttes til at deklarere heltals variabler
"block"	Keyword	Benyttes til at deklarere block variabler
"build"	Keyword	Bruges på to forskellige måder: 1. "build" efterfulgt af navnet på en block for at bygge én block på nuværende position 2. "build" efterfulgt af en orientation i parenteser for at starte en større build konstruktion
"end build"	Keyword	Benyttes til at afslutte en build konstruktion
"if"	Keyword	Benyttes til at lave en kontrolstruktur der udfører en række opgaver i tilfælde af at et logisk udtryk har en boolsk værdi "true"

"else"	Keyword	Benyttes til at lave en kontrolstruktur der udfører en række opgaver i tilfælde af at et logisk udtryk i et if udtryk ikke er blevet opfyldt
"else-if"	Keyword	Benyttes til at lave en kontrolstruktur der udfører en række opgaver i tilfælde af at et logisk udtryk i et if udtryk eller et else-if udtryk ikke er blevet opfyldt, med sin egen betingelse
"end else"	Keyword	Benyttes til at markere enden på de opgaver der skal udføres af else kontrolstrukturen
"end if"	Keyword	Benyttes til at markere enden på de opgaver der skal udføres af if kontrolstrukturen eller else-if strukturen
"repeat"	Keyword	Kontrolstruktur der benyttes til at gentage et antal opgaver flere gange
"end repeat"	Keyword	Benyttes til at markere enden på de opgaver der skal udføres af repeat kontrolstrukturen
"function"	Keyword	Markerer starten på en funktion
"end function"	Keyword	Markerer enden på en funktion
"forward"	Keyword	Bruges til at bevæge "dronen" frem
"back"	Keyword	Bruges til at bevæge "dronen" tilbage
"up"	Keyword	Bruges til at bevæge "dronen" op
"down"	Keyword	Bruges til at bevæge "dronen" ned
"left"	Keyword	Bruges til at bevæge "dronen" til venstre
"right"	Keyword	Bruges til at bevæge "dronen" til højre
"front"	Value(orientation)	Værdi der bruges med build til at bestemme hvordan en konstruktion skal vende
"side"	Value(orientation)	Værdi der bruges med build til at bestemme hvordan en konstruktion skal vende

"top"	Value(orientation)	Værdi der bruges med build til at bestemme hvordan en konstruktion skal vende
'='	Assignment operator	Bruges til at lave opgaver
'+'	Arithmetic operator	Additions operator
'-'	Arithmetic operator	Subtraktions operator
'*'	Arithmetic operator	Multiplikations operator
'/'	Arithmetic operator	Division operator
'%'	Arithmetic operator	Modulo operator
'=='	Logical operator	Lig med operator
"<="	Logical operator	Mindre end eller lig med
operator		
">="	Logical operator	Større end eller lig med
operator		
'<'	Logical operator	Mindre end operator
'>'	Logical operator	Større end operator
'/='	Logical operator	Ikke lig med operator
"AND"	Logical operator	Og operator
"OR"	Logical operator	Eller operator

<pre>("AIR" "STONE" "GRASS" "DIRT" "BEDROCK" "WATER" "LAVA" "SAND" "GRAVEL" "GOLDORE" "IRONORE" "COALORE" "LOG" "LEAVES" "LAPISORE" "SANDSTONE" "DIAMONDRE" "REDSTONEORE" "ICE" "SNOW" "CACTUS" "CLAY" "PUMPKIN" "BROWNMUSHROOMBLOCK" "REDMUSHROOMBLOCK" "MELONBLOCK" "MYCELIUM" "COCOA" "EMERALDORE" "HARDCLAY" "LEAVES2" "LOG2" "STAINEDCLAY" "PACKEDICE" "COBBLESTONE" "PLANKS" "GRAVEL" "SPONGE" "DISPENSER" "SANDSTONE" "STICKYPISTON" "WOOL" "GOLDBLOCK" "TNT" "BOOKSHELF" "MOSSYCOBBLESTONE" "TORCH" "MOBSPAWNER" "OAKSTAIRS" "CHEST" "REDSTONEWIRE" "CRAFTINGTABLE" "FARMLAND" "FURNACE" "STONESTAIRS" "FENCE" "MONSTEREGG" "STONEBRICK" "IRONBARS" "GLASSPANE" "CAULDRON" "SANDSTONESTAIRS" "FLOWERPOT" "PRISMARINE" "SEALANTERN" "NETHERRACK" "SOULSAND" "GLOWSTONE" "QUARTZORE" "OBSIDIAN")</pre>	Value(blocktypes)	Værdier til block variabler, som hver især svarer til en bestemt block i Minecraft
---	-------------------	--

3.2 Type checker

I dette afsnit vil der blive set på VMPs regler omkring scopes og typer i form af en kontekstuel analyse. Derefter vil der blive set på dele af VMPs formelle semantik, i form af operationel semantik.

3.2.1 Kontekstuel analyse

Den kontekstuelle analyse udformer et uformelt beskrevet regelsæt, i forbindelse med scopes og typer, der skal overholdes af programmører i dette programmeringssprog.

VMPs typesystem

Typesystemet indeholder regler for typeanvendelse i VMP.

Typesystemet indeholder eksplisitte typer, som en programmør kan deklarere, henholdsvis **function**, **int** eller **block**, samt en implicit type der ikke deklarereres af programmøren ved anvendelse af typen. **Boolean** er en implicit type i VMP, der kommer til udtryk gennem boolske udtryk.

Eksplisitte typer

For hver type som skal deklarereres eksplisit af programmører i VMP, er der specifikke retningslinjer for anvendelse af hver af disse typer. Fælles for alle eksplisitte typer er, at der skal anvendes et keyword, før en variabel, som beskriver datatypen af variablen.

- **Function** typen kræver at programmøren tildeler variable, af denne type, en funktionskrop. Funktionskroppen kan indeholde programkode, som udføres ved et funktionskald af en variabel af typen **function**. Når en funktionskrop deklarereres af programmører i VMP, er alle operationer, undtagen deklarering af nye funktioner, lovlige. Ved deklarering af en funktion, skal der anvendes et **function** keyword, efterfulgt af variablenavn, samt start- og slutparentes. Ved funktionskald angives funktionsvariablens navn, efterfulgt af start- og slutparentes.
- **Integer** datatypen kræver at programmøren ved deklarering initialisere variablen med et positivt heltal, som er inden for de tilladte værdier i JavaScript, som nævnt i afsnit 2.3.1. Når en integer variabel deklarereres, skal der anvendes et **int** keyword.
- **Block** datatypen kræver at programmøren tildeler variable, af **block** typen, en værdi der kan genkendes af **block** datatypen. Disse værdier er baseret på bloktyper i Minecraft, og en liste af **block** types værdier ses i den kontekstfrie grammatik, under kodeeksempel 3.2. Når en **block** variabel deklarereres, skal der anvendes et **block** keyword.

Implicitte typer

For hver anvendelse af **if**-kontrolstrukturen, anvendes VMPs eneste implicitte datatype, **boolean**.

- **Boolean** datatypen bruges til at angive værdien for et logisk udtryk, der udformes under anvendelse af logiske operatorer, og indeholder to værdier; sandt & falsk.

Logiske operatorer

Logiske udtryk udformes med logiske operatorer. Reglerne for anvendelse af disse operatorer gennemgås i dette afsnit. De logiske operatorer understøttet af VMP er følgende:

- Mindre end ($a < b$) - ' $<$ ' operatoren kontrollerer om værdien af venstresiden er mindre end højresiden. Hvis a er mindre end b evalueres udtrykket til sandt, ellers falsk.
- Større end ($a > b$) - ' $>$ ' operatoren kontrollerer om værdien af venstresiden er større end højresiden. Hvis a er større end b evalueres udtrykket til sandt, ellers falsk.
- Lig med ($a = b$) - ' $=$ ' operatoren kontrollerer om værdien af venstresiden er lig med højresiden. Hvis a er lig med b evalueres udtrykket til sandt, ellers falsk.
- Ikke lig med ($a \neq b$) - ' \neq ' operatoren kontrollerer om værdien af venstresiden er lig med højresiden. Hvis a er lig med b evalueres udtrykket til falsk, ellers sandt.
- Mindre end eller lig med ($a <= b$) - ' $<=$ ' operatoren kontrollerer om værdien af venstresiden er mindre end **eller** lig med højresiden. Hvis a er mindre end eller lig med b evalueres udtrykket til sandt, ellers falsk.
- Større end eller lig med ($a >= b$) - ' $>=$ ' operatoren kontrollerer om værdien af venstresiden er større end **eller** lig med højresiden. Hvis a er større end eller lig med b evalueres udtrykket til sandt, ellers falsk.
- Konjunktion ($(a < b) AND (a = 2)$) - ' AND ' operatoren kontrollerer om to Boolean værdier, den ene til venstre og den anden til højre for operatoren, begge er sande. Hvis begge værdier er sande, evalueres udtrykket til sandt, ellers falsk.
- Disjunktion ($(a < b) OR (a = 2)$) - ' OR ' operatoren kontrollerer op til to Boolean værdier, den ene til venstre og den anden til højre for operatoren, om en af dem er sande. Hvis en sand værdi er fundet, evalueres udtrykket til sandt, ellers falsk.

De logiske operatorer kan kun anvendes på udtryk af datatypen **int**. Når de variable deklarereres til enten **block** eller **int**, og opfylder reglerne angivet i typesystemet, så kan disse variable anvendes inden for variablens scope.

VMPs scope

Inden for scopes skelnes der mellem statiske og dynamiske scopes. Med statisk scope menes der, at strukturen af programkoden bestemmer, hvilken variabel der henvises til ved et kald til en variabel. Se kode eksempel 3.10 for statisk scope eksempel. Dynamisk scope er, modsat statisk scope, ikke låst af programkodens struktur, hvilket betyder at variable værdier skifter alt efter positionen af de forskellige variable i programkoden. Se kode eksempel 3.11 for dynamisk scope eksempel.

```

1 const int b = 5;
2 int foo()
3 {
4     int a = b + 5;
5     return a;
6 }
7
8 int bar()
9 {
10    int b = 2;
11    return foo(); \\ uses const int b = 5 value in the assignment operation
12 }
13
14 int main()
15 {
16     foo(); \\ returns 10
17     bar(); \\ returns 10
18     return 0;
19 }
```

Kodeeksempel 3.10. Eksempel på static scope

```

1 const int b = 5;
2 int foo()
3 {
4     int a = b + 5;
5     return a;
6 }
7
8 int bar()
9 {
10    int b = 2;
11    return foo(); \\ uses int b = 2 value in the assignment operation
12 }
13
14 int main()
15 {
16     foo(); \\ returns 10
17     bar(); \\ returns 7
18     return 0;
19 }
```

Kodeeksempel 3.11. Eksempel på dynamisk scope

VMP er udviklet med et statisk scope i tankerne, da de undersøgte børnevenlige programmeringssprog primært anvender statisk scoping. Fokus i dette afsnit vil derfor være på mulighederne inden for statisk scoping.

Scope for en variabel er det sted i programkoden, hvor variablen kan tilgås[35, 36]. Når en variabel bliver kaldt, læses værdien i det tætteste scope, for den funktion der kalder variablen. Hvis variablen ikke findes, mangler der en deklarering af variablen. Forskellige niveauer af scope kan opdeles følgende[37, 38]:

- **Block/Local scope** er et scope, der er indkapslet af blokke. F.eks. anvender loops, der initialiserer en ny variabel ved hvert gennemløb, block scope da variablen der initialiseres i loopet ikke er tilgængelig andre steder i programkoden, og variablen bortfalder når loopet, hvor variablen er initialiseret, er gennemløbet. Se kodeeksempel 3.12

- **Function scope** er et scope, som indkaptes af funktioner. Variable i dette scope kan kun tilgås inde i funktionerne, hvor de deklarereres, og det er derfor ligegyldigt for en funktion A, om en anden funktion B har deklareret en variabel identisk med den variabel, som funktion A vil deklarere. Ændringer på værdier af variabler i function scope er dermed fastlagt til variabler inden for funktioners indkapsling (`end function`). Se kodeeksempel 3.13
- **Program/Global scope** er det yderste scope, hvor alle deklarationer, og dermed værdier, af variable kan tilgås af alle funktioner i programmet. 3.14

```

1 unsigned int sum_of_squares(const unsigned int N)
2 {
3     unsigned int ret = 0;
4     for (unsigned int n = 0; n <= N; n++)
5     {
6         const unsigned int n_squared = n * n; //block scope
7         ret += n_squared; //block scope
8     }
9     return ret;
10 }
```

Kodeeksempel 3.12. Eksempel på block scope

```

1 int function0()
2 {
3     int i; // Function scope
4     void function2(int i); // Nested function scope
5     void function1(void); // Nested function scope
6     return 0;
7 }
```

Kodeeksempel 3.13. Eksempel på function scope i C

```

1 int x = 0; // Global scope
2 double y = 0.0; // Global scope
3 int function0() // Global scope
4 {
5     int i;
6     int function2(int i);
7     void function1(void);
8     return 0;
9 }
```

Kodeeksempel 3.14. Eksempel på program scope i C

I VMP er der foretaget valget om, at anvende function scope, da dette scope kan opfylde det gældende behov for opdeling af variabler, så hver funktion har sit eget scope med egne variabler. Ud over function scope, er et begrænset global scope valgt. Det begrænsede global scope anvendes kun i sammenhæng med `function` typen, så det er muligt for funktioner at kalde andre funktioner i global scope. Uden global scope ville det ikke være muligt at udføre funktionskald, af forskellige funktioner, i kroppen af funktioner. Med function scope og det begrænsede global scope, er det muligt at angive funktioner i global scope, som kan ses, og dermed kaldes fra, alle andre funktioner. Variabler kan kun deklarereres i function scope, og hver funktion kan dermed kun arbejde med variabler deklareret i funktionen.

3.2.2 Symboltabel

Symboltabellen i compileren til VMP, bliver benyttet i type checker fasen. Den bruges i *funktion scopes* til at opbevare midlertidigt information i form af de variabler der er deklareret, samt hvilken værdi der er assigget til dem. Når type checkerne forlader det enkelte funktion scope, bliver informationen slettet, da der nu ikke længere er brug for den. Herved kan det sikres, at alle deklarationer som benyttes i funktionen eksisterer og overholder de scope rules, der er bestemt for VMP.

3.2.3 Operationel semantik

I dette afsnit vil der blive gennemgået den formelle semantik for VMP. Formel semantik kan beskrives med small-step eller big-step semantik, hvor small-step semantik i sidste ende er identisk med big-step semantik, men udformningen af den formelle semantik er mere detaljeret under anvendelse af small-step semantik end big-step semantik[39]. Big-step semantikken fokuserer på start- og slutresultat af en beregning, hvor small-step semantikken undersøger processen for udregningen mellem start og slut.

Der vil først blive grundigt beskrevet semantikken for '+' operatoren, hvor '+' henviser til det aritmetiske '+' symbol, både for small-step og big-step semantik. Hvorefter der vil blive redejort for resten af den formelle semantik for VMP.

Small-step semantik beskriver hvert enkelt trin i en beregning, og ankommer efter et vilkårligt antal trin til den endelige formelle beskrivelse af beregningen.

Trinene i small-step semantik har formen $a \Rightarrow a'$, hvor a' kaldes den mellemliggende konfiguration, som kan være enten en terminal n eller et aritmetisk udtryk.

Når '+' operatoren anvendes, skal der i small-step semantik beskrivelsen angives en formel beskrivelse for alle de mulige anvendelsesmuligheder af operatoren, så i tilfælde af $2 + 3$ skal der f.eks. angives en formel semantisk beskrivelse, af både $2 + 3$ samt $3 + 2$ for de to mulige anvendelser af operatoren.

Formen for den formelle beskrivelse af '+' operatoren med small-step semantik vil nu blive opstillet:

- $a ::= a_1 + a_2$
- $[PLUS - L_{SS}] \frac{a_1 \Rightarrow a'_1}{a_1 + a_2 \Rightarrow a_1 + a'_2}$
- $[PLUS - R_{SS}] \frac{a_1 \Rightarrow a'_1}{a_2 + a_1 \Rightarrow a_2 + a'_1}$

$[PLUS - L_{SS}]$ og $[PLUS - R_{SS}]$ beskriver de formelle regler for anvendelse af '+' operatoren. Der mangler nu en formel beskrivelse for værdien af et udtryk, der anvender '+' operatoren, som forlængelse af ovenstående regler. Værdien angives med symbolet 'v'.

- $[PLUS - V_{SS}] v_1 + v_2 \Rightarrow v$ hvor $v_1 + v_2 = v$

Dermed er den formelle semantiske beskrivelse af '+' operatoren, med small-step semantik, blevet beskrevet. Den formelle beskrivelse vil nu blive gennemgået med **big-step semantik** af formen $a \rightarrow a'$:

- $a ::= a_1 + a_2$
- $[PLUS]_{BS} \frac{a_1 \rightarrow v_1 \quad a_2 \rightarrow v_2}{a_1 + a_2 \rightarrow v}$ hvor $v = v_1 + v_2$

Denne formelle beskrivelse angiver reglen for anvendelse af '+' operatoren, i VMP, med big-step semantik.

Formelle semantiske beskrivelser der anvender big-step semantik er, som vist i forskellen mellem eksemplerne for big-step og small-step semantik, væsentligt mindre detaljerede end small-step semantik beskrivelser, men indeholder samme endelige konklusion for den angivne additionsregel. Den formelle semantik for VMP vil blive gennemgået med big-step semantik.

Semantikken for VMP er baseret på en tilstand s . Når tilstanden s angives på formen $s \vdash (...)$, menes der at alt efterfølgende \vdash udføres i tilstanden s .

Big-step semantik for aritmetiske udtryk a angiver et aritmetisk udtryk, v angiver en værdi.

- $a ::= n | a_1 + a_2 | a_1 - a_2 | a_1/a_2 | a_1 \% a_2 | a_1 * a_2 | (a_1) | x = v$
- $[PLUS]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 + a_2 \rightarrow_a v}$ hvor $v = v_1 + v_2$
- $[MINUS]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 - a_2 \rightarrow_a v}$ hvor $v = v_1 - v_2$
- $[DIV]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1/a_2 \rightarrow_a v}$ hvor $v = v_1/v_2$ for $v_2 > 0$ OR $v_2 < 0$
- $[MOD]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \% a_2 \rightarrow_a v}$ hvor $v = v_1 \% v_2$
- $[MULT]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 * a_2 \rightarrow_a v}$ hvor $v = v_1 * v_2$
- $[PARENT]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1}{s \vdash (a_1) \rightarrow_a v_1}$
- $[NUM]_{BS} \frac{}{s \vdash n \rightarrow_a v}$ hvor $\mathcal{N}[\![n]\!] = v$
- $[VAR]_{BS} s \vdash x \rightarrow_a v$ hvor $s(x) = v$

De beskrevne aritmetiske udtryk er først og fremmest addition, subtraktion, division, modulus og multiplikation ($PLUS(+)$, $MINUS(-)$, $DIV(/)$, $MOD(\%)$, $MULT(*)$). Reglerne for disse udtryk er relativt identiske, da de udføres i en tilstand s , og kræver to aritmetiske udtryk, noteret ved \rightarrow_a , v_1 og v_2 . Disse udtryk ender alle med en værdi v , og dette gælder

for alle værdier af v_1 og v_2 . Division har en ekstra regel vedrørende værdien af v_2 , som kræver at denne værdi ikke er tallet nul, og dermed kan ingen værdier divideres med nul. Den endelige værdi v er produktet af operatoren ($+$, $-$, $/$, $\%$, $*$) brugt på værdierne v_1 og v_2 . Reglen for anvendelse af parenteser defineres i **PARENT**, og beskriver at ethvert aritmetisk udtryk kan anvende parenteser. Denne regel gør det muligt at opdele udtryk i parenteser. Operator hierarkiet, beskrevet i afsnit 2.2, angiver at udtryk i parenteser skal udføres før andre udtryk, og dermed kan programmører selv bestemme rækkefølgen for udregning i et udtryk, med parenteser. **VAR** reglen definerer anvendelsen af variabel deklarering. En variabel kan deklareret i en tilstand s hvor en værdi v tildeles til en variabel x . **NUM** angiver reglen for numeral, som gør det muligt i en tilstand s , at få den numeriske værdi af en variabel n .

Big-step semantik for boolske udtryk a angiver et aritmetisk udtryk, b angiver et boolsk udtryk og v angiver en værdi.

- $b ::= a_1 / = a_2 | a_1 = a_2 | a_1 > a_2 | a_1 < a_2 | a_1 \leq a_2 | a_1 \geq a_2$
 $|(b_1)|b_1 AND b_2 | b_1 OR b_2$
- $[EQUAL1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 = a_2 \rightarrow_b tt}$ hvor $v_1 = v_2$
- $[EQUAL2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 = a_2 \rightarrow_b ff}$ hvor $v_1 \neq v_2$
- $[NOTEQUAL1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 / = a_2 \rightarrow_b tt}$ hvor $v_1 \neq v_2$
- $[NOTEQUAL2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 / = a_2 \rightarrow_b ff}$ hvor $v_1 = v_2$
- $[GREATER1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 > a_2 \rightarrow_b tt}$ hvor $v_1 > v_2$
- $[GREATER2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 > a_2 \rightarrow_b ff}$ hvor $v_1 \not> v_2$
- $[LESSER1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 < a_2 \rightarrow_b tt}$ hvor $v_1 < v_2$
- $[LESSER2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 < a_2 \rightarrow_b ff}$ hvor $v_1 \not< v_2$
- $[LEQ1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \leq a_2 \rightarrow_b tt}$ hvor $v_1 \leq v_2$
- $[LEQ2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \leq a_2 \rightarrow_b ff}$ hvor $v_1 \not\leq v_2$
- $[GEQ1]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \geq a_2 \rightarrow_b tt}$ hvor $v_1 \geq v_2$
- $[GEQ2]_{BS} \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \geq a_2 \rightarrow_b ff}$ hvor $v_1 \not\geq v_2$
- $[PARENT]_{BS} \frac{s \vdash b_1 \rightarrow_b v_1}{s \vdash (b_1) \rightarrow_b v_1}$

- $[AND1]_{BS} \frac{s \vdash b_1 \rightarrow_b tt \quad s \vdash b_2 \rightarrow_b tt}{s \vdash b_1 AND b_2 \rightarrow_b tt}$
- $[AND2]_{BS} \frac{s \vdash b_i \rightarrow_b ff}{s \vdash b_1 AND b_2 \rightarrow_b ff}$ hvor $b_i \in \{1, 2\}$
- $[OR1]_{BS} \frac{s \vdash b_1 \rightarrow_b tt \quad s \vdash b_2 \rightarrow_b ff}{s \vdash b_1 OR b_2 \rightarrow_b tt}$
- $[OR2]_{BS} \frac{s \vdash b_1 \rightarrow_b ff \quad s \vdash b_2 \rightarrow_b tt}{s \vdash b_1 OR b_2 \rightarrow_b tt}$
- $[OR3]_{BS} \frac{s \vdash b_1 \rightarrow_b ff \quad s \vdash b_2 \rightarrow_b ff}{s \vdash b_1 OR b_2 \rightarrow_b ff}$

De boolske udtryk indeholder, som de aritmetiske udtryk, regler der er relativt identiske. Disse regler er EQUAL1(=), EQUAL2(≠), NOTEQUAL1(≠), NOTEQUAL2(=) GREATER1(>), GREATER2(≶), LESSER1(<), LESSER2(≶), LEQ1(≤), LEQ2(≶), GEQ1(≥) og GEQ2(≶). Alle disse udtryk er udtryk der sammenligner to aritmetiske værdier v_1 og v_2 , og tjekker henholdsvis efter; lig med, større end, mindre end, mindre end eller lig med, større end eller lig med. Et boolsk udtryk konkluderer på, om udtrykket er sandt(tt), eller falsk(ff).

Udtrykkene AND1(AND), AND2(AND), OR1(OR), OR2(OR) eller OR3(OR) er ligeledes relativt identiske regler. Disse udtryk behandler en boolsk værdi b . AND1, AND2, OR1, OR2 og OR3 følger samme operand principper. AND1 og AND2 kræver to udtryk b skal være sande, hvis hele udtrykket med AND skal være sandt, hvor OR1, OR2 og OR3 kun kræver at et udtryk b er sandt, for at udtrykket med OR er sandt. Boolsk udtryk kan, som aritmetiske udtryk, også opdeles i parenteser som defineret i reglen **PARENT**.

Big-step semantik for udtryk Semantikken for udtryk har formen $\langle S, s \rangle \rightarrow s'$ for at vise, at udtrykket S udført i tilstanden s , medfører et tilstandsskift til s' . a angiver et aritmetisk udtryk, b angiver et boolsk udtryk, S angiver et udtryk og v angiver en værdi.

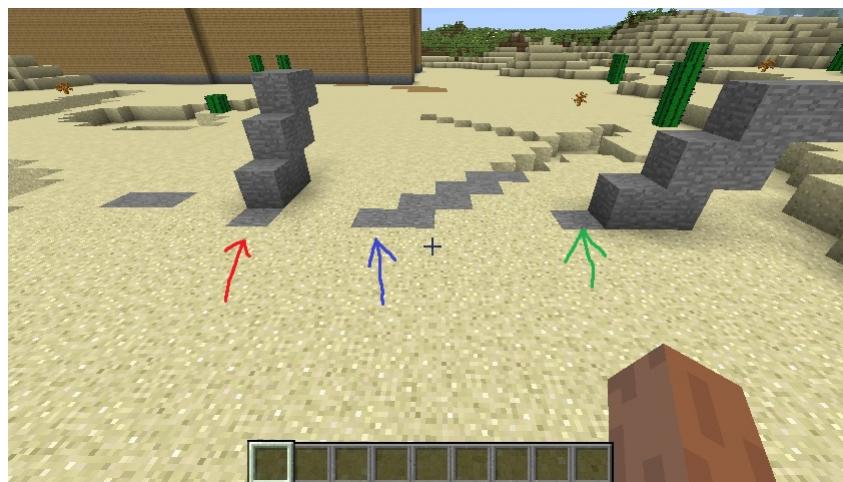
- $S ::= x = a | S_1; S_2 | if b then S_1 else S_2 | REPEAT(N)S$
- $[ASSIGN]_{BS} \langle x = a, s \rangle \rightarrow s[x \mapsto v]$ hvor $s \vdash a \rightarrow_a v$
- $[COMP]_{BS} \frac{\langle S_1, s \rangle \rightarrow s'' \quad \langle S_2, s'' \rangle \rightarrow s'}{\langle S_1; S_2, s \rangle \rightarrow s'}$
- $[IF_{true}]_{BS} \frac{\langle S_1, s \rangle \rightarrow s'}{\langle if b then S_1 else S_2, s \rangle \rightarrow s'}$ hvor $s \vdash b \rightarrow_b tt$
- $[IF_{false}]_{BS} \frac{\langle S_2, s \rangle \rightarrow s'}{\langle if b then S_1 else S_2, s \rangle \rightarrow s'}$ hvor $s \vdash b \rightarrow_b ff$
- $[REPEAT_{>0}]_{BS} \frac{\langle S, s \rangle \rightarrow s'' \quad \langle REPEAT(N')S, s'' \rangle \rightarrow s'}{\langle REPEAT(N)S, s \rangle \rightarrow s'}$
hvor $N' = N^{-1}(N - 1)$ hvor $N(N) > 0$
- $[REPEAT_0]_{BS} \langle REPEAT(N)S, s \rangle \rightarrow s$ hvor $N(N) \leq 0$

Ovenstående indeholder de definerede regler for udtryk. *ASSIGN* reglen for tildeling angiver, at for et udtryk a og en tilstand s kan en variabel x tildeles en værdi v . *COMP*, sammensætning, reglen tillader at en række af udtryk kan forekomme sekventielt. *IF_{true}* og *IF_{false}* tillader valg af et udtryk, ud fra værdien af et boolsk udtryk. *REPEAT_{>0}* tillader løkken at kalde *REPEAT* igen med et rekursivt kald, så længe tallet indtastet for N er sandt for $\mathcal{N}(N) > 0$. *REPEAT₀* angiver at *REPEAT* løkken stopper, hvis værdien af tallet er $\mathcal{N}(N) \leq 0$.

Big-step semantik for build build-udtrykket er den del af VMP, der fungerer som tegnebræt til de konstruktioner en programmør vil konstruere i Minecraft. Semantikken for build-udtrykket er relativt kompleks, og denne del vil derfor fokusere på big-step semantikken for build-udtrykket alene.

Semantikken for build-udtrykket er opdelt i flere dele som samlet set angiver de semantiske regler for build; orientation (position og orienteringsvinkel for spilleren i Minecraft), variabler, MCtilstand (typen af blokke i Minecraft verdenen) og Ascii-art (tegning af konstruktion i et plan). Hver af disse dele vil blive gennemgået, for at komme frem til helheden af semantikken for build-udtrykket.

Orientation er en regel, der omhandler værdier brugt til at skabe relationer, inde i Minecraft, mellem konstruktionen der skal bygges og spilleren som kalder funktionen der bygger konstruktionen. Reglen består af tre værdier: **front**, **side** og **top**. Disse værdier er angivet som keywords i VMP, og betydningen af disse keywords afhænger af ScriptCrafts fortolkning af henholdsvis **front**, **side** og **top**. Figur 3.7 viser effekten af orientation, inde i Minecraft, hvor værdierne **front**, **side** og **top** er markeret ved henholdsvis den grønne, røde og blå pil. Koden som konstruerer trappen ses i kodeeksempel 3.15.



Figur 3.7. Minecraft: stair(4) orientation

```

1 function stair(lenght)
2   block ! = STONE
3   block ? = STONE
4   block b = STONE
5   repeat(lenght)
6     build(front) //orientation angives som parameter til build
7     - ?
8     ! b

```

```

9     end build
10    end repeat
11 end function

```

Kodeeksempel 3.15. Kodeeksempel for en trappe

Var er en semantisk regel allerede gennemgået under **Big-step semantik for aritmetiske udtryk**. build-udtrykket benytter variabler af typen block, og disse variabler bruges som basis til at udforme Ascii-art reglen. Værdier for typen block, som genkendes i VMP, findes i sprogtabellen 3.1.6, under Value(blocktypes). Alle variablerne deklareret, kan findes under den givne tilstand s .

MCtilstand er den regel, som omhandler tilstanden for blokkene inde i Minecraft verdenen. Denne regel bruges for at beskrive hvordan build-udtrykket skal ændre på tilstanden af blokkene inde i Minecraft, når der f.eks. inde i Minecraft kaldes en funktion som bygger en konstruktion, ud fra det definerede Ascii-art i build-udtrykket. Tilstanden udformes ud fra den orientation, som en spiller har inde i Minecraft.

MCtilstanden er defineret som det rum Minecraft verdenen er opbygget af. Rummet i Minecraft består af tre akser; x, y og z, hvor ét givent punkt, defineret ud fra tre heltalsværdier n for henholdsvis x, y og z, på disse tre akser, har en ukendt værdi af typen block. Maximum værdierne for længden x, bredden y og højden z er henholdsvis 68, 719, 476, 736, 68, 719, 476, 736 og 256 for en 32-bit Minecraft verden.[3]

- $[MCtilstand]_{BS} \frac{s \vdash x \rightarrow n \quad s \vdash y \rightarrow n \quad s \vdash z \rightarrow n}{s \vdash (x, y, z) \rightarrow_a v}$ hvor v er type block.

Ændringen på MCtilstand kan opfattes som en tildeling af en ny blok, til den blok valgt ud fra de tre specificerede heltal for x, y og z.

Ascii-art reglen omhandler selve udformningen af kroppen af build-udtrykket. Kroppen indeholder block variabler, som har en plads der er relativ til andre block variabler i Ascii-art kroppen. Ascii-art kroppen gør brug af nogle specifikke control blocks; startpunkt !, slutpunkt ? og startpunkt samt slutpunkt &. Disse identifiers er block variable, som også angiver den start- og slutposition dronen, som bygger konstruktionen i Minecraft, har før og efter en konstruktion. Start- og slutposition er inkluderet, for at der kan defineres en algoritme der gennemløber det Ascii-art, som en programmør har angivet i kroppen af build-udtrykket. Denne algoritme bruges, for at kunne angive hvordan et Ascii-art omdannes til en reel konstruktion i Minecraft.

Ascii-art bygger ovenpå reglen for $[var]_{BS}$, og går ud fra tilstanden s indeholdende de variable af typen block, og tilføjer mening til de visuelle relationer mellem block variabler, ud fra følgende eksempel for Ascii-art:

```

1 block b = STONE      //block var
2
3   b b b
4   b b b      //Ascii-art
5   b b b

```

Kodeeksempel 3.16. Eksempel på Ascii-art

Ascii-art bygges ud fra et to-dimensionelt billede, som et plan, hvor programmøren 'tegner' et plan med variabler, som gerne skulle forestille den konstruktion som programmøren vil bygge i Minecraft. I kodeeksempel 3.16 forestiller konstruktionen, lavet af block variablen b, et plan i form af en 3×3 mur af block værdien STONE. Dette Ascii-art gennemløbes af algoritmen for build-udtrykket, og konstruktionen formes inde i Minecraft med samme form, som det "tegnede" Ascii-art.

Ascii-art kan kun indeholde block variable, og udvider alfabetet, der er tilgængeligt til deklarering af variabelnavne, med de tre controlblock identifiers !, ? og &.

Nu hvor reglerne for de separate dele af build-udtrykket er gennemgået, kan den formelle semantiske regel for build-udtrykket blive opstillet.

Den endelige build regel, som sammenkobler reglerne; orientation bestående af playerpos og mousepos, tilstanden s der indeholder alle variable, MCtilstand der beskriver tilstanden af blokke i Minecraft og Ascii-art der beskriver opsætningen af build-udtrykkets krop, er som følgende:

$$\begin{aligned} [build]_{BS} (playerpos, mousepos, s) \vdash & < build(orientation) Ascii - art, MCtilstand > \\ & \rightarrow MCtilstand' \end{aligned}$$

Denne formelle regel for build, udtrykker ikke hvordan overgangen fra $MCtilstand$ til $MCtilstand'$ forekommer, da denne overgang bestemmes ud fra en algoritme, der håndterer det angivede Ascii-art, og på basis af dette udformer konstruktionen i $MCtilstand$, således at overgangen til $MCtilstand'$ forekommer. Algoritmen anvendt i VMP, som implementerer overgangen fra $MCtilstand$ til $MCtilstand'$, ses i kodeeksempel 4.29 under afsnit 4.5. Overgangen fra $MCtilstand$ til $MCtilstand'$ kan opfattes som en tildeling af en ny blok til det valgte koordinat.

3.3 Opsamling af compilerdesign

I dette afsnit er der redegjort for hvordan en compiler fungerer, og noget teori omkring nogle af de emner vi benytter i forbindelsen med udviklingen af VMP. Da det ikke er tiltænkt at programmeringssproget skal kompileres til andre målsprog end JavaScript med ScriptCraft-extension, ses det ikke nødvendigt, at oversætte sproget til en IR (Intermediat Representation). Da der ikke oversættes til en IR, vil fasen "oversætter" ikke indgå i rapporten. Samtidig benyttes der heller ikke en optimizer, som beskrevet i afsnit 3.0.6. Den optimering, der er i denne compiler, sker i kodegenereringsfasen, se afsnit 4.6. Optimering sker i kodegenereringsfasen, da det er i denne fase, at blokkene læses ind, og det ses mest logisk at optimere på dem på dette tidspunkt i compileren.

Selve kodegenereringen af denne compiler, vil blive beskrevet i afsnit 4.5, i det efterfølgende kapitel, kapitel 4, omkring implementeringen af compileren.

Implementering 4

I dette kapitel vil implementering af softwareløsningen blive gennemgået. Løsningen er baseret på beskrivelserne i kapitel 3. Compileren er kodet i programmeringssproget Java og målsproget er JavaScript med ScriptCraft-udvidelse, termer fra disse programmerings-sprog vil derfor blive anvendt gennem kapitlet.

Compilerens struktur og relevante dele af compilerens funktionalitet vil blive beskrevet. Overordnet består compileren af fire dele; scanner, parser, type checker og code generator. Disse fire dele vil hver især blive beskrevet, med relevante kodeeksempler. Yderligere vil optimeringen også blive beskrevet, og til sidst en beskrivelse af den grafiske brugergrænseflade.

4.1 Compiler generatorer

I dette afsnit vil der blive beskrevet to compiler generatorer. En compiler generator tager en grammatik og genererer en parser og en scanner (omtalt som lexer i JavaCC og SableCC, men er det samme som en scanner). De to valgte compiler generatorer er JavaCC og SableCC, de er valgt på baggrund af, at de er to af de mest anvendte inden for området, og fordi de tager to forskellige typer af grammatik, nemlig LALR og LL. JavaCC og SableCC vil også blive sammenlignet ud fra, hvordan grammatikken indskrives i de to compilere og hvordan output er. Formålet med afsnittet er at bestemme hvilken compiler generator der er bedst udrustet til at generere dette projekts scanner og parser.

4.1.1 JavaCC

JavaCC er en top-down compiler generator, der kan generere parsere til alle LL(k) sprog. I forbindelse med at bruge JavaCC, har vi valgt at bruge et plugin der er blevet lavet til Eclipse. Dette gør det muligt, at skrive grammatikken ind i Eclipse og få feedback løbende under indskrivningsprocessen, og ved eventuelle ændringer eller fejlfinding i grammatikken.

Anvendelse af værktøjet

Ved indskrivning af sprogspecifikation i JavaCC bruges metasyntaksen EBNF (Extended Backus-Naur Form), denne sprogspecifikation er en anden måde at skrive en KFG op på, med en anderledes syntaks. Metasyntaksen EBNF omfatter strukturer og

sammensætningen af sætninger. De vigtigste inddelinger af JavaCC's sprogspecifikationen er:

- **Tokens** Med JavaCC er det muligt at definere alle terminaler, og give dem navne, inden beskrivelsen af selve grammatikken. Et eksempel på dette kan ses på figur 4.1. Derudover kan et eksempel på en token der bliver brugt i en non-terminal ses på figur 4.2. Det er også muligt, at skrive terminaler ind uden tokens, dette gøres ved at sætte terminalen ind mellem gåseøjne.

```

39 TOKEN : /* OPERATORS */
40 {
41   < BUILD : "build">
42 | < PLUS : "+" >
43 | < MINUS : "-" >
44 | < MULTIPLY : "*" >

```

Figur 4.1. Definering af token i JavaCC

- **Non-terminaler** Defineringen af non-terminaler kan ses på figur 4.2, hvor en non-terminal BLKID bliver defineret til at indeholde en token ved navn ALPH. Hvis man i selve definitionen af non-terminalen, vil kalde en anden non-terminal, skriver man det op, med navnet på non-terminalen og parameter parenteser, ligesom et funktionskald i Java eller C#. Eksempler på kald til non-terminaler kan ses under fejlmeddelelserne figur 4.4 og 4.5.

```

217 void BLKID() :
218 {
219 {
220   < ALPH >
221 }

```

Figur 4.2. Definering af en non-terminal i JavaCC

- **Skip** Det er muligt at ignorere bestemte terminaler, f.eks. ignoreres der i dette tilfælde mellemrum, hvilket kan ses på figur 4.3

```

32 SKIP :
33 {
34   " "
35 | "\r"
36 | "\t"
37 }

```

Figur 4.3. Definering af terminaler der skal ignoreres i JavaCC

Brugen af tokens kan gøre det lettere at rette i grammatikken, og gøre grammatikken mere overskuelig. Det tager tid at definere dem inden brug, og der er mulighed for at indskrive tokens direkte ind i definitionen af non-terminaler. Hvis grammatikken er lavet på forhånd, og overholder LL(k), er det hurtigt og let at se, om grammatikken er korrekt, og få lavet en scanner og en parser til sin grammatik.

I tilfælde af fejl i grammatikken giver JavaCC pluginet fejlmeddelelser og advarsler når den indskrives i Eclipse. F.eks. er der fejlmeddelelser hvis man bruger en non-terminal der ikke er defineret, dette kan ses på figur 4.4. Der er også fejlmeddelelser hvis grammatikken ikke passer ind i LL(k) grammatik, som f.eks. left recursion problemer. Der kommer også advarsler hvis sproget er tvetydigt, og den giver endda et forslag til, hvor meget lookahead der skal til for at løse problemet, dette kan ses på figur 4.5.

```

96 void PCLS() :
97 {}
98 {
99   DCL()
100  | CTRL()
101  | BLD()
102  | FUNC() PCLS() Error: Line 102, Column 3: Non-terminal FUNC has not been defined.
103  | MOV() PCLS()
104  | {}
105 }
```

Figur 4.4. Fejlmeddelse ved manglende non-terminal, i JavaCC pluginet til Eclipse

```

96 void PCLS() :
97 {}
98 {
99   DCL()
100  | CTRL()
101  | BLD()
102  | FUNC() PCLS()
103  | MOV() PCLS()
104  | {}
105 }
```

Warning: Choice conflict involving two expansions at
 line 99, column 3 and line 102, column 3 respectively.
 A common prefix is: <ALPHS>
 Consider using a lookahead of 2 for earlier expansion.

Figur 4.5. Advarsel ved tvetydigt sprog, i JavaCC pluginet til Eclipse

Output af JavaCC

JavaCC genererer syv forskellige filer ud fra sprogspecifikationen, som er:

*NAME = selvvalgt navn til parseren.

- **TokenMgrError.java** er en fejklasse der bruges, når der opdages fejl af scanneren.
- **ParseException.java** er en anden fejklasse der bruges, når der opdages fejl af parseren.
- **Token.java** bruges til at repræsentere tokens. Hver token har et integer felt med navnet "kind", der bestemmer hvilken type token man har med at gøre, som eksempelvis plus, minus eller string. Den har samtidig også et string felt med navnet "image", som repræsenterer den sekvens af karakterer fra input som udgør selve tokenet.
- **SimpleCharStream.java** er en adapterklasse som udleverer karakterer til scanneren.

- **NAMEConstants.java** definerer de forskellige tokens og angiver en værdi til dem, som er den værdi der i token klassen bruges som "kind", til at bestemme hvilken token der er i brug.
- **NAMETokenManager.java** er selve scanneren.
- **NAME.java** er selve parseren.

Med disse klasser kan der nu scannes en tekst igennem og findes tokens, samt parses tokens i forhold til grammatikken beskrevet i sprogspecifikationen. Klasserne der er blevet genereret er et slags skelet, og kan ikke meget mere end dette. Med disse klasser kan der bygges videre på compileren ved at ændre i skelettet.

4.1.2 SableCC

SableCC er en open-source compiler generator som kan generere scannere og parsere ud fra alle LALR(1) kontekstfrie grammatikker.[40]

Anvendelse af værktøjet

For at benytte SableCC skal der bruges en sprogspecifikation, som kan laves ud fra en LALR(1) KFG. SableCC's sprogspecifikation benytter, som JavaCC, også metasyntaksen EBNF (Extended Backus-Naur Form). Dette betyder at opskrivningen af grammatikken har samme strukturer og sætningsopbygning, men indskrivningen er anderledes. Den fulde SableCC sprogspecifikation kan ses i bilag B. Sprogspecifikationen deles op i et antal forskellige dele, lignende dem fra JavaCC, hvor de vigtigste er:

- **Helpers** bruges til at definere "hjælpere" ud fra regulære udtryk, som så kan bruges under definitionen af tokens. Et eksempel kan se på figur 4.6.

```
3Helpers
4    digit = ((['1' .. '9'] ['0' .. '9'])* | '0');
5    alphabet = ('a' .. 'z') | ('A' .. 'Z');
```

Figur 4.6. Definering af helpers i SableCC

- **Tokens** bruges til at definere terminaler, som kan bruges under definitionen af productions. Et eksempel kan ses på figur 4.7.

```
22    repeat_keyword = 'repeat';
23    endrepeat_keyword = 'end repeat';
24    plus = '+';
25    minus = '-';
26    divide = '/';
27    multiplication = '*';
```

Figur 4.7. Definering af tokens i SableCC

- **Ignored Tokens** bruges til at bestemme hvilke tokens der skal springes over, et typisk eksempel ville være mellemrum (' '). Ignored tokens svarer til JavaCC's skip.

- **Productions** bruges til at definere alle produktioner i grammatikken, ud fra non-terminaler og de terminaler/tokens der er blevet defineret. Et eksempel kan ses på figur 4.8.

```
statementsdcl = {declare} declaration
                  |{control} control
                  |{build} build
                  |{move} move
                  |{function} functioncall
                  |{assign} assignment;
```

Figur 4.8. Definering af productions i SableCC

Definering af terminaler skal foregå i tokens-delen, og kan ikke blive gjort sammen med produktionerne. Dette kan gøre, at det tager en smule længere tid at skrive sprogspecifikationen, i forhold til at lave en KFG, hvor der typisk ikke er en lignende restriktion.

Derudover er det ikke alle EBNF udtryk der kan bruges i SableCC, og dem der kan benyttes har en anden notation. Repetition og valg kan stadig bruges sammen med SableCC, med de tre symboler; '+', '*' og '?'.

- **block+**, betyder at block kan gå igen en eller flere gange.
- **block***, betyder at block kan gå igen nul eller flere gange.
- **block?**, betyder at block kan bruges nul eller én gang.

Navnene på de forskellige non-terminaler er også lidt anderledes, da der ikke kan bruges store bogstaver til at definere disse i SableCC.

Det er ikke muligt at definere flere tokens med den samme terminal, uden at skabe problemer, som kan ske når der laves operator overloading. Dette kunne blive et problem da '-' bruges både som en **operator** i **expressions**, samt som en tom blok i **build**-udtrykket. For at komme uden om problemet har terminalen '-' fået sin egen token, og som det ses i **expressions** i kodeeksempel 4.1, er der både et alternativ som tager en **operator** og et som tager **minus**, selvom det egentligt også er en operator. Ligeledes bruges **minus** token også under definitionen af en tom **block** variabel.

```
1  expressions ={multi} expression operator expressions
2      |{minus} expression minus expressions
3      |{single} expression;
4  expression = {val} value
5      |{paran} s_paran expressions e_paran;
```

Kodeeksempel 4.1. Udsnit af sableCC sprogspecifikation

Det samme problem er ligeledes opstået i **logicalexpression**, som kan ses i kodeeksempel 4.2, **assignment** og under **declaration**, hvor de alle tre bruger tegnet '=' der skrives

som `assignment_keyword`, for at løse dette er `logicalexpression` blevet delt op i to muligheder ligesom `expressions` i kodeeksempel 4.1.

```
1 logicalexpression = {others} [first]:expression logicaloperator [follow]:expression
2   |{equals}[first]:expression assignment_keyword [follow]:expression;
```

Kodeeksempel 4.2. Udsnit af sableCC sprogspecifikation

SableCC giver en god og præcis fejlrapportering under udvikling af sprogspecifikationen. Desværre er det kun når man kører sprogspecifikationen igennem SableCC værktøjet, at den ser efter fejl i syntaksen, og det er altid kun den første fejl, som bliver fundet der returneres. Dette kan gøre det lidt besværligt, hvis der er mange fejl i sprogspecifikationen. Der kan på figur 4.9 ses et eksempel på en fejl, der er returneret fordi at `production` på linje 54 er blevet stavet forkert. Fejlmeddelelsen viser, at der er en fejl på linje 50 ved tegn nummer 16, og at den hverken kan finde en token med navnet `function` eller en anden produktion med navnet `function`.

```
46
47     processes = {declare} declaration processes
48         |{control} control processes
49         |{build} build processes
50         |{function} function processes
51         |{move} move processes
52         |{epsilon} ;
53
54     function = word s_paran parameters e_paran;
|java.lang.RuntimeException: [50,16] PFunction and TFunction undefined.
```

Figur 4.9. Fejlmeddeelse i SableCC

Output af SableCC

Efter at SableCC har fået kørt sprogspecifikationen igennem, vil der bliver genereret en række nye filer. Disse filer er delt op i fire pakker:

- **Lexer pakken** som indeholder en scanner/lexer klasse og en exception klasse til scanneren. Scanner klassen er selve scanneren og exception klassen bruges til eventuelle exceptions under scanningen.
- **Parser pakken** indeholder en parser og en parser exception klasse, og ligesom i lexer pakken, er parser klassen her selve parseren, og exception klassen bruges i tilfælde af exceptions under parsing.
- **Node pakken** indeholder alle klasser der bruges til at definere det abstrakte syntakstræ, der er lavet ud fra sprogspecifikationen.
- **Analysis pakken** indeholder et interface, samt tre klasser, som bliver brugt til at bestemme, hvordan man bevæger sig ad det abstrakte syntakstræ. Der er lavet to metoder til at bevæge sig igennem AST'et, `DepthFirst` og `ReversedDepthFirst`, som følger med i pakken.

Ved at bruge objektorienterede metoder, er den genererede kode adskilt fra den menneskeskabte kode. Dette gør, at man ikke behøver at ændre i den genererede kode, og dermed ikke risikerer at ødelægge den. Man kan implementere oversættere, fortolkere og andre ting ved bare at lave en ny klasse, som forlænger klassen `DepthFirst` eller `ReversedDepthFirst`. Opbygningen giver også mulighed for, at bruge visitor-pattern til at udvikle strukturer, hvilket gør det mere enkelt at tilføje nye node typer, uden at ændre i dem som allerede er lavet.

4.1.3 Sammenligning

Under sammenligningen bliver der set på forskelle under indskrivningen af selve sprogspecifikation, og outputtet fra de to compiler generatorer.

Sprogspecifikation: Der er små forskelle på de to compiler generatorers sprogspecifikation. I JavaCC er der fejlmeddeling på tilhørende linje og hver gang der bliver gemt eller kompileret, samt mulighed for hints om hvordan fejlen kan løses. I SableCC får man kun én fejlmeddeelse af gangen, og dette sker kun på kompilingstidspunktet. En anden forskel er at der i JavaCC er mulighed for at skrive terminaler direkte ind under defineringen af non-terminalerne. I SableCC skal terminaler derimod defineres under token opdelingen. SableCC bruger samme notation som standard EBNF formatet (ISO/IEC 14977) der er blevet valgt af International Organisation for Standardization, hvilket kan gøre det nemmere at overføre en grammatik, hvis den er skrevet i EBNF. JavaCC udskiller sig derimod lidt mere fra standard EBNF.

Outputfiler: SableCC anvender en objektorienteret opbygning, hvilket gør at man kan adskille den computergenererede kode fra den brugerskabte, hvilket gør det lettere at fejlfinde, da der kun skal ledes i den brugerskabte kode del. I JavaCC skal man lave ændringer og udvide den computergenererede kode. I JavaCC bliver der som standard ikke lavet et AST (abstract syntax tree), hvor man enten selv skal implementere det, eller bruge et eksternt værktøj som JJtree eller JBT. Hvorimod der med SableCC som standard bliver genereret et AST, samt et sæt klasser til at gå igennem AST'et.

Der er blevet gennemgået de forskellige fordele ved de to compiler generatorer, og beslutningen er faldet på at vælge SableCC, da vi tror det vil egne sig bedst til vores projekt, fordi de steder hvor JavaCC har sine fordele er mest i sprogspecifikationen, og det har ikke nogen væsentligt betydning, andet end at det kan spare tid, når grammatikken skal indskrives. Hvorimod SableCC's outputfiler bedre stemmer overens med gruppens tankegang og måde at programmere på, og den tid der måske ville være sparet ved JavaCC's sprogspecifikation bliver indhentet igen når AST'et skal laves.

4.2 SableCC implementering

SableCC er blevet brugt til at generere en lexer (det samme som en scanner, men omtalt som lexer i dette afsnit), parser og et abstrakt syntakstræ. I dette afsnit vil disse dele blive uddybet, og udvalgte dele af koden vil blive gennemgået. Til sidst i afsnittet, vil der blive gjort rede for nogle af de muligheder og redskaber dette har givet, til at viderebygge på

compileren. Delene er alle blevet genereret ud fra KFG'en i bilag B. Dette afsnit bruger generelt et speciale[40] skrevet af Gagnon, omhandlende SableCC, som kilde.

4.2.1 Lexer

Ud fra KFG'en har SableCC genereret en DFA (Deterministisk Finit Automat) baseret lexer. De vigtigste dele af lexeren vil her blive uddybet.

Lexer klassen er hovedsageligt bygget op af fire members der kan findes i kodeksempel 4.3.

```

1 public Lexer(PushbackReader in) //constructor
2
3 public Token peek() throws LexerException, IOException
4
5 public Token next() throws LexerException, IOException
6
7 protected Token getToken() throws IOException, LexerException

```

Kodeksempel 4.3. Et udsnit af nogle members fra lexeren

Som der kan ses på kodeksemplet, skal constructoren bruge en `PushbackReader`, i form af en `java.io.PushbackReader`. Lexeren "pusher" lookahead karakterer ind i `PushbackReaderen`, når den genkender en token (readeren er en unicode karakter strøm). `next()` metoden returnerer den næste token i strømmen og skifter samtidig til den. `Peek()` metoden returnerer også den næste token i strømmen, men skifter ikke til den, hvilket vil sige at flere kald til `peek()` stadig vil returnere den samme token. Metoden `getToken()` kaldes af `next()` og `peek()`, det er denne metode der kontrollerer, hvilken token der bliver accepteret som den næste, og sender svaret tilbage til `next()` og `peek()`.

Klassen indeholder også en række members der specielt kan bruges til at modifcere lexeren, disse kan ses i kodeksempel 4.4.

```

1 protected Token token;
2
3 protected State state;
4
5 protected void filter();

```

Kodeksempel 4.4. Et udsnit af nogle members fra lexeren

Disse private members kan bruges til at modifcere lexerens opførsel. Hovedsageligt er det `filter()` metoden der skal bruges til at modifcere lexeren, da den bliver kaldt hver gang `next()` eller `peek()` genkender en token. Dette kan bruges ved at lave en ny klasse som arver fra lexer klassen, hvor `filter()` metoden så kan overskrives i. Metoden bruges typisk, når man vil have lexeren til at gøre noget, der ikke er muligt udelukkende ved hjælp af sprogspecifikationen.

Vi har anvendt denne mulighed, da det er nødvendigt at vide hvor mange `newlines` der bliver lavet i sammenhæng med `build` keywordet. I selve sprogspecifikationen til SableCC er `newlines` blevet sat til at skulle ignoreres, hvilket de også skal i alle situationer bortset fra når man befinner sig inde i en `build` konstruktion. En token, i dette tilfælde `newline`, kan ikke både blive ignoreret og samtidig bruges under `productions`. Dette er et problem,

da det er vigtigt, at det skal være muligt at lave så mange `newlines` man vil overalt i programmet, samtidig med at der kan holdes styr på dem, hvis de bruges i sammenhæng med `build` keywordet. For at løse dette problem er der blevet lavet en token, som bliver kaldt `nl_replacer`, denne bruges i stedet for `newline` under produktionen `block`, som kan ses på figur 4.10.

```
build = {blocks} build_keyword s_paran orientation e_paran block* endbuild_keyword
|{block} build_keyword letter;

block = {empty} minus
|{ctrl} ctrl_blocks
|{letter} letter
|{nlreplace} nl_replacer;
```

Figur 4.10. Udsnit af KFG fra SableCC - `nl_replacer`

Dette er dog ikke tilstrækkeligt for at få det til at fungere. `Newlines` vil stadig ikke blive opfanget, og parseren vil forvente tokens af typen `nl_replacer`. For at løse dette er der blevet lavet tre forskellige tilstande, som programmet kan være i; NORMAL, PREP og BUILD. Disse tilstande skal bruges til at holde styr på om lexeren er i standard tilstand (tilstand = NORMAL) eller om den er inde i en `build` konstruktion (tilstand = BUILD). PREP tilstanden bruges fordi keywordet `build` kan bruges til to forskellige konstruktioner; den lille der kun bygger én blok (her er `newlines` ligegyldige, og der skal derfor ikke skiftes til tilstanden BUILD), og den store hvor der kan bygges flere blokke (her skal `newlines` ikke ignoreres). PREP bruges derfor når der opfanges en `build` token, hvorefter den næste token bestemmer om der skal skiftes til BUILD tilstanden, hvis næste token er en `TLetter` token (`TLetter` er navnet på en klasse, som er blevet genereret til letter tokenen fra sprogspecifikationen), skal der ikke ske noget, men hvis den er af typen `TOrientation` (`TOrientation` er navnet på en klasse, som er blevet genereret til orientation tokenen fra sprogspecifikationen) betyder det at det er en stor `build` konstruktion, og der derfor skal skiftes til BUILD tilstanden). For at skifte imellem de forskellige tilstande, er der blevet defineret nogle transitions som kan ses i kodeeksempel 4.5. Den første transition bliver udført, hvis nuværende tilstand er NORMAL og lexeren genkender tokenen `build`. Dette vil gøre at tilstanden skifter fra NORMAL til PREP. Den anden transition bliver brugt sammen med tokenen `end build`, og skifter tilstanden fra BUILD til NORMAL.

```
1 {normal->prep, prep}build_keyword = 'build';
2 {build->normal, normal}endbuild_keyword = 'end build';
```

Kodeeksempel 4.5. Tokens - State transitions

For at færdiggøre denne løsning, udnyttes muligheden for at overskrive `filter()` metoden. Den har til formål at kontrollere om den nuværende token er af typen `newline`, og hvis lexeren også er i tilstanden BUILD, vil den ændre den nuværende token (`newline` token) til `nl_replacer` (hvilket parseren vil acceptere). Udover det bliver den også brugt til at skifte fra tilstanden PREP til BUILD, hvis den befinner sig i tilstanden PREP og opfanger tokenen `TOrientation`. Koden for `filter()` kan ses i kodeeksempel 4.6.

```
1 protected void filter()
```

```

2 {
3     if (state.equals(State.BUILD))
4     {
5         if (token.getClass() == TN1.class)
6         {
7             token = new TN1Replacer();
8         }
9     }
10    else if (state.equals(State.PREP))
11    {
12        if (token.getClass() == TOrientation.class)
13        {
14            state = State.BUILD;
15        }
16    }
17 }

```

Kodeeksempel 4.6. Filter override

4.2.2 AST (Abstrakt syntakstræ)

SableCC genererer ud fra sprogspecifikationen adskillige klasser, som parseren bruger til at bygge et AST under parsing af et program. Der findes som udgangspunkt tre typer af disse klasser.

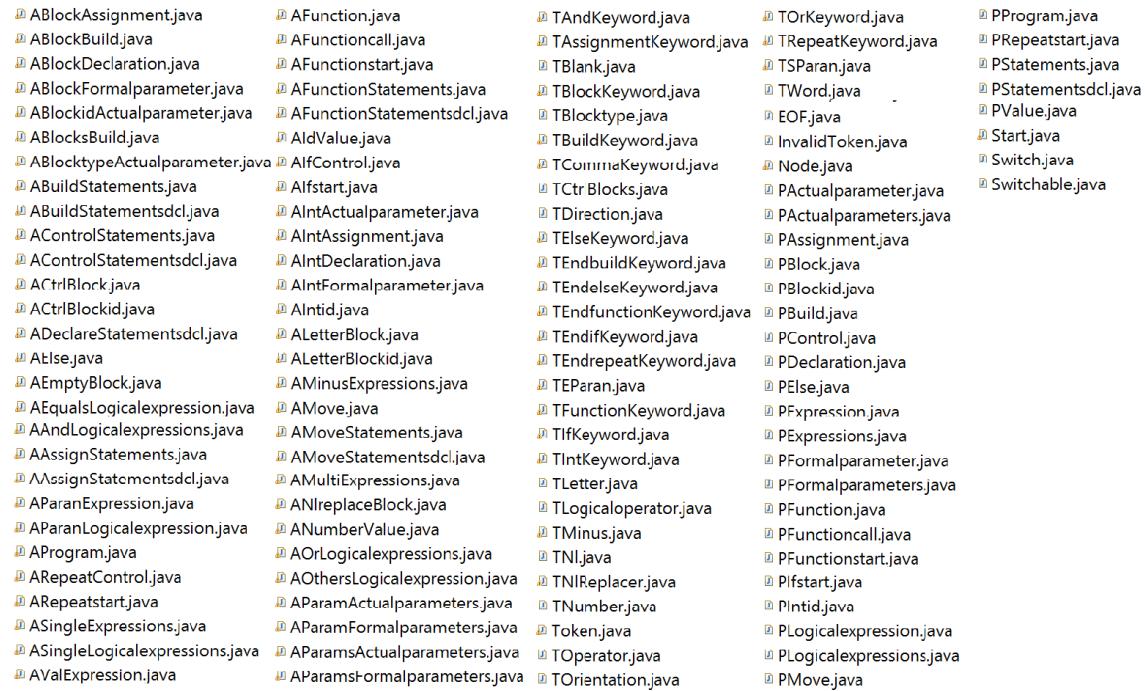
Productions klasser bliver oprettet ud fra hver production i sprogspecifikationen. De bliver hver især navngivet ud fra det navn der er på venstre side af en production, med et præfiks 'P', som definerer at dette er en production.

Alternatives klasser bliver oprettet fra hver mulighed/alternativ en production har. Hvis en production kun har et alternativ, vil en klasse blive oprettet med samme navn, som er på venstre side af lighedstegnet, med et præfiks 'A'. Hvis der er mere end ét alternativ vil der bliver oprettet en klasse for hvert alternativ, med et præfiks 'A', efterfulgt af et navn som er skrevet imellem tuborgklammer foran hvert alternativ. Dette er så efterfulgt af production navnet på venstre side af lighedstegnet.

Token klasser bliver oprettet ud fra hver token der er defineret i sprogspecifikationen. De får hver især et præfiks 'T' efterfulgt af navnet på tokenen.

For alle klasserne gælder det at det første bogstav efter deres præfiks P/T/A også skrives med stort, ligeledes bliver alle bogstaver efter en bundstreg også skrevet med stort. På figur 4.11 kan der ses en liste over alle klasserne som er genereret til at danne det abstrakte syntakstræ.

Production klasserne er alle abstrakte klasser som arver fra `Node` klassen. `Node` klassen er det øverste led i klasseshierarkiet, som alle andre P/T/A klasser arver fra, enten direkte eller igennem en anden klasse. I klassen `Node` ligger alt det funktionelle, som er fælles for alle klasserne, og som giver mulighed for at give instanser af klasserne en parent, hvilket bruges til at danne selve strukturen af det abstrakte syntakstræ. Alternatives klasserne indeholder information der er specielt for hvert alternativ en produktion har. Alternatives klasserne arver fra deres tilsvarende production klasser, og på den måde bliver de forskellige

**Figur 4.11.** AST klasser

alternativer grupperet under en produktionsklasse. Token klasserne svarer til hver af de forskellige tokens, defineret i sprogspecifikationen. De enkle token klasser indeholder den information som er forskellig for hver token. De arver fra klassen ved navn `Token`, som indeholder alle dele der er tilfælles for hver token. `Token` klassen arver selv fra `Node` klassen. Et eksempel kan ses herunder:

```

1 Tokens
2     int_keyword = 'int';
3     string_keyword = 'string';
4     assignment = '=';
5     var = ['a' .. 'z'];
6 Productions
7     declare = {int} int_keyword assignment var
8         | {string} string_keyword assignment var

```

Kodeeksempel 4.7. Eksempel på sprogspecifikation

Sprogspecifikationen fra kodeeksempel 4.7 vil danne klasserne som kan ses på kodeeksempel 4.8

```

1 abstract class Node {}
2 abstract class Token extends Node {}
3
4 abstract class PDeclare extends Node {}
5 class AIntDeclare extends PDeclare {}
6 class AStringDeclare extends PDeclare {}
7
8 class TIntKeyword extends Token{}
9 class TStringKeyword extends Token{}
10 class TAAssignment extends Token{}
11 class TVar extends Token{}

```

Kodeeksempel 4.8. Eksempel klasser

4.2.3 Parser

Ud fra sprogspecifikationen danner SableCC en LALR(1) parser i parser pakken. **Parser** klassen bygger automatisk det abstrakte syntakstræ, imens den parser. Dette sker ved at flytte tokens modtaget fra lexeren over på parse stakken. For hver reduktion, laver parseren en ny instans for det reducerede alternativ. Den popper så elementerne af parse stakken og tilføjer dem til den nye instans. Derefter pusher den så instansen tilbage på parse stakken.

To members fra **Parser** klassen kan ses i kodeeksempel 4.9.

```
1 public Parser(Lexer lexer); // constructor
2 public Start parse() throws ParserException, LexerException, IOException
```

Kodeeksempel 4.9. Parser - metoder

Constructoren skal bruge en lexer som parameter, som skal give den elementerne der skal parses. Metoden **parse()** parser input fra lexeren og returnerer en reference til begyndelsen af AST'et som altid er af typen **Start**. **Start** produktionen er altid defineret som "Start = first production EOF" som i dette tilfælde vil være "Start = Program EOF".

Ligesom med lexeren er det også muligt at modificere parseren, dette kan gøres ved hjælp af de to members som kan ses i kodeeksempel 4.10.

```
1 protected void filter();
2 protected Node node;
```

Kodeeksempel 4.10. Parser - modificering members

Modificering af parseren foregår på samme måde som lexeren ved hjælp af **filter()** metoden, her bliver metoden kaldt ved hver reducering, som gør det muligt at ændre den **Node** som bliver "pushed" på parse stakken. Dette kan bruges i tilfælde hvor det er nødvendigt at ændre på AST'et under parsing. Det bruges mest i tilfælde hvor hukommelsesforbruget vil blive for stort, når der skal laves et helt AST, hvilket specielt sker hvis der skal parses meget store filer med store sprog. Det kan så ved hjælp af **filter()** metoden være muligt at gøre træet mindre. Det kan dog skabe problemer at bruge **filter()** metoden under parsing, da det nemt kan skabe fejl, eftersom at compileren ikke kontrollerer, om det man ændrer en alternative node til, er en subklasse til den samme abstrakte production klasse, som den node der bliver ændret.

4.2.4 Visitor pattern

En vigtig del af at kunne arbejde med SableCC's output, er muligheden for at kunne bevæge sig ad det abstrakte syntakstræ, og besøge de forskellige nodes. Til at gøre dette bruger SableCC et visitor pattern, der giver mulighed for, at kunne udføre forskellige opgaver ved hver type af nodes. En vigtig del af visitor pattern er, at det giver mulighed for at tilføje operationer på objekter, uden at ændre på klasserne. Dette gør at man ikke nødvendigvis skal ind og lave om på den computergenererede kode, og derved risikerer at skabe problemer. Dette bruges eksempelvis ved; lexer(med filter() metoden), typechecker/scope og til kodegenerering.

For at forklare hvordan visitor pattern virker i SableCC vil der blive taget udgangspunkt i sproget beskrevet i kodeeksempel 4.7 og 4.8.

I SableCC bruges et **Switch** interface, til at indeholde metoder for de nodes der skal kunne besøges.

```

1 public interface Switch
2 {
3     // Empty body
4 }
5
6 public interface Analysis extends Switch
7 {
8     void caseAIntDeclare(AIntDeclare node);
9     void caseAStringDeclare(AStringDeclare node);
10
11    void caseTIntKeyword(TIntKeyword node);
12    void caseTStringKeyword(TStringKeyword node);
13    void caseTAssignment(TAssignment node);
14    void caseTVar(TVar node);
15 }
```

Kodeeksempel 4.11. Switch interface

Interfacet **Switch** er det øverste led af alle switch interfaces der oprettes, dette skal alle andre switch interfaces arve fra. Dette gør det muligt at have flere forskellige switch interfaces, hvis det skulle være nødvendigt. Interfacet bliver dog kun benyttet ved **Analysis** interfacet. Som kan ses i kodeeksempel 4.11 indeholder **Analysis** interfacet en tom metode for hver token klasse og alternative klasse. Dette gør så enhver klasse der implementerer **Analysis** interfacet, er klar over at der er en funktion med hver af disse navne.

```

1 public interface Switchable
2 {
3     void apply(Switch sw);
4 }
5
6 public abstract class Node implements Switchable, Cloneable {}
```

Kodeeksempel 4.12. Switchable interface

Alle klasser som skal gøre brug af **Switch** interfacet skal implementere **Switchable** interfacet, som vist i kodeeksempel 4.12, som indeholder den tomme metode **Void apply(Switch sw)**. Denne metode danner en forbindelse til **Switch** interfacet, som der så kan gøres brug af. Da alle token og alternative klasser arver fra **Node** klassen, implementerer de automatisk interfacet **Switchable**, eftersom **Node** klassen implementerer det.

```

1 public final class AIntDeclare extends PDeclare
2 {
3     public void apply(Switch sw)
4     {
5         ((Analysis) sw).caseAIntDeclare(this);
6     }
7 }
```

Kodeeksempel 4.13. apply method override

Hver token og alternative klasse indeholder så en override til metoden `apply(Switch sw)`, hvor metoden der svarer til den enkelte klasse bliver kaldt, som vist i kodeeksempel 4.13. `Switch sw` bliver typecasted til `Analysis` som er det interface der indeholder den tomme metode.

```

1 public class AnalysisAdapter implements Analysis
2 {
3     public void caseAIntDeclare(AIntDeclare node)
4     {
5         defaultCase(node);
6     }
7     public void caseAStringDeclare(AStringDeclare node)
8     {
9         defaultCase(node);
10    }
11    public void caseTIntKeyword(TIntKeyword node)
12    {
13        defaultCase(node);
14    }
15    public void caseTStringKeyword(TStringKeyword node)
16    {
17        defaultCase(node);
18    }
19    public void caseTAssignment(TAssignment node){
20        defaultCase(node);
21    }
22    public void caseTVar(TVar node)
23    {
24        defaultCase(node);
25    }
26    public void defaultCase(Node node)
27    {
28        // do nothing
29    }
30 }
```

Kodeeksempel 4.14. AnalysisAdapter klassen

`AnalysisAdapter` klassen, som vist i kodeeksempel 4.14, implementerer `Analysis` interfacet, og har en definition af alle metoderne fra `Analysis` interfacet. Denne klasse kan bruges til at implementere et `Switch` interface ved at arve fra `AnalysisAdapter`. Det smarte ved at bruge denne klasse er, at den allerede har defineret en standard metode, for alle de forskellige token og alternative nodes, der bliver kaldt i tilfælde af at de ikke er blevet overskrevet et andet sted. Ved at overskrive dem kan man udføre sine egne brugerdefinerede operationer.

```

1 void ThisNode(Node node)
2 {
3     node.apply(new AnalysisAdapter
4     {
5         void caseTIntKeyword(TIntKeyword node)
6         {
7             System.out.println("This is an int node");
8         }
9         void defaultCase(Node node)
10        {
11            System.out.println("This is not an int node");
12        }
13    }
14 })
15 }
```

På kodeeksempel 4.2.4 kan der ses et eksempel på en metode `ThisNode`, som får en `Node` som input og bruger metoden `apply()` som alle nodes indeholder. Her bliver der oprettet en ny `AnalysisAdapter` hvori, `caseTIntKeyword()` og `defaultCase()` metoderne bliver overskrevet. Hvis den node som `ThisNode` modtager er af typen `TIntKeyword` vil der blive skrevet "This is an int node" i output, og hvis node typen er en hvilken som helst anden, vil der blive udskrevet "This is not an int node".

4.2.5 AST Walkers

For at kunne udnytte dette visitor pattern ved at gå igennem det abstrakte syntakstræ, skal der kigges på de `AST Walkers` som også bliver genereret af SableCC. En `AST Walker` bruges til at besøge alle nodes i et abstrakt syntakstræ, i en prædefineret rækkefølge.

Der medfølger to forskellige `AST Walkers` i SableCC:

Depth-first traversal løber igennem nodes i samme rækkefølge, som de bliver indlæst af lexeren.

Reversed depth-first traversal går igennem nodes i modsatte rækkefølge, som de bliver indlæst af lexeren.

I kodeeksempel 4.15 ses et eksempel på en `DepthFirstAdapter` som laver depth-first traversal. Dette eksempel er lavet ud fra sprogspecifikationen i kodeeksempel 4.7.

```

1  public class DepthFirstAdapter extends AnalysisAdapter
2  {
3      void caseStart(Start node)
4      {
5          inStart(node);
6          node.getADeclare().apply(this);
7          node.getEOF().apply(this);
8          outStart(node);
9      }
10     void inStart(Start node) {}
11     void outStart(Start node) {}
12
13     void defaultIn(Node node) {}
14     void defaultOut(Node node) {}
15
16     void defaultOut(Node node){}
17
18     void inAIntDeclare(AIntDeclare node){defaultIn(node);}
19     void outAIntDeclare(AIntDeclare node){defaultout(node);}
20     void caseAIntDeclare(AIntDeclare node)
21     {
22         inAIntDeclare(node);
23         if(node.getIntKeyword() != null)
24             node.getIntKeyword.apply(this);
25         if(node.getAssignment() != null)
26             node.getAssignment.apply(this);
27         if(node.getVar() != null)
28             node.getVar.apply(this);
29         outAIntDeclare(node);
30     }
31     void inAStringDeclare(AStringDeclare node){defaultIn(node);}
32     void outAStringDeclare(AStringDeclare node){defaultout(node);}
33     void caseAStringDeclare(AStringDeclare node)
```

```

34     {
35         inAStringDeclare(node);
36         if(node.getStringKeyword() != null)
37             node.getStringKeyword.apply(this);
38         if(node.getAssignment() != null)
39             node.getAssignment.apply(this);
40         if(node.getVar() != null)
41             node.getVar.apply(this);
42         outAStringDeclare(node);
43     }
44 }
```

Kodeeksempel 4.15. DepthFirstAdapter eksempel

Denne klasse indeholder logikken for at bevæge sig gennem det abstrakte syntakstræ. Hver caseA*** metode har en tilsvarende outA*** og inA*** metode (hvor *** er en pladsholder for klassens navn), som gør det muligt at afvikle kode både i starten og slutningen af en caseA*** metode. For at gøre det muligt at genbruge denne **DepthFirstAdapter** klasse i flere tilfælde, vil man undgå at skrive kode som brugeren vil have afviklet inde i klassen. Istedet laves der en ny klasse som arver fra **DepthFirstAdapter**, hvor metoderne så kan overskrives til brugerens behov. I kodeeksempel 4.16 vil alle **AIntDeclare** og **AStringDeclare** blive skrevet ud i output.

```

1  public class Printer extends DepthFirstAdapter
2  {
3      void caseAIntDeclare(AIntDeclare node)
4      {
5          System.out.println(node);
6      }
7      void caseAStringDeclare(AStringDeclare node)
8      {
9          System.out.println(node);
10     }
11 }
```

Kodeeksempel 4.16. Printer eksempel

Klasser som denne **Printer** klasse bruges eksempelvis i projektet til at lave type checker og kodegenerering, hvilket vil blive gennemgået i de følgende afsnit.

4.3 Scope

Scope regler er en væsentlig del af et programmeringssprog, det er disse regler, der f.eks. sikrer, at det ikke er muligt at have to deklarationer af samme navn inden for samme scope i symboltabellen. I VMP bliver der brugt static scoping, som beskrevet i afsnit 3.2.1, med en struktur der ligner meget Flat Block Structure, ved at der kun er to scope niveauer, global og lokal. Den eneste forskel er, at i VMP er det kun funktionsdeklarationer, som kan blive deklareret i det globale scope.

Scope reglerne er skrevet i klassen **Identification**, som er bygget op af én constructor og fire metoder. Metoden **OpenScope** bliver kaldt i klassen **TypeChecker**, hver gang der er en funktionsdeklarering i projektets sprog. Det eneste metoden gør, er at forøge variablen **level**, som er af typen integer, med én. Dette fortæller compileren at den har bevæget sig ind i et nyt scope. Se koden 4.17.

```

1 public void OpenScope()
2 {
3     level++;
4 }
```

Kodeeksempel 4.17. OpenScope metoden i klassen Identification

Når compileren bevæger sig ud af den deklarerede funktion igen, bevæger den sig også ud af det lokale scope og tilbage ind i det globale scope. Dette bliver gjort ved at kalde metoden `CloseScope`. `CloseScope` formindsker variablen `level` med én, men inden dette bliver gjort skal den seneste deklarerede identifier, som er deklareret i det globale scope, blive fundet og tildelt til variablen `latest` som er et objekt af klassen `InsertedId`. `InsertedId` er en brugerdefineret klasse, hvor hver instans af denne klasse bliver tildelt nødvendig information af den seneste deklarerede identifier. Ved at gøre dette, er det ikke muligt at få adgang til de variabler, som er deklareret inde i en funktions lokale scope, udenfor den funktions lokale scope. Se kodeeksempel 4.18.

```

1 public void CloseScope()
2 {
3     InsertedId input, local;
4
5     input = latest;
6     while(input.level == level)
7     {
8         local = input;
9         input = local.previous;
10    }
11    level--;
12    latest = input;
13 }
```

Kodeeksempel 4.18. CloseScope metoden i klassen Identification

Hver gang en identifier bliver deklareret skal compileren kontrollere, om der allerede findes en identifier af samme navn og scope niveau i symboltabellen. Metoden som bliver brugt til dette er `EnterSymbol`, som er af typen boolean. Ved at metoden er af typen boolean, er det muligt at returnere *true*, hvis der allerede findes en identifier med samme navn i det scope, som den er deklareret i, og hvis ikke, returnerer den *false*. Måden metoden kontrollerer dette på, er ved at den har en string som en af dens parametre, som modtager navnet på den deklarerede identifier. Inde i metoden bliver der deklareret en `InsertedId` variabel `input` som bliver tildelt den seneste deklarerede identifier, og en boolean variabel `search` som bliver tildelt værdien *true*. En while-løkke bliver oprettet med variablen `search` som dens betingelse. Inde i while-løkken er der en `if else` kæde som har en betingelse som siger, hvis der endnu ikke er deklareret nogen identifier eller om `input` ikke ligger i det nuværende scope niveau, så bliver `search` tildelt boolean værdien *false* og løkken afsluttes. Hvis den forrige betingelse ikke er sand, kontrolleres den næste betingelse i `if else` kæden, som ser om navnet på identificeren i `input` er af samme navn, som det parameteren har modtaget. Har de identiske navne, så afsluttes metoden med at der returneres *true*. Hvis ingen af de to forrige betingelser er sande, starter løkken forfra med den forrige identifier. Hvis `search` på noget tidspunkt bliver *false*, afsluttes løkken, og en ny instans af `InsertedId` bliver oprettet, ud fra information fra den nylige deklarerede identifier, som så bliver tilføjet til identifikationstabellen. `EnterSymbol` metoden returnerer *false*, da

der ikke findes nogen identifier med samme navn inden for samme scope. Se kodeeksempel 4.19.

```

1  public boolean EnterSymbol(String id, Node AST)
2  {
3      InsertedId input = latest;
4      boolean search = true;
5
6      while (search)
7      {
8          if(input == null || input.level != level)
9          {
10              search = false;
11          }
12
13          else if (input.id.equals(id))
14          {
15              return true;
16          }
17
18          else
19          {
20              input = input.previous;
21          }
22      }
23
24      input = new InsertedId(id, level, AST, latest);
25      latest = input;
26      return false;
27 }
```

Kodeeksempel 4.19. EnterSymbol metoden i klassen Identification

Kodeeksemplet 4.20 viser hvordan metoderne `OpenScope` og `EnterSymbol` bliver brugt i type checker'en. I metoden `caseTWord` bliver `EnterSymbol` brugt som en betingelse i en if-sætning, og hvis den returnerer *true* modtager bruger'en en fejmeddeelse, om at der er blevet deklareret to funktioner af samme navn. I samme metode, direkte efter denne if-sætning, bliver metoden `OpenScope` kaldt og et nyt lokalt scope er blevet åbnet.

```

1  public void caseTFunctionKeyword(TFunctionKeyword node)
2  {
3     isFunction = true;
4  }
5
6  public void caseTWord(TWord node)
7  {
8      if (isFunction){
9          if (Scope.EnterSymbol(node.toString().trim(), node)){
10              reporter.ReportError("dobbelt function declaration \\"%\"", node.toString()
11                  , node);
12          }
13          Scope.OpenScope();
14         isFunction = false;
15      }
16  }
```

Kodeeksempel 4.20. Kodeeksempel på hvor scope reglerne bliver implementeret i klassen TypeChecker

4.4 Type checker

Type checker'en er lavet til, at sørge for at alle regler for VMP bliver overholdt. Disse regler udgør både scope- og typereglerne. Den sikrer at der ikke er mulighed for: dobbelt identifier deklarering; at man ikke kan tildele værdier til variabler, som ikke er deklarerede; at funktionskald har den rigtige type parameter og det korrekte antal parametre; og at sprogets build-udtryk både har en start- og en slutposition. Måden den sikrer at disse regler bliver opfyldt, er ved at returnere en passende fejlmeddeelse, om hvad fejlen er og hvor den kan blive fundet, hvor programmøren så selv skal gå ind og rette det, indtil der ikke er flere fejl at finde.

Kodeeksempel 4.21 viser hvordan type checker'en tjekker for fejl og hvordan den håndterer fejlene. Dette er et eksempel af metoden der sørger for, at sprogets build-udtryk kun indeholder gyldige `block` variabler, som er deklarerede, og at der er den rigtige mængde start- og slutpositioner. Det første metoden kontrollerer, er om der er mere end én start control block (!), slut control block (?) eller start- og slut control block (&) variabel i brug. Hvis type checker'en finder mere end den burde, bliver der returneret en fejlmeddeelse til programmøren. Dette bliver kontrolleret ved at bruge en `for`-løkke, som går igennem hver `block` variabel i build-udtrykket og ser om tegnet som den repræsenterer, er det tegn der står for start, slut eller start og slut. Hvis der allerede har været en variabel med det tegn, så er der en fejl. Herefter bliver der kontrolleret om hver `block` variabel, der bliver brugt i build, er deklareret. Måden det bliver gjort på er at bruge en `for`-løkke, som det ses i koden 4.21, for at kunne tjekke om hver enkelt variabel indeholder værdien `null`. Hvis den gør, bliver der returneret en fejlmeddeelse om, at der er en `block` som ikke er deklareret. Det sidste der bliver kontrolleret for, i denne metode, er om der overhovedet er et start control block tegn og et slut control block tegn, eller et start og slut control block tegn, at finde i build-udtrykket, hvis ikke så bliver der returneret endnu en fejlmeddeelse om den pågældende fejl.

```

1 public void caseABlocksBuild(ABlocksBuild node)
2 {
3     ArrayList<String> blocks = new ArrayList<String>();
4     boolean isStart = false, isEnd = false, isBoth = false;
5     String block;
6
7     //Læser variabler ind(en af hver variabel), og ser om der er for mange End og
8     //Start tegn.
9     for (int i = 0; i < node.getBlock().size(); i++)
10    {
11        block = node.getBlock().get(i).toString().trim();
12        //if else kaeden leder efter End og Start tegn, og laver fejl ved flere end
13        //et set.
14        if (block.equals("!"))
15        {
16            if (isStart || isBoth)
17                reporter.ReportError("There are 2 or more declared Start positions,
18                only one is allowed.", node.getBlock().get(i).toString().trim(),
19                node.getBuildKeyword());
20            else
21                isStart = true;
22        }
23        else if (block.equals("?"))
24    }
25    //... Koden for ? og & er blevet fjernet i eksemplet her da det er ligner koden for
26    // ...
27 }
```

```

22     //Laeser en af hver variabel til blocks.
23     if (!blocks.contains(block))
24         && !(block.equals("-"))
25         && !(block.equals("$")))
26     {
27         blocks.add(block);
28     }
29 }
30
31 //Ser om symboler er declared.
32 for (int i = 0; i < blocks.size(); i++){
33     if (Scope.RetrieveSymbol(blocks.get(i).trim()) == null)
34     {
35         reporter.ReportError("\%" block is not declared!",
36             blocks.get(i).toString(), node.getBuildKeyword());
37     }
38 }
39 //Ser om der er Start og End tegn. da der SKAL vaere 1 set.
40 if (!(isStart && isEnd) && !isBoth)
41 {
42     reporter.ReportError("There must be a Start and End position.", "&",
43         node.getBuildKeyword());
44 }
45 }
```

Kodeeksempel 4.21. Udsnit af koden fra metoden `caseABlocksBuild` i klassen `TypeChecker`

4.4.1 FindFunctions

I programmeringssproget VMP er det muligt at kalde andre funktioner inde i en funktion. I sprog som C skal man lave prototyper af funktioner i starten af programmet, for at kunne lave funktionskald fra en funktion, som er deklareret inden den kaldte funktion. En årsag til dette kan være, at når type checkeren laver en gennemgang af koden, og finder et funktionskald inden den har fundet en deklarering af funktionen, vil den returnere en fejl. I sprog som VMP, der er udviklet til at være simple og nemme at programmere i, vil det at have prototyper være unødvendigt besværligt. For at undgå dette, i VMP, er der blevet skabt en separat klasse, `FindFunctions`, for `TypeChecker` klassen, som programmet laver en gennemgang af, inden den laver en gennemgang af `TypeChecker` klassen. `FindFunctions` klassen går igennem koden og finder alle deklareringer af funktioner, og ligger deres navne og parametre ind i symboltabellen. Ved at gøre dette, er alle funktionsdeklareringer blevet kontrolleret, allerede inden type checkeren kontrollerer funktionskald. Koden til `FindFunctions` klassen kan ses i kodeeksemplet 4.22.

```

1 public class FindFunctions extends DepthFirstAdapter
2 {
3     ArrayList<Function> functionList;
4
5     public FindFunctions(ArrayList<Function> FunctionList)
6     {
7         functionList = FunctionList;
8     }
9
10    @Override
11    public void caseAFunction(AFunction node)
12    {
13        functionList.add(new Function(
14            ((AFunctionstart)node.getFunctionstart()).getWord().toString(),
15            ((AFunctionstart)node.getFunctionstart()).getFormalparameters()));
16    }
17 }
```

16 }
17 }

Kodeeksempel 4.22. Klassen FindFunctions

4.5 Code Generator

I dette afsnit vil compilerens code generator blive beskrevet. Code generatoren oversætter et program, skrevet i VMP, til et program i JavaScript med ScriptCraft extension. Code generatoren giver et komplet program, skrevet i ScriptCraft, som en outputfil. Den indeholder derfor forskellige metoder, til at oversætte specifikke udtryk, f.eks. er der en metode til at oversætte **if**-sætninger, en til at oversætte **repeat**-løkker osv. For at gøre det mere overskueligt, er metoderne i code generatoren, i denne beskrivelse, blevet inddelt i fem kategorier; diverse metoder, deklarationer og tildelinger, kontrolstrukturer, funktioner og **build**-udtrykket.

Diverse metoder

For at kunne forstå og læse metoderne i code generatoren, er der nogle overordnede metoder, som går igen, og som der først skal beskrives.

blockToNumber metoden består stort set udelukkende af én lang **if else** kæde, som har til formål at konvertere det input brugeren skriver, som f.eks. **STONE** eller **SAND** til tal, som forstås af ScriptCraft. **STONE** og **SAND** bliver så henholdsvis konverteret til 1 og 12.

indent er en enkel metode, der sørger for at printe indrykning i den ScriptCraft outputfil programmet giver. Det er ikke strengt nødvendigt, men gør det lettere at læse den, hvis man skulle have lyst til det.

caseAMove bruges til at konvertere de bevægelsesretninger, som skrives i VMP. Metoden består af en if else-kæde, med 6 forskellige udfald; **left**, **right**, **up**, **down**, **forward** og **back**. Eksempelvis bliver **forward(4)** til **this.fwd(4)**, og **up(3)** bliver til **this.up(3)**, som er den korrekte ScriptCraft-syntaks.

Deklarationer og tildelinger

Der er fire metoder der hører til deklarationer og tildelinger. Den første metode, **caseAIntDeclaration** skal oversætte en deklarering, og initialisering, af en integer fra VMP, til ScriptCraft. I VMP ser en integer deklaration således ud:

```
int eksempel = 5
```

I ScriptCraft ser den samme deklaration ud på følgende måde:

```
var eksempel = 5;
```

Som det ses i kodeeksempel 4.23, tager funktionen en node som input, og skriver til outputfilen, **Intid = Expression** med de passende tegn foran (**var**), mellem (=) og bagefter (;). Metoden **caseAIntAssignment** fungerer på samme måde, blot uden ordet "var" foran, som ikke er nødvendigt efter en integer er blevet deklareret.

1 public void caseAIntDeclaration(AIntDeclaration node)
2 {

```

3     indent();
4     writer.println("var " + node.getIntid().toString() + " = " + node.getExpressions
5         .toString() + ";");
}

```

Kodeeksempel 4.23. Metode til deklarering af integers

Den næste metode, `caseABlockDeclaration` bruges til at deklarere en `block`. Ligesom med integers er forskellen mellem en `block` deklaration, i VMP og ScriptCraft, brugen af "var" og et afslutningstegn ";". Desuden bliver der tilføjet `blockToNumber` foran, således at det bliver konverteret fra vores indskrivning af blocks f.eks. STONE, til et tal. Derudover er det relevant at se på den første kontrolstruktur, der starter på linje 7 i kodeeksempel 4.24, som bliver brugt til at deklarere control blocks !, ? og &, som indikerer start- og slutposition. Disse control blocks bliver navngivet henholdsvis `s1`, `e1` og `b1`, da ScriptCraft ikke kan benytte specielle tegn til variabler, og tallet er for at sikre at de ikke overlapper mulige `block` variabler programmøren kan lave.

```

1 public void caseABlockDeclaration(ABlockDeclaration node)
2 {
3     String blockId = node.getBlockid().toString().trim();
4
5     indent();
6
7     if (node.getBlockid().getClass().equals(ACtrlBlockid.class))
8     {
9         if (blockId.equals("!"))
10        {
11            writer.println("var s1 = blockToNumber(\"" + node.getBlocktype().toString
12                .trim() + "\");");
13        }
14        else if (blockId.equals("?"))
15        {
16            writer.println("var e1 = blockToNumber(\"" + node.getBlocktype().toString()
17                .trim() + "\");");
18        }
19        else if (blockId.equals("&"))
20        {
21            writer.println("var b1 = blockToNumber(\"" + node.getBlocktype().toString
22                .trim() + "\");");
23        }
24    }
25    else
26    {
27        writer.println("var " + blockId + " = blockToNumber(\"" + node.getBlocktype()
28            .toString().trim() + "\");");
29    }
}

```

Kodeeksempel 4.24. Metode til deklarering af blocks

Kontrolstrukturer

Der er syv metoder vedrørende kontrolstrukturer; to til `if`, to til `else`, en til `else-if` og to til `repeat`. Hver kontrolstruktur har to metoder, én til at starte og en til at slutte, med undtagelse af `else-if` som bliver afsluttet med den samme, som afslutter en `if` kontrolstruktur. Metoderne er forholdsvis enkle, og ligner umiddelbart hinanden meget. Der vil derfor kun blive beskrevet to metoder i dette underafsnit; `caseAIfstart` og `caseEndifKeyword`.

Den væsentligste forskel mellem en **if**-kontrolstruktur, i ScriptCraft og VMP, er de logiske operatorer. **caseAIfstart** starter derfor med at skrive keywordet **if**, efterfulgt af en startparentes. Herefter bliver metoden **ConvertExpressions** brugt, som skal sikre at de logiske operatorer bliver konverteret fra VMP, til ScriptCraft.

```

1 public void caseAIfstart (AIfstart node)
2 {
3     Indent();
4
5     writer.print("if (");
6
7     ConvertExpressions(node.getLogicalexpressions());
8
9     writer.println(") {");
10
11    IndentLevel++;
12 }
```

Kodeeksempel 4.25. Start på en **if**-kontrolstruktur

ConvertExpressions bliver brugt til at konvertere logiske udtryk fra VMP, til ScriptCraft. Det essentielle er, at operatorerne **AND** og **OR** skal konverteres til tegnene **&&** og **||**. **ConvertExpression** bliver brugt til enkle udtryk, som **=** (lig med) og **/=** (ikke lig med), samt større end **>** og mindre end **<**, hvor **ConvertExpressions** sørger for at konvertere **AND** og **OR**, som består af to boolske udtryk. I **ConvertExpressions** er det eneste der bliver tilføjet til outputfilen; parenteser, **&&** og **||**. Derudover skal **ConvertExpressions** kalde **ConvertExpression**, hvilket er den metode der står for at printe de enkelte logiske udtryk. I **ConvertExpressions** og **ConvertExpression** bliver der brugt navne på noder i AST'et fra KFG'en, hvor nodeklasserne **PLogicalExpressions**, **PLogicalExpression** og **ASingleLogicalexpressions** bliver refereret til, som henholdsvis **expressions**, **expression** og **SingleExpression**, i dette afsnit.

ConvertExpressions består overordnet af en **if else**-kæde, med tre forskellige udfald. Den første, som starter på linje 3 i kodeeksempel 4.26, tager en **SingleExpression**, der enten henviser til én **expression** eller **expressions** omsluttet af parenteser. En **expression** skal skrives ud og derfor skal **ConvertExpression** kaldes. Hvis den i stedet for henviser til flere **expressions** mellem to parenteser, skal de to parenteser udskrives, og **ConvertExpressions** skal kaldes rekursivt. **SingleExpression**'s bestemmelse og behandling kan ses på linje 3 til 15 i kodeeksempel 4.26.

Den næste mulighed i **if else**-kæden er **AND**, starter på linje 16 i kodeeksempel 4.26. Denne del består af tre dele; en **SingleExpression**, **AND** tegnet og en **expressions**. Første del **SingleExpression** **expressions** er ligesom den første der er beskrevet, hvor udfaldet enten er flere udtryk med parenteser, og så kører den rekursivt, eller blot ét udtryk, hvor at **ConvertExpression** bliver kaldt. Del to ses på linje 26 i kodeeksempel 4.26, og her udskrives keywordet for **AND**, som det ser ud i JavaScript: **"&&"**. Del tre af **AND**, indeholder **expressions**, og der bliver altid kaldt **ConvertExpressions** rekursivt med sidste del af noden.

Den tredje og sidste del i **ConvertExpressions** er **OR**, som ligner **AND** til forveksling, med den undtagelse af den naturligvis skriver **||** som det ser ud i JavaScript, og at typecasts ikke er af typen **AAndLogicalexpressions**, men typen **AOrLogicalexpressions**.

På linje 50 i kodeeksempel 4.26 starter metoden `ConvertExpression`, som igen består af en `if else`-kæde med tre udfald. Den første, linje 52, er i tilfælde af en lig med operator, som skal konverteres til JavaScript, hvor det skrives `==`. Den næste på linje 57, er for den logiske operator `/=`, som i JavaScript skrives `!=`. Der bliver derfor først brugt `to.String()` metoden, og så sammenlignet med teksten `/=`, hvorefter den skriver `!=` i mellem de to logiske udtryk. Den sidste mulighed, linje 64, er større end `>` og mindre end `<`, hvor syntaksen er ens med JavaScript, og det derfor bliver udskrevet uden ændringer.

```

1  private void ConvertExpressions(PLogicalexpressions node)
2  {
3      if (node.getClass().equals(ASingleLogicalexpressions.class))
4      {
5          if (((ASingleLogicalexpressions)node).getLogicalexpression().getClass().
6              equals(AParanLogicalexpression.class))
7          {
8              writer.print("(");
9              ConvertExpressions(((AParanLogicalexpression)((ASingleLogicalexpressions)
10                 .node).getLogicalexpression()).getLogicalexpressions());
11             writer.print(")");
12         }
13         else
14         {
15             ConvertExpression(((ASingleLogicalexpressions)node).getLogicalexpression());
16         }
17     }
18     else if (node.getClass().equals(AAndLogicalexpressions.class))
19     {
20         if (((AAndLogicalexpressions)node).getLogicalexpression().equals(
21             AParanLogicalexpression.class))
22         {
23             ConvertExpressions(((AParanLogicalexpression)((AAndLogicalexpressions)node
24                 .getLogicalexpression()).getLogicalexpressions());
25         }
26         else
27         {
28             ConvertExpression(((AAndLogicalexpressions)node).getLogicalexpression());
29         }
30         writer.print(" && ");
31         ConvertExpressions(((AAndLogicalexpressions)node).getLogicalexpressions());
32     }
33     else if (node.getClass().equals(AOrLogicalexpressions.class))
34     {
35         if (((AOrLogicalexpressions)node).getLogicalexpression().equals(
36             AParanLogicalexpression.class))
37         {
38             ConvertExpressions(((AParanLogicalexpression)((AOrLogicalexpressions)node
39                 .getLogicalexpression()).getLogicalexpressions());
40         }
41         else
42         {
43             ConvertExpression(((AOrLogicalexpressions)node).getLogicalexpression());
44         }
45     }
46     private void ConvertExpression(PLogicalexpression node)
47     {
48         if (node.getClass().equals(AEqualsLogicalexpression.class))
49         {
50             writer.print(((AEqualsLogicalexpression)node).getFirst() + " == " + ((
51

```

```

49         AEqualsLogicalexpression)node).getFollow());
50     }  

51     else if (((AOthersLogicalexpression)node).getLogicaloperator().toString().trim()  

52             .equals("/="))  

53     {  

54         writer.print(((AOthersLogicalexpression)node).getFirst() + " != " + ((  

55             AOthersLogicalexpression)node).getFollow());  

56     }  

57     else  

58     {  

59         writer.print(node.toString());  

60     }
61 }
```

Kodeeksempel 4.26. ConvertExpressions og ConvertExpression

En **if**-kontrolstruktur, i ScriptCraft, bliver afsluttet med **'}'**, hvilket er det eneste afslutningen gør. Der gælder det samme for afslutningerne af de andre kontrolstrukturer.

```

1 public void caseTEndifKeyword (TEndifKeyword node)
2 {
3     indentLevel--;
4     indent();
5     writer.println("}");
6 }
```

Kodeeksempel 4.27. Afslutning på **if**

Funktioner

Der er tilknyttet syv metoder til brugen af funktioner, hvor ikke alle metoder er relevante at vise i deres fulde længde. Her er derfor et overblik over alle syv, samt en kort beskrivelse af deres væsentligste formål.

- **public void caseAFunctionstart (AFunctionstart node)**

Metoden tilføjer en funktion til dronen, samt kalder den næste metode **ConvertFormalParams**, i tilfælde af at funktionen har parametre.

- **public void ConvertFormalParams (PFormalparameters node)**

Metoden kalder **ConvertFormalParam** det korrekte antal gange, i forhold til antallet af formelle parametre, ved at kalde sig selv rekursivt.

- **public void ConvertFormalParam (PFormalparameter node)**

Tilføjer **blockToNumber** metoden, til de parametre som er af typen **block**, således at **blockToNumber** kun skal udregnes én gang pr. **block** formel parameter og ikke hver gang en formel parameter bliver benyttet, da formelle parametre også kan være af typen **integer**.

- **public void caseTEndfunctionKeyword (TEndfunctionKeyword node)**

Slutter funktionen ved at tilføje **'}'** til sidst.

- **public void caseAFunctioncall(AFunctioncall node)**

Bruges til funktionskald, og skal derfor ændre de aktuelle parametre så de passer til ScriptCraft. Dette gøres ved at bruge `ConvertActualParams`. Metoden tilføjer desuden `this.` foran funktionsnavnet. Da det er `this` keywordet der afgør om det er den aktuelle drone, der skal fortsætte funktionen, eller om det er en ny og nulstillet drone der skal tage funktionskaldet.

- `public void ConvertActualParams(PActualparameters node)`

Fungerer på samme måde som `ConvertFormalParams` og kalder `ConvertActualParam` det korrekte antal gange, i forhold til antallet af aktuelle parametre, ved at kalde sig selv rekursivt.

- `public void ConvertActualParam(PActualparameter node)`

Sørger for at sætte gåseøjne rundt om `block` værdien, og gør ingenting ved integers.

Selvom `ConvertActualParams` og `ConvertFormalParams` gør to forskellige ting, fungerer de på samme måde. Som det ses i kodeeksempel 4.28 er der på linje 3 en `if`-sætning, som siger at hvis der kun er én parameter til funktionen, kaldes `ConvertFormalParam`, ellers bliver `ConvertFormalParam` kaldet alligevel, sammen med metoden selv. Således kører den til, at der før eller siden kun er én parameter. `ConvertFormalParam` assigner parameteren til sig selv efter at `blockToNumber` metoden er kørt. STONE ville således blive til 1. `ConvertActualParams` fungerer på samme måde. Bortset fra at der ikke skal konverteres parametre, men i stedet tilføjes gåseøjne omkring block strengen, f.eks. skal STONE skrives "STONE".

```

1  public void ConvertFormalParams (PFormalparameters node)
2  {
3      if (node.getClass ().equals (AParamFormalparameters.class))
4      {
5          ConvertFormalParam (((AParamFormalparameters)node).getFormalparameter ());
6      }
7      else
8      {
9          ConvertFormalParam (((AParamsFormalparameters)node).getFormalparameter ());
10         ConvertFormalParams (((AParamsFormalparameters)node).getFormalparameters ());
11     }
12 }
13
14
15 public void ConvertFormalParam (PFormalparameter node)
16 {
17     if (node.getClass ().equals (ABlockFormalparameter.class))
18     {
19         writer.println (node + " = blockToNumber (" + node + ");");
20     }
21 }
```

Kodeeksempel 4.28. ConvertFormalParams og ConvertFormalParam

Build-udtrykket

Build-udtrykket er en metode til at konstruere blokke i Minecraft på en alternativ måde, dette kan være både mere effektivt og sjovere, se evt. afsnit 2.3.2 for generel beskrivelse af build-udtrykket. Som ses på billedet i figur 4.12, er det sted i koden hvor der er et &

control block tegn, det sted hvor muren bliver bygget. Det ses, på billedet i højre side af figuren, at i midten af skærmen, er den blok, som & repræsenterer, bygget i jorden under hvor musen peger. Derudover kan det ses, at muren er bygget én blok nede i jorden, da den blok der blev peget på, da muren blev konstrueret, lå i jordoverfladen og den blok der bliver peget på, når konstruktionen startes, bliver overskrevet med & blokken. Da der er blokke på den samme linje kode som &, til venstre og højre, kommer der også blokke nede i jorden til venstre og højre. Det ville også være muligt at bruge ! eller ? i stedet for &, hvor konstruktionen af muren starter der hvor ! er i koden og efter bygningen er konstrueret, vil dronen være ved ? i koden. Dette kan være hjælpsomt hvis man skal lave flere ting der skal være i forlængelse af hinanden.

```

1  block s = STONE
2  block & = AIR
3
4  build(front)
5    s - s - s - s - s - s - s - s - s -
6    s s s s s s s s s s s s s s s s s s
7    s s s s s s s s s s s s s s s s s s
8    s s s s s s s s - - - s s s s s s s s
9    s s s s s s s s - - - s s s s s s s s
10   s s s s s s s s - & - s s s s s s s s
11 end build

```



Figur 4.12. Grafisk visning af build-udtrykket

Build-udtrykket bliver gennemgået af en algoritme, som vist i kodeeksempel 4.29, som fungerer ved at bruge alle `block` variabler læst ind i et array, som er en del af klassen `BlkArray`, som er en klasse med tre hovedfunktioner. De tre hovedfunktioner er henholdsvis at tilføje blocks til et array og holde styr på information omkring de indlæste blocks (`AddBlk`), optimere på runtime (`OptizeCol` og `AddBlk`), og konvertere build-udtrykket til JavaScript kode som virker med ScriptCraft (`Convert`). Optimeringsmetoden vil blive gennemgået i afsnittet optimering af build-udtrykket 4.6.2. De to andre hovedfunktioner af `BlkArray` vil blive forklaret i dette afsnit. Når man ser på den kode der bliver kaldet når "treewalkeren" finder et build-udtryk, kodeeksempel 4.29, kan man se at det er opsat sådan, at det er muligt at læse output filen. Der bliver først indsatt en kommentar, der indikerer hvor build-udtrykket starter og slutter, af den grund at `build` ikke eksisterer i ScriptCraft, men er en samling *moves* og *box placements*. Dette kan ligne det kode, som programmøren eventuelt har skrevet over eller under build-udtrykket. Derudover bliver der også lavet en højere indrykning til koden build-udtrykket genererer.

På linje 7 i kodeeksempel 4.29, kan man se at der bliver oprettet et objekt af klassen `BlkArray`, og initialiseres til et nyt objekt. I constructoren til `BlkArray` skal der bruges information til at klare oversættelsen af build-udtrykket. Første parameter skal indeholde den orientering som `build` indeholder i dens parenteser, se evt. figur 4.12 til venstre, hvor der står `front`. Orienteringen er det perspektiv funktionen er skrevet i, og skal ændre hvilket plan det todimensionelle build-udtryk skal konverteres til. Anden parameter skal indeholde `PrintWriteren` vi bruger til at skrive til filen, så vi kan printe til den aktuelle output fil. Tredje parameter skal indeholde den aktuelle indrykning, og den fjerde

parameter skal indeholde en boolean, der beskriver om funktionen skal være med eller uden optimering. Da det eventuelt kunne være sjovt for brugeren, at se hver blok blive bygget enkeltvis.

På linje 10 i kodeeksempel 4.29, starter en **for**-løkke der skal gennemgå alle dele af det array, som indeholder de blocks der skal udskrives til **build**-udtrykket. De blocks kan findes i **node.getBlock()**, den indeholder udelukkende det der er skrevet efter slutparentesen i **build(orientation)** og indtil **end build** keywordet er fundet. I tilfældet med muren figur 4.12, ville den bestå af 96 S blocks, 17 tomme blocks angivet med -, 7 linjeskift oversat til "\$" af scanneren og én block af typen & som bliver kaldet **b1** når den er konverteret til ScriptCraft. Hver eneste **block** i arrayet bliver tilføjet til objektet **blkArray**, med metoden **AddBlk**, denne metode tilføjer alle blocks til et array, som er et member i **ArrayBlk**, derudover holder metoden styr på hvor ! og ?, eller & er placeret til senere brug. Metoden er også et led i optimeringen, dette vil blive gennemgået i afsnit 4.6.2.

```

1 public void caseABlocksBuild(ABlocksBuild node)
2 {
3     indent.AddIndent();
4     writer.println("// Build Function start:");
5     indent.IncreaseIndent();
6
7     BlkArray blkArray =
8         new BlkArray(node.getOrientation(), writer, indent, isOptimized);
9
10    for (int i = 0; i < node.getBlock().size(); i++)
11    {
12        blkArray.AddBlk(node.getBlock().get(i).toString().trim());
13    }
14    blkArray.Convert();
15
16    indent.DecreaseIndent();
17    indent.AddIndent();
18    writer.println("// Build Function ended");
19}
```

Kodeeksempel 4.29. BlockBuild-metoden

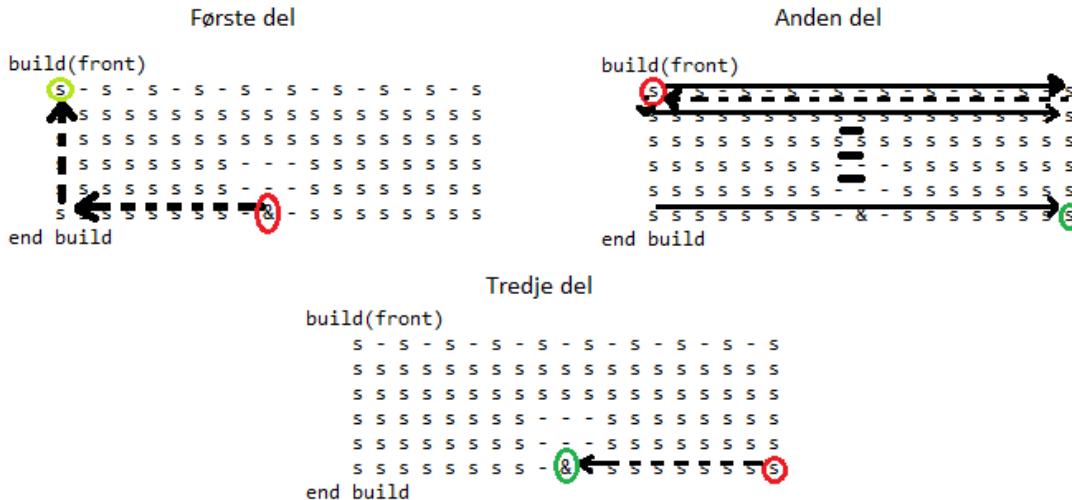
På linje 14 kodeeksempel 4.29, er alle blocks læst ind i arrayet i objektet **blkArray**, og den er klar til at konvertere arrayet til drone flyt og blok placeringer, og derefter printe resultatet til output filen. Metoden **Convert** har tre dele, som kan ses på figur 4.13, hvor de stiplede linjer symboliserer drone flyt, og de fulde linjer symboliserer flytning af drone og placering af blokke. Den røde cirkel symboliserer der hvor dronen er placeret, inden at den del af koden bliver kørt, og den grønne er der hvor dronen står efter koden er kørt.

Først flytter den dronen fra startpunktet, markeret med & eller !, til toppen til venstre. Vi starter i øverste venstre hjørne, da det er her at arrayet's første blok er placeret. Når **build** bliver indlæst af scanneren tager den fra højre mod venstre, og oppe fra og ned.

Anden del går ud på at skrive alle blocks ud, siden at dronen er placeret i venstre øverste hjørne, skal hele arrayet med blocks gennemgås. Hver der kommer et linjeskift, markeret med \$ i arrayet, skal dronen flyttes tilbage, vist med stiplede linjer på figur 4.13, og én ned for at komme på næste linje. Hver gang der skal skrives en **block** ud bliver der kaldet en **PrintBlock**-metode, som printer en **block** ud. Når der ikke er flere blocks i arrayet,

er alle blocks skrevet ud, og dronen bliver efterladt på sidste blocks plads, nederst højre hjørne i **build**-udtrykket.

Tredje del flytter dronen til den rigtige position. Det gør den ved at trække sidste blocks position, fra positionen på slutpunktet, markeret med & eller ?.



Figur 4.13.

For at tage forhold til orienteringen, er der lavet fire string variabler med navn **left**, **right**, **up** og **down**, disse string variabler repræsenterer vores todimensionelle **build**-udtryk. Hvis orienteringen er **front**, skal indholdet i streng variablerne passe til deres navne. Derimod hvis orienteringen er **top**, skal **up** og **down** ikke indeholde op og ned moves, men i stedet indeholde **forward** og **back moves**, og lignende hvis orienteringen er **side**, skal left og right være **forward** og **back**. På den måde kommer orienteringen automatisk ind i udskrivninger, ved at skrive de rigtige *moves* ind i streng variablerne.

4.6 Optimering

Der er flere måder at optimere en compiler på. Overordnet skelner man ofte mellem hovedområderne **completetime**- og **runtimeoptimering**. Optimering på **completetime** skal sænke den tid det tager at kompilere et program, og optimering på **runtime** skal øge hastigheden på det program der bliver kompileret. Optimering af **completetime** kan sænke hastigheden på **runtime**, og ligeledes kan optimering af **runtime** sænke hastigheden på **completetime**. Nogle optimeringer har deres eget gennemløb at AST'et, men for denne optimering er dette ikke nødvendigt, og optimeringen vil ske som et led af code generatoren. I dette afsnit vil der blive redegjort for hvorfor og hvordan, compileren er blevet optimeret samt resultaterne af optimeringen.

4.6.1 Valg af optimering

I processen med valg af optimering, er der blevet set på hvilket hovedområde der er det vigtigste for compileren. Programmerne der skal skrives i VMP, vil som udgangspunkt ikke være store i forhold til store programmer skrevet i C# eller Java. En grund til at

programmerne ikke bliver lige så store, er at compileren ikke er udviklet til at kunne understøtte kompilering af samlede projekter, kun enkelte filer. Hver fil bliver kompileret hver for sig, det er dog muligt at lave en stor fil, men dette kan gøre det svært at holde overblik. VMP er desuden udviklet til personer der er nye til programmering, og det er endnu en grund til at programmerne, der skrives i VMP, som udgangspunkt ikke bliver helt så store. Derudover er sproget forholdsvis enkelt, så der er ikke en stor belastning på compiletime. En af grundene til dette er, at compileren oversætter fra et høj-niveau sprog til et andet høj-niveau sprog, og der derfor flere steder i code generation ikke behøves at oversætte alt, da der er ligheder imellem VMP og ScriptCraft, som er målsproget.

Efter at have lavet compileren uden optimering, og kørt programmer til fejlfinding, er det observeret at det, at placere én blok eller at flytte dronen én placering, sker som én operation ad gangen, som tager tid i Minecraft, hvilket vil kunne få bygning af større konstruktioner til at tage lang tid. For bestemte tider før og efter optimering se afsnit 4.6.3. Det er muligt med ScriptCraft at lave flere blokke på samme tid med ScriptCrafts egen funktion, `box()`. Funktionen er optimeret fra ScriptCraft, så den er hurtigere, end at placere de samme blokke enkeltvis. Når man placerer blokke enkeltvis, skal man også flytte dronen imellem hver blok. Dette er ikke tilfældet med `box()`, hvor dronen bliver stående, så der bliver sparet runtime tid, både på placering af blokke og på at dronen ikke skal flyttes lige så ofte mellem placeringerne.

På baggrund af at compiletime i forvejen er meget kort, og at VMP ikke er udviklet til store indviklede systemer, men til et begrænset brug inde i Minecraft, og at runtime er langsom, hvilket kan være frustrerende i store konstruktioner, er der valgt at fokusere på optimering af runtime, på bekostning af compiletime. Dette kan gøres ved at optimere, således at bloks bliver samlet og placeret med ScriptCraft funktionen `box()`, hvor der er observeret væsentlige hastighedsforskelle. Derudover en generel optimering, så det sikres at dronen ikke bliver flyttet unødvendigt, eller laver andet unødvendigt arbejde på runtime. En anden og lidt mere normal mulighed for at optimere, er at udregne alle expressions på compiletime. Denne optimering kan med fordel udregnes inden code generation, men er fravalgt, da udregninger kun kan benytte integers og har en begrænset mulighed for store komplicerede udregninger, samtidig vil den øgede hastighed af programmet være minimal.

4.6.2 Optimering af build

Der er observeret langsom placering af blokke, når man sætter hver enkelt blok hver for sig. Ud over det, er output koden blevet analyseret, for at finde ud af om dronen bliver flyttet unødvendigt, når udtrykket *build* bliver oversat. Der er fundet nogle helt basale flyt, som kan spares væk. I forbindelse med optimering af `build`-udtrykket har vi valgt at se på de to hovedområder; `box` optimering og `drone` optimering.

Box optimering

Det første punkt til optimering vi fandt var, at det tog lang tid at bygge konstruktioner. Dog findes der nogle indbyggede funktioner i ScriptCraft som f.eks. `box()`. De indbyggede funktioner kan bygge forudbestemte konstruktioner, hvilket var meget hurtigere, end dem bygget med vores kompilerede kode. De konstruktioner der er lavet i ScriptCraft, fungerer

ved at placere flere blokke på samme tid. Optimeringen er dermed en algoritme der kan finde blokke der ligger i nærheden af hinanden, og lave kasser, bestående af samlede blokke, ud af dem. Da det eventuelt kunne være sjovt for en bruger, at se sin konstruktion blive bygget blok for blok, er det muligt at slå optimeringen fra.

I ScriptCraft er det muligt at lave en blok med funktionen set i kodeeksempel 4.30, en anden mulighed er at bruge samme funktion, men nu med flere parametre, som ses i kodeeksempel 4.31. I kommentarene, kodeeksempel 4.31, vises der hvad der skal stå ved de forskellige parametre, den første er en integer der bestemmer blok typen. I kode eksempel 4.30 skal `box()` kun bruge en blok type.

```
1 this.box(1);
```

Kodeeksempel 4.30. En STONE block

```
1 this.box(1, 2, 2, 2); //this.box(Block type, bredde, højde, dybde)
```

Kodeeksempel 4.31. Otte STONE blokke placeret i en kasse

Hvis man vil lave flere blokke inde i Minecraft på samme tid, skal man bruge funktionen med fire parametre, hvor de sidste tre parametre beskriver størrelsen. For at gøre dette er det vigtigt, at have information omkring hvor de enkelte blokke skal placeres. For at holde styr på dette er der lavet en klasse til en blok, ved navn `Blks`, se kodeeksempel 4.32, som kan indeholde en *String* der skal holde styr på, hvilken blok der skal placeres. En position, bestående af koordinaterne *x* og *y*. Hvor *x* beskriver placering på rækken og *y* beskriver hvilken række blokken skal placeres i. Derudover indeholder klassen også størrelse, en `sizeX`, som beskriver hvor stor blokken er fra blokkens position, og til højre derfra, og en `sizeY`, som beskriver hvor stor blokken er fra blokkens position, og nedad derfra. 4.32 holder kun styr på to dimensioner, så det ikke er muligt at lave kasser, kun todimensionelle flader. Ved at holde styr på position og størrelse, er det muligt at skrive flader af blokke ud på én gang, med funktionen vist i kodeeksempel 4.31.

```
1 public class Blks
2 {
3     public String block;
4     public int sizeX = 1;
5     public int sizeY = 1;
6     int x;
7     int y;
8
9     public Blks(int X, int Y, String Block)
10    {
11        x = X;
12        y = Y;
13        block = Block;
14    }
}
```

Kodeeksempel 4.32. Blks klassen med constructor

Optimeringen består i at finde klynger af blokke. Den første metode der bliver taget i brug, er når blokkene bliver læst ind med `AddBlk` metoden, se kodeeksempel 4.33. Denne optimering tilføjer hver blok i rækkefølge, en række af gangen, adskilt af linjeskift, hvilket har tegnet '\$'. Dette giver mulighed for at gemme en `lastBlk`, og hver gang der bliver læst noget nyt ind, bliver `lastBlk` sammenlignet med `newBlock`. Hvis de to blokke er

ens, skal størrelsen på sidste blok laves større på x-aksen, da der indlæses fra venstre mod højre. Hvis ikke de to nyligt indlæste blokke er ens, kan blokkene ikke samles, og der skal tilføjes en ny blok til arrayet, og derudover skal **lastBlk** ændres til den aktuelle blok. Hvis ikke optimeringsmuligheden er tilvalgt, skal alle blokke blot tilføjes til arrayet, og skrives ud enkeltvist. Efter indlæsningen kan der være blokke i forskellige størrelser. Når de skal skrives ud, bliver de skrevet ud i den størrelse, som er angivet til hver blok. Dette giver samme slutresultat inde i Minecraft, men for hver optimeret blok, vil der være ét flyt med dronen og en blokplacering mindre i output koden.

```

1 if (isOptimized)
2 {
3     //hvis man gerne vil have den hurtige bygning inde i minecraft.
4     if (block.size() > 0 && newBlock.equals(block.get(lastBlk).block))
5     {
6         block.get(lastBlk).sizeX++;
7     }
8     else
9     {
10        lastBlk++;
11        block.add(new Blks(blocksSinceNl, noOfNl, newBlock));
12    }
13 }
14 else
15 {
16     block.add(new Blks(blocksSinceNl, noOfNl, newBlock));
17 }
```

Kodeeksempel 4.33. Optimering ved indlæsning af blokke til BlkArray

Hver eneste række er nu optimeret så meget som muligt, men ønsker vi at indsætte en flade der er mere end én høj, skal yderligere optimering tilføjes. Til det formål er der en optimeringsfunktion der bliver kaldt på arrayet inden udskrivning til outputfilen. Metoden hedder **OptimizeCol**, og bliver kun kaldt hvis optimering er valgt. **OptimizeCol** består af to arrays af **Blks**, et array til at indholde den række der bliver indlæst blokke på, **currentRow**, og et array til at indholde den forrige række **lastRow**. Når **currentRow** ikke længere er den aktuelle række, bliver den læst over i **lastRow**. Den eneste opgave **currentRow** har, er at holde på de blokke der senere skal læses over i **lastRow**.

OptimizeCol indeholder to løkker, første løkke går igennem alle blokke der er tilføjet til det overordnede array, og tilføjer dem til variablen **curBlock**. Hvis der kommer linjeskift sørger første løkke for at **currentRow** læses ind i **lastRow**, og derefter nulstiller **currentRow** så den er klar til næste række.

Den anden løkke er inde i den første løkke, og bliver kun kørt, hvis ikke den aktuelle blok læst ind i første løkke er linjeskift eller en tom block, markeret med ' - ', og kun hvis **lastRow** ikke er tom. Hvilket vil sige, at hvis der kommer en blok ind, som kan optimeres, og at det ikke er første linje, skal muligheden for optimering kontrolleres. Den anden løkkes krop er vist i kodeeksempel 4.34, den første del er en **if**-sætningen der sammenligner den aktuelle blok, **curBlock**, med alle blokke i forrige række, **lastRow**. Hvis der er en blok der i forrige række har samme placering på rækken (x-aksen), samme blok type, og samme størrelse, skal der laves en optimering.

Optimeringen kan kun ske én gang pr. række, og derfor kaldes **break** i slutningen af

optimeringen. Optimeringen består af to dele, en del der skal tælle størrelsen op, og en del der skal ændre forrige række, til at være en tom række. Når blokken skal tælles op, er det vigtigt at både blokken i `currentRow` og den aktuelle blok i arrayet bliver talt op. Da `curBlock` bliver til en del af `lastRow`, og den skal stadig indeholde korrekte blokke i næste gennemkørsel. Optællingen sker ved at finde sidste blok i `currentRow` og tælle den op, og derefter finde den aktuelle blok og tælle den op. Den aktuelle blok er beskrevet af den integer der bliver talt op i første løkke, henholdsvis `i`, se linje 12 i kodeeksempel 4.34.

Den anden del af optimeringen skal lokalisere den blok, der ligger lige over `curBlock` i `build`-udtrykket. Denne blok laves til en tom `block` og skal ændre størrelse til én. På linje 14 kodeeksempel 4.34, bliver forrige blok lokaliseret og overskrevet med tom `block`. For at lokalisere forrige blok, skal `currentRow.size()` adderes med `lastRow.size() - j` som beskriver afstanden fra den blok som skal optimeres til starten af `currentRow`. Hvis en blok både er optimeret i `AddBlk` og i `OptimizeCol`, vil den udskrive en kasse. Den blok der er optimeret på, vil befinde sig i nederste venstre hjørne, af den flade der skal udskrives. Dette er optimalt da `Box()` bygger netop fra dette hjørne og op, og til højre. Så dronen skal ikke flyttes unødvendigt, når blokken altid kommer nederst til venstre, i den flade den optimerer på.

```

1 if (lastRow.get(j).block.equals("-"))
2 {
3     continue;
4 }
5
6 if(curBlock.x == lastRow.get(j).x
7     && curBlock.block.equals(lastRow.get(j).block)
8     && curBlock.sizeX == lastRow.get(j).sizeX)
9 {
10    //saa current block til at talt op til rigtig antal stoerrelse.
11    currentRow.get(currentRow.size() - 1).sizeY = ++lastRow.get(j).sizeY;
12    block.get(i).sizeY = lastRow.get(j).sizeY;
13    //sidste række skal nulstilles
14    block.get(i - (currentRow.size() + (lastRow.size() - j))).block = "-";
15    block.get(i - (currentRow.size() + (lastRow.size() - j))).sizeY = 1;
16    break;
17 }
```

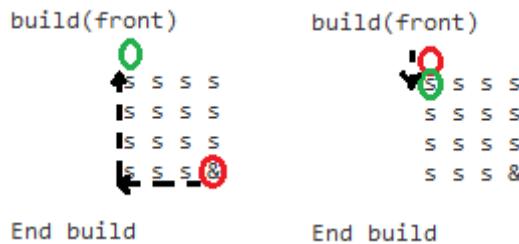
Kodeeksempel 4.34. Kolonnevis optimering

Efter `OptimizeCol` har været igennem alle blokke, og der er optimeret alle de steder hvor det er muligt, skal resultatet skrives ud på outputfilen. I processen med at skrive blokke og udføre drone flyt til outputfilen bliver `box()` funktionen med fire parametre, kodeeksempel 4.31, brugt hvis den aktuelle blok har en størrelse større end én blok. Hvis blokken til udskrivning kun indeholder information til en blok, bliver `box()` med en parameter brugt, se 4.30.

Drone optimering

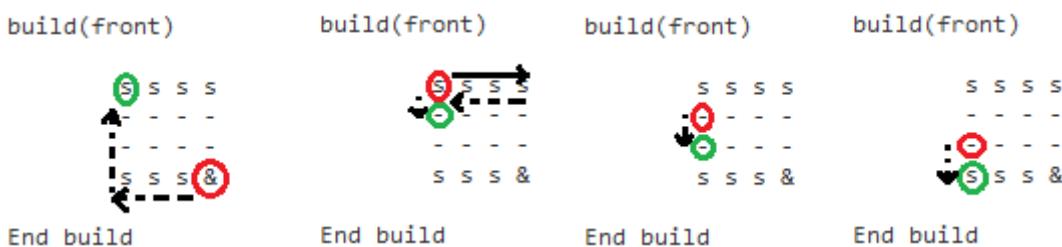
Som et led i optimeringen blev koden til de kompilerede testprogrammer analyseret, for at se om der var unødvendige dele i den genererede kode. Der blev fundet fire mulige optimeringer, som alle er blevet valgt, altid skulle være aktive, da det generelt gjorde det hurtigere, men at alle optimeringerne kun var med usynlige "moves", noget der ikke ville kunne observeres inde fra Minecraft. De fire optimeringer vil kort blive beskrevet:

Første, når der skrives en `build`, er der muligheden for at starte med et linjeskift for at få en mere overskuelig kode, og et godt billede af hvad der kommer i Minecraft. Disse linjeskift var også med i koden, sådan at dronen flyttede op på øverste række kun for at gå ned igen, da der kun var linjeskift, se figur 4.14. Samme problem var der selvfølgelig til sidst, hvor man også begrundet læslighed kan vælge at have linjeskift efter. For at løse dette problem blev der tilføjet kode i `AddBlk`, der ignorerede alle linjeskift inden første blok var indlæst. For at løse de sidste linjeskift, blev linjeskift ændret, til at blive skrevet ind når der blev fundet en blok, i stedet for når der blev fundet et linjeskift.



Figur 4.14. Unødvendigt droneflyt på højre side

Anden optimering omhandler også linjeskift. Når et linjeskift var fundet blev linjeskiftet udskrevet, i form af at flytte dronen tilbage til venstre side af `build`-udtrykket og ned på næste linje. I det tilfælde at der er en linje kun med tomme blokke eller ingenting på linjen, vil dronen så gå ned en række af gangen, se figur 4.15. Optimeringen blev implementeret i udskrivningen af arrayet, ved at tælle alle linjeskift op, og når der var en blok der skulle skrives ud, blev linjeskiftene skrevet ud, hvis der var registreret nogle siden sidste blok. Denne optimering bliver udnyttet ekstra meget i det tilfælde, at optimeringsmuligheden er valgt, da box optimeringen indsætter en tom `block` hver gang den optimerer. Hvis der er to, eller flere, ens rækker i et `build`-udtryk, vil alle linjer der bliver optimeret på, bortset fra den sidste, blive til tomme linjer.



Figur 4.15. Unødvendigt at dronen skal gå en linje ned ad gangen

Tredje og **Fjerde** optimering havde samme løsning. Hver gang der kom en blok af typen tom `block`, markeret med ' - ', blev dronen programmeret til at flytte sig forbi den. Dette gav problemerne, at dronen flyttede sig lige inden linjeskift, og derfor flyttede sig til højre, for at flytte sig til venstre lige bag efter. Problemet er at hvis der er flere tomme blocks lige efter hinanden, ville dronen flytte dem separat. Løsningen er, ligesom med linjeskift, at tælle dem op og udskrive dem først når der kommer en blok. Det er dog vigtigt at variablen, der indeholder hvor meget dronen skal flytte sig, nulstilles ved linjeskift. På

den måde flyttes dronen ikke separat, og dronen vil ikke flytte sig fordi den finder en tom `block` lige før et linjeskift.

Disse optimeringer har optimeret på antallet af flyt dronen skal lave, samt fjernet en del unødvendige linjer kode, inde i output filen.

4.6.3 Resultater af optimering

Der er blevet udført test af koden, for at undersøge hvorvidt koden er blevet optimeret vedrørende samling af blokke. På baggrund af disse test, er der fremsat en hypotese:

Efter hvordan optimeringsmetoden fungerer, vil worst case med optimering give samme resultat som uden optimering. Dette skyldes at optimeringen i worst case, ikke har nogen mulighed for at samle blokke til større grupper, og der bliver derfor ikke foretaget nogen ændring i koden. Så snart den har mulighed for at samle blokke vil optimeringen træde i kraft. Dermed vil enhver konstruktion i værste tilfælde være lige så hurtig, som koden genereret uden optimeringen. I alle andre tilfælde end worst case, vil optimeringen kunne sætte blokke sammen, og der forventes at se både ændring i hastighed inde i Minecraft og på antallet af linjer koden genererer i processen.

Opsummering på hypotesen er at i værste tilfælde vil der være samme hastighed både med og uden optimering. Men i bedste tilfælde ville der være væsentlig forskel på både hastighed og antal linjer.

Test

Ud fra den fremsatte hypotese, er der udført en række test, som tester både på worst case og best case. Til disse tests, skal konstruktionen være simpel og derfor er det valgt at lave én kasse, i forskellige størrelser. Best case vil bestå af samme type blok, og i worst case skal der ikke kunne være mulighed for optimering, og derfor er der valgt en skakternet blokoverflade. Da det er umuligt at optimere noget, når der ikke er to ens blokke ved siden af hinanden. Der bruges samme computer til alle testene, da tiderne muligvis kan variere fra computer til computer. Tiden bliver taget fra den første blok er synlig, til at den sidste blok er sat. Tiden bliver taget med et stopur, og derfor er det kun en vejledende tid, og der skal tages højde for fejlmargin. Koden kørt i første test med at lave en 10x10x10 kasse, best case:

```

1 function tixti()
2   block s = STONE
3   block & = STONE
4
5   repeat(10)
6     build(top)
7       & s s s s s s s s s
8       s s s s s s s s s
9       s s s s s s s s s
10      s s s s s s s s s
11      s s s s s s s s s
12      s s s s s s s s s
13      s s s s s s s s s
14      s s s s s s s s s
15      s s s s s s s s s
16      s s s s s s s s s

```

```

17     end build
18     up(1)
19   end repeat
20 end function

```

Kodeeksempel 4.35. Best case gennemkørsel af 10x10x10

Resultat af best case blev:

Med optimering:

Hastighed: 2-3 sekunder

Antal linjer: 19 (7 linjers build)

Uden optimering:

Hastighed: 1 minut 30 sekunder

Antal linjer: 221 (211 linjers build)

Efter første test er det tydeligt at se, at optimeringen kan hjælpe både med hastighed og linjeantal. Den næste test omhandler worst case, hvor der ikke er mulighed for at sammensætte flere blocks. Det forventes at koden er af samme størrelse og at hastigheden er den samme:

```

1 function tixti()
2   block s = STONE
3   block g = GLASS
4   block & = STONE
5
6   repeat(10)
7     build(top)
8       & g s g s g s g s g
9       g s g s g s g s g
10      s g s g s g s g s
11      g s g s g s g s g
12      s g s g s g s g s
13      g s g s g s g s g
14      s g s g s g s g s
15      g s g s g s g s g
16      s g s g s g s g s
17      g s g s g s g s g
18   end build
19   up(1)
20 end repeat
21 end function

```

Kodeeksempel 4.36. Worst case gennemkørsel af 10x10x10

Resultat af worst case blev:

Med optimering:

Hastighed: 128 sekunder

Antal linjer: 222

Uden optimering:

Hastighed: 129 sekunder

Antal linjer: 222

Efter resultaterne på de første to forsøg, er det tydeligt at se en forskel på hastigheden og antal linjer, desuden observeres det at koden i worst case er identisk med og uden optimering. På grund af at måleudstyret er et stopur vil der ikke testes på ekstremt store konstruktioner, men det er dog stadig muligt at se på antallet af linjer. Så der vil også blive lavet tests med 2x2x2 og 100x100x100. I worst case holder vi igen øje med om der er identisk kode.

Best case:

2x2x2

linjer med optimering: 19 (7 linjers build)
linjer uden optimering: 22 linjer (10 linjers build)

100x100x100

linjer med optimering: 19 (7 linjers build)
linjer uden optimering: 20112 (20100 linjers build)

Worst case:

2x2x2

linjer med optimering: 22 (10 linjers build)
linjer uden optimering: 22 (10 linjers build)

100x100x100

linjer med optimering: 20112 (20100 linjers build)
linjer uden optimering: 20112 (20100 linjers build)

Efter kompilering af mere ekstreme forhold, er det observeret at tests med 10x10x10 og 100x100x100, med og uden optimering, har samme antal linjer. Derudover er koden i worst case med og uden optimering identisk. For at teste på nogle lidt mindre teoretiske konstruktioner, er det valg at benytte nogle af de konstruktioner, brugt i forbindelse med udokumenterede tests på om compileren fungerede som den skulle. Der er udvalgt to slotte, den første bestående af 21 `build`-udtryk, den anden bestående af 6 `build`-udtryk.

Test af stort slot, 4.16:

Med optimering:

Hastighed: 24 sekunder
Antal linjer: 381

Uden optimering:

Hastighed: 14 minutter og 53 sekunder
Antal linjer: 4798

Test af lille slot, 4.17:



Figur 4.16. 21 build(top)-udtryk



Figur 4.17. 6 build(blandet) med en tilhørende trappe lavet uden build.

Med optimering:

Hastighed: 10,5 sekunder
Antal linjer: 371

Uden optimering:

Hastighed: 3 minutter og 12 sekunder
Antal linjer: 4502

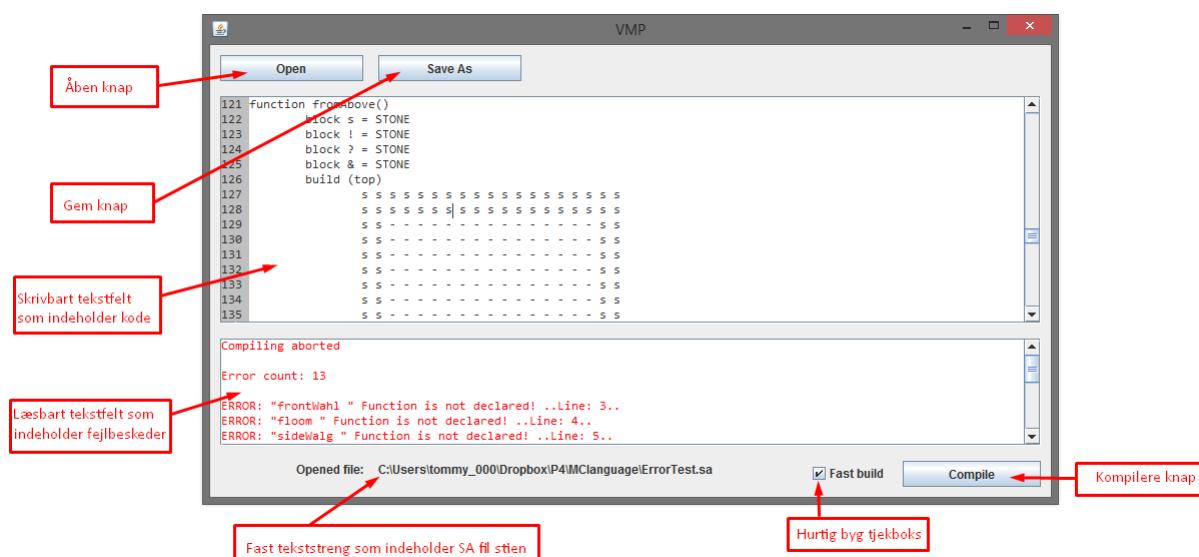
Linjeantallet på de to slotte er meget tæt, selvom det første er noget større end det andet. Grunden til dette, er at den ene har loops omkring sine `build`-udtryk, og genbruger derfor mange linjer. Det andet slot har tilgengæld flere detaljer, som trappe og vinduer. Med tests er der set på flader, der giver en forståelse for hvor meget optimeringen kan hjælpe. Og tests af to slotte, som giver et billede af hvordan optimeringen kan hjælpe i større konstruktioner, med en del blokke af samme type.

4.6.4 Opsamling og vurdering

Optimeringen fungerer efter hensigten, worst case med optimering, genererer kode som er identisk med koden uden optimeringen. På trods af optimeringen i worst case giver længere compiletime og ikke ændrer på runtime, er der stor tilfredshed med resultatet. På konstruktioner med mulighed for optimering, kan optimeringen på runtime gøre en stor forskel, og målingerne fra forsøget med de to slotte oversteg alle forventninger. Koden er også lettere læselig efter at det er kompileret, da det kan være svært at overskue 4500 linjer, som siger det samme som 370 linjer.

4.7 Grafisk brugerflade

Til VMP er der lavet en simpel grafisk brugergrænseflade, som kan bruges til at programmere i og til at kompilere sprogets kode. Dette er valgt, for at gøre det så nemt som muligt, specielt for børn, at kunne programmere i VMP og kompilere koden bagefter, uden der er alt for mange knapper og funktioner, som er svære at finde rundt i. En grafisk brugergrænseflade er også mere brugervenlig, end hvis der skulle bruges en konsol til at kompilere koden, da der ved brug af en konsol kræves, at man skal kunne huske de forskellige kommandoer til at finde og kompilere filen som indeholder koden. Med en grafisk brugergrænseflade kan det gøres ved bare at trykke på en knap.



Figur 4.18. Screenshot af den grafiske brugergrænseflade.

Figur 4.18 viser den grafiske brugergrænseflade, som består af; et vindue, tre knapper, en tjekboks, et skrivbart tekstfelt, et læsbart tekstfelt og en fast tekststreng som kun

systemet kan ændre på. Den første knap *Open* åbner en *open file dialog box*, som bliver brugt til at finde og åbne eksisterende filer som indeholder kode, der er skrevet i VMP. Koden i den fil, man har valgt at åbne, bliver så lagt over i det skrivbare tekstfelt, som gør det muligt at redigere i koden, og den faste tekststreg bliver tildelt stien til den åbnede fil. Hvis man ikke har en eksisterende kodefil, er det muligt at skrive en, og det skrivbare tekstfelt kan bruges til dette, hvor der kan skrives en helt ny funktion fra bunden af. Inden man kan kompilerer koden, skal det gemmes i en fil først. Dette gøres ved at trykke på knappen *Save As*. Når der trykkes på denne knap åbner en *save file dialog box*, som kan bruges til finde den ønskede mappe, at gemme sin kode i. Når det er gjort, gemmer den koden i en fil, og den faste tekststreg bliver tildelt stien til filen. Den sidste knap *Compile* bliver brugt til at kompilere koden til ScriptCraft kode, men inden det kan blive gjort, er der krav som skal opfyldes. Den faste tekststreg skal innehölde stien til en kodefil, som betyder at der enten skal åbnes en eksisterende kodefil, eller skrives noget nyt kode i VMP og gemme det. Det andet krav er, at koden i skrivbare tekstfelt skal passe med koden i den kodefil, som den faste tekststreg innehölde stien til, hvis dette ikke er opfyldt, bliver man bedt om at gemme sine ændringer. Ved at dette er et krav, sikrer man at alt tilføjet, eller redigeret kode, er gemt inden kodefilen bliver kompilert. Når disse krav er opfyldt, åbnes der en *save file dialog box* for at finde destinationen som den kompilerede JavaScript fil skal gemmes til. Når destination er fundet kompileres koden til ScriptCraft og filen bliver gemt, hvis og kun hvis, der ikke er nogen fejl at finde. Hvis der bliver fundet fejl, kompileres der ikke, og i stedet bliver det læsbare tekstfelt tildelt de fejl som er fundet, samt hvilke linjenumre fejlene er fundet på. Når fejlene er rettet, kan man forsøge at kompilere igen. Der er også en valgmulighed, som skal vælges inden man kompilerer koden ved at bruge tjekboksen *Fast build*, til enten at kompilere koden til ScriptCraft kode, som bygger hurtigt i Minecraft, eller til at kompilere koden til ScriptCraft kode som bygger én blok ad gangen i Minecraft. Denne valgmulighed er blevet tilføjet, da det kan være sjovt for børn at se hvordan deres Minecraft konstruktioner, som de selv har programmeret, bliver bygget en blok af gangen. Som standard er valgmuligheden sat til hurtig byg.

Test 5

I dette kapitel vil der blive beskrevet, hvordan vi har tænkt os at teste vores compiler, samt de resultater som testene giver. Grunden til at compileren bliver testet, er for at sikre at den producerer korrekt kode.

Måden compileren bliver testet på, er ved at benytte VMP til at lave små programmer, som bliver undersøgt, om de oversætter til ækvivalent kode i ScriptCraft. Dette vil blive illustreret med kodeeksempler igennem kapitlet, samt illustrationer af de endelige konstruktioner i Minecraft. Til at sammenligne og sikre, at den kompilerede kode er korrekt JavaScript, benyttes *w3schools*[41] som kilde.

5.1 Test

I dette afsnit vil der blive benyttet whitebox test til at sikre, at den kompilerede kode repræsenterer den kode som programmøren har skrevet. Dette undersøges ved at se på koden før og efter kompilering, og sammenligne disse. Derudover vil der blive analyseret eksempler, hvor der bevidst er lavet fejl i koden, for at teste fejlhåndteringen i compileren. Eksemplerne i første del af dette afsnit, indtil fejlhåndtering, vil ikke stå inde i funktioner, da det ikke ses som en nødvendighed, for illustrere hvad den aktuelle kode gør.

5.1.1 Kontrolstrukturer

Der startes med at se på de to kontrolstrukturer i VMP `if` og `repeat()`.

If else kontrolstruktur

Et eksempel på et `if else` kontrolstrukturen i VMP ser således ud:

```
1 if(6 /= 5)
2     //Krop
3 end_if
4 else-if(1 = 1)
5     //Krop
6 end_if
7 else
8     //Krop
9 end_else
```

Kodeeksempel 5.1. If else kæde i VMP

Kompileres kodeeksempel 5.1, ender vi ud med følgende kode:

```

1 if (6 != 5) {
2     //Krop
3 }
4 else if (1 == 1) {
5     //Krop
6 }
7 else {
8     //Krop
9 }
```

Kodeeksempel 5.2. Kompileret resultat af kodeeksempel 5.1

Sammenlignes den kompilerede `if else` kæde med det fra w3schools, kan der konkluderes at compileren oversætter `if else` kæden korrekt, og kodeeksempel 5.1 og 5.2 er ækvivalente.

Repeat()

Et kodeeksempel på et `repeat()` udtryk i VMP kan se således ud:

```

1 repeat(5)
2     //Krop
3 end repeat
```

Kodeeksempel 5.3. `repeat()` udtryk i VMP

Kompileres kodeeksempel 5.3, ender vi ud med følgende kode:

```

1 for (var i0 = 0; i0 < (5); i0++) {
2     //Krop
3 }
```

Kodeeksempel 5.4. Kompileret resultat af kodeeksempel 5.3

Da VMP `repeat()` bygger på en for-løkke, vil det resulterende kompilerede kode ikke umiddelbart ligne det oprindelige kode. Det har dog samme funktion, uover at i VMP er variablen ikke tilgængelig, da *kroppen* af kontrolstrukturen gennemløbes samme antal gange.

5.1.2 Operatorer

For at sikre at alle operatorer oversættes korrekt, ses her først en liste af operatorer i brug:

```

1 int aa = 5 * 5
2 int bb = 4 + 4
3 int cc = 3 / 3
4 int dd = 2 % 2
5 int ee = (aa - bb) * cc
```

Kodeeksempel 5.5. Operatorer skrevet i VMP

Kompileres kodeeksempel 5.5 forventes der at få kode, som ligner det, som blev kompileret, da operatorerne er ens i programmeringssprogene.

```

1 var aa = 5 * 5 ;
2 var bb = 4 + 4 ;
3 var cc = 3 / 3 ;
4 var dd = 2 % 2 ;
5 var ee = ( aa - bb ) * cc ;

```

Kodeeksempel 5.6. Kompileret resultat af kodeeksempel 5.5

Sammenlignes kodeeksempel 5.5 og 5.6, ses det tydeligt at resultatet af kompileringen giver noget kode, som ligner det fra VMP. Sammenlignes det kompilerede kode med hvad w3school afsnit om operatorer, kan der ses at koden er kompileret til korrekt JavaScript kode.

5.1.3 build()

Et eksempel på en `build()` i VMP kan se ud som i kodeeksempel 5.7.

```

1 function Wall()
2   block & = GLASS
3   block s = STONE
4   block g = GLASS
5
6   build(front)
7     g s g s g
8     s g s g s
9     & s g s g
10    end build
11 end function

```

Kodeeksempel 5.7. Et eksempel på et `build()`-udtryk i VMP

Kompileres kodeeksempel 5.7 returnerer kompileren det kode som ses i kodeeksempel 5.8.

```

1 var Drone = require('drone');
2 Drone.extend('Wall', function()
3 {
4   var b1 = blockToNumber("GLASS");
5   var s = blockToNumber("STONE");
6   var g = blockToNumber("GLASS");
7   // Build Function start:
8   this.up(2)
9   this.box(g)
10  this.right(1)
11  this.box(s)
12  this.right(1)
13  this.box(g)
14  this.right(1)
15  this.box(s)
16  this.right(1)
17  this.box(g)
18  this.left(4)
19  this.down(1)
20  this.box(s)
21  this.right(1)
22  this.box(g)
23  this.right(1)
24  this.box(s)
25  this.right(1)
26  this.box(g)
27  this.right(1)
28  this.box(s)

```

```

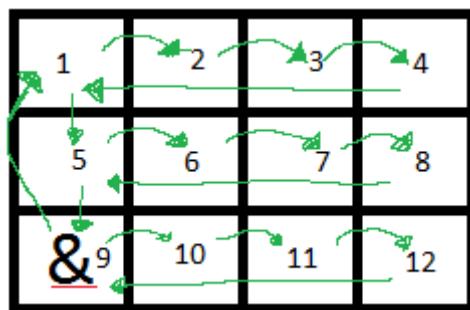
29     this.left(4)
30     this.down(1)
31     this.box(b1)
32     this.right(1)
33     this.box(s)
34     this.right(1)
35     this.box(g)
36     this.right(1)
37     this.box(s)
38     this.right(1)
39     this.box(g)
40     this.left(4)
41 // Build Function ended
42 );

```

Kodeeksempel 5.8. Kompileret resultat af kodeeksempel 5.7

For at sikre, at den kompilerede kode er korrekt, sammenlignes den med ScriptCrafts API[28]. Der kan ud fra API'en ses at syntaksen er korrekt. Der oprettes en drone, som benyttes til at bygge med. Den navigeres rundt med, og bruges til at placere de forskellige blokke, i forhold til hvor de er bestemt til at skulle være. `this` keywordet refererer til dronen, som det også er nævnt i afsnit 2.2.1.

Analyses den kompilerede kode, og følges hver enkelt linje, kan man se, at det passer med det "grafiske" kode i kodeeksempel 5.7. Der startes og sluttes i nederste højre hjørne, som er indikeret med tegnet &. Dette ses også i det kompilerede kode, ved at dronen flyttes to blokke op, hvorefter rækken bygges en efter en. Se evt. figur 5.1 for illustration af hvordan dronen bevæger sig, og placerer blokke i kodeeksempel 5.8.

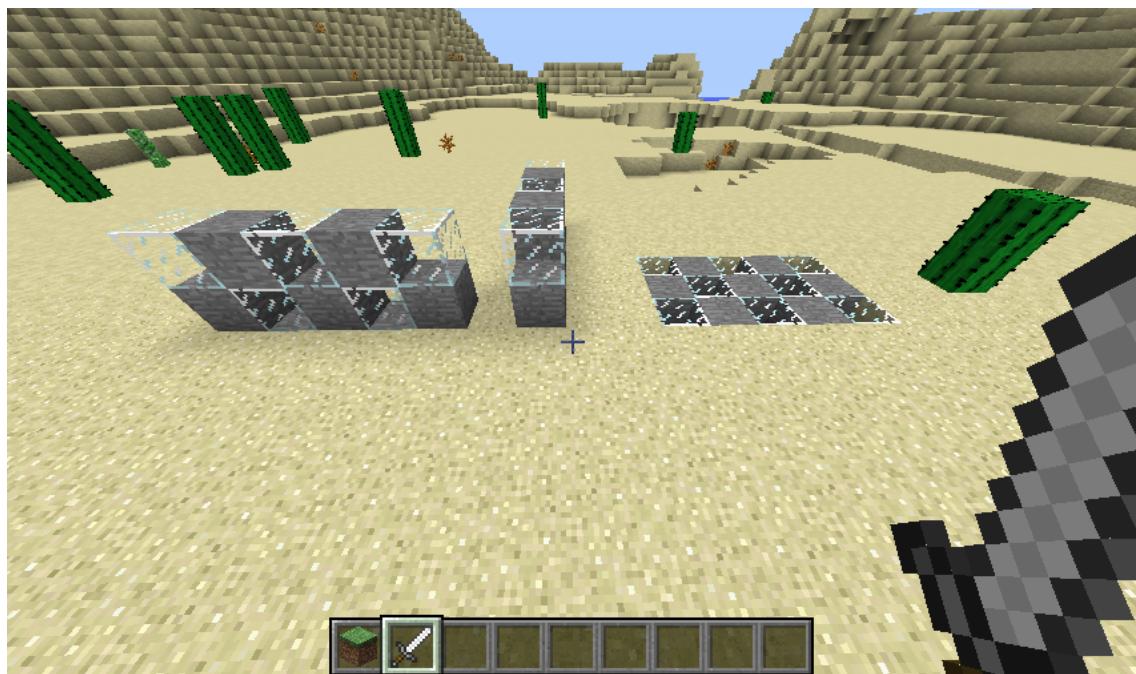


Figur 5.1. Illustration af hvordan dronen bevæger sig i kodeeksempel 5.8

For at teste orienteringsparametrene i keywordet `build()`, er der benyttet samme kode, som i kodeeksempel 5.7, dog med henholdsvis `front`, `side` og `top` som parametre i `build()`. Resultatet med forskellige parametre, kan ses på figur 5.2

Det ses ud fra figur 5.2, at orienteringsparametrene fungerer efter hensigt, ved at konstruktionen i Minecraft, placeres alt efter parametren.

Er der flere blokke med samme type placeret ved siden af hinanden i `build`-udtrykket som ses i kodeeksempel 5.9, benytter compileren optimering, se afsnit 4.6.4. Dette vil resultere i, at de to første linjers blokke i `build`, vil kompileres til én linje kode.



Figur 5.2. Kodeeksempel 5.7 med forskellig build() orienteringsparameter. Fra venstre mod højre ses med parameter: front, side og top

```

1 function Wall()
2   block & = STONE
3   block s = STONE
4
5   build(front)
6     s s s s s
7     s s s s s
8     & s s s s
9   end build
10 end function

```

Kodeeksempel 5.9. Et eksempel på et build-udtryk hvor der kun benyttes en type af **block** variabler

```

1 var Drone = require('drone');
2 Drone.extend('Wall', function()
3 {
4   var b1 = blockToNumber("STONE");
5   var s = blockToNumber("STONE");
6   // Build Function start:
7   this.up(2)
8   this.down(1)
9   this.box(s, 5, 2, 1)
10  this.down(1)
11  this.box(b1)
12  this.right(1)
13  this.box(s, 4, 1, 1)
14  this.left(1)
15 // Build Function ended
16 });

```

Kodeeksempel 5.10. Kompileret resultat af kodeeksempel 5.9

I kodeeksempel 5.9 linje 8, ses resultatet af optimeringen. Kaldet `this.box(s, 5, 2, 1)` giver en rektangel i Minecraft, som har målene $5 \times 2 \times 1$, som er et hurtigere kald, end at kalde hver enkel `block` hver for sig, samtidig med, at dronen skal flyttes for hver blok. Derudover kan det gøre den kompilerede kode mere overskuelig.

5.2 Fejlhåndtering

I dette afsnit undersøges og beskrives fejlhåndteringen som sker af henholdsvis parseren og type checkerden. Fejlhåndtering er til for at gøre det lettere for programmøren, at finde eventuelle fejl i deres kode.

Parser fejlhåndtering

Den fejlhåndtering der sker af parseren er syntaksfejl. Disse fejl kan være ved deklarering af variabler, som det ses i kodeeksempel 5.11, ved kontrolstrukturer og generelt igennem hele koden. Parseren vil ved hver fejl, give linjenummer og karakternummer, på hvor fejlen opstår, og angive hvad parseren forventer, at der skal skrives i stedet. Der vil nu komme nogle enkelte eksempler på syntaksfejl og parserens fejlhåndtering.

```
1 function ForkertDeklarering()
2     int a = 5
3     block aa = STONE
4 end function
```

Kodeeksempel 5.11. Fejl i deklarering af variabler

```
1 language.parser.ParserException: [2,6] expecting: word
```

Kodeeksempel 5.12. Fejlmeddeelse ved kompilering af kodeeksempel 5.11

I kodeeksempel 5.11 ses det, at både `block` og `integer` variablen er blevet deklareret forkert, da `block` variabelnavnet indeholder to karaktere, og `integer` variabelnavnet kun indeholder én karakter. Dette er ikke syntaktisk korrekt i VMP, og derfor vil parseren give en fejlmeddeelse, som det ses i kodeeksempel 5.12. Parseren returnerer kun en fejlmeddeelse på den første fejl den finder, derfor ses der kun én linje i fejlmeddeelsen. Derudover kan det ses, at parseren forventede et `word` på linje 3 karakter 6. Hvilket er korrekt i forhold til VMP, som kræver at integer variablene navne mindst indeholder to karaktere (`word` betyder et ord på to eller flere karakterer).

Hvis den boolske betingelse i et `if`-udtryk skiftes ud med et matematisk udtryk, som det ses i kodeeksempel 5.13, eller kun består af en enkelt integer variabel, vil parseren returnere en fejl, som det ses i kodeeksempel 5.14

```
1 function IfBetingelseIkkeBoolsk()
2     if(5 + 5)
3         //Krop
4     end if
5 end function
```

Kodeeksempel 5.13. `if`-udtryk hvor den boolske betingelse er erstattet

```
1 language.parser.ParserException: [2,7] expecting: '=', logicaloperator
```

Kodeeksempel 5.14. Fejlmeddeelse ved kompilering af kodeeksempel 5.13

Fejlmeddeelsen giver besked om, at der i linje 2, karakter 7 forventes en logisk operator. Der gives altså en fejlmeddeelse, når betingelsen i et `if`-udtryk ikke er et boolsk udtryk.

Benyttes der andet end en integer eller en expression som variabel i `repeat()`, som det ses i kodeksempel 5.15, vil parseren returnere en fejlmeddeelse som det ses i kodeksempel 5.16

```
1 function RepeatMedBlock()
2   repeat(a)
3     //Krop
4   end repeat
5 end function
```

Kodeksempel 5.15. `repeat()` med input variabel andet en integer

```
1 language.parser.ParserException: [2,9] expecting: '(', number, word
```

Kodeksempel 5.16. Fejlmeddeelse ved kompilering af kodeksempel 5.15

Fejlmeddeelsen giver besked om, at der i linje 2, karakter 9 forventes enten en start parentes, som en del af et matematisk udtryk, eller en integer i form af et nummer eller en variabel.

Parseren vil desuden opfange fejl som at `build`-udtrykket mangler orienteringsparameter, at en variabel, værende integer eller block, skal initialiseres, og ikke kun deklarereres.

Type checker fejlhåndtering

Type checker tager sig af de fejl, som ikke er syntaksfejl, men derimod ulovlig variabel håndtering. Derudover er det også muligt for fejlhåndteringen i type checker til at returnere alle fejl fra hele koden, og ikke kun den første fejl der stødes på, som i parseren. Et eksempel på en fejl type checkeren vil registrere, kan være noget såsom en dobbelt deklarering af en variabel, som det ses i kodeksempel 5.17.

```
1 function DobbeltIntDeklarering()
2   int aa = 4
3   int aa = 2
4 end function
```

Kodeksempel 5.17. Dobbelt deklarering af en integer variabel

```
1 ERROR: dobbelt declaration "aa" ..Line: 3..
2
3 Error count: 1
4 Compiling aborted
```

Kodeksempel 5.18. Fejlmeddeelse ved kompilering af kodeksempel 5.17

Kompileres kodeksempel 5.17, vil type checkeren returnere en fejlmeddeelse, som kan ses i kodeksempel 5.18. Ud fra fejlmeddeelsen kan det ses, at der bliver meddelt at der er en fejl, i form af en dobbelt deklarering, samt navnet på den variabel der er blevet deklareret flere gange, i dette tilfælde `aa`. Uover det bliver der givet et linje nummer, som fortæller hvor fejlen er, og til sidst angives der hvor mange fejl type checkeren i alt er stødt på, og at kompileringen stoppes. Samme fejlmeddeelse vil returneres af type checkeren, hvis det er et block variabelnavn der deklarereres flere gange. En lignende fejlmeddeelse vil returneres,

hvis der sker flere deklareringer af et funktionsnavn. I denne fejlmeddeelse vil der dog blive beskrevet, at det er en dobbelt funktionsdeklarering, samt informationen om linje og antal fejl i alt.

Type checker'en opfanger også, hvis der benyttes variabler og funktioner, som ikke er blevet deklareret. Et eksempel på hver af disse, kan ses i kodeeksempel 5.19, hvor fejlmeddeelsen kan ses i kodeeksempel 5.20.

```

1 function IkkeDeklareret()
2   int aa = bb
3   AndenFunktion()
4 end function

```

Kodeeksempel 5.19. Brug af ikke deklareret variabel og funktion

```

1 ERROR: "bb" integer is not declared! ..Line: 2..
2 ERROR: "AndenFunktion" Function is not declared! ..Line: 3..
3
4 Error count: 2
5 Compiling aborted

```

Kodeeksempel 5.20. Fejlmeddeelse ved kompilering af kodeeksempel 5.19

Det ses i fejlmeddeelsen, kodeeksempel 5.20, at type checker'en har fundet henholdsvis en integer variabel, 'bb', og en funktion, 'AndenFunktion', som ikke er deklareret i koden. Det ses også ud fra denne fejlmeddeelse, at type checker'en er i stand til, som nævnt, at meddele om mere end en fejl.

Type checker'en håndterer også fejl i `build`-udtrykket, angående *start* og *stop* positionen inde i udtrykket. Et eksempel på dette kan ses i kodeeksempel 5.21

```

1 function miniwallfront()
2   block s = STONE
3   build(front)
4     s s s s
5   end build
6 end function

```

Kodeeksempel 5.21. `build()`-udtryk uden *start* eller *stop* position

Kompileres kodeeksempel 5.21 vil type checker'en returnere en fejlmeddeelse, som den ses i kodeeksempel 5.22

```

1 ERROR: There must be a Start and End position. ..Line: 3..
2
3 Error count: 1
4 Compiling aborted

```

Kodeeksempel 5.22. Fejlmeddeelse ved kompilering af kodeeksempel 5.21

En lignende fejl, se kodeeksempel 5.23, vil blive returneret, hvis der benyttes for mange *start* eller *stop* positions variabler.

```

1 ERROR: There are 2 or more declared Start positions, only one is allowed. ..Line:
2   5..
3
4 Error count: 1
5 Compiling aborted

```

Kodeeksempel 5.23. Fejlmeddeelse ved kompilering af kode hvor der benyttes mere end en *start* eller *stop* position

Benyttes der parametre i funktioner, er det også type checker'en, der håndterer hvis typerne af parametrene er forkert. Et eksempel på kode, der benytter forkerte parametre i funktionskald, kan ses i kodeeksempel 5.24

```

1  function FunkMain()
2    FunkA(STONE)
3    FunkB(5)
4  end function
5
6  function FunkA(aa)
7    int bb = aa + 2
8  end function
9
10 function FunkB(b)
11   block & = STONE
12   build(front)
13   b b b &
14 end build
15 end function

```

Kodeeksempel 5.24. Funktionskald med forkerte parametre

Det kan ses i kodeeksempel 5.24, at henholdsvis **FunkA** bruger en integer variabel som parameter, og **FunkB** bruger en block variabel som parameter. Fejlen i dette kodeeksempel sker i funktionen **FunkMain**, hvor **FunkA** kaldes med en block variabel og **FunkB** kaldes med integer variabel. Fejlmeddelelsen som type checker'en returnerer ved denne fejl, kan ses i kodeeksempel 5.25.

```

1  ERROR: In functioncall: "FunkA", parameter number 1
2  Expected type: "Integer" but received: "Block Type" ..Line: 2..
3  ERROR: In functioncall: "FunkB", parameter number 1
4  Expected type: "Block variable" or "Block Type" but received: "Integer" ..Line:
   3..
5
6  Error count: 2
7  Compiling aborted

```

Kodeeksempel 5.25. Fejlmeddeelse ved kompilering af kodeeksempel 5.24

Som det ses i fejlmeddelelsen, kodeeksempel 5.25, opfanger type checker'en at der er fejl i funktionskaldene, og der gives besked om, hvilken type funktionen forventede, og hvilken type funktionen faktisk modtog.

5.3 Opsamling af test

Ved at gennemføre en whitebox test og teste fejlhåndteringen, er det sikret, at compileren oversætter og opfanger fejl korrekt og efter hensigten. Koden oversættes til det ækvivalente JavaScript og ScriptCraft kode, som kan benyttes af ScriptCraft plugin'et. Fejlhåndteringen sker i to faser, parser og type checker, hvor parseren opfanger syntaksfejl, og stopper videre indlæsning af kode. Derved kan den også kun returnere en enkelt fejlmeddeelse om én syntaktisk fejl, og angive hvilken linje den er på. Type checker'en

returnerer fejl, der er relateret til ulovlig håndtering af variabler og funktionskald, såsom dobbelt deklarering og kald af ikke deklarerede variabler og funktioner. Type checker'en kan returnere en fejlmeddeelse, der beskriver flere fejl i samme kode, og angiver linjenummer.

Opsamling 6

6.1 Konklusion

I dette afsnit vil der blive konkluderet på dette projekts problemformulering, for at finde ud af hvordan den endelige implementering opfylder de kriterier og mål, der blev opsat for projektets programmeringssprog inden implementationen.

Vi har designet et sprog ved navn VMP (Visual Minecraft Programming), og konstrueret en compiler der kan oversætte fra VMP til ScriptCraft. Da ScriptCraft kan bruges sammen med Minecraft til at lave konstruktioner, har dette gjort det muligt, at udnytte VMP til at lave konstruktioner i Minecraft. Dette er gjort ud fra følgende problemformulering:

Hvordan skal et børnevenligt programmeringssprog designes, så det er let anvendeligt til at konstruere både enkle og komplekse konstruktioner i spillet Minecraft?

Formålet med projektet har været at udvikle et programmeringssprog til Minecraft, som er sjovt og nemt at bruge. Under udviklingen af sproget, har vi fulgt principper fra andre programmeringssprog, som er udviklet med henblik på at lære børn at programmere. Vi kan dog ikke konkludere, om vores sprog er nemt eller sjovt for børn at bruge, da dette vil kræve en brugertest af sproget.

For at gøre sproget børnevenligt, har vi forsøgt at gøre sproget så enkelt som muligt, ved kun at inkludere helt basale udtryk som *if-kontrolstruktur* og *repeat-kontrolstruktur*, samt at begrænse os til to datatyper, *integer* og *block*. Disse dele modellerer Minecraft verdenen, så en programmør kan kode konstruktioner, som samtidigt også vil være mulige at lave i Minecraft.

Ved at bruge **build**-udtrykket, som vi har designet, kan man visuelt lave et lag af blokke, som kan sættes sammen med andre lag til at lave en større konstruktion lagvis. Dette giver et overblik over, hvad der laves og gør det derved nemmere at forestille sig, hvordan en konstruktion vil se ud i Minecraft, fremfor at se på koden i ScriptCraft. Dette er specielt gældende for store konstruktioner, som kan være uoverskelige i ScriptCraft, men kan være nemmere at visualisere i sammenhæng med VMP.

Læsbarhed har været vigtigt i dette projekt, da det netop er vigtigt at kunne skabe sig et overblik over koden, for at kunne forestille sig hvordan konstruktioner vil se ud i Minecraft.

For at opnå høj læsbarhed, er der ændret på de logiske operatorer, så de bliver læselige, og matematiske operatorer som børn normalt bruger i logiske og aritmetiske sammenhænge.

Skrivbarhed har ikke været så højt prioriteret, da programmerne som skal skrives i VMP, sandsynligvis ikke vil være så store. Dog er der blevet implementeret to forskellige *build*-udtryk, som hver især kan være fordelagtige at benytte i specifikke situationer. Eksempelvis til meget små konstruktioner, kan det være lettere at skrive med det lille **build**-udtryk (det som kun kan bygge en blok af gangen). Der er derfor to forskellige *build*-udtryk, udelukkende for at øge skrivbarheden.

Overordnet synes vi, at det er lykkedes os at konstruere et fuldendt sprog med gode brugsmuligheder, og at vi har opnået vores sprogrriterier tilfredsstillende.

6.2 Fremtidigt arbejde

I dette afsnit vil der blive beskrevet nogle af de forbedringer, og eventuel videreudvikling, der kunne være relevante eller interessante at arbejde med.

På nuværende tidspunkt er det ikke muligt at anvende samtlige bloktyper der er tilgængelige i Minecraft i VMP, hvilket kunne være relevant at tilføje. Det betyder dog ikke noget for sprogets funktionalitet, men vil give større fleksibilitet til brugeren.

Det kunne også være relevant at tilføje prædefinerede funktioner, som kunne gøre det muligt at konstruere nogle forudbestemte ting. Eksempelvis kugler, kasser, pyramider og prismaer. Derudover kunne der også tilføjes en mulighed for at øge ens **build**-udtryk med en bestemt faktor. F.eks. kunne man skrive `2x` foran, og så ville konstruktionen blive skaleret. Dette ville øge skrivbarheden, af bestemte konstruktioner, da det ikke ville være nødvendigt at skrive så meget i **build**-udtrykket. Efter en fremtidig brugertest, vil der sandsynligvis være flere ændringer af sproget som skal fortages.

Scratch anvendte muligheden for at benytte grafik som syntaks til programmering, og tilføjelse af dette til VMP kunne være en god mulighed for at gøre sproget nemmere at bruge.

For at gøre den grafiske brugergrænseflade yderligere anvendelig, kunne der tilføjes en liste af de bloktyper der er mulige at bruge. Dette ville gøre det lettere at huske, og skabe overblik over de bloktyper der er mulige at benytte.

Bibliografi

- [1] Geodatastyrelsen. *Danmarks frie geodata i en Minecraft-verden*. URL: <http://gst.dk/emner/frie-data/minecraft/>.
- [2] Mojang AB. *What is Minecraft?* URL: <https://minecraft.net/game>.
- [3] Gamepedia. *The Overworld*. URL: http://minecraft.gamepedia.com/The_Overworld.
- [4] Fætter BR. *MINECRAFT – Byg. Skræddersy. Socialiser.* URL: <http://www.br.dk/maerker/minecraft>.
- [5] Wikipedia. *Code.org*. URL: <http://en.wikipedia.org/wiki/Code.org>.
- [6] Amy Schatz. *Obama: Everybody's Got to Learn How to Code*. URL: <http://recode.net/2015/02/14/obama-everybodys-got-to-learn-how-to-code/>.
- [7] Stuart Dredge. *Coding at school: a parent's guide to England's new computing curriculum*. 2014. URL: <http://www.theguardian.com/technology/2014/sep/04/coding-school-computing-children-programming>.
- [8] Mikkel Wenzel Andreasen. *Nu skal programmering ind i folkeskolen*. 2014. URL: <http://www.computerworld.dk/art/231742/nu-skal-programmering-ind-i-folkeskolen>.
- [9] Natasha Friis Saxberg. *Programmering er fremtidens globale sprog*. URL: <https://www.mm.dk/blog/programmering-er-fremtidens-globale-sprog>.
- [10] Wikipedia. *List of educational programming languages*. URL: http://en.wikipedia.org/wiki/List_of_educational_programming_languages#Children.
- [11] CodeMonkey. *CodeMonkey home page*. URL: <http://www.playcodemonkey.com/>.
- [12] CodeMonkey. *CodeMonkey FAQ*. URL: <http://www.playcodemonkey.com/faqs/>.
- [13] CoffeeScript. *CoffeeScript home page*. URL: <http://coffeescript.org/>.
- [14] Scratch. *Scratch - For Parents*. URL: <http://scratch.mit.edu/parents/>.
- [15] Mitchel Resnick & John Maloney & Andrés Monroy Hernández & Natalie Rusk & Evelyn Eastmond & Karen Brennan & Amon Millner & Eric Rosenbaum & Jay Silver & Brian Silverman & Yasmin Kafai. *Scratch: Programming for All*. URL: <http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf>.
- [16] Telegraph Jessica Saulter. *Coding for kids - School children learn computer programming*. URL: <http://www.telegraph.co.uk/technology/10468460/Coding-for-kids-schoolchildren-learn-computer-pr%5C&ramming.html>.

- [17] Scratch. *Learning with Scratch*. URL: <https://llk.media.mit.edu/scratch/Learning-with-Scratch.pdf>.
- [18] Sabdra Akdaba. *Snakk Basic 1.0 is here!* URL: <http://blogs.msdn.com/b/smallbasic/archive/2011/07/12/small-basic-1-0-is-here.aspx>.
- [19] Philip Conrod & Lou Tylee. *Beginning Microsoft Small Basic*. URL: <http://social.technet.microsoft.com/wiki/contents/articles/16766.beginning-microsoft-small-basic.aspx>.
- [20] SmallBasic. *Small Basic home page*. URL: <http://smallbasic.com/>.
- [21] Wikipedia. *RoboMind*. URL: <http://en.wikipedia.org/wiki/RoboMind>.
- [22] RoboMind. *RoboMind introduction*. URL: <http://www.robomind.net/en/introduction.htm>.
- [23] Freewaregenius. *Robomind syntax screenshot*. URL: <http://cdn2.freewaregenius.com/wp-content/uploads/2013/03/Robomind-Screenshot1.jpg>.
- [24] Robert W. Sebesta. *Concepts of Programming Languages*. 2013.
- [25] Wikipedia. *Type system*. 2015. URL: http://en.wikipedia.org/wiki/Type_system#Type_checking.
- [26] Walter Higgins. *ScriptCraft*. URL: <http://scriptcraftjs.org/>.
- [27] Walter Higgins. *A young persons guide to programmin in minecraft*. URL: <https://github.com/walterhiggins/ScriptCraft/blob/master/docs/YoungPersonsGuideToProgrammingMinecraft.md>.
- [28] Walter Higgins. *ScriptCraft API Reference*. URL: <https://github.com/walterhiggins/ScriptCraft/blob/master/docs/API-Reference.md>.
- [29] W3Schools. *JavaScript Function Parameters*. URL: http://www.w3schools.com/js/js_function_parameters.asp.
- [30] Jr. Charles N Fischer Ron K. Cytron og Richard J. Leblanc. *Crafting a Compiler*. 2009.
- [31] Andrew W. Appel. *Modern Compiler Implementation in ML*, Cambridge University Press. URL: <http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf>.
- [32] The International Organization for Standardization. *EBNF standard*. URL: <http://standards.iso.org/ittf/licence.html>.
- [33] Wikipedia. *Bottom-up parsing*. URL: http://en.wikipedia.org/wiki/Bottom-up_parsing.
- [34] Ander Møller. *dk.brics.grammar*. URL: <http://www.brics.dk/grammar/>.
- [35] Dr. Axel Rauschmayer. *Programming language variables: scope and extent*. 2015. URL: <http://www.2ality.com/2011/05/programming-language-variables-scope.html>.
- [36] Amruth N. Kumar. *Scope in programming languages*. 2015. URL: <http://phobos.ramapo.edu/~amruth/grants/problets/courseware/scope/home.html>.
- [37] Wikipedia. *Scope levels*. 2015. URL: [http://en.wikipedia.org/wiki/Scope_\(computer_science\)#Expression_scope](http://en.wikipedia.org/wiki/Scope_(computer_science)#Expression_scope).

- [38] Tony Zhang. *Hour 14 - Scope and Storage Classes in C*. 2015. URL: <http://aelinik.free.fr/c/ch14.htm>.
- [39] Hans Hüttel. *Transitions and Trees*. 2010.
- [40] Étienne Gagnon. *SableCC Thesis*. URL: <http://sablecc.sourceforge.net/downloads/thesis.pdf>.
- [41] w3schools. *JavaScript Tutorial*. URL: <http://www.w3schools.com/js/default.asp>.

Bottom up, rightmost derivation

A

```
1 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
2 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build" "up(1)"
3 "end repeat" "end function"
4
5 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
6 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build"
7 "up""("number")" "end repeat" "end function"
8
9 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
10 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build"
11 "up""("value")" "end repeat" "end function"
12
13 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
14 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build"
15 direction"("value")" "end repeat" "end function"
16
17 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
18 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build"
19 move "end repeat" "end function"
20
21 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
22 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" "b" "end build"
23 processes "end repeat" "end function"
24
25 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
26 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" blockID "end build"
27 "end repeat" "end function"
28
29 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
30 "repeat""("2")" "build""("top")" "b" "b" "/n" "&" block "end build"
31 "end repeat" "end function"
32
33 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
34 "repeat""("2")" "build""("top")" "b" "b" "/n" blockID block "end build"
35 "end repeat" "end function"
36
37 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
38 "repeat""("2")" "build""("top")" "b" "b" "/n" block block "end build"
39 "end repeat" "end function"
40
41 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
42 "repeat""("2")" "build""("top")" "b" "b" block block block "end build"
43 "end repeat" "end function"
44
45 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
46 "repeat""("2")" "build""("top")" "b" blockID block block block "end build"
```

```

48 "end repeat" "end function"
49
50 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
51 "repeat""("2")" "build""("top")" "b" block block block block "end build"
52 "end repeat" "end function"
53
54 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
55 "repeat""("2")" "build""("top")" blockID block block block block "end build"
56 "end repeat" "end function"
57
58 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
59 "repeat""("2")" "build""("top")" block block block block block "end build"
60 "end repeat" "end function"
61
62 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
63 "repeat""("2")" "build""("orientation")" block block block block block "end build"
64 "
65 "end repeat" "end function"
66
66 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
67 "repeat""("2")" build "end repeat" "end function"
68
69 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
70 "repeat""("2")" processes "end repeat" "end function"
71
72 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
73 "repeat""("2")" "end repeat" "end function"
74
75 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
76 "repeat""(number)" "end repeat" "end function"
77
78 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
79 "repeat""(value)" "end repeat" "end function"
80
81 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
82 control "end function"
83
84 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
85 processes "end function"
86
87 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" "STONE"
88 "end function"
89
90 "function" "box""(" ")" "block" "b" "=" "STONE" "block" "&" "=" blockType
91 "end function"
92
93 "function" "box""(" ")" "block" "b" "=" "STONE" "block" blockID "=" blockType
94 "end function"
95
96 "function" "box""(" ")" "block" "b" "=" "STONE" declaration "end function"
97
98 "function" "box""(" ")" "block" "b" "=" "STONE" processes "end function"
99
100 "function" "box""(" ")" "block" "b" "=" "STONE" "end function"
101
102 "function" "box""(" ")" "block" "b" "=" blockType "end function"
103
104 "function" "box""(" ")" "block" blockID "=" blockType "end function"
105
106 "function" "box""(" ")" declaration "end function"
107
108 "function" "box""(" ")" processes "end function"
109
110 "function" "box""(" ")" "end function"
111
112 "function" "box""("parameters")" "end function"

```

```
113 "function" word("parameters")" "end function"
114
115
116 program
```

SableCC grammar B

```
1 Package language;
2
3 Helpers
4     digit = ([‘1’ .. ‘9’] [‘0’ .. ‘9’]* | ‘0’);
5     alphabet = ([‘a’ .. ‘z’] | [‘A’ .. ‘Z’]));
6
7 States
8     normal, prep, build;
9
10 Tokens
11     int_keyword = ‘int’;
12     block_keyword = ‘block’;
13     assignment_keyword = ‘=’;
14     letter = alphabet;
15     blocktype = (‘AIR’ | ‘STONE’ | ‘GLASS’ | ‘GRASS’ | ‘DIRT’ | ‘BEDROCK’ | ‘WATER’
16         | ‘LAVA’ | ‘SAND’ | ‘GRAVEL’ | ‘GOLDORE’ | ‘IRONORE’ | ‘COALORE’ | ‘LOG’
17         | ‘LEAVES’ | ‘LAPISORE’ | ‘SANDSTONE’ | ‘DIAMONDORE’ | ‘REDSTONEORE’ | ‘
18         ICE’ | ‘SNOW’ | ‘CACTUS’ | ‘CLAY’ | ‘PUMPKIN’ | ‘BROWNMUSHROOMBLOCK’ | ‘
19         REDMUSHROOMBLOCK’ | ‘MELONBLOCK’ | ‘MYCELIUM’ | ‘COCOA’ | ‘EMERALDORE’ | ‘
20         HARDCLAY’ | ‘LEAVES2’ | ‘LOG2’ | ‘STAINEDCLAY’ | ‘PACKEDICE’ | ‘
21         COBBLESTONE’ | ‘PLANKS’ | ‘SPONGE’ | ‘DISPENSER’ | ‘STICKYPISTON’ | ‘WOOL’
22         | ‘GOLDBLOCK’ | ‘TNT’ | ‘BOOKSHELF’ | ‘MOSSYCOBBLESTONE’ | ‘TORCH’ | ‘
23         MOBSPAWNER’ | ‘OAKSTAIRS’ | ‘CHEST’ | ‘REDSTONEWIRE’ | ‘CRAFTINGTABLE’ | ‘
24         FARMLAND’ | ‘FURNACE’ | ‘STONESTAIRS’ | ‘FENCE’ | ‘MONSTEREGG’ | ‘
25         STONEBRICK’ | ‘IRONBARS’ | ‘GLASSPANE’ | ‘CAULDRON’ | ‘SANDSTONESTAIRS’ | ‘
26         FLOWERPOT’ | ‘PRISMARINE’ | ‘SEALANTERN’ | ‘NETHERRACK’ | ‘SOULSAND’ | ‘
27         GLOWSTONE’ | ‘QUARTZORE’ | ‘OBSIDIAN’);
28     operator = (+ | * | / | %);
29     orientation = (‘front’ | ‘top’ | ‘side’);
30     s_paran = ‘(’;
31     e_paran = ‘)’;
32     ctrl_blocks = (&|!|?);
33     {normal->prep, prep}build_keyword = ‘build’;
34     {build->normal, normal}endbuild_keyword = ‘end build’;
35     logicaloperator = (‘<=’ | ‘>=’ | ‘<’ | ‘>’ | ‘/=’);
36     and_keyword = ‘AND’;
37     or_keyword = ‘OR’;
38     if_keyword = ‘if’;
39     endif_keyword = ‘end if’;
40     repeat_keyword = ‘repeat’;
41     endrepeat_keyword = ‘end repeat’;
42     elseif_keyword = ‘else-if’;
43     minus = ‘-’;
44     comma_keyword = ‘,’;
45     number = digit;
46     function_keyword = ‘function’;
47     endfunction_keyword = ‘end function’;
48     direction = (‘forward’ | ‘back’ | ‘up’ | ‘down’ | ‘left’ | ‘right’);
49     else_keyword = ‘else’;
```

```

38     endelse_keyword = 'end else';
39     word = alphabet++;
40     blank = (' ' | '\t' | '\n') 13;
41     nl = 10;
42     nl_replacer = '$';
43
44
45 Ignored Tokens
46     blank, nl;
47
48 Productions
49     program = function*;
50
51     function = functionstart statementsdcl* endfunction_keyword;
52
53     functionstart = function_keyword word s_paran formalparameters? e_paran;
54
55     statementsdcl = {declare} declaration
56         |{control} control
57         |{build} build
58         |{move} move
59         |{function} functioncall
60         |{assign} assignment;
61
62     statements ={control} control
63         |{build} build
64         |{move} move
65         |{function} functioncall
66         |{assign} assignment;
67
68     functioncall = word s_paran actualparameters? e_paran;
69
70     formalparameters ={param} formalparameter
71         |{params} formalparameter comma_keyword formalparameters;
72
73     formalparameter ={int} intid
74         |{block} blockid;
75
76     actualparameters ={param} actualparameter
77         |{params} actualparameter comma_keyword actualparameters;
78
79     actualparameter ={int} value
80         |{blocktype} blocktype
81         |{blockid} blockid;
82
83     declaration = {int} int_keyword intid assignment_keyword expressions
84         |{block} block_keyword blockid assignment_keyword blocktype;
85
86     assignment = {int} intid assignment_keyword expressions
87         |{block} blockid assignment_keyword blocktype;
88
89     expressions ={multi} expression operator expressions
90         |{minus} expression minus expressions
91         |{single} expression;
92
93     expression = {val} value
94         |{paran}s_paran expressions e_paran;
95
96     move = direction s_paran expressions e_paran;
97
98     control = {if} ifstart statements* endif_keyword elseif* else?
99         |{repeat} repeatstart statements* endrepeat_keyword;
100
101    ifstart = if_keyword s_paran logicalexpressions e_paran;
102
103    else = else_keyword statements* endelse_keyword;

```

```
104     elseif = elseifstart statements* endif_keyword;
105
106     elseifstart = elseif_keyword s_paran logicalexpressions e_paran;
107
108     repeatstart = repeat_keyword s_paran expressions e_paran;
109
110     value = {number} number
111         |{id} intid;
112
113     intid = word;
114
115     blockid ={letter} letter
116         |{ctrl}ctrl_blocks;
117
118
119     logicalexpressions = {and}logicalexpression and_keyword logicalexpressions
120         |{or}logicalexpression or_keyword logicalexpressions
121         |{single}logicalexpression;
122
123     logicalexpression ={others} [first]:expression logicaloperator [follow]:
124         expression
125             |{equals}[first]:expression assignment_keyword [follow]:expression
126             |{paran} s_paran logicalexpressions e_paran;
127
128     build = {blocks} build_keyword s_paran orientation e_paran block*
129         endbuild_keyword
130             |{block} build_keyword letter;
131
132     block = {empty} minus
133         |{ctrl} ctrl_blocks
134         |{letter} letter
135         |{nlreplace} nl_replacer;
```

Vejledning i at bruge VMP



Dette afsnit indeholder en tutorial for programmering i projektets programmeringssprog VMP. Tutorialen indeholder en gennemgang af den viden der er krævet, for at kunne anvende dette programmeringssprog, og kodeeksempler samt billeder fra Minecraft af forskellige konstruktioner, der er konstrueret ved anvendelse af programmeringssproget.

C.0.1 Krav

For at anvende VMP til at bygge ting, er det et krav at have adgang til følgende:

- **Tekst editor** til at udforme koden i VMP. Evt. Eclipse IDE, Notepad++ eller compilerens egen tekst editor kan benyttes.
- **Sprogets compiler** som bruges til at compile programmørens kode, til kode der kan læses af ScriptCraft.
- **ScriptCraft** plugin til Minecraft, som sørger for selve konstruktionen efter programmørens instruktioner i kode.
- **Minecraft** spillet, til fremvisning af de endelige konstruktioner.

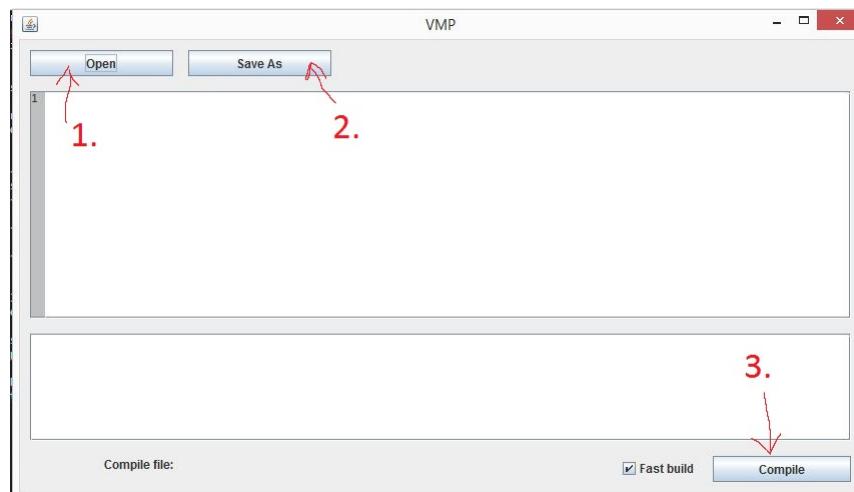
Denne tutorial forudsætter at ScriptCraft er opsat korrekt med Minecraft, og denne del vil derfor ikke blive gennemgået. Der henvises i stedet til Scriptcrafts egen tutorial.[27]

Valget af tekst editor har ingen betydning for kodning i VMP, og det er en mulighed at kode i compileren, og kompilere koden gennem samme program.

C.0.2 Gennemgang af VMP.jar

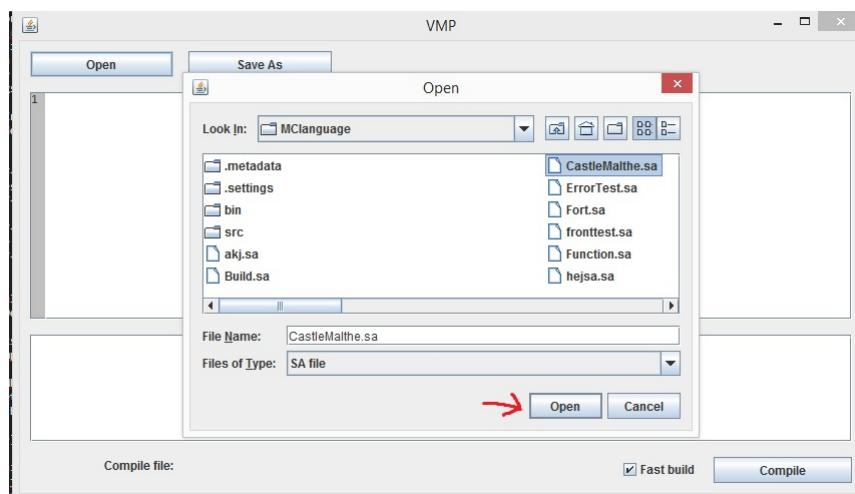
Programmet VMP.jar indeholder selve compileren for VMP til ScriptCraft kode, samt en brugergrænseflade der indeholder funktionalitet der understøtter kompileringsprocessen. På figur C.1 ses VMP.jar programmet efter det er blevet eksekveret.

Figuren indeholder tre knapper. Knappen ”Open” som pilen nummer 1. henviser til, på figuren, åbner et vindue som bruges til at lede efter den fil som skal kompileres af VMP compileren. På figur C.2 ses det nye vindue, hvor den fil der skal kodes i, eller kompileres,



Figur C.1. VMP.jar opstart

kan vælges. Når filen er valgt, som i dette tilfælde med CastleMalthe.sa, trykkes der på knappen ”Open” udpeget af den røde pil.

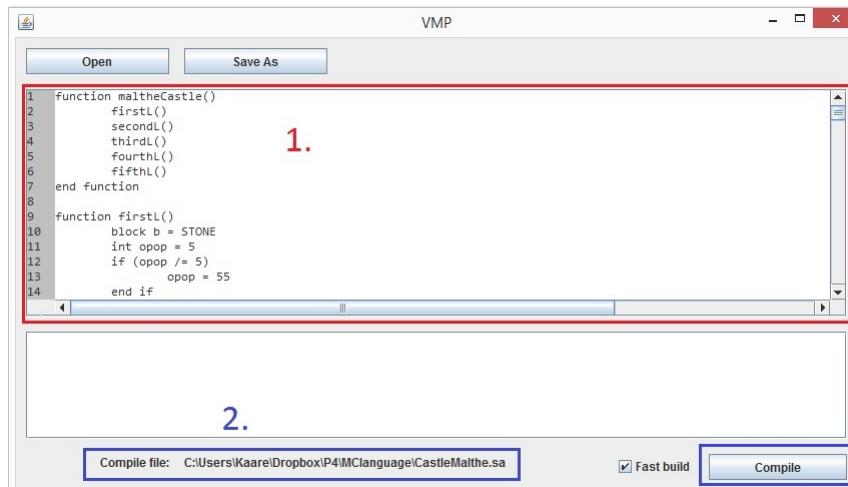


Figur C.2. VMP.jar åben en fil

Når den valgte fil er blevet åbnet, tilføjes indholdet af filen til programmet, og stien til filen tilføjes til compile file, som viser hvilken fil programmets compiler kompilerer, hvis der trykkes på knappen ”Compile”. Se figur C.3, hvor filen CastleMalthe.sa er åbnet, og indholdet tilføjet til programmet.

I den røde kasse 1., ses der indholdet af den åbnede fil CastleMalthe.sa i ren tekst. Den grå vertikale linje, der indeholder tal, i venstre side af den røde kasse, angiver linjetal for indholdet i CastleMalthe.sa filen. Linjetallene bruges senere, under kompileringsprocessen, til at angive linjenummeret for stedet hvor der er en fejl i koden, som gør at koden ikke kan blive kompileret af VMP compileren.

I den venstre blå kasse 2., ses stien til den valgte CastleMalthe.sa fil. Denne fil er den åbnede fil, og dermed defineret som filen der skal kompileres af VMP compileren. For at



Figur C.3. VMP.jar åbnet fil

kompile den valgte fil, trykkes der på knappen ”Compile”, som vist ved den 3. pil på figur C.1.

Til venstre for ”Compile” knappen, er der en checkboks for ”Fast build”. Denne checkboks gør det muligt at fortælle VMP compileren om det kompiledede kode skal kunne bygge en konstruktion i Minecraft hurtigt eller langtsomt, alt efter om spilleren skal kunne se konstruktionen blive bygget blok for blok, eller om konstruktionen skal bygges meget hurtigt.

Hvis der tilføjes ændringer til koden i en åbnet fil, skal disse gemmes til filen før den kan blive kompileret. Knappen ”Save as”, som vist ved den 2. pil på figur C.1, anvendes, hvor der så vil komme et vindue lignede det på figur C.2, hvor den åbnede fil vælges, og derefter gemmes ændringerne ved at trykke på knappen ”Save”, som har erstattet knappen ”Open” på figuren C.2.

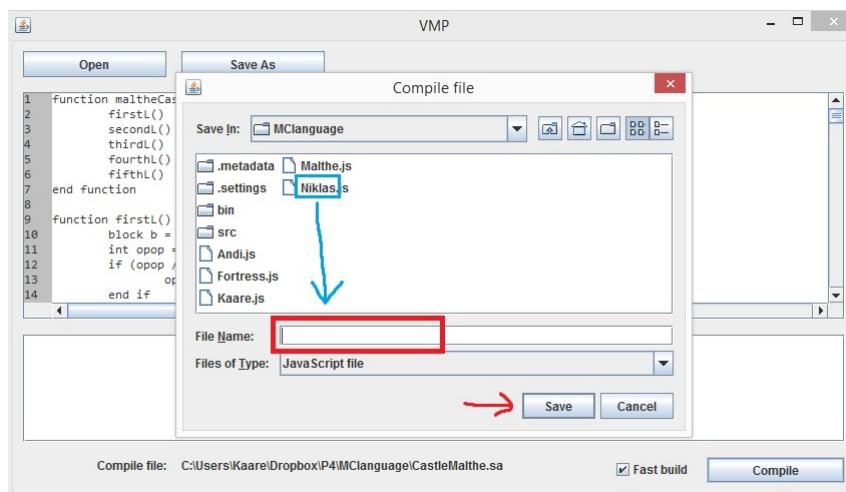
Ud over knappen ”Save as”, kan også knappen ”Compile” anvendes, selv om ændringerne tilføjet til koden ikke er gemt til den valgte fil. Hvis dette er tilfældet, kommer der en pop-up besked, som vist på figur C.4, som fortæller at filen skal gemmes før den kan kompileres.



Figur C.4. VMP.jar compile gem fil popup

Hvis knappen ”No” vælges, lukkes pop-up vinduet, og programmet går tilbage til vinduet vist på figur C.3. Hvis knappen ”Yes” vælges, gennemføres den samme proces, som hvis knappen ”Save as” var valgt, dog hvor programmet selv gemmer filen. Nu hvor filen er gemt, vil ”Compile” knappen bruges til kompilering.

Hvis der trykkes på ”Compile” knappen, åbnes der et nyt vindue som vist på figur C.5.

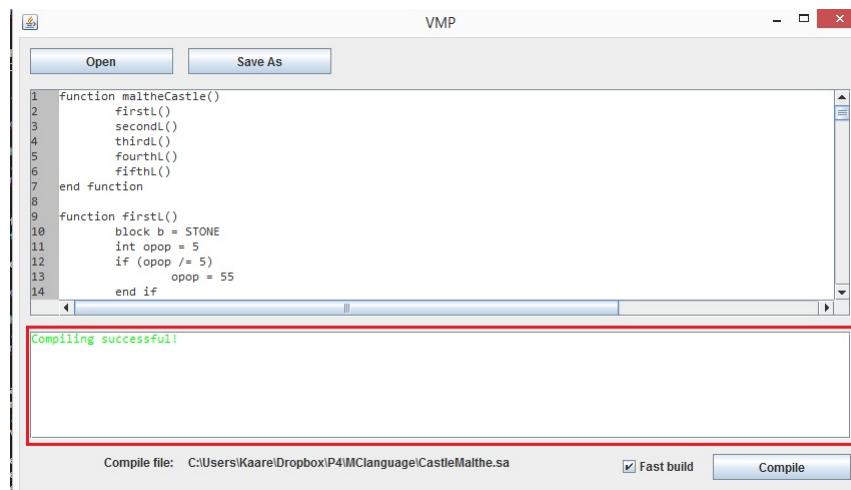


Figur C.5. VMP.jar compile vindue

Når vinduet er blevet åbnet, skal der vælges en mappe, hvori filen med det kompilerede VMP kode skal gemmes. Den optimale sti, hvor filen burde gemmes, er mappen ”plugins” under ScriptCraft. Denne mappe indeholder alle filer med kode der kan bruges af ScriptCraft, til at bygge programmerede konstruktioner i Minecraft.

Den røde boks på figur C.5 viser stedet hvor navngivning, af den kompilerede fil, finder sted. Den blå boks omkranser et eksempel på et muligt filnavn. Når en mappe, og et filnavn, er valgt, trykkes der på knappen ”Save”, som henvist ved den røde pil.

Når filen i dette eksempel, CastleMalthe.sa, kompileres, vælges navnet CastleMalthe, og knappen ”save” trykkes. På figur C.6 ses programmets tilstand efter filen er kompileret.



Figur C.6. VMP.jar kompileret fil

Den røde boks omkranser succes/fejl feltet, som angiver de eventuelle fejl koden i den åbnede fil indeholder. Hvis compileren ikke finder kodefejl i den åbnede fil, vil kompileringen lykkes, og beskeden ”Compiling successful!” bliver fremvist.

Dermed er gennemgangen af VMP.jar overstået, og der vil nu komme en tutorial for selve programmeringssproget, hvor der er fokus på selve koden.

C.0.3 Kodning

Tekst editoren bruges til at frembringe kode, der senere kan bruges til at konstruere konstruktioner i Minecraft ved anvendelse af kald til funktioner understøttet af ScriptCraft. Disse funktioner udformes i tekst editoren, og compiles derefter så de kan læses af Scriptcraft. Kodningsafsnittet vil fokusere på eksempler af kode udformet i VMP i en tekst editor, med kort beskrivelse af de indbyggede elementer der er til rådighed i VMP. Regler for kode i VMP, samt dybere beskrivelse af elementerne tilgængelige i VMP, er gennemgået i afsnit C.0.4.

Kodeeksempel C.1 viser hvordan en funktion der laver en trappe kan udformes i VMP.

```

1 function stair(length)
2   block ! = STONE
3   block ? = STONE
4   block b = STONE
5   repeat(length)
6     build(side)
7       -
8       ?
9       !
10      b
11    end build
12  end repeat
13 end function

```

Kodeeksempel C.1. Kodeeksempel for en trappe

Funktionen **stair** fungerer som en ”byggemanual” for det ScriptCraft plugin der bygger selve konstruktionerne, der er kodet i VMP, inde i Minecraft. Hvis manualen følger de definerede krav til kode i VMP, vil compileren oversætte koden. Funktionen i eksempel C.1 indeholder kode for en trappekonstruktion. Funktionen bruger en parameter **length**, som angiver at når funktionen anvendes inde i Minecraft gennem ScriptCraft, skal der der indtastes en integer værdi mellem parenteserne.

Hvis f.eks. kommandoen `/js stair(10)` udføres gennem ScriptCraft i Minecraft, vil der blive bygget en konstruktion som vist på figur C.7, hvor **length** angiver antal trappetrin.

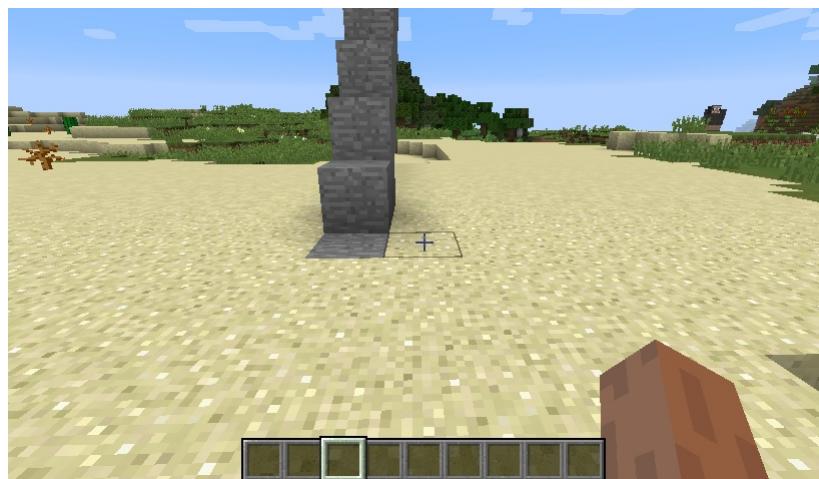
Som det ses på figuren, er trappekonstruktionen bygget i en bestemt retning ud fra hvor spilleren står. Når **build**-udtrykket anvendes i funktionen **stair()** er der angivet en parameter **side** til udtrykket, og denne parameter angiver hvordan ScriptCraft skal bygge konstruktionen, i forhold til spilleren der anvender funktionen, inde i Minecraft. Orientationsparametren, som **side** parametren kaldes, gennemgås mere dybdegående i afsnit C.0.4.

Kodeeksempel C.1 viser en simpel lille konstruktion, men det er også muligt at lave meget større, og mere gennemtænkte, konstruktioner i VMP. Kodeeksempel C.2 viser et eksempel på en skyskraber.

```

1 function skyscraper(floors)
2   block b = PLANKS
3   block & = PLANKS
4   block g = GLASS

```



Figur C.7. Minecraft: stair() funktion

```

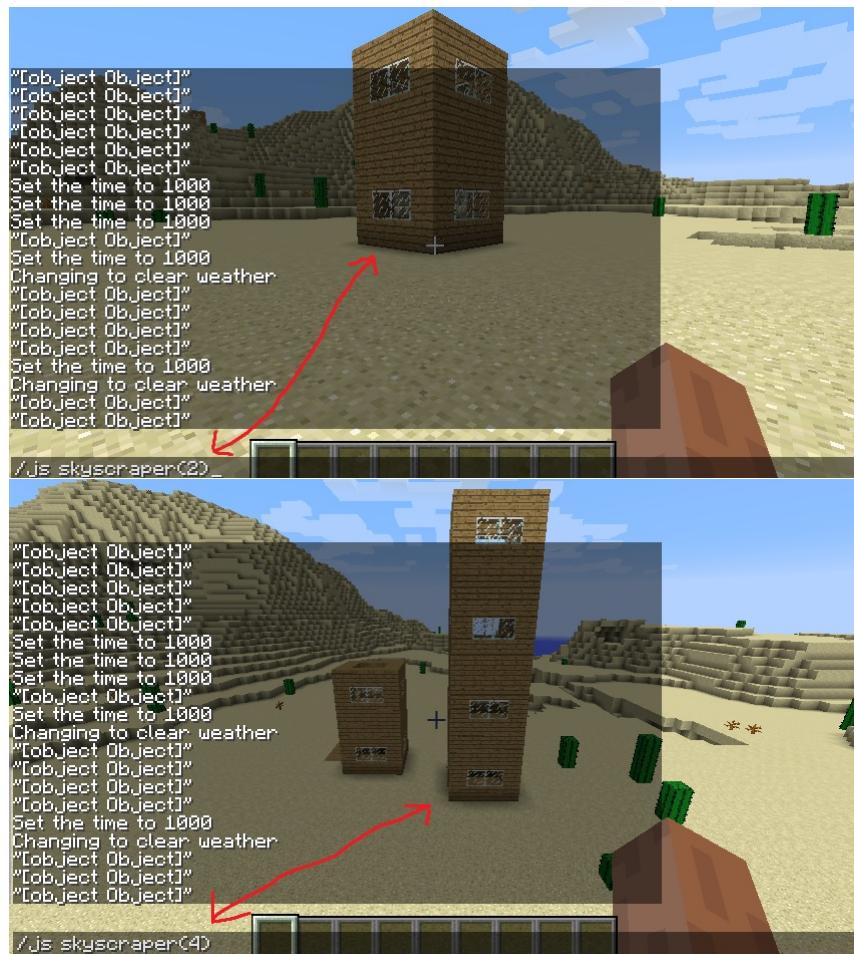
5   int level = 0
6   repeat(floors)
7     repeat(4)
8       if(level == 0)
9         build(top)
10        & b b b
11        b b b b
12        b b b b
13        b b b b
14      end build
15      level = level + 1
16    end if
17    else-if(level == 1 OR level == 3)
18      build(top)
19        & b b b
20        b - - b
21        b - - b
22        b b b b
23      end build
24      if(level == 1)
25        level = level + 1
26      end if
27    else
28      level = 0
29    end else
30  end if
31  else-if(level == 2)
32    build(top)
33      & g g b
34      g - - g
35      g - - g
36      b g g b
37    end build
38    level = level + 1
39  end if
40  up(1)
41 end repeat
42 end repeat
end function

```

Kodeeksempel C.2. Kodeeksempel for en skyskraber

I dette kodeeksempel har programmøren anvendt if-kontrolstrukturen til at angive

forskellige lag af skyskraberen, hvor én etage af skyskraberen består af *fire* lag bestemt ud fra koden `repeat(4)....end repeat`. For hver gennemgang af den inderste `repeat`-løkke, tjekkes der med `if`-kontrolstrukturen hvilken del af etagen der skal bygges. Hvis etagen lige er startet, har variablen `level` integer værdien 0, og efter hver del af etagen er bygget færdig, optælles værdien af `level` med 1, og næste del af etagen bygges. Dette fortsætter indtil 4 dele er bygget, og en hel etage er bygget, og der startes forfra indtil alle etager er færdige med at blive bygget. På figur C.8 ses konstruktionen for kodeeksempel C.2, for henholdsvis 2 og 4 etager.



Figur C.8. Minecraft: skyscraper(floors) funktion

I stedet for at bruge `if`-kontrolstrukturen hver gang der skal kontrolleres hvilken del der skal bygges, kan der bruges funktioner indkapslet af én funktion. Figurerne i bilag D viser funktionen `maltheCastle()` defineret som bestående af 5 andre funktioner. Disse funktioner fungerer på samme måde som i eksemplet med skyskraberen, hvor der defineres ét lag der skal bygges ad gangen, men fordi der anvendes funktioner, i stedet for `if`-kontrolstrukturen, er det muligt for spillere i Minecraft at udføre kald til hver indre funktion i ScriptCraft, så de selv kan designe et slot ud fra de definerede lag af et slot, som er henholdsvis `firstL`, `secondL`, `thirdL`, `fourthL` og `fifthL`. Figuren D.8 viser konstruktionen i Minecraft for `maltheCastle()`.

C.0.4 Elementer i VMP

De forskellige elementer tilgængelige for programmører i VMP vil blive gennemgået i dette afsnit. VMP indeholder følgende emner over elementer, som vil blive beskrevet:

- **Typer, værdier og variable**
- **Operatorer**
- **Function**
- **build**
- **Kontrolstrukturer**

Der vil nu komme en gennemgang af disse emner. **Typer, værdier og variable** inkluderer en beskrivelse over typerne, integer og block, som er i VMP samt de mulige værdier der kan tildeles variable af disse typer.

Variable af typen integer kan tage imod heltalsværdier, og integer variables navne skal være 2 eller flere karakterer langt. Variable af typen block kan tage imod en prædefineret liste af blokke som værdier, og variabelnavnet må kun være på én karakter. Listen af disse værdier kan ses under listen Value(blocktypes) i afsnit 3.1.6. I eksempel C.3 vises der hvordan tildeling af typer og værdier til variable fungerer.

```

1 function assignVars()
2   block a = STONE //foerste gang en variabel deklarereres, skal den tildeles en
3     vaerdi
4   block b = PLANK
5   int aa = 1
6   int bb = 2
7   a = AIR           //variabler kan efter deklareringen tildeles vaerdier inden for
8     samme function
9   aa = 3
end function

```

Kodeeksempel C.3. Kodeeksempel for en trappe

Variabler der har fået tildelt en værdi, kan kun bruges inden for de functions hvor tildelingen er udført, f.eks. kan variabler mellem `function..end function` ikke tilgås uden for den specificerede function.

Operatorer i VMP omhandler de aritmetiske og logiske operatorer samt tildelingsoperatorer.

De aritmetiske operatorer inkluderer operatorerne (+, -, *, /, %), og disse operatorer fungerer på samme måde som i et matematisk miljø, og de bruges til at udføre beregninger på værdier af int typen.

- **Addition** angives ved + mellem værdier eller variabler, ” $x + y$ ”.
- **Subtraktion** angives ved – mellem værdier eller variabler, ” $x - y$ ”.

- **Multiplikation** angives ved * mellem værdier eller variabler, ” $x * y$ ”.
- **Division** angives ved / mellem værdier eller variabler, ” x/y ”. Division vil altid afrunde til laveste heltal.
- **Modulus** angives ved % mellem værdier eller variabler, ” $x \% y$ ”. Modulus vil returnere resten fundet efter division.
- **Parenteser** angives ved (før og efter) værdier eller variabler, ” $(x+y)*x$ ”. Parenteser gør det muligt for programmøren at ændre prioritering af operatører i udregningen af et udtryk.

De logiske operatorer inkluderer operatorerne ($=, / =, <, >, \leq, \geq, AND, OR$). De første seks operatorer udfører boolske beregninger på basis af værdier af int typen. AND, OR operatorerne udfører beregninger baseret på boolske beregninger.

Logiske udtryk udformes med logiske operatorer. Reglerne for anvendelse af disse operatorer, i VMP, er følgende:

- Mindre end ($a < b$) - ' $<$ ' operatoren kontrollerer om værdien af venstresiden er mindre end højresiden. Hvis a er mindre end b evalueres udtrykket til sandt, ellers falsk.
- Større end ($a > b$) - ' $>$ ' operatoren kontrollerer om værdien af venstresiden er større end højresiden. Hvis a er større end b evalueres udtrykket til sandt, ellers falsk.
- Lig med ($a = b$) - ' $=$ ' operatoren kontrollerer om værdien af venstresiden er lig med højresiden. Hvis a er lig med b evalueres udtrykket til sandt, ellers falsk.
- Ikke lig med ($a / = b$) - ' $/ =$ ' operatoren kontrollerer om værdien af venstresiden er lig med højresiden. Hvis a er lig med b evalueres udtrykket til falsk, ellers sandt.
- Mindre end eller lig med ($a \leq b$) - ' \leq ' operatoren kontrollerer om værdien af venstresiden er mindre end **eller** lig med højresiden. Hvis a er mindre end eller lig med b evalueres udtrykket til sandt, ellers falsk.
- Større end eller lig med ($a \geq b$) - ' \geq ' operatoren kontrollerer om værdien af venstresiden er større end **eller** lig med højresiden. Hvis a er større end eller lig med b evalueres udtrykket til sandt, ellers falsk.
- Konjunktion ($((a < b) AND (a = 2))$) - ' AND ' operatoren kontrollerer om to Boolean værdier, den ene til venstre og den anden til højre for operatoren, begge er sande. Hvis begge værdier er sande, evalueres udtrykket til sandt, ellers falsk.
- Disjunktion ($((a < b) OR (a = 2))$) - ' OR ' operatoren kontrollerer op til to Boolean værdier, den ene til venstre og den anden til højre for operatoren, om en af dem er sande. Hvis en sand værdi er fundet, evalueres udtrykket til sandt, ellers falsk.

De logiske operatorer kan kun anvendes på værdier og variable af datatypen **int**.

Den sidste operator er tildelingsoperatoren `=`. Denne operator bruges kun hver gang variable skal tildeles værdier.

Eksempel C.4 viser hvordan de forskellige operatorer anvendes.

```

1  function assignVars()
2    block a = STONE //foerste gang en variabel deklarereres, skal den tildeles en
3      vaerdi
4    int aa = 3
5    int bb = 2
6    int cc = 0
7
8    cc = bb * aa // cc tildeles vaerdien 6
9    cc = aa % bb // cc tildeles vaerdien 1
10   cc = aa + bb // cc tildeles vaerdien 5
11   cc = aa - bb // cc tildeles vaerdien 1
12   cc = bb / aa // cc tildeles vaerdien 0
13
14   cc /= aa OR cc = aa //true hvis cc ikke er ligmed aa eller hvis cc er ligmed aa
15   cc > aa OR cc < aa //true hvis cc er stoerre eller mindre end aa
16   cc <= aa OR cc >= aa //true hvis cc er mindre end eller ligmed, eller stoerre
17   end eller ligmed aa
16   cc < aa AND aa > bb //true hvis cc er mindre end aa og aa er stoerre end bb
17 end function

```

Kodeeksempel C.4. Anvendelse af operatorer

Function indeholder et scope, som indkaples af de forskellige deklarerede functions. Variable i dette scope kan kun tilgås inde i funktionerne, hvor de deklarereres, og det er derfor lige gyldigt for en function A, om en anden funktion B har deklareret en variabel identisk (med samme navn og type) med den variabel, som function A vil deklarere. Ændringer på værdier af variabler i function scope er dermed fastlagt til variabler inden for funktioners indkapsling (`end function`). Functions kan kalde andre functions, som vist i kodeeksempel C.5, under function `layers()`. Et kald til function `layer()` vil dermed udføre koden i de functions som bliver kaldt af `layers()`.

```

1  function layers()
2    layer1()
3    layer2()
4    layer3()
5    layer4()
6  end function
7
8  function layer1()
9  ...
10 end function
11
12 function layer2()
13 ...
14 end function
15
16 function layer3()
17 ...
18 end function
19
20 function layer4()
21 ...
22 end function

```

Kodeeksempel C.5. Kodeeksempel for en trappe

Ud over function, er det muligt at anvende **build**-udtrykket samt nogle kontrolstrukturer, i VMP.

build-udtrykket opbygges af **block** variabler i kroppen, og det er i denne krop der 'tegnes' en konstruktion. Ved hjælp af de kontrolstrukturer, der vil blive gennemgået efter **build**-udtrykket, kan denne konstruktion så bygges som ønsket inde i Minecraft. Kodeeksempel C.6 viser en 'tegning' af en 4x4 blok, opbygget af sten, hvorefter der bliver placeret én enkelt blok, med **build b** udtrykket. Dette **build**-udtryk, der bygger én blok, starter der hvor et tidligere **build**-udtryk stoppede, hvis de 2 udtryk bruges sekventielt i samme funktion.

```

1 block b = STONE
2 block & = STONE
3 block a = AIR
4
5 build(top)
6   & b b b
7   b b b b
8   b b b b
9   b b b b
10 end build
11
12 build b

```

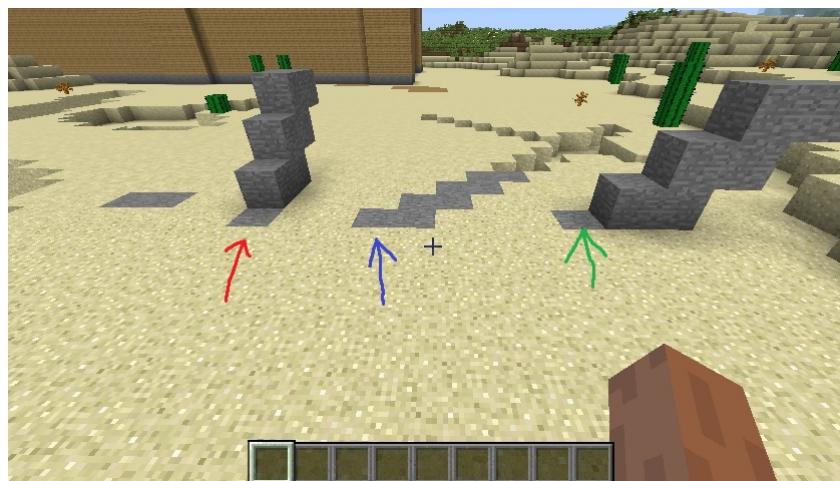
Kodeeksempel C.6. Kodeeksempel for build

Når der 'tegnes' i kroppen af **build**-udtrykket, skal følgende regler være opfyldt:

- **Control blocks** skal være placeret i 'tegningen'. Disse control blocks er de 3 symboler '!', '?' og '&'. Control blocks bruges til at vise hvor en konstruktion skal bygge fra, og slutte ved. Symbolerne '!' og '?' er henholdsvis start- og slut control blocks, og disse skal deklarereres som en **block** og tildeles en dertilhørende værdi. Symbolet '&' fungerer som både start- og slut control block.

Orientation parameter, der angiver hvordan ScriptCraft skal bygge den programmerede konstruktion, baseret på orienteringen af spilleren i Minecraft. Figuren C.9 viser eksempler på **stair(4)** function, hvor de 3 orientation værdier **side**, **top** og **front** er anvendt. Trappen ud for den røde pil angiver **side** orientation, blå pil angiver **top** orientation og grøn pil angiver **front** orientation. Kommandoen **stair(4)** blev, i eksemplerne, udført fra samme vinkel som billedet er taget fra, hvor musen var fikseret på de blokke pilene peger på.

Kontrolstrukturer der er anvendelige i VMP er **if** **else**- og **repeat**-kontrolstrukturerne, som henholdsvis er betinget selektiv og iterativ kontrolstrukturer. **if** **else**-kontrolstrukturen kræver et boolsk udtryk, som tidligere vist i kodeeksempel C.4, og udfører kode alt efter om udtrykket er sandt eller falsk. **repeat**-kontrolstrukturen tager imod ét heltal, hvor koden i kroppen af kontrolstrukturen bliver udført det antal gange som angivet ved heltallet. Det er muligt at udføre alt kode i kroppen af disse kontrolstrukturer, undtagen ny deklarering af variable samt functions. Eksempler på anvendelse af disse 2 kontrolstrukturer ses i kodeeksempel C.7.



Figur C.9. Minecraft: stair(4) orientation

```

1 function assignVars()
2     int aa = 3
3     int bb = 2
4     int cc = 0
5
6     repeat(5)      //koden mellem repeat(5) linje 6 og end repeat linje 26 gentages
7         5 gange
8         repeat(2)
9             if (cc /= aa OR cc = aa)
10                if (cc > aa OR cc < aa)
11                    if (cc <= aa OR cc >= aa)
12                        ...
13                    end if
14                    else      //udfoeres hvis if (cc <= aa OR cc >= aa) evaluerer
15                        til false
16                    ...
17                end else
18            end if
19        end if
20    end if
21
22    aa = aa + 1
23    bb = bb + 2
24    cc = cc + 3
25 end repeat
26 end repeat
27 end function

```

Kodeeksempel C.7. brug af kontrolstrukturer

Scriptcraft funktionskald

Når funktionerne skal kaldes inde fra den Minecraft server, som anvender ScriptCraft pluginet samt filerne med javascript kode for de definerede funktioner, skal kommandoen `/js <funktionsnavn>()` anvendes. Hvis funktionen for `maltheCastle()` skal anvendes, skal kommandoen udformes som `/js maltheCastle()`. Når der foretages funktionskald gennem ScriptCraft, vil konstruktionerne defineret i de kaldte funktioner opbygges ud fra

positionen af spilleren samt spillerens museposition. Der hvor spilleren peger sin mus, vil konstruktionens startpunkt være, når funktionskald foretages.

maltheCastle() figures D

```
function maltheCastle()
    firstL()
    secondL()
    thirdL()
    fourthL()
    fifthL()
end function
```

Figur D.1. Minecraft: maltheCastle() funktion

Figur D.2. Minecraft: maltheCastle() funktion

Figur D.3. Minecraft: maltheCastle() funktion

Figur D.4. Minecraft: maltheCastle() funktion

Figur D.5. Minecraft: maltheCastle() funktion

Figur D.6. Minecraft: maltheCastle() funktion

Figur D.7. Minecraft: maltheCastle() funktion



Figur D.8. Minecraft: maltheCastle() konstruktion