Language Design and Compiler Construction

from an Interdisciplinary Perspective

Vejleder:	Gruppe:	Retning:
Kurt Nørmark	b602a	BAIT6

Af:

Christoffer Hviid Poulsen

Daniel Neupart Hansen

Kristian Hove Andersen

Michael Kvist Nielsen

BAIT 6. Semester

Projekt gruppe: B602a

Titel:	Language design and compiler construction from an interdisciplinary perspective
Tema:	Sprog og Oversættere
Projekt periode:	BAIT6 1. Februar - 27. Maj 2011
Projekt gruppe:	B602a
Studieretning:	Bachelor i informations-teknologi (BAIT)
Gruppemedlemmer:	Christoffer H. Poulsen Daniel N. Hansen Kristian H. Andersen Michael K. Nielsen
Vejleder:	Kurt Nørmark
Side antal:	94
Afleverings dato:	27/05/2011

Synopsis:

Dette projekt er motiveret af en undren over ligheden mellem forskellige programmeringssprog. Denne undren baserer sig på filosofien:

"Hvorfor skal programmeringssproget styre programmøren, i stedet for at det er programmøren, der styrer sproget?"

Projektets mål er derfor at skabe et nyt sprog, hvis primære idé er strukturmæssigt at ligne et talesprog. Helt konkret vil målet være at erstatte specialtegn med ord, som er en inkluderet del af syntaksen.

Resultatet blev sproget MOCCA. MOCCA henvender sig til ikke-eksperter, som ønsker at lave små opgaver på computeren.

Det er lykkedes at skabe et sprog som er fri for specialtegn. Desværre resulterer det i en kompliceret syntaks, som kræver et stort overblik. Dette står i skarp kontast til ønsket om et mere intuitivt sprog. Dog står det også klart at idéen bestemt kan muliggøres, og at erfaringerne fra dette projekt kan være til stor hjælp.

Forord

Denne rapport er skrevet som et bachelorprojekt på 6. semester ved den tekniske retning af bacheloruddannelsen i informationsteknologi ved Institut for Datalogi på Aalborg Universitet. Projektperioden strækker sig fra 1. februar til 27. maj 2011.
Projektets tema er "Sprog og Oversættere". Målet med projektet er, at læringsfasen undervejs skal munde ud i, at vi kan demonstrere bredt kendskab til sprogdesign og oversætterkonstruktion.

Rapporten behandler temaet ved dele rapporten op i 4 dele: Sprogdesign, compilerkonstruktion, test og afprøvning samt refleksion.
Sprogdesign specificerer det sprog, vi vil udvikle.
Compilerkonstruktion redegør for opbygningen og udviklingen af en compiler til sproget.
Test og afprøvning efterprøver den konstruerede compiler i forhold til specifikationen.
Refleksionen diskuterer og konkluderer på metode og resultater. Ydermere vil en perspektivering kigge på MOCCAs fremtid.

Det forventes af læseren, at denne har datalogisk viden svarende til en bachelor i informationsteknologi. I denne rapport vil kildehenvisninger fremstå således: [Kilde, År].

Kildekoden til den compiler, der konstrueres ud fra denne rapport, kan findes på:

`http://mocca.prosphere.eu`.

A resume of this report, written in english, can be found in "Bilag A".

Christoffer H. Poulsen

Daniel N. Hansen

Kristian H. Andersen

Michael K. Nielsen

INDHOLD

KAPITEL 1

INTRODUKTION

"Dette kursus er ligesom levertran - Det er ikke sikkert, at I synes, det smager godt, men det er godt for jer!". Sætningen her blev sagt af vores forelæser Bent Thomsen i kurset "Sprog og Oversættere" (SPO) allerede til første kursusgang. Kurset er grundlag for projekttemaet af samme navn på 6. semester under den teknologiske retning af bacheloruddannelsen i informationsteknologi (BAIT) ved Institut for Datalogi på Aalborg Universitet. SPO er en grundlæggende datalogisk disciplin. Den beskæftiger sig med teorien bag udvikling og oversættelse af programmeringssprog. I denne rapport vil vi redegøre for, hvordan vi i praksis vil konstruere et nyt sprog, og under hvilke forudsætninger dette er sket. Vi vil så at sige præsentere den måde, hvorpå vi får gjort levertranen spiselig og til et grundlag for vores fremtidige datalogiske virke.

1.1 Formelle Krav

For at få den bedst mulige indlæring af den datalogiske grundsten, som SPO udgør, har instituttet fastsat nogle mål og krav i den studieordning vores uddannelse følger [og Sundhedsvidenskabelige Fakulteter De Ingeniør-, 2009]. Målene lyder som følger:

Vi skal kunne:

- Dokumentere kendskab til og overblik over de berørte teknikker og begreber inden for sprogdesign og oversætterkonstruktion.
- Beskrive, analysere og implementere en oversætter eller fortolker til et konkret programmeringssprog eller til en udvidelse til et eksisterende programmeringssprog.
- Redegøre for de enkelte faser og sammenhængen mellem faserne i en oversætter.
- Redegøre for de anvendte implementationsteknikker i den konstruerede oversætter eller fortolker.

- Benytte korrekt fagterminologi.
- Ræsonnere datalogisk om og med de berørte begreber og teknikker.

Disse mål er sat som et udgangspunkt for, hvad vi skal vurderes på til den afsluttende prøve. For at gøre os i stand til det er der sat nogle krav til indholdet i rapporten.
Den skal omfatte:

- En analyse af en datalogisk problemstilling, hvis løsning naturligt kan beskrives i form af et design af væsentlige begreber for et konkret programmeringssprog.
- I tilknytning hertil skal konstrueres en oversætter eller fortolker for sproget, som viser dels at man kan vurdere anvendelsen af kendte parserværktøjer og -teknikker, dels at man har opnået en forståelse.

1.2 Vores Baggrund

Inden vi fortæller, hvordan vi vil arbejde med læringssituationen samt opstiller en beskrivelse for det projekt, vi vil udføre for at imødekomme de formelle mål og krav, vil vi gerne sætte os selv i scene.

Vi er den lille gruppe af BAIT studerende, der efter 4. semester valgte at gå i den teknologiske retning. På 1. til 4. semester har vi tværfagligt beskæftiget os med kommunikation, forretning og teknologi. Den samlende kraft de 3 fagretninger imellem har været at se dem i et IT perspektiv. Kommunikationsfagene har hovedsageligt lært os, hvordan IT udvikles og optimeres i samspil med brugeren. Forretningsdelen har udover tal og marketing også fokuseret på, hvordan IT kan skabe værdi i en organisation.

Vi tænker på teknologi som byggestenen for IT forstået på den måde, at koncepter skabt med for eksempel et bruger eller organisatorisk perspektiv virkeliggøres af teknologien. Det har for os været en af hovedgrundene til at vælge den teknologiske vej på vores bachelor. Vi vil gerne nævne, at vi med vores valg ikke ønsker at underkende det at kunne skabe IT koncepter. Det interesserer os stadig lige meget, da det er en stor del af at skabe det gode IT system. Vi har dog et ønske om at kunne virkeliggøre dem selv ved hjælp af teknologi. Teknologi dækker for vores uddannelses vedkommende fagligt, såvel teoretisk som praktisk, over elementer fra det datalogiske felt.

Inden for datalogi har vi hidtil beskæftiget os med forskellige ting, vi mener, der kan knyttes til dette projekt. Det har vi gjort både igennem kurser og projekter. Det drejer sig hovedsageligt om sprog. Compilere har vi tidligere hørt om, men i høj grad abstraheret over.

Først og fremmest har vi i kurser mødt 2 programmeringsparadigmer, nemlig det imperative og det objektorienterede. Det betyder også, at vi har mødt nogle programmeringssprog. Vi har i meget lille skala programmeret imperativt i C i forskellige kurser og

i lidt højere grad web-scripting-sproget PHP i et projekt. Vi har i lidt større skala programmeret både imperativt og objektorienteret i C# i både projekt og kurser. Vi har også i enkelte kurser skrevet eksempelvis algoritmer med imperativ pseudokode.

1.3 Metode

På dette semester befinder vi os endnu en gang i en læringssituation. Vi skal igen, som i Kierkegaards filosofi, tage springet ud på de 70000 favne vands dybde. Som sædvanlig har vi umiddelbart ingen forudsætninger for projektet, og alligevel kaster vi os ud i det med krum hals. Der er uvished derude, dog virker det ikke så skræmmende som tidligere. Vi har som sagt prøvet denne tilgang før. Derfor er vi i dette afsnit i stand til at redegøre for, hvordan vi rent faktisk vil gribe projektet an.

1.3.1 Projektarbejde

I et projekt hvor læring er drivkraften og viden produktet, starter vi med at afstemme forventninger med hinanden. Det handler om forventninger i forhold til vinklingen af projektet og det enkelte gruppemedlems fremmøde, arbejdsindsats og tidsforbrug. Det gør vi for at få et fælles udgangspunkt og en god start. Godt begyndt er halvt fuldendt. Denne måde at starte et projekt på er et resultat af 5 tidligere projektrapporter, hvor mange erfaringer er gjort.

I forhold til dette projekt vil vinklingen fremgå af afsnittene Motivation (afsnit 1.4) og Afgrænsning (afsnit 1.6). For at få en balanceret arbejdsindsats gruppemedlemmerne imellem evaluerer vi hinandens arbejde fra uge til uge gennem en fælles kommentarrunde. Projektarbejde hvor gruppen er samlet foregår i tidsrum, hvor der er skemafrit, og som ikke er vores aftalte reserverede arbejdsdage.

Projektarbejdets omdrejningspunkt vil være morgenmøder. Vi vurderer, at vi kender hinanden så godt, at den åbne dialog der hersker på møderne, er det mest givende for os. Disse kan enten have karakter af planlægnings- eller arbejdsmøder. Planlægningsmøder er med nogenlunde fast dagsorden, hvor vi informerer hinanden om, det vi hver især arbejder med samt modtager kommentarer til det. Derudover vil vi aftale og uddele nye arbejdsopgaver og aftale arbejdsmøder, hvis der er behov for det. Behovet for arbejdsmøder opstår, hvis vi vurderer, at en problemstilling er så kompleks, at en fælles diskussion vil gavne. Møderne fordrer fælles forståelse. På møderne opdaterer vi også vores oversigtsværktøjer bestående af tidsplan og rapportdisposition. Antallet af møder stiger i takt med, at antallet af forelæsninger falder.

Vores ugentlige morgenmøder tjener også et andet formål. Siden vi tager referat i møderapporter, har vi ved projektet afslutning en god ressource til at dokumentere projektets forløb. Yderligere indeholder de også problemer og bekymringer, vi har haft omkring

opbygningen af compileren. Disse overvejelser vil få stor fokus i diskussionen og danne grundlag for hele læringsprocessen, som vi konkluderer på til sidst.

1.3.2 Rapportskrivning

Fokuset er naturligvis på at skabe en rapport, som kan danne grundlag for eksamenen jævnfør de formelle krav. En skriftlig rapport bærer naturligvis altid præg af, hvilket fagområde den er skrevet indenfor. Heraf fremkommer et paradoks. Vi har skrevet fem rapporter på tidligere semestre under tre forskellige fagområder. Hver af fagområderne vægter emner forskelligt og stiller forskellige krav til rapportens læsere. Dog har vi gennem disse fem rapporter kunne se en gylden tråd i opbygningen.

Vi følger således strukturen: Introduktion, analyse, design, diskussion, konklusion og perspektivering.

Introduktionen ender ud i en problemformulering med udgangspunkt i vores formulerede motivation, vores visioner og en afgrænsning.

Analysen knytter sig primært til foranalysen, der består af et oplæg for projektets præmisser og en problemformulering.

Designfasen dækker over sprogdesignvalg og implementering.

Diskussionen er en kombination af refleksion over projektets læringsmæssige grundlag og erfaringsopsamling. Konklusionen har så til formål at opsamle begge dele af diskussionen i dens hovedpunkter. Sidst vil vi slutte af med perspektivering. Det dækker over MOCCAs fremtidsudsigter.

Undervejs vil vi referere til materiale i bilagene. Disse er ikke tænkt nødvendige at læse for at forstå rapporten i sin helhed, men blot for yderligere information.

Terminologi

Vi har valgt at skrive rapporten på dansk. Det giver nogle udfordringer i forhold til terminologien, da datalogiske udtryk er af engelsk oprindelse. For at imødekomme disse udfordringer har vi som hovedregel besluttet at bruge de engelske navne fremfor danske. Et eksempel kunne være, at vi skriver compiler i stedet for den danske term oversætter. Det begrunder vi med, at vi anser det for at være korrekt fagterminologi, hvilket var et formelt læringsmål. Endvidere har nogle begreber slet ingen eller kun ringe danske pendanter. Når vi beskriver de enkelte begreber, bruger vi derimod danske verber. Det betyder, at en parser ikke er beskrevet som, at den parser et input, men som at den analyserer et input. Det gør vi for at vise, at vi har forstået de forskellige begreber.

1.3.3 Sprogudvikling

Ligesom ved alt andet fremkommer der ikke noget ud af intetheden. Alt har altså en enten en forgænger eller en inspirationskilde. Vores inspirationskilde vil være sproget C#, som også er det sprog, vi er mest trænet i. Men målene for henholdsvis C# og vores sprog er jo naturligvis dybt forskellige. Derfor vil vi kun kunne bruge en brøkdel af C#'s ideer til at danne sprogets syntaks. Disse kodeeksempler vil vi kunne benytte til at skabe vores eget sprogs syntaks.

Indledningen til kurset "Sprog og Oversættere" gav os et indblik i, hvor de dominerende sprog i dag har deres rødder. Størstedelen kan naturligvis sende en stor tak i retning af folkene bag ALGOL60 og de erfaringer, man har gjort sig herfra. Men måske mere interessant er der opstået et helt videnskabeligt felt til studiet af, hvorledes forskellige design principper kan påvirke et sprog i forskellige retninger. Disse design principper er blevet præsenteret for os i kurserne "Sprog og Oversættere" og "Syntaks og Semantik". Udviklen af sproget vil være meget præget af undervisningsmetodikken. Den overordnede metodiske tilgang, der former sprogdesign kapitlet, har rødder i en af "Sprog og Oversættere" kursets lærebøger [Watt and Brown, 2007]. Den præsenterer 2 forskellige måder at specificere designet af et programmeringssprog på. Den første måde er en uformel specifikation skrevet i et naturligt sprog og gjort forståelig for enhver, der gør brug af sproget. Den anden måde kaldes formel specifikation. Denne er skrevet i en præcis notation, der udmærker sig ved at understøtte beskrivelsen i at være utvetydig, konsistent og fuldendt. Dog vil notationen i en formel specifikation kræve et vist forhåndskendskab.

1.3.4 Compilerudvikling

I kurset "Sprog og Oversættere" har man valgt at dele compileren i tre dele, hvilket også er den normale gængse tilgang. Endvidere er de tre dele splittet op, således at de passer i forelæsningsdele, hvortil man kan lave opgaveløsning. Det betyder, at vi kan overføre erfaring fra opgaveregningen direkte til vores projekt løbende. Vi skal altså blot følge oplægget fra kurset og herved også få hjælp til problematiske valg fra kursusholder. Kurset benytter sig af David Watt og Deryck Browns "Programming Language Processors in Java"[Watt and Brown, 2007]. Bogen leverer en tredelt forståelse af compilerkonstruktion. Den følger en eksempel orienteret tilgang, der gennemgår compiler konstruktion skridt for skidt. Dette er ledsaget af Java kode eksempler. C# eksempler findes på bogens hjemmeside. Bogen præsenterer altså alle sine koncepter i konkrete eksempler, som kan overføres direkte til vores projekt. Undervejs skal der naturligvis også foretages mange valg.

1.3.5 Test og Afprøvning

For at belyse sammenhængen mellem sprogdesign og den compiler vi konstruerer, vil vi bruge et kapitel på at teste og afprøve den. Tilgangen vil være noget, der er inspireret af `test driven development`. Altså at vi sætter testkode op, der bør kunne verificeres af compileren. Hvis dette ikke kan lade sig gøre, tilretter vi og kører testen igen, indtil det virker. Det betyder, at vi vil udfordre vores compiler i forhold til det udviklede sprogs syntaks, gætte på et udfald, se hvilket resultat det giver og til sidst evaluere på det. Giver disse test resultater, vi ikke regner med, kan det give anledning til ændringer i compileren. Ændringerne vil blive beskrevet.

1.3.6 Refleksion

Vi vil sidst i rapporten bruge et kapitel på at reflektere over denne rapport og projektet. Vi vil dele refleksionen op i tre dele: Diskussion, konklusion og perspektivering.
Diskussion stiller spørgsmål ved både de valgte metoder og de resultater, vi har opnået ved at bruge dem.
Konklusionen samler op på rapporten og svarer på, hvordan vi fik løst de opgaver, vores afgrænsning stiller op (afsnit 1.6).
Perspektiveringen vil se på MOCCAs fremtid som sprog. Det vil den gøre ud fra en tilgang, der kigger på, om MOCCA har tiden foran sig, når det læringsaspekt som projektarbejdet lægger ned over det forsvinder.

De enkelte kapitler vil yderligere redegøre og uddybe specifikke metoder, efterhånden som de benyttes.

1.4 Motivation

Vores motivation til at opfylde de formelle mål gennem vores projektarbejde knytter sig meget til en undren over udformningen af programmeringssprog i de forskellige paradigmer, vi har mødt igennem tidligere undervisning. Det kan bunde i vores tværfaglige baggrund, at vi primært er motiveret af denne undren.

Vi er tit blevet præsenteret for det synspunkt, at hvis man kan lære ét programmeringssprog, så kan man lære at programmere i langt i de fleste. Der synes altså at være visse ligheder mellem sprogene - i hvert fald inden for hver enkelt af de 4 programmeringsparadigmer; imperativt, objektorienteret, funktionel og logisk. Kan man få lært sig et sprog inden for hvert paradigme, skulle den grundlæggende forståelse for programmeringens natur være på plads. Vi har dog gjort den observation undervejs i vores studie, at medstuderende, som vi betragter som havende et teknisk godt udgangspunkt, ikke har formået, selv med undervisning i faget, at lære at programmere. Mere specielt finder vi det dog, at de faktisk har været i stand til tale omkring programmering, som løsning af bestemte problemer i forbindelse med gruppearbejde, på lige fod med gruppemedlemmer,

som har fundet programmering væsentligt nemmere. De har altså haft en anderledes opfattelse af det imperative paradigme. Det er ikke lykkedes dem at krydse den grænse, hvor den definition, samtalen udmundede i, har kunnet blive til datatyper, specialtegn, funktioner osv. i eksempelvis C#, som har været vores fælles sprogreference.

Den observation, som vi lige har beskrevet, giver anledning til en motivationsfaktor. Vi kan godt tænke os at lave et programmeringssprog, der gør det lettere at programmere ud fra den fælles referenceramme, vi oplever, de fleste har, når de har beskæftiget sig med IT på et plan, hvor skabelsen af små imperative programmer er næste skridt.

1.4.1 Den Store Vision

Programmeringssproget, vi vil skabe, vil vi kalde "MOCCA - Speech Empowered Programming". Dette afsnit vil beskrive, hvad et færdigudviklet MOCCA kan gøre for verden, og kan derfor godt ses som en salgstale. Den brede målgruppe (brugere) af sproget vil være typen af IT brugere, hvor udvikling af små imperative programmer er den næste IT færdighed i værktøjskassen.

Vi kender til et studie på AAU, nærmere bestemt Global Forretningsudvikling, som modtager undervisning i regneark. Her lærer de, hvordan simple `if` sætninger i en macro kan forvandle et regneark til et brugbart budgetstyringsværktøj. Efter vores overbevisning er der ikke særlig langt fra forståelsen af `if` sætninger og andre kontrolstrukturer i regneark og til at komme i gang med at følge generel kommando baseret programmerings tankegang. Derfor danner de en grundlæggende idé om, hvem vores målgruppe er.

"MOCCA - Speech Empowered Programming" er for den kræsne og ikke-nørdede IT bruger. MOCCA byder ikke på alle de finurligheder, der i vores øjne gør læringskurven på konventionelle sprog stejl. Du vil således ikke længere opleve at skulle balancere typer af parenteser, du ikke anede fandtes. Det samme gælder eftersøgningen af den linje, der ikke lige er afsluttet med den rette type kolon. MOCCA gør nemlig op med den lange liste af specialtegn og gør brug af det sprog, du alligevel benytter i din dagligdag.

Tilnavnet er "Speech Empowered Programmering", og det er ikke en tilfældighed. MOCCA leveres nemlig ikke med et stort, skræmmende og ikke mindste nyt og ukendt program. Den grænseflade du skal beskæftige dig med er et stykke stemmegenkendelsessoftware. For MOCCA programmeres med stemmen og med naturligt sprog.

1.5 Problemformulering

Inden vi afgrænser os til en række specifikke mål for projektet, vil vi her kort opsummere hvilke forhold, der gør sig gældende i forhold til løsningen, hvad problemet dækker over og en beskrivelse af, hvad løsningen bør basere sig på.

Der er i afsnit 1.1, "Formelle mål og krav", fastsat nogle krav til indholdet af rapporten. Projektet skal løse problemet på en måde, så disse kan opfyldes.

Det konkrete problem kan opstilles som spørgsmålet:

Kommer vi nærmere en realisering af "Den Store Vision" ved at udforme et lille programmeringssprog med dertilhørende compiler, der eliminerer specialtegn, og sætter os i stand til at udforme små test programmer?

Løsningen til problemet ligger i de forskellige metoder, som SPO kurset har præsenteret for os. Der skal formuleres et sprogdesign, konstrueres en compiler og sproget skal derefter afprøves med denne.

1.6 Afgrænsning

Vi er klar over, at vi ikke vil stå med et markedsklart sprog, alla det vi beskriver i "Den Store Vision", ved dette projekts deadline. Det er der flere grunde til. For det første er visionen ikke realistisk tid og ressourcer taget i betragtning. Sekundært er det i forhold til projektets rammer vores egentlige ambition at leve op til de formelle mål og krav - ikke at levere et programmeringssprog til markedet.

Vi vil derfor afgrænse visionen og klarlægge, hvad det er, vi beskæftiger os med i denne rapport. Vi ser bort fra idéen omkring at programmere ved hjælp af tale i sin helhed. Vi kommer ikke til at udforme et programmeringssprog, der er baseret fuldstændigt på naturligt sprog. Det er alt for omfattende, hvis ikke umuligt.

Der er dog et element i visionen, vi gerne vil tage med i vores projekt. Det er noget, der kan bringe programmeringssproget tættere på naturligt sprog. Det er det element at fjerne specialtegn. Specialtegn er i denne sammenhæng dem, der på engelsk kaldes "punctuation marks". Det dækker blandt andet over alle typer parenteser og koloner samt almindelige tegnsætningstegn. Derudover ser vi også aritmetiske symboler som specialtegn, det vil sige plus, minus, gange, dividere osv.

Grunden til at vi gerne vil afgrænse MOCCA, til at være et lille sprog uden specialtegn, skal også findes i praktisk erfaring. Det virker for os som den første hindring på den lange vej til programmering med naturlig tale. Det er ofte her, vi oplever, at vi selv går galt i byen, når vi programmerer. Det er ofte italesættelsen af et specialtegn, der får en

samtale i grupperummet omkring et stykke kode til at gå i stå. Til gengæld beriger det grupperummet med diverse lyde og fagter.

Sproget skal kunne eksekvere små test programmer ved at oversætte til C#. Vi mener ikke, det er realistisk, at vi indenfor projektets tidsramme kan designe en compiler, der oversætter vores sprog direkte til maskinkode. Samtidig er C# det sprog, vi er oplært i fra de tidligere semestre.

Vi synes, at skabelsen af dette sprog virker realistiske mht. tid, ressourcer og indlæring, således at vi kan komme i mål med projektet.

Den konkrete afgrænsning kan altså opstilles som følgende punkter, vi mener opfylder problemformuleringen:

- Vi vil designe et lille imperativt programmeringssprog uden specialtegn.
- Vi vil konstruere en compiler, der kan oversætte MOCCA kode til ækvivalent C# kode.
- Vi vil teste om vores compiler opfylder sprogets syntaks.
- Vi vil i refleksionen perspektivere vores sprog i forhold til "Den Store Vision".

KAPITEL 2

SPROGDESIGN

I dette kapitel vil vi gøre rede for de designvalg, vi har truffet for MOCCA, hvorfor vi har truffet dem, og hvilken betydning de har medført. Som nævnt under "Metoder" i afsnit 1.3 vil kapitlet tage udgangspunkt i kurset "Sprog og Oversættere"s litteratur [Watt and Brown, 2007].

Watt og Brown lægger op til, at man træffer et valg af specifikationstype. Valget står mellem en formel og en uformel specifikation. Dog understreges det, at man ofte ender med en hybrid, hvor den formelle specifikation støttes af en uformel. Men hvor Watt og Brown ikke eksplicit sigter mod en hybrid, vælger vi på forhånd at lave både en uformel og en formel specifikation. Vi vælger den tilgang, fordi vi, i egenskab af vores baggrund, mener at have gode forudsætninger for at skrive en uformel specifikation af MOCCA. Samtidig er vi klar over, at den formelle specifikation er en nødvendighed, når vi senere i rapporten giver os til at konstruere en compiler til MOCCA. Vi mener, at der i spændet mellem de 2 specifikationer ligger en rigtig god læringsproces. Det begrunder vi med, at der i den proces skal træffes en masse valg.

Der vil også sidst i kapitlet blive præsenteret en række kodeeksempler, hvor små programmer skrevet i MOCCA vil blive gennemgået. De har en iterativ funktion, forstået på den måde at de medvirker til det formelle sprogdesigns udvikling. Der har således været en simultan udvikling af syntaks og kodeeksempler, hvor eksemplerne har påvist mangler i syntaksen, og syntaksen har påvist uoverensstemmelser i eksemplerne.

2.1 Uformel Specifikation

Vi har tidligere beskrevet den målgruppe, som vi har haft i tankerne med MOCCA. Målgruppen var kendetegnet ved, at arbejde ofte med computer, men aldrig at have gjort sig i programmeringens finurlige univers. Som eksempel brugte vi en Global Forretningsudvikling studerende og regneark.
Ud fra dette grundlag vil vi nu bestemme nogle mere faste rammer vi ønsker sproget skal holde sig indenfor. Først og fremmest ser vi en mulighed og et potentielt behov for at lave et sprog, som vægter at ligne talesprog mest muligt. Jævnfør vores afgrænsning (afsnit 1.6) skal dette forstås som muligheden for udelukkende at have ord fremfor specialtegn. Håbet er, at dette skifte vil øge læsbarheden af programkoden og medføre en mere intuitiv forståelse af kode for målgruppen.

Ud fra vores undervisning i SPO har vi gjort os den forståelse, at programmeringssprogs primære struktur i dag oftest er bygget op omkring specialtegn. Dette har vi fået flere forklaringer på. Blandt andet gør det kodearbejdet væsentligt nemmere for en erfaren programmør. Det minimerer antallet af tastaturtryk for at udføre et stykke arbejde. En anden vigtig ting er, at man også forsøger at undgå tvetydighed, for eksempel ved brug af semikolon, til at vise, at nu er kommandoen færdig. Havde man valgt, at linieskift skulle udføre denne funktion, kunne man måske komme i en situation, hvor linieskift ikke betød enden af kommandoen, men snarere et brud i kommando sekvensen. Her er det naturligvis interessant, hvordan vi vil håndtere denne problematik, og dette forsøger vi at gøre klart med vores formelle beskrivelse.

Det er hensigten, at MOCCA skal anvendes af folk, der ellers ikke er vant til at programmere. Vi vælger derfor at designe det som et højniveau sprog. Det, der karakteriserer et højniveau sprog, kan være brug af datatyper, kontrol strukturer, erklæringer, abstraktion og data indkapsling [Watt and Brown, 2007].

MOCCA søger at imødekomme en del af disse karakteristika, idet vi gerne vil langt væk fra det uforståelige for "det almindelige menneske" - i dette tilfælde maskinkode. Derfor er abstraktion, kontrolstrukturer og erklæringer sat i fokus.

Vi fokuserer på at udvikle MOCCA som et lille imperativt sprog ud fra overvejelser om projektets begrænsede ressourcer, som beskrevet i afgrænsningen (afsnit 1.6). Imperativt vil sige et sprog, der består af en række kommandoer, der skal udføres af computeren.

2.1.1 Design Kriterier

Vi vil her beskrive de forskellige designkriterier i forhold til MOCCA. Et designkriterie i forhold til MOCCA skal ses som en måde at opstille en række krav til, hvad sproget skal kunne. Idet der er mange valg i forhold til at designe et nyt sprog, er dette en måde at strukturere arbejdet på og skabe et solidt fundament for sproget. Dette hører ind under det uformelle design, fordi det ikke direkte nævner nogle specifikke løsninger, men mere overordnede mål. Metoden er blevet introduceret i SPO kurset.

De forskellige kriterier bliver vurderet til at være enten "Meget vigtigt", "Vigtigt", "Mindre vigtigt" eller "Ikke vigtigt". Disse kan ses i en oversigt i figur 2.1. Samtidig er den primære funktion for sproget at introducere os for den udfordring, der ligger i at udvikle programmeringssprog og implementere det i en compiler. Det er derfor ikke hensigten, at sproget skal være specielt omfattende.

Kriterie	**Meget vigtigt**	**Vigtigt**	**Mindre vigtigt**	**Ikke vigtigt**
Læsbarhed		X		
Skrivbarhed	X			
Pålidelighed			X	
Ortogonalitet			X	
Uniformitet		X		
Vedligeholdelse				X
Generalitet			X	
Udvidelse				X
Implementérbarhed	X			

Figur 2.1: *Tabel over vurdering af designkriterier.*

Læsbarhed

Læsbarhed er defineret som evnen til at læse og forstå programmet. Denne disciplin lægger sig meget op af programmøren, men også op ad udefrakomne, der skal kunne forstå koden.
Læsbarheden i MOCCA er "Vigtig", forstået på den måde at det skal være nemt for ikke-programmører at forstå koden. Dog er det primært folk selv, der skal bruge det kode, de frembringer. Derfor vil man have et forhåndskendskab til, hvad koden gør.

Skrivbarhed

Skrivbarhed er defineret som, hvor let sproget skal være at skrive. Her er der fokus på klarhed, korrekthed, præcision og hastighed. Denne disciplin lægger sig mest til programmøren, idet det er denne person, der bruger sproget.
Skrivbarheden er "Meget vigtig", da målet med sproget netop er at sætte ikke-kyndige i stand til at programmere på egen hånd.

Pålidelighed

Pålidelighed er defineret som, at et program vil opføre sig som planlagt, når det udføres. Denne disciplin er især henvendt til udvikleren og stiller krav til sprogets design, der skal sørge for, at en applikation fortsætter med at køre som planlagt.
MOCCA skal ikke bruges til kritiske applikationer men blot lette hverdagen på kontoret. Pålideligheden vurderes derfor til at være "Mindre vigtig".

Ortogonalitet

Ortogonalitet er defineret som muligheden for at kombinere alle sprog facetter på en meningsfyldt måde. Dette stiller i høj grad krav til sprogdesignet og kan øge kompleksiteten af programmer. Ortogonalitet er "Mindre vigtigt" da den tænkte bruger ikke har et kodekendskab, der dækker alle facetter af programmeringsdisciplinen.

Uniformitet

Uniformitet dækker over, at ens funktioner i sproget skal se ens ud, men også opføre sig på samme måde. Uniformitet er "Vigtigt", da det skal være nemt at lære MOCCA for folk, der ikke er trænede i programmering. En ensartet opbygning vil fremme indlæringen.

Vedligeholdelse

Vedligeholdelse er defineret ved evnen til at kunne finde og rette de fejl, der måtte være efter udgivelsen af sproget. Dette lægger sig op ad udvikleren, da der stilles krav til at vedligeholde sproget. Da vi ikke forventer at udvikle videre på MOCCA efter dette projekt, vurderes det, at maintainability er "Ikke vigtigt".

Almindelighed

Almindelighed definerer mængden af generel abstraktion af programkode. Mange moderne programmeringssprog vægter dette højt, men generel abstraktion er kun vigtigt i det omfang, at det har noget at sige for, hvor nemt det skal være at sætte sig ind i brugen af MOCCA. Det er derfor vurderet til at være "Mindre vigtigt".

Udvidelse

Udvidelse definerer muligheden for, at programmøren selv kan tilføje elementer til koden så som egne typer eller structs. Da sproget designes til folk uden specielt meget programmeringserfaring, er udvidelse "Ikke vigtigt".

Implementérbarhed

Implementérbarhed lægger sig op af evnen til at kunne implementere en compiler eller fortolker til sproget. I forhold til de tænkte brugere af MOCCA er implementérbarheden mindre vigtig. Den vurderes dog alligevel til "Meget vigtigt", da dette projekt indebærer, at sproget implementeres i en compiler. Dette gør også, at andre kriterier eventuelt kan blive nedprioriteret af hensyn til projektet.

2.2 Formel Specifikation

Den formelle specifikation af sproget er i modsætning til den uformelle en matematisk tilgang til at beskrive sproget på. Den beskriver mere konkret, hvad sproget skal kunne og stiller helt klare retningslinier op for, hvordan syntaksen er bygget op. Dette vil blive gjort ved hjælp af et metasprog, der kan løse denne problemstilling. Selve den formelle specifikation består af to dele:

- `Scoperegler`
  Scopereglerne fortæller noget om, hvordan variable optræder i programmerne. Dette er den *kontekstsensitive* del. Den del der ikke kan læses direkte ud fra syntaksreglerne.
- `Syntaksregler`
  Syntaks reglerne er en helt konkret beskrivelse af, hvordan et program skal opbygges. Dette er den *kontekstfrie* del af sproget. Altså den den del man kan læse direkte ud fra syntaksreglerne.

Begge emner vil blive behandlet i det følgende afsnit. Det er den formelle specifikation, der i sidste ende danner grundlag for implementeringen af compileren til sproget.

På baggrund af at vi i compilerkonstruktions kapitlet, se kapitel 3, har valgt at anvende en *recursive descent parser* [Sebesta, 2010], er vores sprogdesign nødt til at følge betingelserne for *LL(1)* sprog. Da parseren ikke kan forudsige kommende *tokens*, er vi nødt til at sikre, at der ikke er *terminalsymboler*, der består af det første ord i andre terminalsymboler. Det kunne for eksempel være to forskellige operatorer, der hed henholdsvis `is` og `is not`. I MOCCA er operatorerne med disse betydninger derfor blevet døbt `is` og `differs from`.

2.2.1 Scoperegler

Scopereglerne, som MOCCA benytter sig af, er den såkaldte *nested block structure* [Watt and Brown, 2007]. I forhold til scopereglerne taler vi om variable og om blokke. En blok starter ved et `begin` og slutter ved et `end`. Nested block structure vil sige, at blokke kan være indlejret i hinanden, og derfor kan der være flere niveauer af scopes. Vi siger, at hvis en variabel er erklæret lokalt i forhold til en blok, er den i det samme "scope level".

Følgende regler gælder om scope levels:

- Erklæringer i den yderste blok er globale, det vil sige, at de kan bruges alle steder i programmet. Den yderste blok har scope level 1.
- Erklæringer i en indre blok er lokale for lige netop den. En indre blok er altid indkapslet i en ydre blok. Hvis en indre blok er indkapslet i den yderste blok, har

denne scope level 2. Hvis en indre blok er indkapslet i scope level 2, så er denne i scope level 3 osv.

Nested block structure har følgende scope regler:

- Ingen globale variable må have lov til at blive generklæret globalt. Dog må de samme variable gerne blive erklæret igen lokalt.
- For hver anvendt forekomst af en variabel I i en blok B, skal der være en erklæring af I. Denne erklæring skal være i selve B. Hvis ikke dette er tilfældet, så skal den være i den blok B', der indkapsler B. Hvis dette heller ikke er tilfældet, skal den være i den blok B", der indkapsler B' osv.

Derudover indeholder MOCCA yderlige regler i forhold til:

- **Variable og frie navne**
  For at begrænse konsekvenserne af stavefejl ved brug af variable vælger vi, at variable skal erklæres på forhånd, altså før de bruges i programmet. Dette gøres ved at bruge `let` efterfulgt af erklæring af variable og funktioner, og derefter følger `in`, hvor programkoden, der anvender funktionerne og variablerne, hører til. Der kan således ikke oprettes variable spontant.
  Da erklærede variable kan bruges i blokke inde i den blok, de er erklæret i, giver det lejlighed til, at funktioner kan benytte sig af frie navne. Funktionerne kan derfor gøre brug af omgivelsernes statiske bundne variable i kroppen eller som parametre. Det skyldes scopereglerne. I MOCCA kan man derfor nøjes med at erklære funktionen lokalt, hvis de variable, man gør brug af i den, er erklærede i omgivelserne.

- **Evaluering af udtryk**
  Det er ikke muligt at bruge paranteser til at opdele større udtryk, jævnfør elimineringen af specialtegn, så de enkelte dele evalueres i en bestemt rækkefølge. Vi er derfor nødt til at tage en beslutning om, at der læses fra venstre mod højre, når udtryk evalueres. Dog gælder det ved tildelinger, at der læses fra højre mod venstre.

Overtrædelser af disse scoperegler bør medføre, at compileren returnerer en fejl.

2.2.2 Metasprog

Der anvendes *Extended Bachus Naur Form* (EBNF) til at beskrive syntaksen i MOCCA. Vi er blevet undervist i dette metasprog i forbindelse med kurserne SPO og SS og kan derfor forholde os til, hvordan det skal bruges. Derfor var EBNF et naturligt valg, da vi skulle beslutte, hvordan vi ville beskrive syntaksen for vores sprog. Samtidig er det effektivt til at beskrive rekursion og præsenterer syntaksen på en mindre kompleks måde, end hvis der var benyttet Bachus Naur Form.

2.2.3 Syntaks

Følgende sektioner beskriver sproget MOCCAs konkrete syntaks. For overskuelighedens skyld vil de enkelte dele af syntaksen blive beskrevet individuelt. En komplet beskrivelse af sproget kan findes i bilag B. Til at støtte beskrivelsen findes der i bilag C syntaksdiagrammer, der visualiserer hver regel i syntaksen.

Syntaksen skelner mellem følgende elementer:

- **Nøgleord**, eller terminalsymboler, er de symboler, som vi rent faktisk indtaster på tastaturet, og som optræder direkte i programmet. Det er i dette sprogs tilfælde for eksempel `while`, `let`, `if`, `function` osv.

- **Nonterminalsymboler** repræsenterer den mængde af sætninger og symboler, der så at sige ikke optræder direkte i programmet. I dette sprogs tilfælde er det for eksempel `single-command`, `command`, `declaration` osv. De kan ses i syntaksen i bilag B, hvor nonterminaler for overskuelighedens skyld er fremhævet.

Variable og Typer

MOCCAs variable kan indeholde 2 typer af værdier, nemlig *integers* og *strings*. Forskellen på de to er, at integers er heltal, og alle andre værdier skal ses som tekststrenge. Et integer er i denne forstand et heltal, det vil sige et naturligt tal. Dog er der begrænsninger, idet vi forholder os til 32 bit heltal og et begrænset antal kombinationer [MSDN, 2011*a*]. En string skal ses som en følge af symboler. Bemærk at navne på identifiers, altså en streng, kun kan starte med et `letter`.

MOCCA benytter sig ikke af eksplicitte typer på variable, det vil sige at programmet i realiteten ikke ved, hvad man giver som input til det. Dette er et valg, da skrivbarhed vægtes højt.

integerlit ::= **digit digit***

identifier ::= **letter** (**digit** | **letter**)*

Da brugerne ikke er vant til at tænke programmering, vælger vi ikke at bruge erklæring af typer for variable. Dette forekommer os også at være i tråd med målet om, at MOCCA på sigt skal anvende programkode bestående af naturlig tale. Samtidig fravælger vi gængse datastrukturer som arrays og hægtede lister for ikke at komplicere sproget unødigt, vores målgruppe taget i betragtning.

Programmer

Program er det mest fundamentale syntaktiske element. Det definerer hele strukturen af selve programmet. I og med at MOCCA kræver, at variable bliver erklæret, før de bliver brugt, består et program af en erklæring. Denne erklæring gælder for den efterfølgende command. Hvis en variabel er erklæret i programmet, er den global. Det vil sige, at den kan tilgås i hele programmet. En `begin command end` beskriver en kodeblok.

program ::= let **declaration** in begin **command** end

Erklæringer

En erklæring er en måde at gøre det klart, at man vil oprette et element, som man har tænkt sig at bruge senere. For at kunne gøre brug af funktioner i MOCCA har vi behov for at kunne erklære dem. Selve betydningen af terminalsymbolet `function` er som de C typer af sprog, vi hidtil har arbejdet med. Det betyder, at `function` både bliver brugt til at abstrahere over en mængde af udtryk, det som andre sprog kalder en funktion, og en sekvens af kommandoer, det som andre sprog kalder en procedure.

I MOCCA er erklæringer specielt vigtige, da sproget kræver, at variable er erklæret på forhånd, altså inden de bruges. En erklæring kan bestå af én eller flere single-declarations. Hvis der er mere end én erklæring skal de adskilles af nøgleordet `and`.

declaration ::= **single-declaration**
| **single-declaration** and **declaration**

single−declaration ::= **identifier**
| function **identifier** **formal-parameters** begin **command** end

formal−parameters ::= ε
| **identifier**
| **identifier** and **formal-parameters**

En enkelt erklæring kan enten være en variabel, eller det kan være en funktion, efterfulgt af formelle parametre. Funktioner er ligesom variable identificeret på et navn, altså en identifier. Formelle parametre kan være identifiers adskilt af `and`, eller de kan være tomme, deraf epsilon.

Kommandoer

Kommandoer i MOCCA består af enten én enkelt kommando eller en sekvens af enkelte kommandoer. Hvis der er mere end én kommando, er de adskilt af `followed by`. Dette er en såkaldt sekvens af kommandoer.

Eftersom MOCCAs imperative natur betyder, at en række kommandoer udføres sekventielt, har vi haft behov for et terminalsymbol, der fortæller, at en kommando er slut, og at en ny starter. I C#, som vi bruger som reference- og målsprog, afsluttes enhver kommando med et semikolon. I første omgang forsøgte vi os med et terminalsymbol, vi kaldte `endline`, fordi vi ligeledes mente, at betydningen skulle være, at kommandoen sluttede her. Vi blev dog klar over, at det ikke udelukkende er den betydning, der ligger i semikolonet. Den første opfattelse mener vi grunder i netop det, at en hver kommando i C# afsluttes med det. Dermed gik den måske vigtigste del af betydningen tabt - nemlig at en ny kommando følger. Med terminalsymbolet `followed by` mener vi, at betydningen bliver repræsenteret stærkest. Det mener vi blandt andet fordi, at kommandoen, som ligger sidst i sekvensen, ikke afsluttes med `followed by`.

command ::= **single-command**
| **single-command** followed by **command**

En enkelt kommando kan bestå af en `identifier ::= expression`. Dette er en såkaldt tildeling, hvilket er når vi tildeler en variabel en værdi. Variablernes navn skal altid stå på venstresiden af en tildeling og udtrykket på højresiden.

MOCCA indbefatter også kontrolstrukturerne `while` og `if`. Disse to er opbygget således, at de består af et udtryk, der skal evalueres, inden kroppen udføres. `if` og `while` giver lidt sig selv, da de læner sig op af den betydning, de har i eksempelvis C#. Vi har valgt at holde fast i disse, da de i sig selv giver betydningen til kende. `for` løkker er ikke umiddelbart intuitive at forstå, og vi mener derfor ikke, at de vil bidrage fornuftigt til sproget med den pågældende målgruppe. Samtidig vælger vi ikke at have `else` med til `if`. Dette gøres for at mindske kompleksiteten af sproget.

Det er muligt at erklære variable inden i kommandoer, ved hjælp af kommandoen `let declaration in begin command end`. Dette skaber en ny blok og giver mulighed for at have lokale variable. Hvis en variabel er erklæret inde i en `let declaration`, kan den kun tilgås inde i den blok.

Kommandoen `call` er et kald til en funktion, der er identificeret på dens identifier. Identifieren skal efterfølges af de aktuelle parametre. I MOCCA skelnes der mellem aktuelle og formelle parametre, da funktionerne benytter sig af *call by value* strategien. De formelle parametre er de kendte parametre i funktionens definition. De aktuelle parametre er dem, der sendes videre til funktionen på kaldstidspunktet. De aktuelle parametre sendes som

værdier videre til funktionen. Hvis der er flere af dem, skal de adskilles af et `and`. Der er ikke mulighed for at bruge udtryk direkte som parametre. Dette mener vi ville gøre sproget mere komplekst, da brugen af ren tekst i forvejen ikke tydeliggør nok, hvad der er parametre.

Da MOCCA ikke tillader, at vi bruger andet end variable i parametrene, er vi nødt til at erklære en variabel på forhånd, lægge en værdi over i den ved hjælp af en tildeling og derefter bruge den.

MOCCA indbefatter funktionen `print`, der printer en værdi, hvilket er basalt, hvis man ønsker at udskrive tekst til en konsol.

Samtidig kan MOCCA returnere en værdi, hvilket er helt basalt, hvis man ønsker rekursion.

Resten af terminalsymbolernes betydning knytter sig til erklæringer, tildeling og funktioner. Alle erklæringer indkapsles af `let` og `in`.

```
single-command  ::= identifier assignment expression
                  | if expression begin command end
                  | begin command end
                  | while expression begin command end
                  | let declaration in begin command end
                  | call identifier actual-parameters
                  | return identifier
                  | print identifier


actual-parameters::= ε
                  | identifier
                  | integerlit
                  | identifier and actual-parameters
                  | integerlit and actual-parameters
```

Udtryk og Operatorer

Et udtryk kan være en enkelt variabel. Samtidig kan det også være en kombination af variable og heltal adskilt af en operator. Et sådant udtryk vil eksempelvis kunne være `x add 1`. I MOCCA kan et udtryk dog også være en streng, der fortæller compileren eksplicit, at den skal håndtere værdien som en tekststreng.

Derudover kan et udtryk også være et `call` udtryk, hvilket er nødvendigt, hvis rekursion skal kunne forekomme. Et eksempel kunne være: `y assignment call Fibonacci y add Fibonacci x`, hvor vi lægger de to funktioners returneringstyper sammen og tildeler værdien til variablen y.

```
expression        ::= primaryexpression
                    | expression (operator primaryexpression)*

primaryexpression ::= integerlit
                    | identifier
                    | string begin graphic* end
                    | call identifier actual-parameters
```

Operatorer i Mocca understøtter mange af de mest basale operatorer til beregning af aritmetiske udtryk samt `modulo` operationen.

```
operator          ::= add
                    | subtract
                    | multiply
                    | divide
                    | greater than
                    | less than
                    | less or equal
                    | greater or equal
                    | is
                    | assignment
                    | differs from
                    | modulo
```

De aritmetiske tegn $+$, $-$, $*$ og / bliver til `add`, `subtract`, `multiply` og `divide`. Vi vælger altså blot at angive dem ved deres fulde navne. Her har der ikke været grund til at opfinde andre.

Det samme gælder for de logiske operatorer. $>$ bliver til `greater than`, $<=$ bliver til `less or equal` osv.

Der har været flere overvejelser, når det gælder de specialtegn, der i et sprog som C# giver struktur. Det drejer sig for eksempel om parenteser. I C# bruges de til at fortælle at noget starter og slutter, alt efter hvilken vej de vender. Det er den tankegang, vi følger, når vi vælger at terminalsymbolet `begin` i MOCCA, har den samme betydning som en startparentes, og `end` gør det ud for en slutparentes.

Kommentarer

MOCCA skal også understøtte kommentarer, så det er muligt for programmøren at lave kommentarer i kildekoden. Almindeligvis indkapsles kommentarer direkte i kode også af nogle specialtegn eller starter med nogle og afsluttes med det specialtegn, der udgør linjens afslutning. Vi har valgt at indkapsle kommentarer med terminalsymbolerne `comment` og `endcomment`. Betydningen fremgår klart af disse terminalsymboler. Vi vurderer chancen for, at en MOCCA programmør i en kommentar vælger at bruge `endcomment`, og derved afslutte kommentaren for tidligt og gøre programmet sårbart for fejl, som værende usandsynlig. Alt hvad der står i kommentarblokken bliver fjernet, når compileren kører. Dette har altså ingen betydning for det kørenede program. Det betyder dog, at man ikke kan bruge ordene `comment` og `endcomment` i koden udover ved kommentarer.

comment ::= comment **graphic**$*$ endcomment

graphic ::= letter
| digit
| space

2.3 Kodeeksempler

Følgende afsnit viser nogle små kodestumper, der er skrevet i MOCCA. Interessante ting vil i den forbindelse blive fremhævet og forklaret. Der kan ses flere komplette kodeeksempler i bilag D, herunder "Hanois Tårne" og "Euklids Algoritme".

Hello World

Den klassiske begynder-kodestump "Hello World" skal selvfølgelig med som eksempel på MOCCA kode. Det giver et førstegangsindtryk af, hvordan MOCCA kan bruges. Der illustreres hvordan en tekststreng udskrives. Det bør her bemærkes, at strengen skal tildeles til en variabel, og først derefter kan variablen udskrives. Variablen skal erklæres mellem `let` og `in`, før den kan bruges. Samtidig vises, hvordan en kodeblok er afgrænset af `begin` og `end`.

```
1 let
2   y
3 in
4   begin
5     y assignment string begin HELLO WORLD end
6     followed by
7     print y
8   end
```

Fibonacci

Et andet eksempel er en algoritme til at finde det x'te Fibonacci tal. Der ses her, hvordan en funktion erklæres i linie 3 til 23 og efterfølgende kaldes i linie 26. Her er x en formel parameter, der angiver, hvilket nummer fibonaccital der ønskes returneret. Der bruges to `if` sætninger. Den første vurderer om x er mindre end 2 og returnerer i så fald bare værdien af x. Den anden vurderer om x er lig eller større end 2 og erklærer i så fald en række variabler, hvoraf to bruges til tildeling af returnerede værdier fra rekursive kald af funktionen. De lægges så sammen og tildeles til den tredje, der så returneres. Alle variabler samt funktionen erklæres før de bruges, jævnfør scope reglerne. Mellem `if` sætningerne, og mellem tildelingerne, ses `followed by`, der bruges ved sekvenser af kommandoer.

```
1  let
2    u and
3    function Fibonacci x
4      begin
5        if x less than 2
6          begin
7            return x
8          end
9        followed by
10       if x greater or equal 2
11         begin
12           let
13             y and z and v
14           in
15             begin
16               y assignment x subtract 1
17               followed by
18               z assignment x subtract 2
19               followed by
20               v assignment call Fibonacci y add call Fibonacci z
21               followed by
22               return v
23             end
24         end
25     end
26 in
27   begin
28     u assignment call Fibonacci 10
29     followed by
30     print u
31   end
```

2.4 Konklusion

Målet med dette kapitel var at beskrive sproget MOCCA. Dette blev gjort ved hjælp af en uformel og en formel specifikation. Den uformelle specifikation skulle stille løse rammer op for sproget. Den formelle skulle samle op på dette og gøre det konkret.

Den uformelle specifikation satte nogle generelle retningslinier op for sproget. Målet for MOCCA var at stille et sprog op, som en relativt uerfaren programmør kunne bruge. Her tænkes især på målgruppen. Løsningen er et højniveau sprog, der lægger vægt på abstraktion og kontrolstrukturer. Samtidig er sproget designet på en måde, så det undgår specialtegn, da der lægges vægt på et sprog, der skal være let at skrive men også let at forstå.

Den formelle specifikation samlede op på de ellers løse koncepter fra den uformelle specifikation. Den stillede konkrete scope- og syntaksregler op for sproget. MOCCA benytter sig af en såkaldt nested block structure, der fungerer på den måde, at en variabel er lokal i forhold til den blok, hvor den er erklæret, men global for indre blokke. MOCCA kræver, at en variabel er erklæret eksplicit, før den kan bruges.

Selve syntaksen af sproget er bygget op af syntaksregler. En række valg er foretaget i forhold til syntaksen. Først og fremmest er MOCCA et lille og imperativt sprog. Derfor findes der ikke klasser, interfaces, datastrukturer og lignende., der ellers kendetegner nogle af de større sprog som C#. Dog indbefatter MOCCA generel abstraktion så som en række kontrolstrukturer.

For at gøre MOCCA mere læse- og skrivevenligt er der lagt vægt på at erstatte specialtegn. Derfor er de fleste specialtegn, som man kender dem fra eksempelvis C#, lavet om til ren tekst. Eksempelvis er operatoren "+" lavet om til `add`, hvilket også er en intuitiv måde at udtale det på. Derudover er knapt så intuitive specialtegn som "{" og "}" lavet om til `begin` og `end`.

Syntaksreglerne afspejler et sprog. Derfor er der stillet små kodeeksempler op, for hvordan MOCCA kan bruges. Dette vil blive brugt til at have noget konkret at holde udviklingen af compileren op imod. Her tænkes blandt andet på test.

KAPITEL 3

COMPILERKONSTRUKTION

Dette kapitel beskriver opbygningen af en compiler for sproget MOCCA samt hvilke valg der er foretaget i forhold til dette. Sproget, der vil blive oversat til, er C#. Det er også det sprog, vi har valgt at skrive compileren i. Før konstruktionen af en compiler kan begynde er der en række overvejelser man skal have fastlagt. Dette inkluderer hvilken type compiler man laver, hvilken tilgang man har til opbygningen af compileren, og hvordan man har tænkt sig at oversætte.

Vi har valgt en tilgang med fire grundfaser i oversættelsen. De fire faser er leksikalsk analyse, syntaktisk analyse, kontekstuel analyse og kodegenerering. Vores lærebog i SPO [Watt and Brown, 2007] lægger op til en tilgang med 3 faser, hvor leksikalsk analyse sammen med parsing udgør den syntaktiske analyse. Vi mener dog at leksikalsk analyse i sig selv indeholder stof nok til at være en fase for sig.

At faseopdele en compiler er en generel tilgang. I hver fase træffer vi dog en række valg i forhold til MOCCA, og der er blandt andet en række særlige tilfælde, vi i kapitlet synes er interessant at kigge nærmere på. Hver fase udgør et afsnit. Her vil selve implementeringen af faserne være beskrevet med kode for de mest interessante dele fra hver fase.

På adressen `http://mocca.prosphere.eu` kan følgende findes:

- Hele koden til compileren.
- En udgave af compileren, der kan eksekveres fra kommandoprompt.
- Vejledning til anvendelsen af compileren.

3.1 Overvejelser

Før konstruktionen af en compiler kan begynde, er der en række overvejelser, man skal have gjort sig. Dette inkluderer hvilken type compiler man vil lave, og hvordan man har tænkt sig at oversætte. Her gør en række faktorer sig gældende. Følgende afsnit handler om disse faktorer, samt hvilke valg vi har truffet i forhold til disse.

3.1.1 Oversættelse

Pointen med at lave et nyt sprog er, at man som udgangspunkt gerne vil have den kode, som man har skrevet, lavet om til noget eksekverbart. Generelt er der to måder man kan gøre dette på. Man kan bruge en fortolker eller en compiler. Begge er programkode, men de har dog hver deres fordele og ulemper. Dette projekt fokuserer på compileren, men for god ordens skyld vil begrebet fortolker også kort blive forklaret.

En *compiler* er et program der tager noget programkode, altså kildekode, og oversætter det til et semantisk ækvivalent program, altså målkode. Formålet med dette kunne være at lave et program, der er skrevet til en bestemt platform, om til noget programkode, der kan eksekveres på en anden type platform.

Dette er dog ikke uden ulemper. Før et program kan eksekveres på en vilkårlig platform, skal det oversættes til maskinkode, som er den type kode, den pågældende platforms underliggende maskine forstår.

Den anden mulighed er at man kan bruge en *fortolker*. Pointen i en fortolker er at man i stedet for at oversætte hele programmet, henter én instruktion af gangen og eksekverer denne. Selv om vi her vælger at konstruere en compiler, vil der i sidste ende også være en fortolker i spil, når et MOCCA program eksekveres. Langt nede i abstraktionslagene over maskinkoden, vil der på et tidspunkt kun blive oversat én instruktion af gangen.

Det at oversætte dækker sædvanligvis over at oversætte programkode fra et højniveau sprog til et sprog på et lavere niveau, som regel med det formål at få programkoden eksekveret.

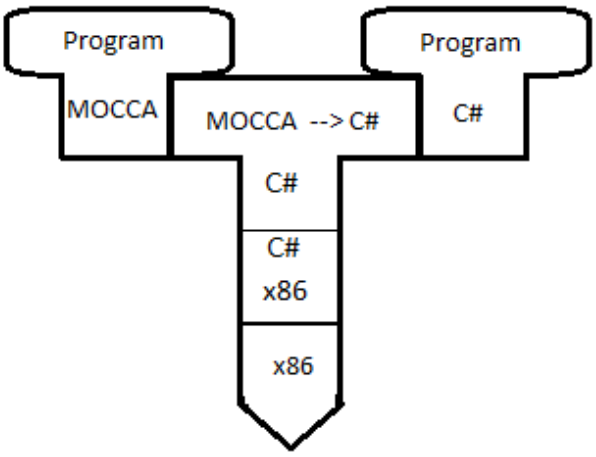
Hvis man oversætter programkode til et sprog, der har samme niveau, hedder det at krydscompile. Det er brugbart, når man ønsker at gøre sit program eksekverbart på flere forskellige platforme.

Oversættes der programkode fra lavniveau til højniveau kaldes det at decompile. Dette kan bruges i forbindelse med programanalyse. Det kan være programmer, man har mistet den oprindelige programkode til eller vil have undersøgt sikkerhedsrisikoen ved og hensigten med.

I forhold til dette projekt vil der blive krydscompilet. Der vil blive oversat til C# kode, der har et højere niveau end MOCCA, jævnfør højniveau karateristikkerne fra forrige kapitel, (afsnit 2.1). De klassificeres dog begge som højniveau sprog, da vi opfatter det at decompile som værende på tværs af sprogklasser, for eksempel fra maskinkode til højniveau sprog.

Målet med compileren er at skabe noget, der kan oversætte sproget MOCCA til en type sprog, der allerede har en eksisterende compiler, der kan oversætte til maskinkode. I vores MOCCA-compilers tilfælde bliver det altså til C# kode. Vi vurderer det som værende en fordel at oversætte MOCCA kode til et sprog, vi i forvejen kender. Pointen er, at vi gør oversættelsesprocessen nemmere for os selv. Ellers skulle vi til at sætte os ind i et lavniveau sprog, hvilket vi anser som værende uden for rammerne af dette projekt.

Måden, hvorpå koden skal oversættes, kan beskrives ved hjælp af et såkaldt *Tombstone diagram* [McKeeman, Wortman and Horning, 1970]. Diagrammet består af en række brikker, som kan bruges til blandt andet at beskrive en compiler. I figur 3.1 kan et sådant diagram for MOCCA-compileren ses.



Figur 3.1: *Dette billede viser compiler designet for MOCCA-compileren. Det viser, at et program skrevet i MOCCA kan compiles til C# ved hjælp af en compiler skrevet i C#. Da der allerede findes en compiler til C#, kan vi undgå at skrive en compiler der oversætter direkte til maskinkode, der kan eksekveres på en processor af x86 arkitekturen.*

3.1.2 Compilerdesign

Udover at tage stilling til hvilket sprog compileren skal skrives i, er det også nødvendigt at tænke over, hvilken type compiler man vil lave, og hvordan den oversætter. Det første og sikkert mest indlysende valg, når man skal designe compileren, er hvilket sprog, den skal skrives i. Her falder det mest naturligt at vælge C#, da vi allerede har kendskab til dette sprog.

Det næste valg, der bør træffes, er hvilken type af compiler man vil lave. Compilere kan fokusere på forskellige ting. Det kan for eksempel være portabilitet, således at compileren kan oversætte til flere forskellige typer platforme. Det kan også være fokus på, hvornår eksekveringen af programmet starter. Ved at kombinere ideer fra compilerens og fortolkerens verden kan man sætte eksekveringen i gang, før et program er oversat. Dette betyder dog ikke nødvendigvis, at programmets fulde afvikling sker hurtigere. For compilere af disse typer er strategien og fokuset i opbygningen et andet, end ved den compiler vi vil opbygge.

Den compiler, vi vil opbygge, fokuserer kun på oversættelse. Portabilitet er ikke vigtigt for os, da C# allerede nu er valgt som målsprog. Eksekveringen af et MOCCA program sker, når C#s compiler har færdiggjort oversættelsen. Størrelsen og typerne af de programmer, som denne version af MOCCA bruges til at udforme, vinder ikke noget ved at eksekvering begynder før. Strategien for opbygningen af vores compiler er, at vi inddeler oversættelsen i faser. Faserne er følgende:

1. **Leksikalsk analyse.** Dette er den første fase. Her scannes MOCCA programkoden. Selve redskabet til at skanne programkoden kaldes en lex Lexeren identificerer de ord i programkoden, den møder. Ordene kaldes tokens. Således vil MOCCA koden `x assignment 42` identificeres med følgende tokens:

   - En **identifier** = x
   - Et **assignment** = assignment
   - En **integerlit** = 42

2. **Syntaktisk analyse.** Dette er anden fase. Her kommer MOCCAs syntaksregler ind i billedet. Derfor er det vigtigt, at lexeren har identificeret de forskellige tokens. Sammensætning af tokens kontrolleres nemlig for eventuelle overtrædelser af syntaksreglerne. Værktøjet til at kontrollere sammensætningen kaldes en parser. Endelig står fasen for en del af fejlhåndteringen, nærmere bestemt den *kontekstfrie del*. Altså den del der lægger sig til produktionsreglerne.

3. **Kontekstuel analyse.** Dette er den tredje fase. Her kontrolleres brugen af variable og typer. Derudover er en del af fejlhåndteringen inkluderet i denne fase, nærmere bestemt den del der håndterer den *kontekstsensitive del*. Den kontekstsensitive del er den del, der ikke umiddelbart kan læses ud fra produktionsreglerne. Eksempelvis at en erklæring af en variabel skal ske før den bruges.

4. **Kodegenerering.** Den fjerde og sidste fase er kodegenerering. Denne fase vil oversætte MOCCAs programkode til C# programkode.

Mens oversættelsen er i gang, vil koden være repræsenteret i noget, der kaldes et *abstrakt syntaks træ*. Et abstrakt syntaks træ (AST) er en syntaktisk trærepræsentation af den kode, der findes i kildeprogrammet. Det abstrakte kommer ind i billedet, idet det ikke er alle ting der er repræsenteret i strukturen. I en kommando bestående af `if x is 2 begin x assignment x add 1 end` vil syntakstræet kun opsamle de tokens den ikke allerede kender ud fra syntaksen. Derfor vil følgende tokens fra førnævnte kommando være repræsenteret, `x`, `is`, `2`, `x`, `add` og `1`.
Dette bliver genereret af parseren og besøges herefter både af den kontekstuelle analyse, men også af kodegenereringen.

Der skelnes mellem to typer compilere. *One-pass compilere* og *multi-pass compilere*.

1. One-pass compilere besøger programmets AST én gang for at generere kode. One-pass compileren har sine fordele, når det kommer til hastighed og den plads, der bruges til generering og opbevaring af træet. Ulempen er, at et gennemløb af træet ikke giver programkode i målsproget, der har den optimale kvalitet.

2. Multi-pass compilere besøger træet flere gange. De vinder på modularitet og fleksibilitet. Det skal forstås sådan, at opgaverne kan deles op på en måde, så man kan generere mere kvalitativ kode i målsproget. Det kan man netop fordi, at træet besøges mere end én gang.

I forhold til dette projekt vil vi udvikle en multi-pass compiler. Det er der flere grunde til. Først og fremmest vil nogle af udfordringerne, vi møder senere i dette kapitel, kræve at træet gennemløbes flere gange. Dernæst vægter vi, at programkoden i målsproget har en vis kvalitet.

3.2 Udvikling

I forhold til selve udviklingen af compileren har vi benyttet forskellige værktøjer og metoder.
Grundlæggende har vi gjort brug af to forskellige udviklingsmiljøer. Det første udviklingsmiljø har været Microsoft Visual Studio 2010. Da C# er blevet udviklet af Microsoft, er Visual Studio derfor ideelt til at kode i, idet der tilbydes mange funktioner blandt andet til debugging [Microsoft, 2011]. Ulempen ved Visual Studio er, at det er blevet udviklet til Microsofts egen platform, Windows. Det har derfor været nødvendigt også at finde et alternativ til Linux baserede styresystemer, da de fleste af gruppens medlemmer benytter sig af dette. Valget er faldet på Mono Develop. Dette er et open source alternativ til Visual Studio, der samtidig også giver mange af de fordele som Microsofts egen løsning gør [Wikipedia, 2011].

I forhold til en specifik udviklingstilgang har vi valgt at programmere objektorienteret. Den objektorienterede tilgang kender vi fra kurserne "Objekt-Orienteret Programmering og Algoritmik" (OOPA) og "System Analyse og Design" (SAD) på vores 2. semester. Opdelingen i de 4 faser synes vi passer rigtig godt med denne tilgang. Vi kan med denne tilgang indkapsle de forskellige faser og deres metoder samt forskellige ansvarsområder under disse. Ydermere giver det os mulighed for at benytte nedarvning og polymorfi, hvilket i udviklingen af vores compiler vil vise sig nyttigt.

3.3 Leksikalsk Analyse

Den første fase er den leksikalske analyse også kendt som scanneren. Formålet med denne er at identificere enkelte tokens og kategorisere dem, alt efter hvilken type det er. Det er samtidig også dens opgave at fjerne alt overskydende, det vil sige "endlines", "white spaces" (mellemrum) og kommentarer.

"White spaces" indikerer hvornår et token slutter, men sproget MOCCA benytter sig af en speciel type tokens, der indeholder mere end ét ord og derfor også indeholder et "white space". Der ses to mulige løsninger på dette.

1. Den første er at scanne alle ordene og identificere dem, for derefter at scanne dem igen og sammensætte dem, der hører sammen, og videregive dem til syntaktisk analyse. Fordelen er, at scanneren allerede har set det næste token og derfor har viden om, hvad der kommer efter. Ulempen er, at det tager tid at scanne alle tokens én gang for derefter at gennemløbe dem.

2. Den anden måde er at sammensætte dem med det samme, det vil sige at der kontrolleres efter nøgleord. Et nøgleord kunne for eksempel være, at hvis scanneren ser et `greater`, så ved den, at det token er sammensat, og derfor er det næste, der kommer, et `or` eller `than`, som det kan ses i syntaksen i bilag B. Derfor skal den scanne det og sammensætte dem til ét token og videregive dem til parseren. Fordelen er, at scanneren ikke skal scanne alle tokens to gange. Ulempen er, at man ikke umiddelbart vil kunne have nøgleord, der både er tokens i sig selv og samtidig indgår som første ord i andre tokens, da scanneren ikke ville kunne se forskel. For eksempel `is` og `is not`.

Den anden metode er valgt, idet det vil spare tid, fordi parseren ikke skal se alle programmets tokens igennem.

3.3.1 Implementation

Som nævnt tidligere er det scannerens opgave at identificere tokens og smide alt overskydende kode væk. Meget af dette arbejde er relativt let ved blot at søge efter "white spaces", "endlines" eller andre ting, man ikke ønsker videreført til parseren. Den vigtigste opgave, scanneren har, er at identificere tokens. Dette vil vi kigge nærmere på her.

Når scanneren ser kildekoden, ved den ikke, hvornår et token er slut. Derfor er det givet i sproget, at et token slutter, når der kommer et "white space". Det er så at sige det eneste den skal bruge til at adskille dem. Det eneste tilfælde, hvor dette ikke gælder, er når vi bruger vores "dobbelt token" operatorer.

Det er en del af scannerens opgave at rydde op i koden, således at vi kun står med en ren tekststreng. Dette betyder, at den skal rydde "endlines" af vejen, der kun er til for at gøre et program let læseligt for programmøren. Eksemplet nedenfor viser før og efter lexeren har scannet koden.

Før:

```
1 let
2   function Random x
3         begin
4             return x
5         end
6 in
7 call Random 4
```

Efter:

```
1 let function Random x begin return x end in call Random 4
```

Herefter beskriver vi "kernen" i den leksikalske analyse. Den leksikalske analyse består af metoden `Scan()`, der står for det centrale i opbygningen af tokens. Metoden udliciterer arbejdet til flere forskellige metoder, der hver især står for deres del af den leksikalske analyse. Selve implementationen kan ses i kodeeksemplet på næste side.

```
1  public Token scan()
2  {
3    Token tok;
4    String newhold;
5    int kind;
6
7    currentlyScanningToken = false
8
9    while (currentChar == '\n'
10         || currentChar == ' '
11         || currentChar == '\r'
12         || currentChar == '\t')
13       ScanSeparator();
14
15   currentlyScanningToken = true;
16
17
18   currentSpelling = new StringBuilder("");
19
20   kind = ScanToken();!
21
22    if (currentSpelling.ToString() == "followed"
23         || currentSpelling.ToString() == "greater"
24         || currentSpelling.ToString() == "less"
25         || currentSpelling.ToString() == "is")
26      {
27          newhold = currentSpelling.ToString();
28          tok = scan2(newhold, pos);
29      }
30
31   else
32      {
33         if (currentSpelling.ToString() == "add"
34             || currentSpelling.ToString() == "iss"
35             || currentSpelling.ToString() == "subtract"
36             || currentSpelling.ToString() == "multiply"
37             || currentSpelling.ToString() == "divide"
38             || currentSpelling.ToString() == "modulo")
39            { kind = (int)Token.Tokens.OPERATOR; }
40
41         else if (currentSpelling.ToString() == "assignment")
42                 { kind = (int)Token.Tokens.ASSIGNMENT; }
43         else if (currentSpelling.ToString() == "let")
44                 { kind = (int)Token.Tokens.LET; }
45         else if (currentSpelling.ToString() == "in")
46                 { kind = (int)Token.Tokens.IN; }
47         else if (currentSpelling.ToString() == "and")
48                 { kind = (int)Token.Tokens.AND; }
```

```
49          else if (currentSpelling.ToString() == "begin")
50                  { kind = (int)Token.Tokens.BEGIN; }
51          else if (currentSpelling.ToString() == "while")
52                  { kind = (int)Token.Tokens.WHILE; }
53          else if (currentSpelling.ToString() == "end")
54                  { kind = (int)Token.Tokens.END; }
55          else if (currentSpelling.ToString() == "function")
56                  { kind = (int)Token.Tokens.FUNCTION; }
57          else if (currentSpelling.ToString() == "string")
58                  { kind = (int)Token.Tokens.STRING; }
59          else if (currentSpelling.ToString() == "print")
60                  { kind = (int)Token.Tokens.PRINT; }
61          else if (currentSpelling.ToString() == "call")
62                  { kind = (int)Token.Tokens.CALL; }
63          else if (currentSpelling.ToString() == "return")
64                  { kind = (int)Token.Tokens.RETURN; }
65
66          tok = new Token(kind, currentSpelling.ToString());
67      }
68    return tok;
69  }
```

Den første del står for indlæsning og opbygning af tokens. Metoden kontrollerer det pågældende symbol, så længe det er et af de symboler der skal sorteres fra kaldes metoden `ScanSeparator()` (linie 13). Denne metode indlæser det næste symbol og returnerer det. Da der kan være mere end et space, tab eller newline i kildekoden er dette implementeret i en while løkke (se linie 9) der først afbrydes hvis det nye symbol der indlæses ikke er et af de dem der skal sorteres fra.

Hvis det pågældende symbol er i sproget og ikke skal sorteres fra bliver `currentSpelling`, der er en StringBuilder (linie 18) initialiseret og metoden `ScanToken()` (linie 20) bliver kaldt. Denne metode sørger for at indlæse hele tokens, indtil den støder på et space.

Den anden del står for at bestemme et tokens type. Et tokens type kan bestemmes ud fra dets stavemåde. I forhold til MOCCA er det interessant, at der er tokens, der indeholder et space. Derfor kontrolleres der om det token, der lige er scannet, er et af de udvalgte, der skal sættes sammen. Dette implementeres ved hjælp af en simpel `if` sætning (linie 22). Hvis det er et af disse nøgleord kaldes metoden `scan2()` (linie 28), der sørger for at scanne det næste token på samme måde som det første. Det første token og det andet token sættes sammen og returneres som ét token, der indeholder et space. Samtidig bliver typen sat til `operator` (linie 33-38), idet det kun er operatorer, der kan være disse specielle tokens. Dernæst returneres det nyindlæste token til parseren.
Hvis ikke det pågældende token er et af de specielle tokens, fastslås typen (linie 41-64) og returneres til parseren (linie 68).

3.4 Syntaktisk Analyse

Den anden fase af compileren hedder den syntaktiske analyse. Denne fase koncentrerer sig om at fastslå kildekodens struktur. Denne fase bliver kaldt for parsing og står for den kontekstfrie del af programmet. Det vil sige, den fanger fejl, der kan fanges ved at følge grammatikreglerne.

Selve parseren kan laves på forskellige måder, *top down* eller *bottom up*. Følgende vil kort beskrive de to undersøgte metoder samt et valg heraf. Begge metoderne lægger sig op af at bygge et abstrakt syntaks træ.

1. Den første hedder bottom up parsing. Denne metode undersøger bladene (terminalsymbolerne) og opefter imod rod noden.
2. Den anden metode kaldes top down. Denne metode bygger et syntakstræ oppe fra roden og ned imod bladene (terminalsymbolerne).

En bestemt type parsing vil blive brugt i parseren for MOCCA. Det vil være en top down parser med en bestemt algoritme kaldet *recursive descent*. Fordelen i denne parser er, at den er forholdsvis simpel at opbygge direkte fra syntaksen.

Formålet med denne algoritme er at inddele parsingen i en række metoder (én for hver nonterminal). I MOCCA-compileren vil der være mange af disse metoder. Når disse metoder arbejder sammen, så kan hele programmer parses.

En recursive descent parser kræver at sprogets grammatik opfylder et krav, nemlig at det skal være LL(1). Termen LL(1) dækker over, at sprogets syntaks skal definere at udtryk evalueres fra venstre mod højre. Derfor skal grammatikken eliminere venstrerekursion. Ellers vil grammatikken i forhold til LL(1) være tvetydig og parseren vil altså ikke være i stand til at konstruere et korrekt syntaks træ. MOCCAs syntaks er på forhånd lavet således, at det opfylder LL(1).

Idet vores syntaks opfylder reglerne for en LL(1) parser, kan kodningen af parseren ske ved hjælp af et værktøj. Alternativt kan man kode den selv. Der er fordele og ulemper ved begge tilgange.

1. Ved at bruge et værktøj, så som YACC [Johnson, 2011] eller SableCC [SableCC, 2011], kan man opbygge parseren automatisk. Den klare fordel ved at bruge et værktøj er selvfølgelig, at man kan spare tid. Den klare ulempe er, at man oftest skal bruge en del tid på at sætte sig ind i disse værktøjer og i forhold til vores projekt tabes meget af læringsprocessen.
2. Fordelen ved at kode parseren selv er, at man selv har styr på, hvilken kode der er underliggende, og at man i forhold til vores projekt gennemgår en læringsproces.

Valget er faldet på selv at opbygge parseren, idet det vil tage lang tid at sætte sig ind i værktøjerne, men også fordi vi gerne skulle komme ud på "den anden side" og stå med en reél idé om, hvordan en parser skal opbygges.

3.4.1 Implementation

Parseren for MOCCA er blevet implementeret som en recursive descent parser. Undervejs vil der blive opbygget et AST for programmet.
Koden herunder definerer strukturen af et program i compilerens AST. Dens constructor tager en erklæring og en kommando som input. Derudover tager den en position - dette bliver brugt i fejlhåndtering. Erklæringen repræsenterer et antal identifiers, der i MOCCAs tilfælde skal erklæres, før man kan bruge dem i den efterfølgende kommando. Kommandoen repræsenterer en kommando i programmet.

```
1  public class Program : AST
2      {
3          public Command C;
4          public Declaration D;
5
6          // Et program består af en let command,
7          // med en deklaration og en command.
8
9           public Program
10         (Declaration dAST, Command cAST, SourcePosition thePosition)
11                 : base(thePosition)
12          {
13             D = dAST;
14             C = cAST;
15          }
16
17         public override Object Visit(Visitor v, Object o)
18         {
19             return v.VisitProgram(this, o);
20         }
21
22     }
```

Parseren kalder lexeren, nærmere bestemt metoden `Scan()`, når den skal bruge et nyt token. Hele parseren er baseret på syntaksen af MOCCA, det vil sige at så længe koden følger syntaksen, så opfører parseren sig som den skal. Koden på næste side viser metoden `ParseProgram()`. Det er den første parse metode der kaldes, når parseren kaldes.

```
1  public Program ParseProgram()
2  {
3     Program programAST = null;
4
5     currentToken = lexicalAnalyser.scan();
6
7    if (currentToken.spelling == "let")
8     {
9       accept((int)Token.Tokens.LET);
10
11      Declaration dAST = ParseDecSeq();
12
13       while (currentToken.spelling == "in")
14        {
15         accept((int)Token.Tokens.IN);
16
17             if (currentToken.spelling == "begin")
18             {
19              accept((int)Token.Tokens.BEGIN);
20              Command cAST = ParseCommand();
21              accept((int)Token.Tokens.END);
22
23              programAST = new Program(dAST, cAST);
24             }
25
26        }
27    return programAST;
28 }
29
30 else
31    {
32
33    errorReporter.ReportError
34         ("% Ikke forventet token:", currentToken.spelling);
35
36    return programAST;
37    }
38
39 }
```

Ifølge syntaksen, der er beskrevet i afsnit 2.2.3, er det første token, der vil forekomme i et program, altid nøgleordet `let`. Derfor er det selvfølgelig oplagt at kontrollere, om et sådant nøgleord findes her. Hvis det gør, så skal programmet acceptere den, det vil sige, at den skal scanne det næste token. Derefter kaldes parsemetoden `ParseDeclaration()`, der returnerer et objekt af typen `Declaration`, jævnfør syntaksen. Resten af metoden burde give sig selv. Til sidst returneres det af parseren opbyggede træ.

Ligesom dette eksempel er resten af parseren bygget op på en lignende måde. På flere forskellige steder kontrollerer parseren, om det token, den rent faktisk kigger på, er det token, den forventer. Dette sker blandt andet i `accept()` metoden fra overnævnte eksempel. Hvis ikke skal den returnere en fejl. Fejlhåndtering vil blive forklaret i afsnit 3.5.

3.4.2 Visitor Design Mønster

Et mønster der går igen i de to sidste faser af MOCCA-compileren, er det såkaldte *visitor design mønster* [Watt and Brown, 2007]. Dette mønster er en måde, hvorpå man kan separere bestemte algoritmer fra resten af koden. Derved fås en kode, der er mere overskuelig og samtidig følger det objektorienterede paradigme. I forhold til MOCCA-compileren vil det sige at man samler alle metoder, der har med kontekstuel analyse at gøre, i én klasse og samler alle metoder, der har med kodegenerering at gøre, i en anden klasse. Alternativet til Visitor mønstret ville være at have hver metode i sin respektive klasse.

Selvom der er meget, der taler for Visitor mønstret, er der også nogle negative sider. Den største værende at man er nødt til at have en såkaldt "callback" metode inde i hver klasse, hvor man ønsker at implementere dette visitor mønster. Det vil sige at metoderne i realiteten ikke bliver helt separeret fra hinanden.

Visitor er dog valgt til compileren, da vi mener, at det i høj grad bidrager til overskueligheden at have metoderne samlet på ét sted i stedet for spredt.

I MOCCA compileren er mønstret implementeret således, at hver subklasse af AST arver visitor metoden (linie 17-20, i eksemplet på side 45). Dette er den såkaldte callback metode, der sørger for at kalde tilbage og besøge bladene på subtræet. Endvidere findes der en klasse, `Visitor`, der definerer vores visitor pattern. I denne klasse er der defineret en metode for hver klasse, der kan være i AST'et. Enhver klasse, der arver fra dette mønster, skal implementere alle metoder. Implementationen af Visitor kan ses i klasserne `Visitor`, `Checker` og `Encoder`.

3.5 Kontekstuel Analyse

Efter compileren har afsluttet den syntaktiske analyse, skal den kontrollere, at den statiske semantik er i orden. Den kontekstuelle analyse står for den kontekstsensitive del af grammatikken. I MOCCA-compilerens tilfælde vil det sige at kontrollere, om variable er erklæret, før de bruges. Dette er altså dét, der ikke kan læses ud fra syntaksen af sproget.

I traditionel kontekstuel analyse vil tilgangen ligesom i MOCCAs tilfælde være at kontrollere variable, men også at kontrollere typer. Hvis man forsøger at addere en heltals type og en string type, er der grundlag for en fejlmeddelelse i compileren, idet dette ikke er hensigtsmæssigt. Dette vil dog ikke være tilfældet for denne compiler, da typer ikke bruges i denne forstand.

Den kontekstuelle analyse kontrollerer det AST, der var et resultat af outputtet fra den syntaktiske analyse, og er relativ let at implementere i compileren.

3.5.1 Implementation

MOCCA har ingen typer, derfor er det eneste vi skal gøre at kontrollere om en variabel er erklæret. Eksemplet herunder er et godt eksempel på, hvad den kontekstuelle analyse skal kunne håndtere.

```
1  let
2    function Fibonacci x
3      begin
4        if x less than 2
5          begin
6
7            ...
8
9          end
10     end
11 in
12   begin
13     call Fibonacci 10
14   end
```

MOCCA benytter sig af variable, der skal være erklæret før brug. I ovenstående kodeeksempel kan der ses, at en erklæring kun gælder for en såkaldt `let-in` kommando. Det vil sige, at hvis ikke variablen eksplicit er nævnt her eller i et scope udenfor det aktuelle scope, skal der returneres en fejl.

Helt konkret er der to sider af den kontekstuelle analyse:

- **Fejlhåndtering**

  Det første, der må tages hånd om, er fejlhåndtering. Idet MOCCA-compileren gerne skulle give konstruktive fejlmeddelelser, skulle der gerne gives en besked om, hvor fejlen kan være. Dette vil også hjælpe med til debugging af programmerne. Klassen `ErrorReporter` tager sig af dette. Denne klasse indeholder metoden `ReportError`, der definerer en fejlmeddelelse. En fejlmeddelelse består af en streng og et navn på et token. Når man kalder metoden vil der blive udskrevet en meddelelse i konsollen i stil med: *ERROR: Denne variabel er ikke erklæret: .. y.*
  Selve implementationen af metoden kan ses i nedenstående kode.

```
1  public void ReportError(String message, String tokenName)
2          {
3              Console.Write("ERROR: ");
4
5              Console.WriteLine(" " + message + ".." + tokenName);
6
7              numErrors++;
8          }
```

- **Kontrol af variable**

  Den anden del af den kontekstuelle analyse er implementeret i klassen "checker" og kontrollerer efter fejl ved brugen af variable.

  Kontrollen bliver udført ved hjælp af Visitor mønstret. Når metoden "Check", der tager et AST som input, kaldes, besøger denne hele syntaks træet. Dette kan ses i nedenstående stykke kode:

```
1  public void Check(Program ast)
2          {
3              ast.Visit(this, null);
4          }
```

  Når en variabel bliver erklæret, skal den skrives i en tabel der indeholder variabelnavne. I variabeltabellen er der et felt til navnet og et felt til det scopelevel, hvor den er erklæret i. Hver node på syntakstræet svarer til en af vores metoder i vores Visitor klasse. Derfor gælder det om at konstruere metoderne på en fornuftig måde.

  Vi ved, at når der optræder en erklæring, skal vi skrive variablen ind i vores tabel og når vi bruger en variabel, skal vi kontrollere om den findes. Vi kan let implementere dette, idet vi bare bruger det før omtalte Visitor mønster. Nedenfor er vist et eksempel på en indskrivning af en værdi i tabellen. Hvis vi besøger en funktions erklæring, skriver vi dens navn ind i tabellen(linie 5) i det pågældende scope.

Da en funktion også består af parametre og en kommando, jævnfør syntaksen i bilag B, besøger vi også disse(linie 6). De andre metoder, der knytter sig til erklæringer, er implementeret på nogenlunde samme måde. Det vil sige, at vi skriver navnet i tabellen, når vi ser en ny erklæring.

```
1  public Object VisitFuncDeclaration
2                          (FuncDec ast, Object o)
3          {
4
5              idTable.enter(ast.I.spelling, ast);
6              ast.P.Visit(this, null);
7              idTable.OpenScope();
8              ast.C.Visit(this, null);
9              idTable.CloseScope();
10
11             return null;
12
13         }
```

På tilsvarende måde kan vi bruge Visitor mønstret til at kontrollere om en variabel findes. Når vi bruger en variabel kan vi kalde metoden `retrieve` til at søge igennem den tabel, der allerede er lavet. I eksemplet herunder ses et sådant eksempel. I en tildeling tildeler vi en variabel en værdi, derfor besøger vi tabellen for at se, om den eksisterer(linie 6). Hvis variablen ikke eksisterer vil værdien `null` returneres, hvorefter et kald til `ReportError`(linie 10) udskriver en fejl med oplysning om, hvilken variabel det drejer sig om.

```
1  public Object VisitAssignCommand
2                  (AssignCommand ast, Object o)
3  {
4
5
6      Declaration binding = (Declaration)ast.V.Visit(this, null);
7
8      if (binding == null)
9        {
10           reporter.ReportError
11         ("\"%\" Denne variabel er ikke erklæret:", ast.V.spelling);
12       }
13     return null;
14
15 }
```

3.6 Kodegenerering

De fleste compilere oversætter kildekode til et sprog med lavere abstraktionsniveau, for eksempel assembler kode. Men for at kunne generere eksekverbar kode til MOCCA vil compileren oversætte til sproget C#, der så kan oversættes til maskinkode via C# compileren.

Fordelen er at det er betydeligt lettere at oversætte til et sprog med højere abstraktionsniveau. Her menes der, at det kan være betydeligt færre valg, der skal træffes. Den klare ulempe er, at man ikke selv står for at designe, hvordan et program eksekveres på maskinniveau, og derfor ikke er i kontrol. MOCCA er derfor nødt til at eksekvere kode præcis ligesom C#, fordi valget er taget på forhånd.

Der stiller sig dog et problem op. Da MOCCA ikke benytter sig af typer, og C# i høj grad gør, stilles der en udfordring i at oversætte koden til noget, der rent faktisk har typer.

Selvom mange af delene er ligetil, er nogle dele også mere vanskelige at implementere. Følgende problem kan opstå i oversættelsen.
Idet C# ikke har helt de samme regler angående variable som MOCCA, er det nødvendigt at skænke dette en tanke. MOCCA gør ikke brug af eksplicitte typer, det gør C#. Der opstår et problem i oversættelsen, i og med at vi ikke kan kontrollere indholdet af en variabel på en fornuftig måde. Dette gør sig gældende for både variablers og funktioners returneringstype. Typerne på disse kan i C# være repræsenteret på formen `public Int x`, der viser noget om typen af indhold, og `public Int dummyfunction()`, der viser noget om indholdet af, hvad der bliver returneret i funktionen.

Hvad vi leder efter er en generel type. Dette giver mere mening, når vi nærmer os lavere abstraktion, og typerne flyder sammen, og det hele alligevel er binær kode. På et højt abstraktionsniveau giver det dog god mening at allokere forskellig plads til variable afhængigt af indholdet. Samtidig sikrer man, at man ikke tildeler forkerte værdier.

Det er vigtigt, at der bliver taget hånd om dette problem, da et program i C# ellers ikke vil kunne oversættes.

3.6.1 Implementation

Selve implementeringen af kodegeneratoren er ligetil. Da vi krydscompiler til C# er mange af de valg, der skulle være truffet, hvis vi oversatte til et sprog med lavere abstraktionsniveau, allerede truffet. Dette betyder dog, at vi ikke har haft den store indflydelse på, hvordan programmerne skal eksekveres.

C# har en bestemt opbygning af sine programmer, der i nogen grad følger syntaksen i MOCCA. Løsningen har været at bruge det førnævnte Visitor Pattern til at besøge

syntaks træet og generere kode.

Selve udskrivningen af koden er ligetil. Ligesom vi skrev til en `StringBuilder` i den leksikalske analyse (afsnit 3.3), skriver vi også til en her. Forskellen er dog, at vi højst sandsynligt har med større tekststrenge at gøre. I koden herunder ses et eksempel på metoden `Runencoder`, der tager et AST og et filnavn med sig som parametre. I dette tilfælde er AST det, der er videregivet fra den syntaktiske og kontekstuelle analyse. Filnavnet er en sti, hvor til source filen skal skrives.

```
1 public void Runencoder(Program ast, string file)
2 {
3     targetcode = new StringBuilder("");
4
5     ast.Visit(this, null);
6
7     System.IO.File.WriteAllText(file, targetcode);
8 }
```

Metoden besøger træet ved hjælp af Visitor mønstret (linie 5). Hvor de relevante visitor metoder skriver til den oprettede tekststreng. Til slut udskrives tekststrengen i en fil (linie 7). Denne fil kan oversættes af en C# compiler til en eksekverbar fil.

Vanskeligere er problemet med at håndtere variable. Idet MOCCA benytter sig af en generel type, opstår der vanskeligheder, når der skal oversættes til de mange forskellige typer, der eksisterer i C#. Løsningen på problemet har været at tildele variable typen `object`. Typen har den egenskab, at man kan tildele en hvilken som helst værdi til den. Vi kan eksplicit caste typen, når vi skal bruge den til eksempelvis aritmetiske udtryk, hvor vi er sikre på, at vi skal bruge integers. Casting af et object til en værdi kaldes *unboxing* og casting fra en værdi til typen object kaldes *boxing* [MSDN, 2011*b*].

Nu da vi ved hvordan vi vil håndtere typer, kan vi tilpasse vores Visitor mønster herefter.

Det kan være givende at stille nogle eksempler op omkring oversættelsen. Kode eksempler afgør, hvordan kodegenerering oversætter en række kommandoer i MOCCA til målet C#. Således har vi opstillet et simpelt eksempel for hvordan MOCCA skal oversættes til C#. Dette kan ses herunder.

```
function Returnint x and y
  begin
    y assignment y add x
    followed by
    return y
  end
```

```
public object Returnint(object x, object y)
{
  y=(int)y+(int)x;
  return y;
}
```

I MOCCA koden til venstre erklærer vi en funktion `Returnint`, der tager to parametre y og x. I kroppen af funktionen tildeler vi y værdien af y + x, og returnerer derefter y.

C# koden til højre beskriver præcis det samme, men da MOCCA ikke indbefatter typer, er vi nødt til eksplicit at caste (unboxe) objekterne, når de bruges.

Således kan vores Visitor mønster tilpasses, så de respekterer denne regel.

Følgende kodestumper er eksempler fra MOCCA compileren. Disse vil fremhæve de vigtigste aspekter af, hvordan Visitor metoderne er modificeret for at kunne konstruere kode ud fra overstående eksempel.

Erklæringer

I vores kodeeksempel erklærer vi en funktion. Hvis vi ved hjælp af Visitor mønstret besøger en funktions erklæring, begynder vi at tilføje symboler til vores tekstbuffer ved hjælp af funktionen `Append()`.

Således kan vi opbygge en hel funktion ud af nogle få kald til `Visitor`, som det kan ses i koden nedenfor. Først tilføjer vi funktionsnavnet (linie 7) ved at besøge dens identifier. Dernæst tilføjer vi parametre (linie 10) og til sidst tilføjer vi kroppen ved at besøge kommandoen (linie 14). Resten af koden tilføjer blot tegn som {} og (), der overholder C# syntaksen.

```
1  public Object VisitFuncDeclaration(FuncDec ast, Object o)
2 {
3
4     targetcode.Append("public static object ");
5
6     targetcode.Append(ast.I.spelling);
7     targetcode.Append("(");
8
9     ast.P.Visit(this, null);
10    targetcode.Append(")");
11    targetcode.Append("{");
12    funcdec++;
13
14    ast.C.Visit(this, null);
15    funcdec--;
16
17    targetcode.Append("return 1 ");
18    targetcode.Append("; ");
19    targetcode.Append("}");
20
21    return null;
22
23 }
```

Der er dog to yderligere ting, man bør bemærke. Idet MOCCA ikke er objekt orienteret men nærmere deklarativ, skal alle funktioner være tilgængelige uden eksplicit at skulle lave en instans af en klasse for at kalde den [MSDN, 2011*c*]. Derfor har vi været nødt til at lave funktioner og variable, der er erklæret statiske.

Dette giver dog et problem idet at en variabel ikke kan være statisk inde i en funktion. Derfor er variablen `funcdec` blevet indført. Så snart vores Visitor mønster besøger noget, der ligger uden for det globale scope, tæller den `funcdec` op, hvilket er illustreret i ovenstående kodeeksempel. Hvis vi erklærer variable via en `let in` inde i kroppen af funktionen, laver den et check for at se, om den bør tilføje et `static` foran variablen.

Kommandoer

Idet vores eksempel også kommer forbi en erklæring er det også passende at beskrive disse. Dette gøres blandt andet i koden nedenfor.

```
1  public Object VisitAssignCommand(AssignCommand ast, Object o)
2  {
3
4
5      targetcode.Append(ast.V.spelling);
6      targetcode.Append("=");
7      ast.E.Visit(this, null);
8
9      targetcode.Append(";");
10
11     return null;
12 }
```

Da en erklæring består af en variabel, tilføjer vi denne på venstre side af et lighedstegn (linie 5). På højre side besøger vi udtrykket (linie 7). Følgende kode er et eksempel på, hvordan udtrykket fra eksemplet kan generere kode.

```
1  public Object VisitIdentifierExpression
2                          (IdentifierExpression ast, Object o)
3          {
4              targetcode.Append(ast.I.spelling);
5
6              return null;
7           }
```

Figur 3.2: *Visitor metoden* `VisitIdentifierExpression`.

I eksemplet kommer vi flere gange ud for at skulle tilføje et variabelnavn til tekstbufferen. Dette gøres ved at tilføje stavemåden til tekststrengen (linie 4).
Til slut i eksemplet skal y returneres. Dette er også ligetil og behøver ingen større introduktion, det er blot at tilføje et `return y;` til tekstbufferen.
Nu kan den færdige tekststreng `public object Returnint(object x, object y) { y=(int)y+(int)x; return y;}` skrives til en fil eller oversættes af en C# compiler.

3.7 Konklusion

Målet med det dette kapitel har været at beskrive, hvordan vi har konstrueret en compiler til sproget MOCCA, der kan krydscompile til C#, og udfordringerne ved dette. Konstruktion blev delt op i 4 faser og udviklingen er foregået objektorienteret.

Den første fase er leksikalsk analyse. Formålet med denne er at identificere de forskellige tokens i program ved at scanne dem. Scanneren i vores compiler er udvidet således, at den kan identificere tokens, der strækker over flere ord i programmet. Det er gjort ved, at det første ord i et sammensat token ikke deles med andre individuelle tokens. Således kan de identificeres og slås sammen i koden.

Anden fase er den syntaktiske analyse. Her valgte vi fremfor at bruge et værktøj selv, at kode en parser, der kunne opbygge det abstrakte syntaks træ. Parseren, vi har kodet, er en top down parser, der gør brug af recursive descent algoritmen. Det betød, at vi konstruerede en LL(1) parser, hvilket i forvejen passede med MOCCAs syntaks.

For at have en repræsentation af det abstrakte syntaks træ i de næste faser valgte vi at implementere Visitor design mønsteret. Dette gøres for at bevare den nuværende objektorienterede klasse struktur, da en hel del nye metoder, der kan skifte med syntaksen, samles her. Derved bevares overskueligheden i koden.

Den tredje fase i vores compiler er den kontekstuelle analyse. Da MOCCA er et typeløst sprog, tager den fase sig af at tjekke, om funktioner og variable er erklæret. Her bruges Visitor mønsteret. Samtidig er en fejlhåndtering implementeret således, at det bliver tilkendegivet, hvad der ikke måtte være erklæret.

Den sidste fase, kodegenerering, tager sig af at oversætte den fejlfrie MOCCA kode repræsenteret i det abstrakte syntaks træ til målsproget C#.
Da C# er et sprog med typer, og MOCCA er et sprog uden, har vi i kodegenereringen valgt at benytte den C# type, der hedder `object`. Denne type kan tildeles alle værdier. Vi bryder dog med dette, når der arbejdes med operatorer, da disse i MOCCA bruges til at arbejde med heltal. Derfor får disse værdier typen `int`.
Et andet problem har været MOCCAs brug af frie navne. Dette understøttes ikke af funktioner i C#, med mindre disse variable gøres statiske. Det har vi måtte indføre en funktion til i Visitormønsteret.

KAPITEL 4

TEST OG AFPRØVNING

Dette kapitel forholder sig til test og evaluering af vores compiler implementation af MOCCA. Der bliver anvendt forskellige kodeeksempler til at teste, at der genereres korrekt C# programkode. Det, der i sidste ende bestemmer udfaldet af testen, er altså Visual Studios C# compiler, samt om det færdige program opfører sig som forventet. Vi har valgt at teste compileren i sin helhed i stedet for at opdele testene i de forskellige faser.

I det følgende afsnit vil vi opstille rækken af test og dele dem op i henholdsvis test, der forventes at blive oversat korrekt, og test, der forventes at være problematiske. Derefter bliver resultatet af testene evalueret, og der opstilles nogle rammer for, hvad der skal gøres for at rette op på eventuelle fejl.

Vi har valgt denne tilgang til test af vores compiler, da vi ikke anser den for at være færdigudviklet. Det er derfor hensigten, at disse test skal give en idé om, hvor langt vi er fra målet med udviklingen af en compiler, der implementerer MOCCA.

4.1 Afprøvning af Kodeeksempler

Vi benytter ikke specielle værktøjer til test. Måden, testene bliver udført på, er, at vi opstiller forskellige kodeeksempler, der har fokus på forskellige aspekter af MOCCA. Disse bliver så oversat af vores MOCCA compiler til C# kode. Hvis den genererede kode kan oversættes korrekt af Visual Studios C# compiler, og det færdige program opfører sig som forventet, anser vi testen for bestået. Først gennemgår vi en række test, som vi forventer, vores compiler uden videre vil bestå. Derefter introduceres en række test, der forventes at ville belyse fejl eller mangler i vores compiler.

TEST 1: Ifølge sprogdesignet skal variable erklæres før brug. Det er naturligvis vigtigt at teste, om compileren respekterer dette. Derfor testes det program, som vises i figur 4.1. Da vi eksplicit har indført et check for dette i den kontekstuelle analyse, forventer vi ikke, at der opstår problemer i forbindelse med denne test. Compileren bør altså komme med en fejlmeddelelse om, at der bruges en ikke-erklæret variabel.

```
1  let
2    x
3  in
4    begin
5      x assignment 1
6      followed by
7      y assignment 2
8      followed by
9      print x
10     followed by
11     print y
12   end
```

Figur 4.1: *Test for om variable kan bruges uden erklæring.*

TEST 2: MOCCA bruger som tidligere beskrevet nested block structure. I forhold til tildelinger af værdier til variable, er det derfor interessant at teste, om en lokal tildeling til en variabel også har konsekvenser globalt. Der er ikke defineret eksplicitte regler for, at en lokalt tildelt værdi til en variabel kun skal gælde lokalt. Derfor forventer vi, at en tildeling gælder, indtil den bliver overskrevet, uanset om man går ud af den kodeblok, som tildelingen forekommer i. Koden i figur 4.2 tester dette.

```
1  let
2    x
3  in
4    begin
5      x assignment 1
6      followed by
7      begin
8        x assignment 2
9      end
10     print x
11   end
```

Figur 4.2: *Test for om en lokal tildeling gælder udenfor kode blokken.*

TEST 3-8: Vi tester naturligvis også de tidligere opstillede kodeeksempler fra afsnit 2.3 og bilag D. Da de er blevet tilpasset løbende i forbindelse med udviklingen af syntaksen, forventer vi, at vores compiler uden problemer vil kunne oversætte disse programmer til korrekt C# kode.

Herefter følger en række programmer, som vi forventer vil påvise problemer ved vores compiler.

Ifølge syntaksen for MOCCA burde det være muligt at lave en kommando bestående af en identifier, der tildeles resultatet af en vilkårlig lang kæde af udtryk og operatorer, uanset hvilke operatorer der benyttes. Der skelnes ikke mellem typer af operatorer i MOCCA. Der fokuseres her på følgende del af syntaksen:

```
single-command  ::= identifier assignment expression

expression      ::= primary-expression
                  | expression (operator primary-expression)*
```

TEST 9: Et nærliggende stykke testkode kunne derfor være som opstillet i figur 4.3. Det er her ikke intuitivt, hvad der egentlig bliver tildelt x i denne situation, og hvad der så efterfølgende returneres. Dette illustrerer et problem med implicitte typer i MOCCA. I kraft af at MOCCA inkluderer konstrolstrukturerne `if` og `while`, gør det også implicit brug af boolske udtryk. MOCCAs variable er dog ikke designet til at håndtere boolske typer, så det bør derfor ikke være muligt at tildele resultatet af et boolsk udtryk til en variabel.

```
1 let
2   x
3 in
4   begin
5     x assignment 1 greater than 2 less than 3
6     followed by
7     return x
8   end
```

Figur 4.3: *Test af en kæde af operatorer og udtryk.*

TEST 10: En anden ting, der tillades i syntaksen, er integers, der bruges som udtryk i `while` løkker. Ligesom ved det foregående eksempel er der her tale om et problem med implicitte typer. Det står ikke umiddelbart klart, hvordan et tal bør evalueres i MOCCA, når det bruges som et boolsk udtryk. Et eksempel på dette ses i figur 4.4.

```
1 let
2   x
3 in
4   begin
5     while 42
6       begin
7         print x
8       end
9   end
```

Figur 4.4: *Testkode for hvordan MOCCA håndterer et tal som expression i en* `while` *løkke.*

TEST 11: Som følge af manglende paranteser i MOCCA er det en regel, at udtryk konsekvent evalueres fra venstre mod højre. Det bør naturligvis testes, om vores compiler respekterer dette. Koden for dette kan ses i figur 4.5. Det forventede resultat må ifølge MOCCAs regler være at 1 først lægges til 2, og derefter ganges med 3, så der altså udskrives 9. Der er dog ikke eksplicit taget højde for denne regel i konstruktionen af vores compiler, så der kan muligvis forekomme en fejl her.

```
1 let
2   x
3 in
4   begin
5     x assignment 1 add 2 multiply 3
6     followed by
7     print x
8   end
```

Figur 4.5: *Test for om compileren overholder MOCCAs regel om at evaluere udtryk fra venstre mod højre.*

TEST 12: En anden ting, der bør testes, er, hvad der sker, hvis det samme navn erklæres flere gange. Dette er angivet i figur 4.6. Variablen x erklæres flere steder i koden men ikke i samme kodeblok. Dette burde være muligt i MOCCA. Den globale variabel y bruges ikke i selve koden i dette eksempel, men ifølge syntaksen er der nødt til at være en erklæring af noget under alle `let` i MOCCA. Da vores compiler oversætter erklæringer

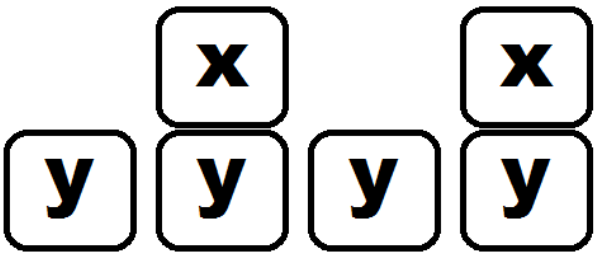
inde i koden til det samme overordnede scope i C#, forventer vi, at der kan opstå en fejl her.

```
1  let
2    y
3  in
4    begin
5      let
6        x
7      in
8        begin
9          x assignment 1
10       end
11     followed by
12     let
13       x
14     in
15       begin
16         x assignment 1
17       end
18   end
```

Figur 4.6: *Test for om det samme navn kan erklæres flere gange.*

TEST 13: Først erklæres y som global variabel. Derefter kommer den første indlejrede kodeblok, hvor x erklæres. Efter kodeblokken forsvinder x så fra stakken af erklærede variable, indtil x igen erklæres i den anden kodeblok. Figur 4.7 illusterer, hvordan stakken af erklærede variable bør udvikle sig i eksemplet fra figur 4.6.

Endelig tester vi, om en lokalt erklæret variabel kan bruges globalt, som det forsøges i koden i figur 4.8. Ifølge scopereglerne for MOCCA burde dette ikke kunne lade sig gøre.



Figur 4.7: *Illustration af stakken af erklærede variale i eksemplet fra figur 4.6.*

```
1  let
2    x
3  in
4    begin
5      let
6        y
7      in
8        begin
9          y assignment 42
10       end
11     followed by
12     print y
13   end
```

Figur 4.8: *Test for om en lokalt erklæret variabel gælder udenfor kode blokken.*

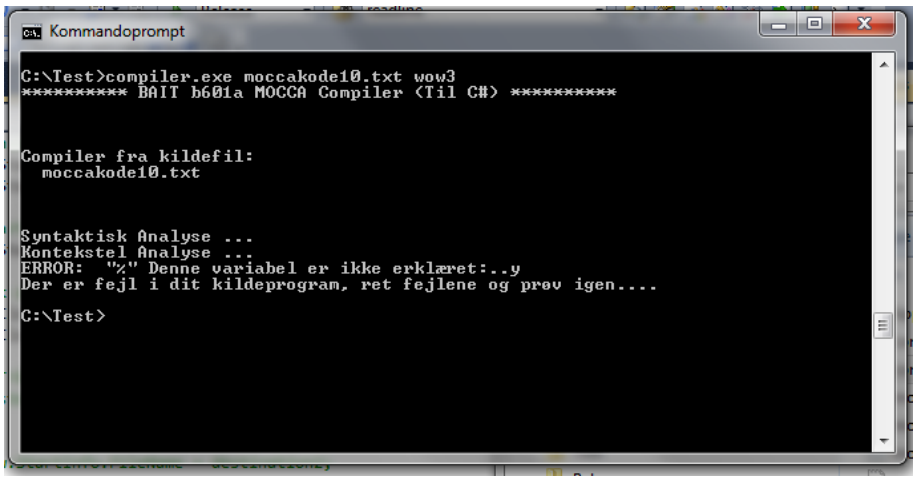
4.2 Evaluering af Resultater

Vi vil her gennemgå resultaterne af de forskellige test. En oversigt over resultaterne fra kodeeksemplerne, som vi forventede ville blive korrekt oversat uden problemer, kan ses i figur 4.9. Resultaterne af de problematiske kodeeksempler fra foregående afsnit kan ses i figur 4.11. De er opdelt, så hver test enten er "Bestået", "Bestået efter ændringer" eller "Ikke bestået". "Bestået efter ændringer" vil sige, at testkoden virker korrekt, efter compileren er blevet rettet til. "Bestået" vil sige, at testens resultat opfylder forventningerne til testen.

Eksempel	Bestået	Bestået efter ændring	Ikke bestået
Test 1: Variable uden erklæring (Figur 4.1)	X		
Test 2: Lokal og global tildeling (Figur 4.2)		X	
Test 3: Hello World	X		
Test 4: Fibonacci		X	
Test 5: Euklids Algoritme		X	
Test 6: Hanois Tårne	X		
Test 7: Primtal	X		
Test 8: Fakultet	X		

Figur 4.9: *Oversigt over test resultater af tidligere kodeeksempler.*

I forhold til erklæring af variable før brug bør det naturligvis testes, om det er muligt at generere C# kode uden at have en erklæring af de anvendte variable. Dette gøres som vist i figur 4.1. Det forventedes, at der ville blive meldt en fejl om, at en anvendt variabel ikke var erklæret, og dette støttede testen os i. Den kontekstuelle analyse konstaterede, at en erklæring manglede, som det kan ses i figur 4.10.



Figur 4.10: *Screenshot af oversættelsen af testen, der omhandler brug af variable uden erklæring.*

I testkode stykke nummer to, se figur 4.2, testes det, om en lokal tildeling af en værdi til en variabel kommer til at gælde globalt istedet for den tidligere tildelte globale værdi. Det forventedes, at dette ville være tilfældet. Dette viste sig ikke at holde stik, da testen udskrev den først tildelte værdi. Det konstateredes, at den genererede C# kode ikke indeholdt den indlejrede kodeblok, der i MOCCA var afgrænset af `begin end`. Efter en ændring af vores compilers parser returneres den sidst tildelte værdi korrekt.

En oversættelse af kodeeksemplet "Fibonacci" fra afsnit 2.3 resulterede i første omgang i C# kode, der genererede fejl. Årsagen til dette er, at den oprindelige implementation i compileren konsekvent erklærede variable som `public static` i C# kode. Dette virker fint nok, så længe variable bliver erklæret udenfor funktioner, men det er ikke tilladt inde i funktioner i C#. Løsningen blev at lave en semafor inspireret variabel, der tæller op, når compileren går ind i en funktion, og ned, når den er færdig med en funktion. Der bliver så kun indsat `public static` foran erklæringer af variable, hvis der ikke er tale om erklæringer inde i en funktion.

"Hanois Tårne", se bilag D, blev for så vidt korrekt oversat i vores compiler, men det viste sig, at der var en fejl i implementationen af algoritmen i MOCCA. Efter fejlen blev korrigeret, var der ingen problemer med oversættelsen.

"Euklids Algoritme", se bilag D, kunne ved første test ikke håndtere `num modulo div is`

`not 0`, da compileren ikke kunne håndtere mere end to `primary-expression` adskilt af en enkelt `operator`. Dette er blevet ændret, så compileren nu kan håndtere en række af `primary-expressions` og `operator` af vilkårlig længde.

De andre eksempler - Hello World, Primtal og Fakultet, se bilag D - oversættes uden problemer ved hjælp af vores compiler, og de færdige programmer opfører sig efter hensigten.

Eksempel	Bestået	Bestået efter ændring	Ikke bestået
Test 9: Operatorer og udtryk (Figur 4.3)			X
Test 10: Tal som expression i while (Figur 4.4)			X
Test 11: Evaluering af udtryk fra venstre (Figur 4.5)			X
Test 12: Erklæring af variable flere gange (Figur 4.6)			X
Test 13: Lokal og global erklæring (Figur 4.8)			X

Figur 4.11: *Oversigt over test resultater af kodeeksempler.*

Der blev i foregående afsnit opstillet en række test, der skulle påvise eventuelle fejl i implementationen af MOCCA i vores compiler.

Ved den første test der er opstillet i figur 4.3, var forventningen, at der ville opstå en fejl, da MOCCA variable ikke understøtter booleans. Dette viste sig at holde stik. Den genererede C# kode indeholdt fejl, da koden forsøger at sammenligne en boolean og en integer.
Løsningen på dette problem kan være at opdele MOCCAs operatorer i forskellige kategorier, afhængigt af om de evalueres som boolske udtryk eller som integers. Derefter kan man så begrænse boolske udtryk til kun at måtte sammenligne to variable eller værdier.

Test nummer 2 anvendte en integer som betingelse i en `while` løkke, som vist i figur 4.4. Det forventedes, at der ville opstå en fejl, da MOCCA ikke er designet til at evaluere et tal som sandt eller falskt. Dette blev også vist, da koden blev oversat. Den færdige C# kode kunne ikke evaluere et tal som en boolean og meldte derfor om fejl.
For at løse dette problem kan man indføre et typecheck af de udtryk, der gives som betingelser til kontrolstrukturer. Hvis ikke udtrykket kan fortolkes som boolean, skal compileren give en fejlmeddelelse. Det kan eventuelt overvejes, om MOCCA bør udvides med regler for, om initialiserede variable og talværdier skal kunne evalueres som booleans.
En af MOCCAs scoperegler siger, at udtryk altid skal evalueres fra venstre mod højre. Dette testes i test nummer 3, se figur 4.5. Det forventede resultat af det anvendte udtryk

må være 9, da 1 først skal lægges sammen med 2, og derefter skal 3 ganges til. Vores compiler respekterede dog ikke dette og oversatte til C# kode, der først gangede 2 og 3 sammen og derefter lagde 1 til. Vores compiler dumpede følgeligt i denne test. Problemet bunder her i, at vi oversætter for direkte til C# kode. Da C# anvender intuitive regler for hvilke operatorer, der bør evalueres først, vil disse også gælde for den C# kode vores compiler genererer, da vi ikke eksplicit sikrer os, at MOCCAs regel om evaluering fra venstre mod højre gælder. Løsningen på dette problem vil være at inkludere et sæt paranteser for hver operator i den genererede C# kode og derved sikre, at operatorer bliver evalueret i den rækkefølge de forekommer.

En af reglerne for MOCCA siger, at variable skal erklæres, før de kan bruges i koden. Dette kan både gøres for hele programmet, men også for de enkelte kode blokke. Det er derfor relevant at teste, om det samme navn kan erklæres flere gange i forskellige kode blokke som illustreret i figur 4.6. Forventningen var, at det godt kunne lade sig gøre at erklære det samme navn i forskellige blokke. Testen viste dog, at dette ikke er tilfældet med vores compiler. Den genererede C# kode har begge erklæringer i samme scope, og den fejler derfor.
Løsningen på dette problem vil være, at hver `let in` blok oversættes til en selvstændig metode i C# udenfor `main`. Metoderne kaldes så i selve programmet, så erklæringer af det samme navn ikke forekommer i samme scope.
Desuden kunne det konstateres, at der eksisterede et problem magen til det, der blev påvist under "Fibonacci" testen. Der blev konsekvent erklæret variable med `public static` både ved globale variable og lokale variable. Dette er dog ikke tilladt inde i metoder i C#. Det var derfor nødvendigt at lave en løsning ligesom ved "Fibonacci", så der nu kun bruges `public static`, hvis erklæringen ikke forekommer i en metode.

Den sidste af testkode stykkerne (se figur 4.8) omhandler lokal og global erklæring af variable. Forventningen til testen var, at en variabel ikke kunne udskrives udenfor den `let in` kodeblok, den var erklæret i. Det viste sig dog, at den endelige C# kode godt kunne udskrive variablen. Denne test dumpedes altså.

4.3 Konklusion

Vi kan på baggrund af dette kapitel konkludere, at der stadig er et stykke vej, før vores compiler indeholder en korrekt implementation af MOCCA, som vi har designet det. De fastlagte regler for MOCCA skal selvfølgelig overholdes, uanset hvilket sprog vi krydscompiler til.

Det kunne på længere sigt have været interessant at teste sproget ud fra "Den Store Vision", der blev beskrevet i afsnit 1.4.1, ved at lade udenforstående programmere MOCCA kode. I første omgang ville det være oplagt at lade medstuderende, der har programmeringserfaring, teste sproget. Dette ville give et billede af hvor intuitivt, og brugervenligt, sproget egentlig er. Hvis de for eksempel overhovedet ikke kunne forholde sig til, hvordan små programmer skulle konstrueres i MOCCA, ville det tyde på, at vi ikke bevæger os i retning af at opfylde "Den Store Vision".

Derudover ville det være hensigtsmæssigt at teste det på en ikke-programmør, da dette er målgruppen for MOCCA. Sproget er dog ikke på nuværende tidspunkt langt nok i udviklingen, til at dette ville være realistisk.

KAPITEL 5

REFLEKSION

5.1 Diskussion

Dette kapitel tager udgangspunkt i mange af de beslutninger, der er blevet truffet i løbet af projektet. Noget af det arbejde vi har udført, bunder i vores grundidéer med MOCCA og kunne måske have været gjort på en anden måde, når vi nu ser tilbage på det. Det vil blive diskuteret for at give projektet et andet perspektiv.

5.1.1 Metode

Undervejs i dette projekt er vi stødt på mange udfordringer. Specielt det faktum, at sprogdesign har været et ukendt univers, har været svært. På nogle punkter har vi kunne drage paralleller til tidligere projekter, hvor forståelsen af en problematik indtræffer sent i projektforløbet. Et klart eksempel på dette har været måden, hvorpå sproget er fremkommet. Vi er startet med en version, haft en forelæsning og herefter fremstår nye problematikker i sproget. Det har været følelsen af tre skridt frem og to tilbage. Dette har imidlertid givet os indblik i mange af de forskellige problemstillinger, man møder ved sprogdesign. Samtidig står dette indblik ganske godt i tråd med de formelle krav, som satte et mål fra starten. Ved projektets afslutning skulle vi evne at ræsonnere datalogisk og gøre rede for en compilers virke.

Det at ræsonnere datalogisk har vi kun prøvet på to af vores tidligere semestre. I forbindelse med dette har vores vejleder været en stor hjælp. På andre semestre har vi skulle ræsonnere efter henholdsvis den samfundsvidenskabelige og den humanistiske tradition. Den datalogiske tilgang adskiller sig primært fra vores synspunkt ved at være en mere pragmatisk tilgang. Hvis ikke der er én sandhed, så træffes beslutningen typisk efter fastlagte variable. Et eksempel på dette kunne være, om compileren skulle oversætte mod C# eller Java. Her befinder man sig i et valg, hvor vi kender de faktorer, vi arbejder med. Vi ved indenfor hvilke felter, de to sprog excellerer. Samtidig kender vi vores

egne begrænsninger, og derfor faldt valget på C#. Omvendt kunne vi også have valgt at oversætte mod et lavniveau sprog. Hertil kendte vi igen faktorerne, givet det faktum at vi aldrig tidligere har arbejdet med et sprog med lavere niveau end C. C#s compiler er endvidere også optimeret så ganske meget, at oddsene for at øge performance ved selv oversætte mod et lavniveau sprog næppe er muligt. Denne tilgang adskiller sig væsentligt fra de to andre videnskabelige traditioner, der i langt højere grad ynder metodebaseret arbejde og metaræsonnering.

Tidligere Projekter

Men målet for projektet har også været anderledes. Vi har naturligvis ikke haft til mål at skabe verdens bedste sprog. Derimod har interessen dvælet ved problematikken om programmeringssprog med talesprogskarakteristika. Vi har primært set udviklingen af compileren som en læringsmæssig del af projektet. Derfor har vi forsøgt at kombinere det med vores forståelse for samspillet mellem menneske og maskine. Denne forståelse førte til en undren over programmeringssprogs stejle indlæringskurve. Derfor endte dette projekt med at blive et forsøg på at introducere mere intuition til programmeringssprog.

Tidligere projekter har også haft en læringsmæssig side, som typisk har bestået af et stykke programmeringsarbejde. Dette har været for opnå forståelse for eksempelvis databaseprogrammering eller det objektorienterede paradigme. I disse projekter har vi typisk været ganske frustrerede over det faktum, at det store billede først indtræffer sent i projektet. Men modsat tidligere projekter, hvor denne fremgangsmåde har været frustrerede, har vi denne gang bedre kunne håndtere det. Det har været et decideret mål for dette projekt at udvikle sproget efter bedste overbevisning på det givne tidspunkt. Efterhånden som det store billede indtræffer, har vi i stedet for at ærgre os over spildt arbejde benyttet chancen til at skrive noter til denne diskussion. Man kan sige, at vi til dels har fulgt trial-and-error metoden.

Projektet

Selve det at udføre et stykke projektarbejde er vi efterhånden begyndt at vænne os til. Allerede fra start overførte vi erfaringer fra tidligere projekter. Her indførte vi mandagsmøder, der fungerede som styringsmøder. Spørgsmålene til os selv på disse møder var: "Hvad er blevet lavet?" og "hvad skal laves i denne uge?" Set tilbage er der faktisk kun én ting, som vores planlægning har slået lidt fejl ved. Vi blev i forhold til programmeringen ved med at vente på indtræfningen af det store billede. Her skulle vi i virkeligheden blot have sprunget ud i det med krum hals, som vi gjorde på lignende vis med syntaksdelen. I stedet prioriterede vi at arbejde med netop syntaksen i stedet for programmeringsarbejdet og forsøgte at få syntaksdelen helt på plads, inden vi begyndte programmeringsarbejdet. Set tilbage skyldtes det muligvis, at vi tidligere har arbejdet med mere distinkte faser i projektarbejdet. Disse distinkte faser skulle være fuldt på plads, inden man kunne påbegynde næste fase. For eksempel skulle vi på 5. semester have en organisationsanalysen fuldt på plads, før vi begyndte at arbejde med ændringer i denne organisation. Vi har

været præget af samme tanke her, men vi har indset, at vi i virkeligheden sagtens kunne have begyndt programmeringsarbejdet samtidig med syntaksen. For eksempel med lexeren der kigger efter nøgleord. Selvom nøgleordene ikke er bestemt endnu, kan man jo sagtens skrive et stykke kode, der genkender nøgleord.

Rapportskrivning

Rapporten bærer naturligvis præg af den naturvidenskabelige tilgang til dokumentation af resultater. Men selv indenfor denne findes der flere forskellige tilgange. Dette har vi også erfaret på egen krop, da forskellige vejledere har forskellige meninger om, hvad der skal vægtes. Eller måske nærmere, at forskellige rapporter har forskellige styrker, og derfor vægter man nogle områder tungere, mens man samtidig nedprioriterer andre områder, hvor man står knapt så stærkt. Her har vejlederen også på dette semester spillet en stor rolle i vægtningen af områder. Vi har tidligere benyttet os af at cirkulere skrevne dele af rapporten rundt mellem hinanden, sådan at vi har kunne sikre en ensrettethed. Vi har i stedet med denne rapport hver mandag haft en retterunde af den sidste uges skrevne materiale. Et værktøj kaldet Crocodoc (`http://www.crocodoc.com`) har været benyttet til dette. Crocodoc kan fremvise en pdf-fil online, som inviterede personer kan vedhæfte kommentarer til. Dette værktøj har vi været særskilt glade for også i forbindelse med vejledermøder. Det har dog medført, at hver gang der er blevet skrevet et nyt kapitel, skal der naturligvis uploades en ny pdf til Crocodoc. Dette har resulteret i mange Crocodoc filer. Her har vi erfaret, at en mere konsistent afslutning af gamle filer er vigtigt, sådan at noter ikke bliver glemt. Derfor har vi mod slutningen måtte tjekke flere filer igennem for gamle kommentarer, som vi eventuelt havde overset.

Selve skrivemetoden kunne sagtens have været gjort anderledes. Her tænkes især på IMRAD [Universitet, 2011] metoden, der er en mere klassisk tilgang til opbygning af rapporter. Ud fra vores egne erfaringer vurderer vi dog, at vores selvkonstruerede rapportopbygninger er langt bedre, idet det somme tider kan være svært at presse modeller ned over projekter. Tilgangen til formidling i denne rapport læner sig meget op af den måde, vi har arbejdet med projektet på. Det har givet bedst mening for os.

5.1.2 Sprog

MOCCA har ændret sig meget siden starten af projektet. Dette har været et resultat af, at mangler er blevet opdaget, eller at problemer er opstået. Selve grundidéen var et talelignende programmeringssprog. Derfor blev det implementeret uden de specialtegn, som kendetegner mange normale programmeringssprog. Mange interessante valg er blevet foretaget. Her er der især to, der har gjort sig gældende.

- Det første valg har været at fjerne specialtegn. Dette har betydet, at vi har fundet alternativer. Symboler som {} og () er blevet til `begin` og `end`. Operatorer som +,-,* og / er blevet til `add`,`subtract`,`multiply` og `divide`. Valget har haft en række konsekvenser, der blandt andet har betydet, at programmer bliver længere og mere uoverskuelige. Om dette skyldes, at vi i forvejen har kigget en del på konventionelle sprog som C#, skal vi lade være usagt. Vi kan dog konstatere, at et program skrevet i MOCCA er noget mere uoverskueligt end et program skrevet i C#. Mange af de specialtegn, vi har omdøbt, er fremkommet på baggrund af, at vi havde en klar idé om, hvordan de skulle udtales.

- Det andet valg der er blevet truffet, har været mængden af de funktioner, der skulle være tilgængelige i sproget. I MOCCA har vi hovedsageligt lagt vægt på `print`, fordi man skulle kunne udskrive tekst til en konsol. Denne funktion er selvfølgelig nødvendig for at kunne programmere mange basale ting, men vi har indset at én specifik mangler. Der tales selvfølgelig om en `Read` funktion, der kan læse input fra brugeren for eksempel via en konsol, hvilket kan være helt essentielt, når man konstruerer små programmer.

  Da vi talte om manglene ting i sproget, var det ikke kun funktioner, der kom på bordet. Datastrukturer blev også nævnt, idet det kunne være belejligt at kunne operere på disse, da de giver en tilpas abstraktion fra typer. Helt basale datastrukturer som arrays eller hægtede lister ville ligesom kontrolstrukturer som for-løkker være rare at have. Også i forhold til MOCCA ville de være rare at have, men ikke en nødvendighed. Derfor er de heller ikke implementeret i sproget.

Udviklingen af sproget har båret meget præg af en løbende tilretning. Det er derfor også vigtigt at påpege, at ting er gået op for os undervejs. Dette har blandt andet betydet, at vi er blevet klar over en ekstra type, som vi implicit kan håndtere i vores programmer. Vi taler om typen bool, som vi kan håndtere i `if` og `while` kommandoer. Dette er naturligvis noget, der skal tages højde for i en eventuel fremtidig sprog specifikation.

5.1.3 Compilerkonstruktion

I forhold til programmering af compileren har vi gjort mange interessante observationer. Når man tager en stor beslutning, har det i mange tilfælde en betydning for resten af projektet.

Oversættelse

I vores situation har vi valgt at krydscompile til C#. Det har haft sine konsekvenser. Her tænkes især på de mange valg, der på forhånd er taget for os. Blandt andet kan vores fravalg af datatyper nævnes. Dette har haft betydning, da vi har oversat til et sprog med mange datatyper. Det har derfor været svært at finde ud af, hvordan vi håndterer heltal på en fornuftig måde. Løsningen har været at lave alle variable til typen `object` for derefter at boxe dem, når de skulle bruges. Dette har dog ikke været helt uden problemer og har ofte givet os mange spekulationer om, hvordan det kunne løses på en fornuftig måde.

Et spørgsmål der melder sig er "kunne det have været mere fornuftigt at compile til et lavniveau sprog?" Der kan være mange fordele i dette. Blandt andet ville vi have haft 100% kontrol over, hvordan den oversatte kode i sidste ende eksekveres. En anden fordel ville være, at vi kunne håndtere vores datatyper på en mere hensigtsmæssig måde, idet vi på lavniveau kunne abstrahere fra typer. Typer er normalt sat i verden for typesikkerhed. Da dette ikke kommer i spil i lavniveau sprog vil vi være ude over dette.

Selv om der er nævnt mange fordele, er der samtidig også nogle ulemper. Den klare ulempe er at for at kunne oversætte til et sprog der er på et lavere abstraktionsniveau kræver det en kendskab til sproget. Idet vi kun har haft lidt kendskab til lavniveau sprog, ville der måske have været en hel del, der skulle gennemgås.

Vi mener stadig at have truffet den rigtige beslutning i forhold til compilerens valg af oversættelse, men en eventuel ny compiler kunne godt oversætte direkte til lavniveau kode.

En anden diskussion er om den firdelte konstruktion af compileren har været den korrekte tilgang. I princippet kunne den konstrueres som en tredelt, jævnfør Watt and Brown [Watt and Brown, 2007], todelt eller måske bare som en enkelt del. Det er selvfølgelig helt op til programmøren selv at træffe disse valg. I forhold til MOCCA-compileren har det dog været vigtigt at holde det overblik, det giver at dele compileren op i tilpas mange dele, således at hver del tager sig af hver sin opgave. Vi mener også, at dette giver en bedre indlæring i forhold til de formelle mål og krav, afsnit 1.1.

Programmering

Når man, som i vores tilfælde, vælger at lave en recursive descent parser, er det vigtigt at have den syntaktiske del af sproget lagt helt fast. Det vil sige, at der helst ikke skulle komme nogle overraskelser, når man senere koder parseren. I vores tilfælde har dette været en vekselvirkningsproces. Dels fordi vores syntaks er blevet rettet til mens vi har kodet, dels også fordi kodningen har påvist fejl i syntaksen. Dette har været en relativt lang proces, der endte i mange timers spekuleren.

En anden tilgang kunne have været at ændre tilgangen. Vi kunne i stedet for at kode parseren selv have gjort brug af et af de såkaldte værktøjer til at generere parsere og lexere. Vi kunne derfor have brugt den ekstra tid på at få rettet syntaksen til, således at den var klar til en af værktøjerne. En ulempe ved at gøre det ville være, at vi kunne være kommet til at skulle sætte os ind i et af disse værktøjer. En proces der kunne komme til at tage lang tid og fjerne fokus fra den formelle del af indlæringen.

Vi vurderer stadig, at vi har truffet det rigtige valg, da vi har skulle gennemgå en læringsproces og gerne skulle have en smule kendskab til, hvordan parseren og lexeren er bygget op.

Den generelle kodestil har været præget af, hvad der virkede på det pågældende tidspunkt. Vi har således taget det til os, men er aldrig nået til at rette det.

Blandt andet kan et specifikt eksempel fra brugen af globale variable nævnes. Kigges der på metoden scan() fra kapitel 3.3 på side 42, vil man flere steder opdage at variablen `currentSpelling` er nævnt. Når hver af hjælpemetoderne kaldes tildeler de variablen en ny værdi. En analogi kunne være en postkasse, hvor en metode lægger noget i, og den næste tager den ud. I stedet for denne løsning kunne vi have lavet metoder, der tog imod input, og derfor videregav værdier imellem dem.

5.1.4 Test

Vi har i forbindelse med test i dette projekt udelukkende fokuseret på implementationen af MOCCA i compileren. I forhold til den endelige mål med MOCCA er dette dog ikke fyldestgørende. Det bør ikke kun være compileren i sig selv, der skal testes, da dette blot er en instantiering af sproget. Sproget i sig selv er jo det overordnede mål, som er beskrevet i "Den Store Vision" i afsnit 1.4.1. Det er derfor på sin vis vigtigst at teste, om sproget lever op til de rammer, der er blevet opstillet i introduktionen i kapitel 1. Grunden til, at vi ikke har gjort dette, er, at vi har afgrænset os til at fokusere på de indledende skridt i udviklingen af MOCCA ved i første omgang bare at fjerne specialtegn fra sproget. MOCCA er derfor ikke på et sted i udviklingen, hvor det ville give mening at teste i forhold til den endelige målgruppe. Da rammerne for semesteret samtidig fordrer, at vi udvikler en compiler, har det virket logisk at fokusere på en test af denne.

Hvis man skulle have opstillet en test for MOCCA som sprog, kunne det have været interessant både at lave test til teknisk orienterede folk og ikke-teknisk orienterede folk. Derved kunne man have belyst, hvordan de ville reagere forskelligt på at skulle opstille små programmer i et "fremmed" sprog. Vi må dog nok se i øjnene, at sproget i sin nuværende form rammer en kende ved siden af sin endelige målgruppe.

De første kodeeksempler, der blev opstillet i udviklingen af MOCCA, kunne uden problemer bestå testen af compileren. Dette var dog også forventet, da de er blevet tilpasset løbende, mens syntaksen er blevet rettet til. Ved at opstille nogle mere kritiske kodeeksempler, med fokus på åbenlyse problematiske forhold i både sprog og compiler, lykkedes det os dog at påvise fejl og mangler. Der er derfor et stykke vej endnu, før både sprog og compiler kan siges at være færdigudviklet. Her kan fejlhåndtering blandt andet nævnes, idet vi påviste, at ikke alle tests leverede en konstruktiv meddelelse.

Et helt andet spørgsmål kunne være, om vi har testet compileren på en fornuftig måde. Vi har testet compileren ved at lave eksempler, der skulle compiles, for derefter at se om den gav en fejl. En helt anden tilgang kunne være at teste performance af compileren. Her tænkes især på køretid, alt efter hvor meget input man giver til den. Vi har dog vurderet, at en korrekt implementation af sproget skulle komme i første række, snarere end at den skulle performe fornuftigt. Ydermere vurderer vi ikke, at der i MOCCA skal skrives store programmer.

At teste en compiler er ikke en specielt nem opgave, idet et sprog kan strække sig over en bred vifte af anvendelsesmuligheder. Det kan derfor derfor svært at komme ud i alle hjørner af koden. En mere grundig tilgang kunne være at bruge en *test suite* [associated, 2011]. En test suite er et værktøj, der kan teste compilere for kvalitet og verifikation. Vi har i første omgang testet de kodeeksempler, vi har brugt i udviklingen af sproget. Derefter har vi forsøgt at opstille test for de dele af MOCCA og vores compiler, der har virket mangelfulde eller problematiske. Dette er naturligvis ikke en fyldestgørende måde at teste på, men det har været udmærket til at indikere, hvor fokus bør lægges i en videre udvikling af både compiler og sprog.

5.2 Konklusion

Denne konklusion vil give et kvalificeret svar på spørgsmålet der blev stillet i problemformuleringen i afsnit 1.5, ved at forholde sig til punkterne i afgrænsningen i afsnit 1.6. Det gøres helt konkret ved at samle op delkonklusionerne i de 3 kapitler "Sprogdesign" (kapitel 2), "Compilerkonstruktion" (kapitel 3) og "Test og afprøvning" (kapitel 4).

Beskrivelsen af sproget MOCCA er udformet ved hjælp af en uformel og en formel specifikation.

Den uformelle specifikation stiller løse rammer op skrevet i et, for målgruppen, forståeligt sprog. Den formelle specifikation samler op på dette, ved at stille konkrete scope- og syntaksregler op for sproget.

I forhold til scoperegler benytter MOCCA sig af såkaldt nested block structure. Det har den egenskab, at en variabel er lokal i forhold til den blok, hvor den er erklæret, men global for indre blokke. MOCCA er designet således, at en variabel skal erklæres eksplicit før den kan bruges.

En række valg blev foretaget i forhold til syntaksen. Syntaksen er udformet ved at bruge EBNF som notation.

Hovedpointerne for syntaksen er, at MOCCA er et typeløst sprog, og at MOCCA eliminerer specialtegn. Det placerer vores lille imperative sprog MOCCA på et lavere niveau end for eksempel C#, der i høj grad sigter mod det objektorienterede paradigme. Dog indbefatter MOCCA generel abstraktion med funktioner og nogle generelle kontrolstrukturer som `if` og `while`. Til at forme syntaksen opstillede vi små kodeeksempler for, hvordan MOCCA kan bruges.

Syntaksen blev ud fra kodeeksemplerne og scopereglerne udformet således, at variable ikke skal erklæres med typer. Der findes et enkelt gråzone tilfælde, da strenge, når de tildeles til en variabel, skal indkapsles af noget kode, der fortæller, at her tildeles en streng.

Specialtegn blev erstattet ved at transformere tegnene, som man kender dem fra eksempelvis C#, til ren tekst. Eksempelvis er operatoren "+" lavet om til `add`, hvilket også er en intuitiv måde at udtale det på. Derudover er knapt så intuitive specialtegn, så som "{"og "}"lavet om til `begin` og `end`.
Compileren til MOCCA blev udviklet efter det objektorienterede paradigme ved at dele konstruktionen op i 4 faser: Leksikalsk analyse, syntaktisk analyse, kontekstuel analyse og kodegenerering.

Den leksikalske analyse scanner MOCCA kildekoden og klargør den til næste fase ved at identificere de enkelte tokens i kildekoden. Dette gøres ved at holde dem op mod syntak-

sen. I MOCCAs tilfælde har scanneren en funktion mere. Den sørger nemlig for at slå de tokens sammen, der i kildekoden spreder sig over flere ord. Dette gøres ved at indføre specielle handlemønstre i scannerens kode for de ord, der i nogle tilfælde efterfølges af andre.

I den syntaktiske analyse blev der konstrueret en recursive descent parser. Denne blev kodet af os selv frem for at benytte et værktøj. Det gjorde vi, fordi vi vurderede, at det var den måde, vi i et læringsperspektiv fik mest ud af det på. Parseren er konstrueret således, at den kan tage sig af kontekstfri fejlbehandling, der konkret kan konstatere, om syntaksens delelementer overholdes.

For at holde vores objektorienterede tilgang overskuelig blev Visitor design mønsteret indført. Det bruges i vores compiler til at gennemløbe en repræsentation af den parsede kode, et abstrakt syntaks træ, i de sidste to faser.

I den tredje fase, kaldet den kontekstuelle analyse, finder vores compilers kontekstsensitive fejlbehandling sted. I MOCCA-compileren drejer det sig om at forholde sig til, om variable er erklæret, når de anvendes, ellers skal en fejlmeddelelse returneres. Dette er den kontekstsensitive del.

Den sidste fase i compileren gennemløber det abstrakte syntakstræ og genererer C# kode. Udfordringerne i forhold til MOCCA har været at oversætte fra noget typeløst til C#, der i høj grad gør brug af typer. Løsningen blev at bruge typen `object` i C# til erklæringer.
Kapitlet "Test og afprøvning" satte MOCCAs syntaks og compiler op mod hinanden. Dette blev gjort for at anskueliggøre styrker og svagheder i begge dele.
På baggrund af disse test konkluderes det, at MOCCA-compileren, som ventet, ikke er en fuldendt implementation. Det er en meget svær disciplin at teste en compiler, men blot disse små test viser, at der er uoverensstemmelser mellem compiler og syntaks, samt at fejlbehandlingen kunne være bedre. Der præsenteres mulige løsninger i kapitlet.

Selvom vi ikke har skabt den løsning, der præsenteres i "Den Store Vision" i afsnit 1.4.1, og heller ikke er kommet et stort skridt videre i forhold til den, konkluderer vi, at de afgrænsede problemstillinger som beskrevet i afsnit 1.6, samt de formelle mål og krav (afsnit 1.1), er opfyldt. På den baggrund konkluderer vi også, at vi har taget vores levertran, og at det har været godt for os. Samtidig er vi ved at have reddet os i land fra de 70000 favne vand.

5.3 Perspektivering

Dette afsnit vil forholde sig til, hvilken fremtid MOCCA har efter dette projekt. Det vil vi gøre ved at kigge på, om det har sin berettigelse at videreudvikle sproget MOCCA og compileren mod "Den Store Vision" (afsnit 1.4.1).

Vi vil først og fremmest kigge på vores egen motivation for at videreføre MOCCA, i forhold til hvor det lander i denne rapport, når læringsaspektet forsvinder. Dernæst vil vi se på, om et færdigudviklet MOCCA har en eksistensberettigelse.

Inden dette semester havde ingen af gruppens medlemmer nogen særlig interesse i forhold til teorien bag programmeringssprog og compilere. De programmeringssprog og programmeringsparadigmer vi har kendskab til, har vi i sammenblandinger betragtet som værktøjer, vi kan brug til at udforme de systemer, vi har haft brug for. Denne indstilling er intakt på trods af det kig under motorhjelmen, vi har fået på dette semester.

Vi har forståelse for den dannelse i forhold til programmeringssprog, dette kig har givet os, og den vil helt sikkert komme os til gode i forhold til indlæring af nye programmeringssprog og paradigmer.

Når det så er sagt, føler vi ikke, at den faglige profil vi i sidste ende sigter mod, som er mere rettet mod informationssystemer og menneske-maskine interaktion, har nogen glæde af, at vi fortsætter med at arbejde med MOCCA projektet. MOCCA når derfor ikke ved vores hjælp videre fra det stadie, hvor det er nu.

På dette stadie kunne der dog være elementer, som vi, i forhold til vores baggrund og den føromtalte fremtidige faglige profil, kunne ønske os at kigge på, hvis muligheden i senere semestre byder sig. Det drejer sig for eksempel om at bringe brugere i spil i forhold til udviklingen af sproget.

Der kunne være tale om decideret brugerdrevet innovation, hvor brugerne bringes i spil så tidligt i forløbet som muligt, og derfor for eksempel ville have indvirkning på specifikationerne. Det kunne også være usabilitytest af den nuværende specifikation i forhold til målgruppen for at finde ud af, om eliminering af specialtegn gør programmeringsdisciplinen lettere for denne.

Selvom MOCCA ikke umiddelbart videreføres, mener vi faktisk, at der kunne være behov for, at programmeringssprog generelt støtter sig op af dele af "Den Store Vision" (afsnit 1.4.1).

Idéen om at bringe brugervenlighed ind i selve kodefasen på et niveau, så indlæringskurven for den almindelige IT bruger falder markant, synes vi er yderst interessant. Vi mener faktisk, at i særdeleshed de store programmeringssprog med store udviklingsmiljøer og

en stor mængde dokumentation forsømmer denne opgave. På sigt mener vi, at man som bruger bliver efterladt bagude i IT sammenhæng, hvis man ikke er i stand til at skabe eller kombinere noget med IT, for eksempel ved at programmere. Det gør sig gældende i mange sammenhænge i en tidsalder, hvor for eksempel sociale medier, dataindsamling og databehandling indgår i arbejdsbeskrivelser, hvor IT, som her er fællesnævneren, kun står skrevet mellem linjerne. Det er en meget interessant problemstilling, der om ikke andet, giver grundidéen i dette projektet sin eksistensberettigelse.

LITTERATUR

associated, ACE. 2011. "SuperTest C/C++." http://www.ace.nl/compiler/supertest.html.

Johnson, Stephen C. 2011. "Yet Another Compiler Compiler." http://dinosaur.compilertools.net/yacc/.

McKeeman, W. M., D. B. Wortman and J. J. Horning. 1970. *A compiler generator [by] W. M. McKeeman, J. J. Horning [and] D. B. Wortman.* Prentice-Hall Englewood Cliffs, N.J.,.

Microsoft. 2011. "Visual Studio." http://www.microsoft.com/visualstudio/da-dk.

MSDN. 2011*a*. "int (C# reference)." http://msdn.microsoft.com/en-us/library/5kzh1b5w%28v=vs.80%29.aspx.

MSDN. 2011*b*. "object (C# reference)." http://msdn.microsoft.com/en-us/library/9kkx3h3c%28v=vs.80%29.aspx.

MSDN. 2011*c*. "static (C# reference)." http://msdn.microsoft.com/en-us/library/98f28cdx.aspx.

og Sundhedsvidenskabelige Fakulteter De Ingeniør-, Natur. 2009. "Studieordning for Bacheloruddannelsen i Informationsteknologi." http://www.sict.aau.dk/digitalAssets/3/3322_baitsept2008_sept2009.pdf.

SableCC, Project. 2011. "SableCC." http://sablecc.org.

Sebesta, Robert W. 2010. *Concepts of Programming Languages.* 9. edition ed. Prentice Hall.

Universitet, Københavns. 2011. "IMRAD." http://folkesundhedsvidenskab.ku.dk/projekter/imrad/.

Watt, David A. and Deryck F. Brown. 2007. *Programming Language Processors in Java - Compilers and Interpreters*. Prentice Hall.

Wikipedia. 2011. "MonoDevelop." `http://en.wikipedia.org/wiki/MonoDevelop`.

BILAG A

RESUME

This report is written as bachelor thesis by students at the technical track of the information technology bachelor at the Department of Computer Science on Aalborg University.

The goal of this report is to demonstrate knowledge about language design and compilers. The knowledge is obtained through courses and project work.

The report is describing the design of a programming language called MOCCA which eliminates special characters like arithmetic symbols and punctuation marks. Furthermore, the construction, implementation and testing of a compiler for MOCCA will be outlined.

The design of MOCCA will be carried out by forming both informal and formal specifications.

The informal specification is plain text. This specification should be readable by MOCCA's target audience. The target audience is daily users of information technology, who are ready to add the skill of creating small imperative programs to their toolbox. This could be users who already know how to use different statements in spreadsheets like Excel.

The formal statement is a presentation of the keywords in MOCCA and a grammar, explaining the language's syntax. The grammar is formed using the metalanguage Extended Backus Naur Formalism (EBNF). Small code examples are built from the grammar to test it's abilities.

To make programs written in MOCCA code executable, a compiler is constructed. Throughout the compiler, MOCCA code is translated to C# code. Then C#'s compiler is translating the code further to execution on the machine. The construction of this compiler is split into four phases.

The first phase is lexical analysis, which focuses on scanning the input of MOCCA code and preparing it for further processing. The preparation is to identify the different tokens in the program according to the grammar. The scanner in the MOCCA compiler also connects special tokens spread over more than one word.

The second phase is called syntactic analysis and focuses on the structure of the identified tokens. The structure must follow the rules specified by the grammar. A recursive descent parser is constructed to manage this phase of the MOCCA compiler. Some context free error handling is part of this phase.

The third phase is contextual analysis. This phase usually focuses on type checking and declarations. Since MOCCA is a type less language, this phase only focuses on checking that variables and functions are declared in the right context. If not, an error is returned. This is called the context sensitive error handling.

The last phase is code generation. This is the phase where the MOCCA code is translated to C# code. C# is a high level language that contrary to MOCCA makes use of types. That gives some challenges in this phase that will be explained.

The constructed MOCCA compiler will in this report be exposed to some tests. The purpose is to compile and execute code examples, made from the syntax. Before the tests, possible outcomes will be considered. If a test gives an outcome other than the expected, the compiler will be evaluated and changes will be made to the code. Possible changes will be described.

At the end of the report, the method to design the language and the compiler will be discussed. So will the results achieved. A conclusion towards the findings of the report will be presented. Furthermore, the future possibilities for MOCCA will be set into perspective. This will be done, considering whether development of MOCCA will continue, when the learning aspect is gone.

BILAG B

SYNTAKS

Her ses den konkrete syntaks for MOCCA, udtrykt i EBNF. Nonterminal symboler er fremhævet med fed.

```
program            ::= let declaration in begin command end

command            ::= single-command
                     | single-command followed by command

single-command     ::= identifier assignment expression
                     | if expression begin command end
                     | begin command end
                     | while expression begin command end
                     | let declaration in begin command end
                     | call identifier actual-parameters
                     | return identifier
                     | print identifier

expression         ::= primaryexpression
                     | expression (operator primaryexpression)*

primaryexpression  ::= integerlit
                     | identifier
                     | string begin graphic* end
                     | call identifier actual-parameters

declaration        ::= single-declaration
                     | single-declaration and declaration

single-declaration ::= ε
                     | identifier
                     | function identifier formal-parameters begin command end
```

integerlit ::= **digit** **digit**∗

identifier ::= **letter** (**digit** | **letter**)∗

formal−parameters ::= ε
| **identifier**
| **identifier** and **formal-parameters**

actual−parameters ::= ε
| **identifier**
| **integerlit**
| **identifier** and **actual-parameters**
| **integerlit** and **actual-parameters**

operator ::= add
| subtract
| multiply
| divide
| greater than
| less than
| less or equal
| greater or equal
| is
| assignment
| is not
| modulo

letter ::= a | b | c | d | e | f | g | h | i | j | k | l
| m | n | o | p | q | r | s | t | u | v | x | y | z
| A | B | C | D | E | F | G | H | I | J | K | L
| M | N | O | P | Q | R | S | T | U | V | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

comment ::= comment **graphic**∗ endcomment

graphic ::= letter
| digit
| space

Terminal Symboler

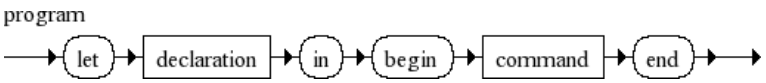
Vi har her listet den række af terminalsymboler der indgår i MOCCA.

```
if
while
begin
end
function
let
in
call
return
print
comment
endcomment
and
followed by
add
subtract
multiply
divide
greater than
less than
less or equal
greater or equal
differs from
assignment
is
```

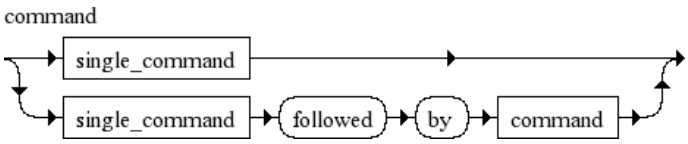
BILAG C

SYNTAKS DIAGRAMMER

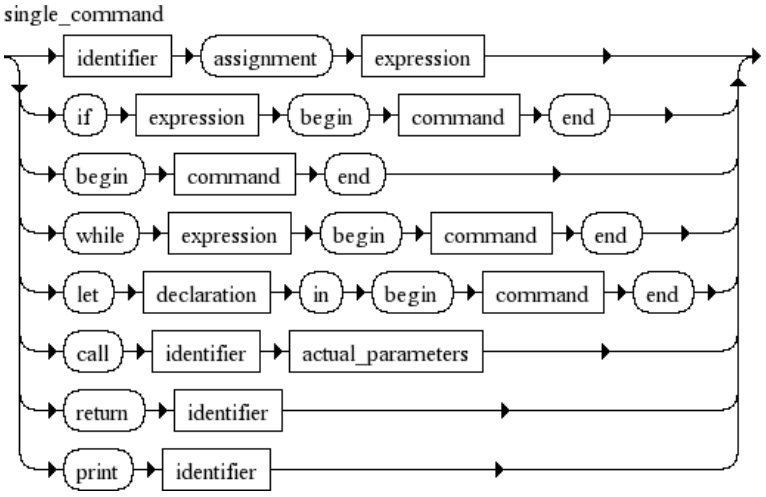
I dette bilag er der præsenteret diagrammer for samtlige non-terminaler i syntaksen for MOCCA. De fulde regler findes i bilag B.



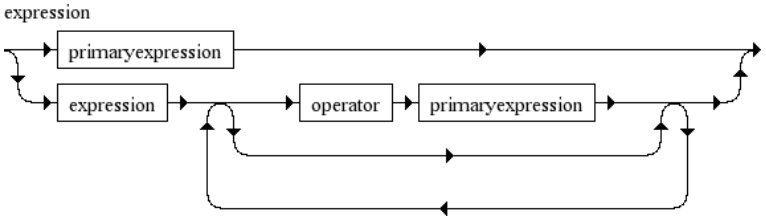
Figur C.1: *Syntaksdiagram for reglen program*



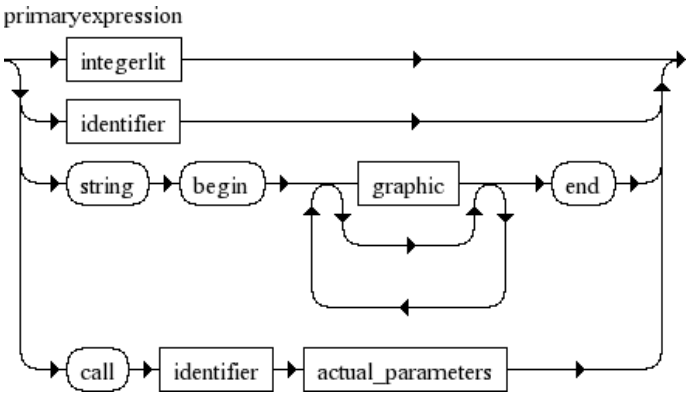
Figur C.2: *Syntaksdiagram for reglen command*



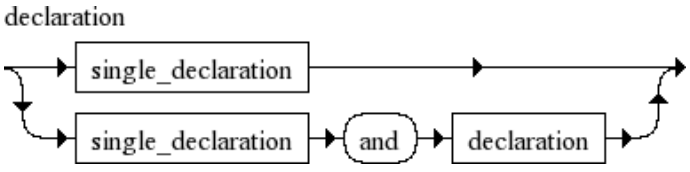
Figur C.3: *Syntaksdiagram for reglen single-command*



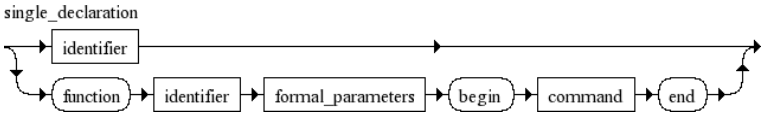
Figur C.4: *Syntaksdiagram for reglen expression*



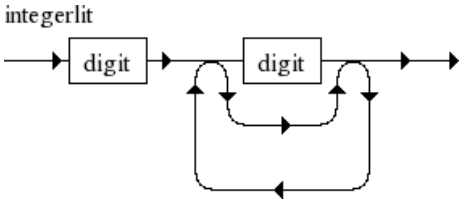
Figur C.5: *Syntaksdiagram for reglen primaryexpression*



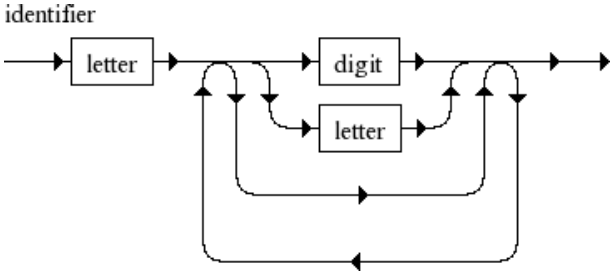
Figur C.6: *Syntaksdiagram for reglen declaration*



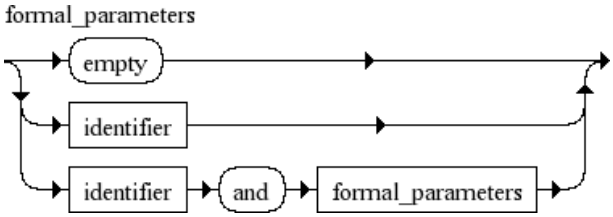
Figur C.7: *Syntaksdiagram for regeln single-declaration*



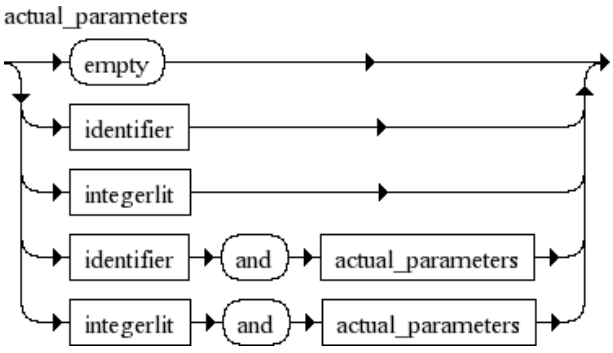
Figur C.8: *Syntaksdiagram for reglen integerlit*



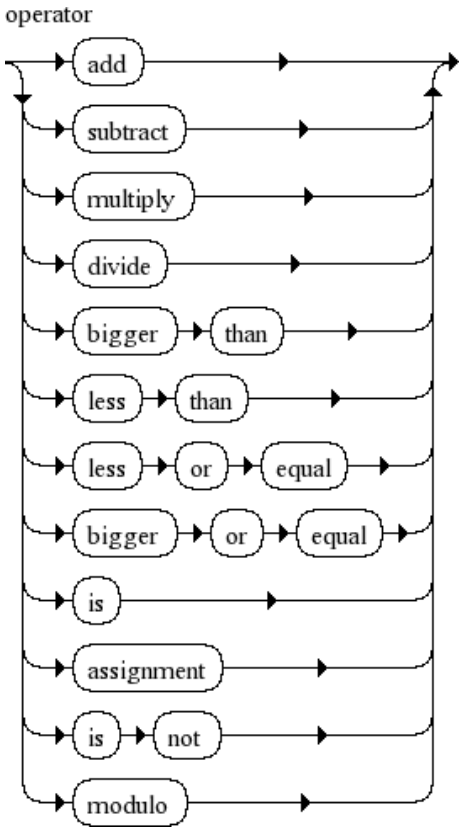
Figur C.9: *Syntaksdiagram for reglen identifier*



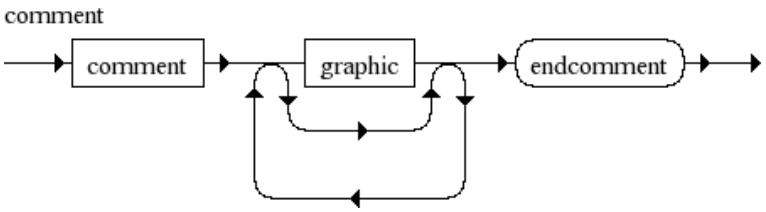
Figur C.10: *Syntaksdiagram for reglen formal-parameters*



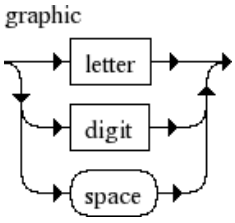
Figur C.11: *Syntaksdiagram for reglen actual-parameters*



Figur C.12: *Syntaksdiagram for reglen operator*



Figur C.13: *Syntaksdiagram for reglen comment*



Figur C.14: *Syntaksdiagram for reglen graphic*

BILAG D

KODEEKSEMPLER

Euklids Algoritme

En implementation af Euklids algoritme, der finder højeste fælles divisor mellem to tal. Her ses blandt andet rekursion i MOCCA.

```
let
  function Euklid x and y
    begin
      if x is 0
        begin
          return y
        end
      followed by
      while y is not 0
        begin
          if x greater than y
            begin
              x assignment x subtract y
            end
          followed by
          if x less or equal y
            begin
              y assignment y subtract x
            end
        end
      followed by
      return x
    end
in
  begin
    call Euklid 12 and 20
  end
```

Hanois Tårne

Hanoi's tårne demonstrerer rekursion i MOCCA.

```
let
  function Hanoi x and from and to and aux
    begin
      let
        str1 and str2
      in
        begin
          str1 assignment string begin MOVE DISK FROM end
          followed by
          str2 assignment string begin TO end
          followed by
          if x is 1
            begin
              print str1
              followed by
              print from
              followed by
              print str2
              followed by
              print to
            end
          followed by
          if x greater than 1
            begin
              let
                y
              in
                begin
                  y assignment x subtract 1
                  followed by
                  call Hanoi y and from and aux and to
                  followed by
                  print str1
                  followed by
                  print from
                  followed by
                  print str2
                  followed by
                  print to
                  followed by
                  call Hanoi y and aux and to and from
                end
            end
        end
    end
in
```

```
47   begin
48     call Hanoi 5 and A and C and B
49   end
```

Primtal

En algoritme der udskriver x antal primtal.

```
1  let
2    function Primes x
3      begin
4        let
5          num and div
6        in
7          num assignment 2
8          followed by
9          print num
10         followed by
11         num assignment 3
12         followed by
13         while num less or equal x
14           begin
15             div assignment 3
16             followed by
17             while num modulo div is not 0
18               begin
19                 div assignment div add 2
20               end
21             followed by
22             if div is num
23               begin
24                 print num
25               end
26             followed by
27             num assignment num add 2
28           end
29     end
30 in
31   begin
32     call Primes 20
33   end
```

Fakultet

MOCCA implementation af fakultet.

```
let
  function Factorial x
    begin
      if x less or equal 1
            begin
              return 1
            end
      followed by
      if x greater than 1
            begin
              let
                z and y
              in
                begin
                  z assignment x subtract 1
                  followed by
                  y assignment x times call Factorial z
                  followed by
                  return y
                end
            end
    end
in
  begin
    call Factorial 5
  end
```