Komet

Programmeringssprog til robotter i Minecraft

Project Report

Group: SW409F15

Aalborg Universitet
4. Semester Software
Selma Lagerløfs Vej 300
DK-9220 Aalborg

AALBORG UNIVERSITET
STUDENTERRAPPORT

4. Semester
Software
Selma Lagerhøfs Vej 300
9220 Aalborg
http://www.cs.aau.dk

Title: Komet
Theme: Design, definition og implementation af programmeringssprog.
Project period:
2. febuar 2015 – 27. maj 2015
Projectgroup:
SW409F15
Group members:
Caspar Rosgaard Kuchartik
Jesper Jensen
Nikolaj Smed
Thomas Krause-Kjær
Thomas Pilgaard Nielsen
Tristan Bendixen
Supervisor:
Claus Skaaning
Number of copies: 2
Number of pages: 100
Ended : 27. maj 2015

Abstract:

Collecting resources in Minecraft is a time consuming task. Different mods tries to make the player's life easier by adding robots which collect resources. These robots have to be programmed in languages, which is not designed to write robots. We have attempted to create programming language, with special construct, which should make programming a robot in Minecraft easier.

Preface

Denne rapport er udarbejdet af fjerdesemesters softwarestuderende på Aalborg Universitet, i foråret 2015. Det er nødvendigt at have en basis viden for programmering, for at få fuld forståelse for rapporten.

Projektforslaget, som dette projekt er baseret på, blev stillet af Claus Skaaning, hvem også er vejleder for projektgruppen.

Kildekoden for kompileren og præprocessoren, som er blevet udviklet i dette projekt, findes på den vedlagte DVD.

På DVD'en findes desuden:

- Den kompilerede kompiler
- Den kompilerede præprocessor
- Biblioteksfilerne: stdlib.lua og robot.lua
- Readme
- Rapporten i PDF–form
- Testprogram
- Test suite

Aalborg University, 26. maj 2015

Caspar Rosgaard Kuchartik
ckucha13@student.aau.dk

Jesper Jensen
jejens13@student.aau.dk

Nikolaj Smed
nsmed13@student.aau.dk

Thomas Krause-Kjær
tkraus13@student.aau.dk

Thomas Pilgaard Nielsen
tpni13@student.aau.dk

Tristan Bendixen
tbendi13@student.aau.dk

Indhold

0.1 Indledning

Minecraft er et populært spil, hvor man kan bygge sin egen verden. Til dette kan der gøres brug af robotter, også kaldet bots, som kan hjælpe spilleren. Disse bots skal overvejende konfigureres i almindelige programmeringssprog, som f.eks. JavaScript eller Lua. Dette giver motivation for undersøgelse af hvordan et dedikeret programmeringssprog til botting i Minecraft ville se ud, hvilket giver følgende initierende problemstilling:

Hvordan vil et domænespecifikt programmeringssprog til botting i Minecraft se ud?

Kapitel 1

Minecraft

I dette kapitel vil spillet Minecraft blive undersøgt og nogle af dets karakteristiske komponenter beskrevet.

Minecraft er et såkaldt open world computer spil med over 100 millioner registrerede brugere[29]. Disse brugere findes primært i den vestlige verden, hvor USA, Storbritannien og Canada står for hovedparten af downloads af Minecraft[17].

Selve spillet består af to spiltyper: *Survival* (overlevelse) og *creative* (kreativ). I Survival skal man overleve mod monstre, selv skaffe ressourcer som f.eks. træ, jern, diamant osv. som f.eks. kan bruges til at konstruere våben eller redskaber. Mad skal også indsamles, da sult er en faktor. Der er forskellige sværhedsgrader i survival, som afgør hvor meget skade monstre giver.

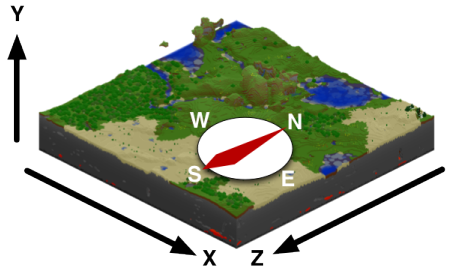
I Creative kan der bygges forskellige ting med de blokke, som er i spillet. Man behøver ikke selv indsamle dem, da man kan finde alle blokke i en menu i spillet. I creative kan man flyve rundt, hvilket gør det nemmere at bygge.[25]

Koordinater

Minecraft repræsenterer verdenen som et tredimensionelt koordinatsystem. De tre dimensioner i rummet bliver ofte omtalt som X, Y, og Z med retninger som vist i figur 1.1 på modstående side. Det er her værd at bemærke, at y–aksen og z–aksen er byttet rundt i forhold til den gængse matematiske brug af tredimensionelle koordinatsystemer.

I denne inddeling har det øverste nord-vestlige hjørne af hver *blok* et koordinat bestående af hele tal. Da der ikke er noget mellemrum mellem hver *blok*, bliver disse derfor præcist 1x1x1 enhed store, denne enhed er ofte omtalt som værende en meter.

Denne opbygning af verdenen som et koordinatsystem gør det oplagt at samle punkterne til en vektor. På denne måde kan omfattende operationer på flere koordinater blive udført på en simpel måde.



Figur 1.1: Koordinatsystemet i Minecraft

Blokke

I Minecraft er verdenen opbygget af kubiske blokke. Den meget firkantede verden, bestående af disse blokke, er kendetegnende for spillet[31].

Figur 1.2: Minecrafts kubiske univers.

Disse blokke har forskellige egenskaber, der alle har en betydning for spillet. Da der ikke kan tages et koordinat ud af et koordinatsystem, så består "tomme" blokke af blokken luft, som spilleren kan bevæge sig i. Øvrige blokke er primært opdelt i solide og flydende, hvor der i standard–spillet, ofte refereret til som *vanilla*, er to typer af flydende blokke: Vand og lava.

Solide blokke er igen opdelt i flere forskellige undergrupper, fra de mest almindelige byggematerialer som træ og forskellige stenarter, samt typer af malm, til de mere eksotiske blokke. Nogle blokke er påvirket af tyngdekræften, som f.eks. grus. Næsten alle bloktyper i Minecraft kan smadres, hvilket oftest kræver værktøj, hvorefter ressourcen kan samles op. Det at en ressource kan samles op repræsenteres som en miniatureudgave af den pågældende ressource, som svæver et lille stykke over jorden, samt er i bevægelse på stedet. På denne måde tydeliggøres det, at den kan samles op.

De opsamlede ressourcer kan efterbehandles til noget andet. Malm kan eksempelvis omsmeltes, i en smelteovn, til en renere form, der så kan kombineres med andre materialer til at danne værktøj, nye blokke eller andet.

Om spilleren er i gang med at indsamle ressourcer til sine projekter, eller har gang i at bygge på et af disse, så benyttes der af blokke. Bygninger kræver placering af blokke til den ønskede form, ligeledes er diverse maskiner og andre hjælpemidler også i blokform.

Blokke identificeres ved hjælp af et unikt navn, der så benyttes til at give hver bloktype et ID'er, som dynamisk tildeles af spillet. Disse ID anvendes til unikt at identificere typer af blokke, så man essentielt kan se hele den tredimensionelle verden, som et tredimensioneret array af heltal.

Blokke er derfor en central del af spillet Minecraft.

1.1 Mods

I Minecraft kan der gøres brug af modifikationer, såkaldte *mods*, til at ændre forskellige ting i spillet. Mods er lavet af spillere, og har ikke nogen relationer med udviklerne bag Minecraft. Det er næsten kun fantasien som sætter grænser mht. mods.

Med hensyn til automatisering af forskellige opgaver i Minecraft, findes der flere forskellige mods og programmer, som gør det muligt. Et eksempel er programmet *Mineflayer*, som emulerer en klient, som kodes i JavaScript til at gøre hvad brugeren ønsker[23]. Et andet eksempel er programmet *SpockBot*, som fungerer på samme måde som *Mineflayer* dog skrives koden i Python[18]. Et tredje eksempel er moddet *OpenComputers*, hvor der bruges robotter, som programmeres i Lua[9].

En grund til at spillere spiller survival, i stedet for creative, er at de gerne vil have noget udfordring. De vil gerne have den følelse af, at de selv bygger ting, indsamler ressourcer o.l. i stedet for at gå ind i en menu i spillet og vælge de blokke som de skal bruge, som i creative spiltype.

1.1.1 OpenComputers

Fokus for dette projekt er at udvikle et programmeringssprog, som gør det nemmere at skrive robotter i Minecraft. For at holde fokus på denne opgave undersøges mulighederne for at bruge en modifikation til spillet, som tilføjer et robotelement samt andre ændringer til spillet for at få robotten til at fungere. Dette gøres for at fokusere arbejdet omkring sproget og ikke selve implementeringen af robotten i Minecraft.

En modifikation til spillet, som er blevet undersøgt i forbindelse med robotter i Minecraft, er *OpenComputers*. Modifikationen tilføjer en række elektroniske komponenter, som bruges til at bygge computere og robotter inde i spillet, som spilleren kan interagere med. Computerne og robotterne har deres eget styresystem, *OpenOS*, som er skrevet i Lua. Robotterne og computerne kan udføre Lua kode på samme måde, som en normal computer kan udføre bytecode, hvilket vil sige de kan køre koden "natively". Det er derfor muligt at kode i Lua, hvorefter koden kan køres på en robot, som udfører instruktionerne. Selve kodningen kan foregå inde i spillet, da operativsystemet, *OpenOS*, har en indbygget teksteditor. Koden kan også skrives udenfor spillet, i andre teksteditorer, såsom Notepad, Notepad++, Sublime Text osv. Kode skrevet udenfor selve

spillet kan derefter gemmes i Minecraft spilmappen, hvorfra computerne eller robotterne inde i spillet kan tilgå den. [9]

Fordelen ved *OpenComputers* er, at det tilføjer blokke og elementer inde i spillet, så spilleren først skal bygge robotten og derefter kode den, hvilket ligger mere op ad Minecraft stilen med at samle ressourcer og konstruere ting. Modsat dette er SpockBot og Mineflayer, da begge disse modifikationer bygger på at emulere en klient version af spillet, og på den måde tilføje en robot. En emuleret klient gør også, at selve spilleren ikke kan bruge sin version af Minecraft, mens robotten kører, med mindre spilleren har to kopier af Minecraft.

1.2 Målgruppe

Som der nævnes i afsnit 1.1.1 på side 4, vil der i dette projekt blive udarbejdet et programmeringssprog, som gør det lettere at programmere robotter i Minecraft. Det er dog ikke tiltænkt, at programmeringssproget skal bruges som introduktionssprog til programmering. Der findes allerede mods og lignende til Minecraft, som har til hensigt at introducere børn til programmering, f.eks. CodeBlocks, som bruger blokke inde i spillet til at programmere en robot [13].

Det sprog der udarbejdes skal i stedet gøre det lettere for personer, der allerede har basale programmeringsevner, at programmere robotter i Minecraft, ved hjælp af mod'et *OpenComputers*. For at være en del af målgruppen for sproget, bør man også have en interesse i at spille Minecraft. Grunden til at nogle vælger at bruge robotter i spillet, er at der er nogle opgaver, der skal udføres for at avancere i spillet, som ellers kan være meget tidskrævende og kedelige. Et eksempel på dette er ressourceindsamling. Der er forskellige mods der kan hjælpe en med dette. Udover robotter kan der bruges maskiner, som graver et stort hul og samler alle ressourcerne ind, som den støder på. Da projektet har til formål at lave et programmeringssprog til kodning af robotter, er målgruppen for projektet brugere, som har interesse i at kombinere spil og programmering. Det antages, der er brugere der har denne interesse, da der er brugere, der har kodet programmer til robotter i deres fritid.

Ud fra ovenstående kan følgende kriterier for projektets målgruppe opstilles:

- Brugeren har basale programmeringsevner.
- Brugeren har interesse for programmering.
- Brugeren spiller Minecraft.
- Brugeren har ønske om at kombinere spil og programmering.

Kapitel 2

Problemformulering

For at finde frem til problematikker i Minecraft, er spillet, samt dets målgruppe, blevet undersøgt. Ud fra undersøgelserne er der fundet frem til at brugerne, som vælger spiltypen *survival* i stedet for *creative*, gerne selv vil lave eller finde materialerne, der bruges under opbygningen af diverse konstruktioner. Indsamlingen af materialer er en meget tidskrævende proces, hvis brugeren vil lave store konstruktioner. Ved brug af robotter kan indsamlingen af materialer automatiseres, uden at fjerne indsamlingsfasen fra *survival*, som brugerne gerne vil have [20].

For at gøre ressourceindsamlingen lettere for brugeren, er muligheder for at inkorporere robotter i Minecraft, blevet undersøgt. *OpenComputers* blev i afsnit 1.1.1 på side 4 vurderet til at være den mod, der bedst implementerer robotter til spiltypen *survival*. Til at programmere robotter i *OpenComputer* benyttes Lua, som er et *General-purpose language*. Det betyder, at sproget er designet til at løse problemer i et bredt domæne. I dette projekt ønskes det at udvikle et programmeringssprog, der specifikt er designet til programmering af robotter i Minecraft.

Problemformuleringen for projektet:

> *Hvordan kan et programmeringssprog designes til at programmere robotter i Minecraft, som samtidigt er beregnet til brugere med erfaring i programmering.*

Til at støtte problemformuleringen er der formuleret en liste med underpunkter til specificering af problemformuleringen.

- Sproget skal kunne interagere med OpenComputers i Minecraft.
- Sproget skal have specifikke sprogkonstruktioner til robotter.
- Der skal være en fordel ved brug af det udviklede sprog til robotter i Minecraft end allerede eksisterende sprog.
- Erfarende brugere skal kunne starte på det udviklede sprog uden at have problemer med syntaksen.

Kapitel 3

Teori

3.1 Kriterier

Til at vurdere et programmeringssprog skal der opsættes en målestok, for hvordan det skal vurderes. Denne målestok gives i form af en række kriterier, som vil blive beskrevet i dette afsnit. Afsnittet afsluttes med, hvilke kriterier der bliver valgt at fokuseres på i udviklingen af et programmeringssprog til styring af robotter, til at finde ressourcer, i Minecraft. Afsnittet vil blive skrevet ud fra afsnit 1.3 i Concepts of Programming Languages[32].

Kriterierne deles ind i tre hoveddele: Læsbarhed, skrivbarhed og pålidelighed.

Læsbarhed: Læsbarhed er, hvor let et program kan læses og forstås. Læsbarhed skal ses i domænets kontekst, f.eks. hvis et program, som skal lave en bestemt form for beregning er skrevet i et sprog, som ikke er beregnet til dette, så kan programmet virke unaturligt og indviklet.

Skrivbarhed: Skrivbarhed er, hvor let et programmeringssprog kan blive brugt til at skrive programmer til et valgt problemdomæne. De fleste karakteristikker ved et programmeringssprog, som har indflydelse på skrivbarheden, har også indflydelse på læsbarheden, da det at skrive programmer ofte kræver at genlæse det, man allerede har skrevet. Som ved læsbarhed skal skrivbarhed ses i domænets kontekst, da f.eks. Visual Basic er bedre til at skrive brugergrænseflader end C, men hvor C er bedre til at skrive systemprogrammer, som f.eks. operativsystemer[32, s. 13].

Pålidelighed: Et program anses som pålideligt, hvis det under alle forhold opfører sig efter hensigten. Pålidelighed er påvirket af læsbarhed og skrivbarhed. Dette kan vise sig i form af et program, som er skrevet på en måde, som ikke er naturligt i det brugte programmeringssprog. Et sådan program har mindre sandsynlighed for at være korrekt. Det vil ligeledes være svært at læse og deraf svært at ændre i.

	Kriterier		
Karakteristik	Læsbarhed	Skrivbarhed	Pålidelighed
Simplicitet	•	•	•
Ortogonalitet	•	•	•
Datatyper	•	•	•
Syntaksdesign	•	•	•
Understøttelse af abstraktioner		•	•
Ekspressivitet		•	•
Typetjek			•
Undtagelseshåndtering			•
Begrænset aliasing			•

Figur 3.1: Oversigt over kriterier for et programmeringssprog. [32]

3.1.1 Karakteristikker

I følgende afsnit vil de forskellige karakteristikker, som hovedkriterierne består af, blive beskrevet. På figur 3.1 kan der ses hvilke karakteristikker, som har indflydelse på de tre kriterier.

Simplicitet: Simplicitet handler om, hvor simpelt et programmeringssprog er. F.eks. hvis et programmeringssprog er meget komplekst, kan der være en tendens blandt programmører til at lære et subset af sproget, og ignorere andre features. Et komplekst programmeringssprog kan også gøre et skrevet program uoverskueligt. Et eksempel på en feature som går mod simplicitet er features, som kan skrives på flere forskellige måder, f.eks. som i Java, hvor disse fire udtryk, pånær enkelte kontekster, betyder det samme:

```
count = count + 1
```

```
count += 1
```

```
count++
```

```
++count
```

Ortogonalitet

Ortogonalitet i et programmeringssprog betyder, at et relativt lille sæt af primitive konstruktioner kan kombineres på mange måder, hvorpå det danner kontrol og datastrukturerne af programmeringssproget. Desuden er alle mulige kombinationer af primitive konstruktioner lovlige og meningsfulde. F.eks. hvis man ser på et programmeringssprog med datatyperne: *integer*, *float*, *double* og *character* samt typeoperatorerne: *array* og *pointer*. Hvis de to typeoperator kan blive anvendt på dem selv samt datatyperne, så kan et stort antal af konstruktioner defineres. En mangel på ortogonalitet leder til undtagelser af programmeringssprogets regler. Det vil sige, at hvis et sprog har *pointers*, så burde det være muligt at definere en *pointer* til at pege på hvilken som helst datatype i sproget. Ekstrem ortogonalitet kan dog lede til unødvendig kompleksitet, da man stort set frit kan kombinere konstruktioner og fejl kan være svære at finde, da næsten alle kombinationer af konstruktioner er valide.

Datatyper

Datatyper giver mulighed for klassificering af data. Datatyper er f.eks. *integer, float* og *boolean*. F.eks. hvis et sprog ikke har *booleans*, kan en numerisk datatype bruges, som f.eks:

```
run = 1
```

Dette udtryk er uklart, hvor det med *booleans* ville være:

```
run = true
```

Dette udtryk er utvetydigt, da der eksplicit står at *run* er *true*.

Syntaksdesign

Syntaksdesign kan deles ind i to hovedgrupper: Specielle ord samt form og mening.

Specielle ord, også kaldet "reserverede ord", er ord som kun kan bruges i en speciel kontekst, som f.eks. *if*, *while* og *for*.

Metoderne til at definere blokke er vigtige. Blokke, som er kode med eget scope, vil blive beskrevet og forklaret, sammen med scopes, i afsnit 4.8 på side 25. I C-lignende sprog startes blokke med krøllet parentes "{" og sluttes ligeledes med en omvendt krøllet parentes "}". Andre sprog afslutter blokke med ***end***. I BASH afsluttes de med det specielle ord, som startede blokken, skrevet bagfra, så en *if*-blok vil afsluttes med *fi*[24]. En helt anden måde at definere blokke, som gjort i Python, er ved indryk. Tekst, som er i samme blok, indrykkes så de står på linje, og når blokken afsluttes, så slettes et indryk[37].

I programmer, hvor blokke strækker sig over mange linjer og som har mange indlejrede blokke i sig, kan det være nødvendigt at skulle scrolle op og ned i dokumentet for at se hvilke blokke som slutter, hvilket giver dårligere læsbarhed. Nogle sprog, som f.eks. Ada og Fortran 95, gør brug af en version af ***end***, hvor navnet eller typen på blokken er med, som f.eks. ***end if*** eller ***end loop***. Denne måde giver bedre læsbarhed, da der direkte kan læses, hvilken type blok som afsluttes uden at skulle scrolle, hvor der f.eks. C kun kan ses at en blok afsluttes. Dog hvis der sker ændringer i selve blokken, f.eks. ændring fra en *if*-blok til en *loop*-blok, skal der ændres to steder, i stedet for ét.

I nogle programmeringssprog kan specielle ord bruges som variabelnavne, som i sproget Fortran 95. Der kan f.eks. ***end*** og ***do*** bruges som variabelnavne, hvilket kan resultere i forvirring omkring et program, som gør brug af specielle ord i variabelnavne.

Form og mening omhandler at designe udtryk, så de til dels viser hvad de skal bruges til. Semantikken skal komme fra syntaksen. Dette vil blive brudt af f.eks. en konstruktion, som betyder forskellige ting afhængig af konteksten. Et eksempel på dette er C's *const*. Hvis en variabel deklareres *const int *var*, skabes der en almindelig *pointer*, men det som den peger til, i dette tilfælde en *int*, kan ikke ændres. Hvis deklarationen lyder *int *const var*, så resulterer det i en *pointer* til en *int*, hvor pointeren ikke kan ændres, men *int*'en kan godt[10].

Understøttelse af abstraktioner

Abstraktion betyder, at det er muligt at bruge komplicerede strukturer eller operationer på en måde, hvor der kan ses bort fra mange af detaljerne. Et eksempel på abstraktion er funktioner, som kan implementere kode, som skal stå flere stedet i programmet. Uden funktioner ville koden kopieres rundt i resten af programmet, hvilket gør koden unødigt længere og mere kompleks.

Ekspressivitet

Ekspressivitet er når et programmeringssprog har lette og praktiske måder at specificere beregninger på. Et eksempel på dette er C-udtrykket:

```
count = count + 1
```

Som kan skrives som:

```
count++
```

I begge udtryk bliver variablen *count* talt én op. Det andet udtryk er kortere at skrive og det udtrykker på den simplere måde, at *count* bliver talt én op. Dette gør det lettere at læse og forstå, hvad der sker.

Typetjek

Typetjek er hvorpå typerne i programmet testes for fejl enten under kompilation eller udførelse af programmet. I en ældre version af programmeringssproget C var der ikke typetjek[32]. Dette betød, at hvis en funktion forventede at få en *integer* ind som parameter, så blev der ikke tjekket på, om funktionen rent faktisk blev kaldt med en *integer*, eller om det var en *float*. I dag bliver alle parametre typetjekket i C.

Et typesystem giver programmøren sikkerhed for, at det data, som en funktion får ind, altid er af den forventede type. Dette gør, at typefejl findes ved kompilering af programmet, så programmøren ikke manuelt skal typetjekke i koden, og der vil ikke forekomme typefejl under kørslen af programmet.

Undtagelseshåndtering

Undtagelseshåndtering er, når et program kan opfange fejl under programudførelse, korrigere fejlen og fortsætte programudførelsen. Sprog som f.eks. Java og C# har undtagelseshåndtering, hvor andre sprog som C og Fortran ikke har.

Begrænset aliasing

Løst defineret er aliasing, når to eller flere navne kan bruges til at tilgå det samme hukommelse. Det er bredt accepteret, at aliasing er en farlig feature i et programmeringssprog[32, s. 16]. I mange sprog kan aliasing optræde som to *pointere*, som peger på det samme hukommelse.

Ud fra beskrivelsen af de forskellige karakteristikker, som har indflydelse på hovedkriterierne, så vælges der at fokuseres på læsbarhed og skrivbarhed. Det er et valg der tages, da programmeringssproget, som skal udvikles, skal bruges til at programmere robotter, til at finde ressourcer, i Minecraft. Det forventes ikke, at programmet har en lang levetid, da koden som skrives, skrives til specifikke problemstillinger i Minecraft og derefter kan smides ud. Da koden skal bruges sammen med Minecraft og ikke skal styre kritiske komponenter, kan der ses bort fra sikkerhed og pålidelighed.

Under beskrivelsen af Komet, i kapitel 4 på side 18, vil der blive argumenteret for, hvordan udvalgte sprogkonstruktioner passer ind i ovenstående kriterier, og derved sikres det, at disse kriterier bliver taget i betragtning i udviklingen af Komet.

3.2 Programmeringsparadigmer

I det følgende afsnit vil tre programmeringsparadigmer blive beskrevet: Imperativ, objektorienteret og funktionel. Et programmeringsparadigme er et regelsæt, for hvordan kode skal opbygges. Beskrivelsen af de tre paradigmer baseres på bogen "Concepts of Programming Languages"[32]. Til sidst i afsnittet vil der blive diskuteret hvilket paradigme, der passer bedst til det domæne, der fokuseres på i dette projekt.

3.2.1 Imperativ

Det imperative programmeringsparadigme er baseret på Von Neumann arkitekturen, som er en computerarkitektur. Næsten alle moderne computere er baseret på Von Neumann arkitekturen. Da det imperative paradigme er baseret på computerarkitekturen, er udførelsen af koden mere effektiv, end kode skrevet i andre paradigmer. I det imperative programmeringsparadigme beskrives *hvordan* en opgave skal udføres. Dette betyder, at der kan bruges mange linjer kode til at udføre simple opgaver.

Karakteristisk for det imperative paradigme er, at et program skrevet i paradigmet har en mængde tilstande, som den bevæger sig imellem. Programmets tilstand ændrer sig gennem udførelsen af programmet. Denne ændring faciliteres af operationers sideeffekter. Derfor skal programmøren være opmærksom på programmets tilstand og være bevidst om operationers sideeffekter. I praksis betyder det, at det samme input til en given funktion, kan have forskellige resultater, afhængigt af programmets tilstand, når funktionen bliver kaldt.

Der kan bruges løkker til at iterere over instruktioner et bestemt antal gange, ved at iterere indtil en variabels værdi er nået. Eksempler på programmeringssprog i det imperative paradigme er Fortran og C.

3.2.2 Objektorienteret

Det objektorienterede paradigme bygger på tre koncepter: Indkapsling, nedarvning og polymorfisme.

Indkapsling betyder, at data og funktionalitet grupperes i den samme enhed, som kaldes et objekt, som er en instans af en klasse. Samtidig giver det mulighed for at skjule noget data for omgivelserne. I den virkelige verden kunne en klasse være "person" og objekter af denne klasse kan være "Anna" og "Jeppe".

Nedarvning gør, at en klasse kan have de samme egenskaber som en anden klasse, samt nogle ekstra egenskaber. Dette gør, at kode kan genbruges.

Polymorfisme betyder, at det er muligt, at to klasser kan nedarve fra samme superklasse, og de to klasser kan derefter omdefinere funktionaliteten i superklassen uafhængigt af hinanden.

C++ og Java er eksempler på objektorienterede sprog.

3.2.3 Funktionel

I det funktionelle paradigme, modsat det imperative paradigme, beskrives der *hvad* der ønskes udført. Udførelsen afhænger herefter af implementeringen af det pågældende sprog. Dette betyder, at programmøren skal skrive færre linjer kode sammenlignet med det imperative- og det objektorienterede paradigme. Derimod skal systemet selv stå for at generere mere kode.

Ved brug af det funktionelle paradigme ændres der ikke på et programs tilstand under udførelsen af programmet. Dette betyder, at når der kaldes en funktion med samme input flere gange, opnås der altid det samme resultat.

Gentagelse af en funktion i et funktionelt sprog opnås ved, at en funktion kalder sig selv, der bruges altså rekursion.

Et eksempel på et funktionelt programmeringssprog er Haskell.

3.2.4 Valg af programmeringsparadigme

I dette projekt vil der blive udarbejdet et programmeringssprog under det imperative paradigme. Dette er valgt, fordi det imperative paradigme er det simpleste af de tre nævnte paradigmer. De funktionaliteter, som de to andre paradigmer tilføjer, opvejes ikke af deres kompleksitet.

Det funktionelle paradigme medbringer sin egen kompleksitet. Da koden til programmeringen af en robot i Minecraft, som regel er forholdsvis kort, vurderes det ikke at det funktionelle paradigmes kompleksitet opvejes af dets udeladelse af tilstande.

Der vurderes ikke at være behov for elementerne fra det objektorienterede paradigme. Robotten er det eneste objekt, som ville få både data og instruktioner, hvilket vil gøre indkapsling overflødigt. Nedarvning er der ikke brug for, da robotten ville få alle instruktionerne knyttet til sin klasse. Til at holde styr på hvilke blokke der har de samme egenskaber, kunne nedarvning bruges. Fordelen ved denne funktionalitet vurderes ikke proportionelt til den tilføjede kompleksiteten af et objektorienteret system.

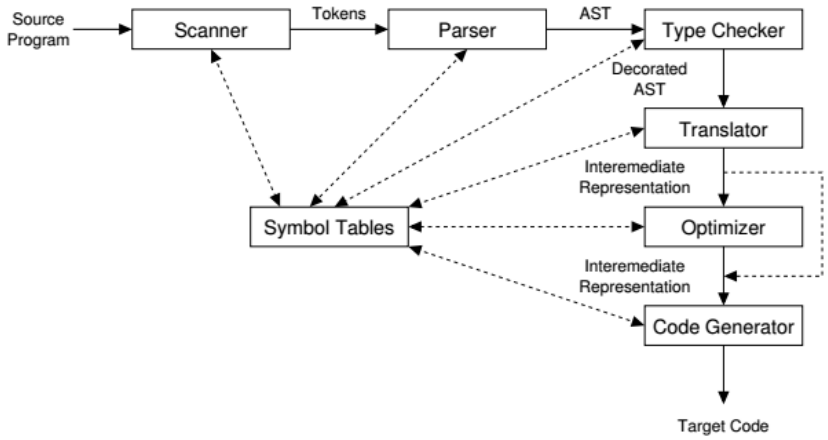
3.3 Udførelsesmuligheder

Indenfor datalogien findes der to muligheder, når det kommer til oversættelse og udførelse af kode. Den ene mulighed er at kompilere hele koden på én gang til *target language* via en kompiler. Den anden mulighed er at lade et andet program, en fortolker, oversætte programmet løbende. I dette afsnit vil begge muligheder blive beskrevet, og til sidst vælges den løsning, som er bedst til dette projekt.

Kompiler

En kompiler er et program, der oversætter det inputsprog den får, også kaldet *kildekode*, til et andet sprog, kendt som *target language*. Det sprog, der bliver oversat til, kan være objektkode, assemblerkode eller et andet sprog på et højere niveau. Ideen med en kompiler er at kunne skrive et program i et højniveausprog, og derefter transformere det til et niveau, som en computer kan forstå, eller til et andet sprog.

Måden hvorpå en kompiler fungerer, er at kildekoden gennemløbes, én eller flere gange afhængig af kompilertypen, hvor der ud fra indholdet genereres en mængde *tokens* svarende til teksten i koden. Samtidigt sikres der, at syntaksen i det skrevne kode er korrekt, da det ellers ikke ville kunne forstås af kompileren, og den ville derfor give en fejlbesked eller crashe. Ud fra disse *tokens* dannes et abstract syntax tree (AST). På AST'et kan der udføres semantisk analyse, som indebærer typetjek og scopetjek. Herefter oversættes indholdet af AST'et til andet kode, hvad enten det er et andet programmeringssprog eller maskinkode. Der kan være yderligere faser i en kompiler, hvor koden optimeres, for at blive hurtigere eller bruge færre ressourcer. På figur 3.2 ses de faser, som en kompiler kan have, og i hvilken rækkefølge de forskellige faser udføres.[12]



Figur 3.2: Faser i en kompiler. [12]

Kompilere anvendes for at kunne programmere i et højniveausprog, som er lettere at skrive

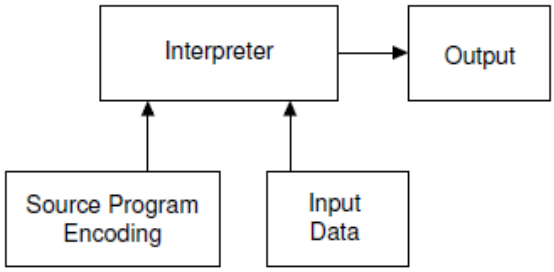
og læse for programmøren, end f.eks assemblerkode. De fleste kompilere kan også indikere syntaksfejl i koden til programmøren, når koden bliver kompilet, hvor programmøren herefter kan rette disse. Da en kompiler finder fejl ved kompilationen af programmet, giver det en form garanti for, at programmet kan køre uden at bryde sammen. Nogle kompilere har desuden indbyggede optimeringsmoduler, som ændrer på kodens struktur, således denne optimeres. Et eksempel på en kompiler som optimerer koden er GCC kompileren [2].

Fortolker

I den kommende sektion vil der blive uddybet hvad en fortolker er, og hvornår den kan bruges i stedet for en kompiler. Det vil blive undersøgt, hvordan en fortolker adskiller sig fra en kompiler. Afsnittet er skrevet ud fra bogen "Crafting a Compiler" [12].

En fortolker er et program, som direkte eksekverer et input, der består af instruktioner givet i et programmerings eller scriptingsprog. Den skal ikke først kompilere programmet før det eksekveres. Da programmet ikke kompileres, som i en kompiler, gives der ikke den samme garanti for, at der ikke kan forekomme fejl, og derved er der større risiko for, at programmet bryder sammen. En fortolker ser brugerprogrammet som almindelig data, som den kan manipulere med og derved forbliver fortolkeren fokuspunktet for eksekveringen.

På figur 3.3 ses en skematisk fremvisning af en fortolker.



Figur 3.3: Eksempel på hvordan en fortolker fungerer fra "Crafting a Compiler" [12]

Udførelsen af et program gennem en fortolker tager længere tid, end udførelsen af et tilsvarende program kompileret til maskinkode. Når koden ændres og en kompiler skal kompilere på ny, eksempelvis når et program er under udvikling, er en fortolker hurtigere[7]. En fortolker er langsommere til at eksekvere et program flere gange, da den skal analysere alle erklæringer hver gang programmet køres.

Nogle systemer tillader samarbejde mellem en kompiler og en fortolker, hvor de kan kalde hinanden og udveksle variabler. Dette medfører, at et program først kan testes med en fortolker, og derefter blive kompileret for at optimere eksekveringshastigheden. En fortolker er derfor nyttig under udviklingen af programmer, hvor det ønskes løbende at lave ændringer og validere outputtet. Under udviklingen af software kan en fortolker forøge udviklernes effektivitet, da koden konstant ændres, og en kompiler ville skulle kompilere den på ny. En fortolker kan være mere effektiv til undervisning, hvor de studerende undersøger og ændrer programmet, i forsøget på at forstå ethvert element af koden. Et eksempel på en fortolker kan være en browser,

som bruger en layout-motor, der som standard kan fortolke HTML, XML, JavaScript og CSS. Ved brug af en browser fortolkes der løbende, da klient og server udveksler data, som skal genfortolkes og vises til brugeren [33].

Valg af udførelsesmetode

Det vurderes til sproget, som skal udvikles, at en kompiler vil være den bedste løsning til oversættelse af kode. Dette valg baseres på, at fejl i koden vil kunne findes med det samme, hvorved det giver færre fejl under eksekveringen.

I næste sektion vil kontekstfrie grammatikker blive beskrevet, da disse skal bruges til definering af et programmeringssprog.

3.4 Kontekstfri grammatik

Formelt set er et sprog et sæt af endelige tekststrenge (*eng.: strings*) over et endeligt alfabet. En kontekstfri grammatik er en kompakt repræsentation af et sprog defineret ud fra følgende fire komponenter:

- Et endeligt terminalt alfabet Σ, hvilket er sættet af *tokens*, hvor \$ signalerer slutningen på inputtet. En token er et leksikalt element (videre uddybelse af tokens i afsnit 5.2 på side 38).
- Et endeligt ikke-terminalt alfabet N. Ikke-terminaler kan også ses som *variabler* i grammatikken.
- Et startsymbol S.
- Et endeligt sæt at produktioner P. En produktion skrives som $\mathrm{A} \rightarrow \alpha$.

Den formelle definition på en kontekstfri grammatik (CFG) er $G = (N, \Sigma, P, S)$.

Helt konkret så beskriver en CFG en grammatik til at validere strenge, for at se om de er lovlige i sproget, som CFG'en beskriver.

Et eksempel på en CFG kan ses på listing 3.1.

```
1  A  → B
2     | λ
3  B  → c
```

Listing 3.1: Eksempel på en CFG.

Ud fra grammatikken i listing 3.1 kan der ses, at A kan omskrives til B eller λ. λ angiver den tomme streng. | angiver at A kan omskrives til *B eller* λ. B kan endvidere omskrives til c. Det vil sige at A bliver udskiftet med B eller λ, hvorefter B bliver udskiftet med c, hvis A blev udskiftet til B. Hvis A blev udskiftet med λ, så slettes A.

A og B er ikke-terminale, da de kan omskrives. c og λ er derimod terminale, da de ikke kan omskrives. Ofte bliver ikke-terminale symboler skrevet med stort og terminale med småt, som i eksemplet på listing 3.1.

Denne måde at angive CFG, som ses på listing 3.1 på side 15, hedder Backus–Naur Form (BNF).[32]

En afledning (*eng.: derivation*) af en streng er et skridt mod startssymbolet i CFG'en, eftersom en streng skal kunne afledes til startssymbolet, for at være valid i sproget, som CFG'en beskriver.

Hvis sproget i listing 3.1 på side 15 får strengen *c*, så vil den blive afledt på følgende måde til startssymbolet:

```
A   ⇒ B
    ⇒ c
```

Listing 3.2: Afledning af listing 3.1 på side 15.

3.4.1 Venstre afledning

En venstre afledning (*eng.: leftmost derivation*) er hvor den første ikke-terminale fra venstre bliver afledt først. For at give et eksempel på dette, bruges grammatikken i listing 3.3.

```
1   E       → Prefix ( E )
2           | v Tail
3   Prefix  → f
4           | λ
5   Tail    → + E
6           | λ
```

Listing 3.3: Et andet eksempel på en CFG.[12]

En venstre afledning af grammatikken i listing 3.3 kan ses i listing 3.4

```
E   ⇒lm Prefix ( E )
    ⇒lm f ( E )
    ⇒lm f ( v Tail )
    ⇒lm f ( v + E )
    ⇒lm f ( v + v Tail )
    ⇒lm f ( v + v )
```

Listing 3.4: Et eksempel på venstre afledning.[12]

I listing 3.4 finder afledningen sted, hvor hver linje er en enkelt afledning. Der er givet strengen "f (v + v)".

Det ses, at det er en venstre afledning, da *Prefix* først bliver afledt til *f*, hvorefter *E* bliver afledt. Hele vejen ned igennem afledningen, så er det altid det venstre symbol, som først bliver afledt. Dette betyder at det er det sidste *v* fra venstre, som bliver afledt sidst i den originale streng.

Der står "lm" i afledningen, da det kommer af det engelske *leftmost derivation*.

3.4.2 Højre afledning

En højre afledning (*eng.: rightmost derivation*) er hvor den første ikke-terminale fra højre bliver afledt først. Denne form for afledning kan godt virke ulogisk, da det ikke stemmer overens med den vestlige læseretning.

Et eksempel på højre afledning kan ses på listing 3.5.

```
E   ⇒rm Prefix ( E )
    ⇒rm Prefix ( v Tail )
    ⇒rm Prefix ( v + E )
    ⇒rm Prefix ( v + v Tail )
    ⇒rm Prefix ( v + v )
    ⇒rm f ( v + v )
```

Listing 3.5: Et eksempel på højre afledning.[12]

Som ved venstre afledning, er der givet strengen "f (v + v)". Til forskel fra venstre afledning, bliver *E* afledt før *Prefix*. *Prefix* er det sidste symbol til at blive afledt, da det er det symbol længst til venstre.

Som ved *lm*, kommer "rm" af det engelske *rightmost derivation*.[12]

3.4.3 Extended Backus-Naur Form

Den tidligere definition på CFG kan udvides med Extended Backus-Naur Form (EBNF).

EBNF tilføjer muligheden for at definere valgfrie og gentagende symboler.

Valgfrie symboler sættes mellem kantede parenteser således:

A $\rightarrow \alpha$ [β γ] δ

$\beta\gamma$ er helt valgfrie i denne grammatik, mens α og δ ikke er.

Gentagne symboler sættes mellem krøllede parenteser således:

A $\rightarrow \alpha$ {β γ} δ

β og γ kan gentages et vilkårligt antal gange, men α og δ kan ikke.

EBNF gør det lettere at lave mere omfattende sprog, da der kan tages højde for gentagelser og valgfrie symboler. Det kan også gøre en CFG lettere at læse, da meningen med en bestemt konstruktion kan være nemmere at skrive.[32]

I næste kapitel vil programmeringssproget Komet blive forklaret med dens grammatik og forskellige sprogkonstruktioner.

Kapitel 4

Komet

Eftersom OpenComputers anvender Lua–kode til at programmere robotter, findes her en kort introduktion til Lua, inden sproget Komet, samt dets sprogkonstruktioner, og deres tilsvarende Lua–konstruktioner bliver beskrevet.

4.1 Lua

Lua er et multiparadigme programmeringssprog, der tillader udviklere at programmere imperativt, funktionelt eller en blanding af disse, afhængigt af personlig præference. Lua bruger dynamiske typer til de simple værdier som tal, tekststrenge booleans, nulls, brugerdata og funktioner, og til alt andet benyttes tabeller. Tabeller spænder derfor bredt, og benyttes til simple arrays såvel som komplicerede objektstrukturer, komplet med nedarvning, som implementeres gennem de såkaldte metatabeller.

Lua er et populært valg som scriptingsprog i spil og programmer. Dette skyldes, at Lua er et letvægtssprog. Desuden er det let at binde til funktioner i det andet software. Det giver også en fordel for brugeren af softwaren, da et kendskab til Lua dermed kan benyttes flere steder, i modsætning til at lære en del nye programmeringssprog at kende. [16]

4.2 Sproget Komet

Komet udvikles til at være et sprog, som skal gøre det nemmere at programmere robotter i Minecraft for erfarende programmører.

Komet gør brug af dynamiske typer, så systemet selv bestemmer typen, når en variabel erklæres. Der findes endvidere ingen *null*-type. Typesystemet vil blive beskrevet dybere i afsnit 4.3 på modstående side.

Følgende liste består af enkelte syntaktiske og semantiske regler for Komet.

- *Statements* afsluttes med semikolon (;).
- Variabelnavne kan kun bestå af bogstaver, som er *case sensitive*, så der er forskel på "i" og "I", og de har ingen begrænsning af længden.
- Til *assignment* bruges operatoren ":=".
  - I Komet står variablen på venstre side af assignment operatoren og udtrykket på højre side.
- Kommentarer startes med //, hvor resten af linjen anses som en kommentar.

Da Komet skal bruges til at styre robotter i Minecraft, er der nogle specielle sprogkonstruktioner, som kan facilitere dette. Disse sprogkonstruktioner vil blive forklaret med beskrivelser og eksempler i de følgende sektioner.

Komets fulde CFG kan findes i bilag på afsnit 13.2 på side 84. Den er skrevet specifikt til ANTLR 4, som vil blive beskrevet i afsnit 5.4 på side 47.

4.3 Komets typer

I Komet er der fire datatyper:

- Number, er en heltalsværdi ($\mathbb{Z}$) med en præcision på 52 bit.
- Boolean $\in$`{true,false}`.
- String, som startes med " og afsluttes ligeledes med ", kan bestå af alle tegn.
- List, som består af enten en *rng* eller listelementer. Lister forklares på afsnit 4.11 på side 31.

4.4 Reserverede ord

I tabel 4.1 på den følgende side kan man se en tabel over reserverede ord. Disse ord er reserverede af forskellige årsager, som angives i tabellen. Da de er reserverede, kan de ikke optræde i andre kontekster i koden, end den kontekst de er beregnet til (tekststrenge undtaget). Tabellen er opdelt således, at første kolonne er ordet og anden kolonne er brugen af ordet.

4.5 Operatorer

Komet består af en række operatorer, som kan ses på tabel 4.2 på side 21. Tabellen består af fem kolonner. Første kolonne betegner operatorprioriteten, hvilket er den rækkefølge, som operatorerne bliver behandlet i, f.eks. at multiplikation behandles før addition.

Tredje kolonne betegner operatortype, om en operator er unær eller binær. En unær operator tager én operand, hvor en binær tager to operander.

Ord	**Brug**
and	Bruges til sammenkædning af to variable til en liste eller som logisk "og".
anywherein	Bruges til deklarering af løkkedelen af *case*.
break	Stopper en kørende løkke eller *case*.
case	Bruges til deklarering af *case*-konstruktion.
continue	Bruges til at fortsætte løkkedelen af en *case*.
define	Bruges til definering af en variabel.
do	Bruges til start af scope i løkker og *if*-konstruktioner.
else	Bruges til deklarering af *else*-konstruktion i forbindelse med en *if*-konstruktion.
end	Bruges til at afslutte et scope.
false	Bruges i forbindelse med erklæring et udtryk *false*.
for	Bruges til deklarering af *for*–løkke.
forever	Bruges til deklarering af *forever*–løkke.
func	Bruges til deklarering af en funktion.
if	Bruges til deklarering af *if*-konstruktion.
is	Bruges som en funktion der tjekker om en variabel er en del af en property.
mod	Modulo, resten af en division, binær operator.
not	Logisk negering, unær operator.
or	Bruges som logisk "eller" (union af to variable).
prop	Bruges til at definere en *proplist*.
return	Returnerer fra en funktion.
true	Bruges i forbindelse med erklæring et udtryk *true*.

Tabel 4.1: Tabel over reserverede ord.

Operator-prioritet	Operator	Betydning	Type	Associativitet
1	( )	Parenteser	N_A	Venstre mod højre
2	not	Logisk negering, unær operator.	Unær	Højre mod venstre
3	*	Matematisk multiplikation.	Binær	Venstre mod højre
	/	Matematisk division.	Binær	
	mod	Modulo, resten af en division.	Binær	
4	+	Matematisk addition.	Binær	
	-	Matematisk subtraktion.	Binær	
5	>	Større end.	Binær	
	<	Mindre end.	Binær	
	>=	Større end eller lig med.	Binær	
	<=	Mindre end eller lig med.	Binær	
6	=	Lig med.	Binær	
	!=	Ikke lig med.	Binær	
7	and	Logisk og.	Binær	
8	or	Logisk eller.	Binær	
9	:=	Assignment	Binær	Højre mod venstre

Tabel 4.2: Tabel over operatorer.

Femte kolonne betegner associativiteten af operatoren, om den er associeret højre mod venstre eller venstre mod højre. F.eks. ved addition, vil der blive beregnet fra venstre mod højre, som det foregår i normal algebra. Ved assignment, foregår det højre mod venstre, da udtrykket på højre side bliver assignet til variablen på venstre side af assignment-operatoren.

4.6 Betinget konstruktion

En betinget (*eng.: conditional*) konstruktion giver mulighed for at vælge mellem et eller flere stykker kode ud fra et udsagn. Dette udsagn evaluerer til *true* eller *false*, og derved vælges koden.

En populær betinget konstruktion er *if else*. Generelt opbygges en *if else*-konstruktion som på listing 4.1.

```
1   if udsagn
2       i = 1
3   else
4       i = 2
```

Listing 4.1: Eksempel på *if else*-konstruktion.

Programmeringssprog implementerer *if-else*-konstruktioner forskelligt, men den generelle funktion er det samme.

I eksemplet på listing 4.1 bliver `i = 1` eksekveret, hvis "udsagn" bliver evalueret til *true*, men hvis "udsagn" bliver evalueret til *false*, så bliver `i = 2` kørt. Ofte er *else* valgfrit, som f.eks.

i C. I nogle sprog kan *else if*-konstruktioner sammenkædes til at kombinere flere forskellige betingelser, så bagefter en *if else*-konstruktion, kommer der en *else if*, som kører videre på kæden og, som afsluttes med en *else*.

Brugbarheden i betingede konstruktioner er at kunne vælge forskelligt kode ud fra et input, udsagnet, hvorved der bl.a. kan lave tjeks på værdier eller "hvis udtrykket er det ene, så gør dette"-konstruktioner, koden bliver mere dynamisk med *if-else*-konstruktionen.

I Komets grammatik er *if else* implementeret som vist på listing 4.2.

```
1            | 'if' expr 'do' ifb=block ('else' elseb=block)? 'end'          #If
```

Listing 4.2: Eksempel på *if else*-konstruktion i Komet.

Et konkret eksempel på kode ud fra grammatikken kan ses på listing 4.3.

```
1    if udsagn do
2        i := 1;
3    else
4        i := 2;
5    end
```

Listing 4.3: Eksempel på *if else*-konstruktion i Komet.

Den største forskel fra det generelle eksempel på listing 4.1 på side 21, er at i Komet, efter udsagnet, har ordet "do", som henviser til at kodeblokken, som skal køres, hvis udsagnet er *true*, starter og "end", som afslutter kodeblokken.

I Komet er det ikke muligt at sammenkæde *else if*, så det er kun muligt at have ét *else* pr. *if*-konstruktion, da Komet skal afslutte blokke med "end", vil det ikke give en pæn struktur. I stedet kan *if else* indlejres efter behov.

Komets version af *if-else*-konstruktionen er designet til at fremme læsbarheden. Læsbarheden forøges ved at eksplicit angive hvornår kodeblokken i *if-else*-konstruktionen startes med ordet "do" og afsluttes med "end".

4.6.1 if-else i Lua

I listing 4.4 ses den tilsvarende Lua-kode for if-else konstruktionen vist i listing 4.3, som er skrevet i Komet. Forskellen på de to er, at i Lua skrives der "then" i stedet for "do". Der behøves desuden ikke at blive brugt semikolon til at afslutte statements i Lua.

```
1    if udsagn then
2        i = 1
3    else
4        i = 2
5    end
```

Listing 4.4: `if-else` i Lua.

4.7 Funktioner

Funktioner, metoder eller procedurer er alle navne, som dækker over en abstraktion af kode. Ofte er det programmeringsparadigmet, som afgør hvilken terminologi som bruges, dog vil navnet "funktion" blive brugt fremadrettet.

Helt konkret er kodeabstraktionen, at koden grupperes, så en blok kode kan kaldes og køres på én gang. Det er også muligt at kalde funktioner med parametre, hvilket gør det muligt at sende værdier med fra funktionskaldet.

Funktioner kan afsluttes med *return*, hvorefter programmet kører videre fra, hvor funktionen blev kaldt. *Return* kan, i nogle sprog, returnere en værdi. Hvad der kan returneres, samt hvor mange forskellige værdier, afhænger også af sproget. F.eks. kan man i C returnere alle de basale datatyper, bortset fra et helt *array* og der kan kun returneres én værdi.

Når en funktion kaldes, gøres det via et funktionskald, som indeholder de parametre, som en funktion kaldes med. Det er sprogspecifikt hvordan et funktionskald virker og agerer, men når der returneres fra en funktion, så kører koden videre fra funktionskaldet. Et funktionskalds konsekvenser ligger derfor i ændringer af global tilstand og returværdier.

I Komet erklæres en funktion med ordet *func*, efterfulgt af funktionsnavnet og parametre i parentes, hvor funktionsblokken startes med ordet *do* og afsluttes med *end*. *Return* kan bruges til at returnere fra funktionen, hvor der kun kan returneres én værdi pr. funktionskald. Hvis der skal returneres mere end én værdi, kan værdierne tilføjes til en liste, som kan returneres (se forklaring af lister i afsnit 4.11 på side 31)

Grammatikken for funktioner og funktionskald kan ses på hhv. listing 4.5 og listing 4.6.

```
1            | 'func' NAME '(' namelst? ')' 'do' block 'end'                #Function
```

Listing 4.5: Grammatikken for funktioner i Komet.

```
1 fcall       : NAME paramlst;
```

Listing 4.6: Grammatikken for funktionskald i Komet.

Formelle parametre i funktioner i Komet angives i parentes i funktionsdefinitionen. Flere parametre adskilles med komma.

Ved funktionskald angives funktionsnavnet, samt de faktiske parametre, som funktionen forventer at få ind. Hvis en funktion skal kaldes i sekvens, med et eller flere forskellige parametre i hvert kald, så kan der bruges notationen: `#parm1,parm2,parm3#`, for den varierende parameter. Et eksempel kan ses på listing 4.8 på den følgende side.

Et eksempel på "almindelige" funktionskald kan ses på listing 4.7.

```
1    myFunc(parm1);
2    myFunc(parm2);
3    myFunc(parm3);
```

Listing 4.7: Eksempel på funktionskald i Komet.

På listing 4.8 kan man se et funktionskald, som gør det samme som listing 4.7 på side 23, men som er skrevet på én linje.

```
1    myFunc(#parm1,parm2,parm3#);
```

Listing 4.8: Eksempel på funktionskald i Komet med speciel parameterkonstruktion.

I listing 4.8 bliver `myFunc` kaldt tre gange. Først `parm1`, derefter med `parm2` og endelig med `parm3`. Det er præprocessoren i Komet, som vil stå for at "udfolde" hver parameter til et separat funktionskald.

Det virker ikke kun ved funktionskald, men listing 4.9 er også lovligt, hvor variablerne *a*, *b* og *c* bliver deklareret med værdien 10.

```
1    define #a,b,c# := 10;
```

Listing 4.9: Eksempel på deklaration af variabler med speciel notation.

På listing 4.10 kan man se et eksempel på en funktion i Komet. Linje 1 til 5 er selve funktionen og linje 7 er funktionskaldet.

```
1    func testFunktion(parm1, parm2) do
2        i = parm1 + parm2;
3        j = parm1 mod parm2;
4        return;
5    end
6
7    testFunktion(1, #2,3,4#);
```

Listing 4.10: Eksempel på en funktion i Komet.

Funktioner bruges i Komet, da det i høj grad forbedrer skrivbarheden ved at kunne skrive komplicerede eller lange konstruktioner på en nem måde. Især med `#parm1,parm2,parm3#`-notationen kan antallet af funktionskald skæres ned i bestemte situationer, som vurderes der let kan forekomme i Minecraft.

Funktioner reducerer også koderedundans, da det samme kode kan kaldes med forskellige værdier med varierende funktionskald, hvilket også forbedrer skrivbarheden.

4.7.1 Funktion i Lua

Listing 4.11 viser, hvordan funktionen fra listing 4.10 bliver defineret og kaldt i Lua. Forskellen på definering af funktioner i Komet og Lua er, at der i Lua skrives "function" frem for blot "func" i starten af funktionsdefinitionen. Desuden står der "do" efter parametrene i Komet, mens der i Lua ikke står noget. I Lua findes der ikke et tilsvarende multiparameter funktionskald, som der gør i Komet, hvilket betyder, at alle mulige kombination af funktionskaldet skal kaldes hver for sig.

```
1  function testFunktion(parm1, parm2)
2    i = parm1 + parm2
3    j = parm1 mod parm2
```

```
    return
  end

  testFunktion(1, 2)
  testFunktion(1, 3)
  testFunktion(1, 4)
```

Listing 4.11: Funktion i Lua.

4.8 Scopes

Et scope er en konstruktion, som bruges til at afgrænse synligheden af variabler. Det er sprogspecifikt, hvordan scopes defineres og virker, som tidligere nævnt i afsnit 3.1.1 på side 9. I et scope kan alle lokalt erklærede variabler tilgås, hvor de ikke kan tilgås udenfor scopet. Der kan ikke eksistere to variabler, i samme scope, med samme navn. En variabel siges at være *lokal* i et program eller en blok, hvis den er deklareret der. En variabel kan også være *global* i et program, hvis den er erklæret i det yderste scope.

Der skelnes mellem flere forskellige typer af scopes, som vil blive beskrevet i de følgende afsnit.

Statisk scope

Et statisk scope, også kaldt *leksikalt scope*, er et scope, som defineres før kodeeksekvering. I et statisk scope vil en procedure *p* kun være i stand til at bruge variabler og andre procedurer, som er kendt, når *p* bliver erklæret[21]. Statiske scopes ændres ikke under kodeeksekveringen.

Statisk scope kan i programmeringssprog implementeres på to måder: Med mulighed for indlejrede subprogrammer, hvor der dannes indlejrede statiske scopes, eller uden mulighed for indlejrede subprogrammer. Helt konkret menes der, om der er mulighed for definering af funktioner inde i funktioner. Bl.a. Ada og JavaScript tillader dette, men C gør ikke. I C danner funktioner statiske scopes, men indlejrede scopes kan kun defineres med blokke.

Blok

En blok er definitionen af et scope midt i koden. Blokke tillader en sektion af kode at have egne lokale variabler.

Nogle aspekter af blokke virker forskelligt fra programmeringssprog til programmeringssprog.

I C tillades *compound statements*, f.eks. kroppen af *if*, at have deklarationer og deraf definere et nyt scope. Dette er et eksempel på en blok. Dette scope, som laves af en blok, bliver behandlet i C som scopes dannet af subprogrammer. Et sådan scope kan også indlejres som del af et større scope.

I C og C++ kan variabelnavne godt genbruges i indlejrede blokke, f.eks. hvis en variabel `count` er blevet brugt i et scope, og en indlejret *while*–løkke generklærer `count` i dens blok. Dette er dog ikke lovligt i Java eller C#, da det blev ment, at være en for høj risiko for fejl[32, s. 221].

Et andet eksempel er JavaScript, som bruger statisk scope for nestede funktioner, men hvor blokke, som ikke optræder i form af funktioner, ikke kan defineres.[32]

Dynamisk scope

Dynamisk scope er baseret på kaldesekvensen af subprogrammer og kan deraf kun blive besluttet i *runtime*. Tidligere versioner af LISP samt APL og SNOBOL4 bruger dynamisk scope. Lua gør brug af dynamiske scopes for variabler, som ikke er erklæret med ordet "local".

I et dynamisk scope, søges der først efter en variabel i den lokale funktion, derefter søges der i funktionen som kaldte den lokale funktion, derefter søges der i funktionen som kaldte den funktion og så videre op af funktionskaldsstakken[15]. Det vil sige, at rækkefølgen af funktionskaldene afgør, hvilken variabel som den lokale funktion kan ende med og deraf er scopet dynamisk.

Scopes i Komet

I Komet vil der blive gjort brug af statisk scopes. Dette vil gøres, for at forøge pålideligheden af et skrevet program, da dynamiske scopes kan variere fra programkørsel til programkørsel.

Som i C, dannes der en blok ved f.eks. en *if* eller *for*, som har sit eget scope. Hver gang der mødes en *block* i Komets grammatik, som f.eks. i listing 4.2 på side 22, dannes der en blok. Et scope afsluttes med ordet *end*.

En begrænsning i Komet mht. blokke er, at der kan ikke laves en blok uden, at der er en eller anden form for kodekonstruktion, som danner blokken, som ved *if* eller *for*.

I Komet er variabler globale, hvis de erklæres i det første scope, så de kan tilgås i alle indlejrede scopes. Der er ikke et reserveret ord til at tvinge variabler til at være globale.

4.9 Case–anywherein

Til Komet er der blevet udviklet en speciel *Case–anywherein* feature. Denne feature er ikke som den gængse *Case switch* fra C, men en blok, hvori der altid udføres bestemte stykker kode, *cases*, hvis bestemte betingelser opfyldes i den efterfølgende *anywherein*–blok.

Brugbarheden i denne konstruktion udspringer i muligheden for at kunne have mange forskellige *cases*, som kan tage højde for mange situationer, hvor *case–anywherein* sørger for, at så snart *case* betingelserne bliver opfyldt, køres deres kode. Ved at bruge *case–anywherein* slipper programmøren for at skulle gentage mange tjek igennem kodeblokken. Konstruktionen er brugbar til robotter, da det skal være muligt at tage højde for mange situationer, som robotten kan komme ud for. Samtidigt gør konstruktionen koden kortere, hvis det samme tjek skal udføres

flere gange i den samme blok. Et eksempel på brugen af *case–anywherein* kunne være, at robotten er ved at grave en tunnel, og der så falder noget grus ned foran den. Med *case–anywherein* ville der så være en *case* med en betingelse, som tjekkede om der var grus foran robotten, hvorefter *case*-indholdet ville være et stykke kode, som ville få robotten til at fjerne gruset.

Grammatikken for *case–anywherein* konstruktionen ses i listing 4.12. *Case* og *anywherein* adskiller de to dele af indholdet i konstruktionen. I *case* findes først en betingelse og dernæst de statements der skal udføres, i tilfælde af at betingelsen bliver opfyldt i *anywherein*–blokken. Grammatikken for *case* betingelserne og statements der skal udføres, er beskrevet i en anden del af grammatikken, som ses på listing 4.13, for at gøre grammatikken lettere at læse. Hver *case* efterfølges af en *casecontrol*, som består af enten *break* eller *continue*, alt efter om det ønskes at *anywherein*–blokken skal stoppe eller fortsætte. Produktionsreglen for *casecontrol* kan ses i listing 4.14.

```
1 cse         : 'case' 'do' caseblk+ 'anywherein' block 'end';
```

Listing 4.12: Grammatik for *Case–anywherein* i Komet.

```
1 caseblk     : ('(' expr ')' block casecontrol);
```

Listing 4.13: Grammatik for *caseblok*.

```
1 casecontrol : ('break' | 'continue') ';';
```

Listing 4.14: Grammatik for *casecontrol*.

Case–anywherein featuren er designet til, at det skulle være simpelt at skrive mange tjeks, til store mængder kode, på en smart måde. Herved undgår programmøren at skulle gentage det samme tjek, hver gang en variabel ændres og i det hele taget skulle tænke på, hvornår variablen ændres, da konstruktionen sammen med kompileren selv finder ud af, hvornår variabler ændres og hvornår hvilke betingelser er opfyldt. Desuden øger konstruktionen skrivbarheden, da programmøren ikke skal skrive tjeks nede i *anywherein*–blokken. Læsbarheden øges også, da *anywherein*–blokken indeholder de statements der skal udføres, mens *case*-delene indeholder de betingede konstruktioner og deres statements. Derved bliver det tydeligere, hvad der foregår i *anywherein*–blokken, og hvilke tjeks der foretages i *case*–delen

Case-anywherein i Lua

I Lua findes der ikke en tilsvarende konstruktion til Komets *Case-anywherein*. Det tilsvarende i Lua vil i stedet være et *if*–tjek efter hvert statement. I listing 4.15 ses *Case-Anywherein* konstruktionen i Komet. Tjekket udføres efter hvert statement i *Anywherein*–blokken ved hvert statement, som har indflydelse på tjekket. I listing 4.16 på næste side ses, hvordan det tilsvarende eksempel ville se ud i Lua–kode.

```
1 define a := 0;
2 define b := 1;
3 define c := 0;
4
5 case do
6     (a = b)
```

```
    print("A is equal B");
    continue;
anywherein
    a := 1;
    a := changeA();
    c := b;
end
```

Listing 4.15: Eksempel på *Case-Anywherein* i Komet.

```
a = 0
b = 1
c = 0

a = 1
if a == b then
    print("A is equal B")
end
a = changeA()
if a == b then
    print("A is equal B")
end
c = b
end
```

Listing 4.16: Lua–kode tilsvarende *Case-Anywherein* eksemplet fra listing 4.15 på side 27.

4.10 Løkker

Til Komet er det gjort muligt at bruge løkker i koden. Ideen med løkker er at kunne køre det samme stykke kode flere gange i træk. Dette ses som værende en god egenskab at have i et sprog til at programmere robotter til Minecraft, da de opgaver der typisk ville involvere en robot, ville være ting der skulle gentages flere gange i træk. Et eksempel på dette kunne være en robot, som skulle indsamle ressourcer og bringe dem til et bestemt sted og derefter gentage processen. I Komet er der blevet tilføjet tre typer af løkker.

Foreach–løkke

Den første er *foreach*–løkken, som ses i listing 4.17.

```
            | 'for' var 'in' listVar 'do' block 'end'                    #Forloop
```

Listing 4.17: Grammatik for *foreach*–løkke i Komet.

Syntaksen for løkken består af fire reserverede ord, *for*, *in*, *do* og *end*, hvorimellem der findes en variabel, en liste og en kodeblok. Variablen kan herefter bruges i resten af løkke-koden til at tilgå det element af listen, som løkken går igennem. Brugen af løkken vil være at få robotten til at gentage et bestemt stykke kode det antal gange, som der er elementer i en liste, f.eks. kunne det være, hvis robotten havde fundet seks forskellige ressourcer, og de herefter skulle sorteres efter deres type. Selve syntaksen minder meget om syntaksen fra *foreach*–løkken i C#.

Foreach i Lua

I Lua skal der bruges funktionen `pairs` til at lave en *foreach*-løkke. `pairs` bruges på den tabel der skal itereres over, hvilket giver en funktion, som giver en *key* og den tilhørende *value*. Et eksempel, på hvordan man bruger `pairs`, kan ses i listing 4.19. Dette svarer til *foreach*-løkken, skrevet i Komet, i listing 4.18. `print(i["key"])` printer alle keys, som eksisterer i listen `list`.

```
for i in list do
      print(i["key"])
end
```

Listing 4.18: Foreach i Komet.

```
for k, v in pairs(list) do
      print(k)
end
```

Listing 4.19: Foreach i Lua.

4.10.1 *For*–løkke

Syntaksen for *for*–løkken er den samme som for *foreach*–løkken, vist i listing 4.17 på side 28. Det der bestemmer, om det er en *for*– eller *foreach*–løkke, er hvad `listInit` indeholder. Grammatikken for `listInit` kan ses i listing 4.20. Hvis `listInit` er en `rng` er det en *for*–løkke. `rng` har formatet: "startværdi..slutværdi, tæller".

```
listInit    : rng                                   #RngListInit
            | list                                  #ListListInit
            ;
```

Listing 4.20: Grammatik for `listInit`.

I listing 4.21 ses et eksempel på en `for`–løkke. Tælleren kan udelades, hvorved der bliver talt op med én for hver iteration.

```
for i in 0..10, 2 do
  a = i;
end
```

Listing 4.21: For–løkke i Komet.

For–løkke med multidimensional liste

I Komet er det muligt at iterere over et multidimensional liste med en *for*–løkke, som vist i listing 4.22

```
for i in (x, y, z)[1..10, 2..5, 1..4] do
  var := i;
end
```

Listing 4.22: For–løkke med multidimensional liste i Komet.

For–løkke i Lua

I listing 4.23 ses, hvordan en `for`–løkke skrives i Lua. `i` bliver sat til 0 og tælles 2 op, hver gang løkken har kørt igennem én gang. Dette gøres indtil `i` har værdien 10. Tilsvarende kode i Komet kan ses i listing 4.21 på side 29. I Lua kan tælleren udelades, hvorved `i` tælles op med 1 hver gang løkken har kørt igennem én gang.

```
1  for i = 0, 10, 2 do
2    a = i
3  end
```

Listing 4.23: For–løkke i Lua.

I Lua er der ikke en tilsvarende konstruktion for Komets *for*–løkke. Den enkelte *for*–løkke bliver i Lua lavet om til flere *for*-løkker inden i hinanden. Listing 4.24 viser, hvordan Komets konstruktion for en *for*–løkke med en multidimensional liste, vist i listing 4.22 på side 29, bliver skrevet i Lua.

```
1  for a in 1..10 do
2    for b in 2..5 do
3      for c in 1..4 do
4        i = {x = a, y = b, z = c}
5        var = i
6      end
7    end
8  end
```

Listing 4.24: Imitering af Komets *for*–løkke, med en multidimensional liste, i Lua

Forever–løkke

Den tredje løkke, som er blevet gjort muligt, er *forever*–løkken.

```
1            | 'forever' 'do' block 'end'                                    #Forever
```

Listing 4.25: Grammatik for *forever*–løkke i Komet.

Ideen med denne løkke er at kunne gentage en handling på ubestemt tid. Dette vil være, hvis en robot skulle gentage en opgave og kun stoppe, hvis bestemte betingelser blev opfyldt. I det tilfælde vil det være nødvendigt at bruge det reserverede ord *break*, som stopper den løkke der køres og går videre til koden efter løkkeN. *Break* virker desuden også for de andre løkkeR. *Forever*–løkkeN minder om C og C#'s *while*–løkke. Forskellen fra *while* er, at *forever* ikke laver et tjek for, om en given betingelse er opfyldt, hver gang løkken gennemløbes. Derfor kan *forever*–løkken køre på ubestemt tid, og kun stoppes hvis et *break*–statement mødes.

Brugen af de reserverede ord *do* og *end*, til at vise hvor løkke blokken starter og slutter, øger læsbarheden for programmøren. *Forever*–løkken øger også læsbarheden, da det ud fra løkkens navn er sigende, at løkken kører *"for evigt"* eller indtil at der mødes et *break* statement.

Forever i Lua

Måden hvorpå der laves tilsvarende Lua-kode for Komets *forever*–løkke fra listing 4.26 ses i listing 4.27. I Lua findes der ikke en konstruktion, der som standard kører indtil den bliver stoppet af `break`, så derfor anvendes der en `while`–løkke, hvor dens *condition* er true.

```
1  forever do
2    testFunktion();
3  end
```

Listing 4.26: Forever i Komet.

```
1  while true do
2    testFunktion()
3  end
```

Listing 4.27: Forever i Lua.

4.11 Lister

I Komet kan der gøres brug af lister. Et element i en liste i Komet består altid af en key og en værdi. Hvis key udelades behandles listen som et 0-indekseret array. På listing 4.28 kan produktionsreglerne for selve listen ses. En liste kan bestå af nul eller flere *namelst*. Derefter kan den bestå af nul eller flere *listpart*.

```
1 list        : ('(' namelst? ')')? '[' listpart (',' listpart)* ']';
```

Listing 4.28: Grammatik for `list`.

På listing 4.29 ses produktionsreglen for *namelst*. *namelst* er nøgle for en værdi i en liste. Dette bruges bl.a. til, at en værdi skal kunne findes ud fra dens nøgle. Syntaksen for opslag i en liste kan ses nederst i listing 4.31 på næste side. Nøgler kan bestå af tal, tekststrenge, enkelte bogstaver samt *true* og *false*.

```
1 namelst     : NAME (',' NAME)*;
```

Listing 4.29: Grammatik for `namelst`.

På listing 4.30 kan produktionsreglen for *listpart* ses. *listpart* agerer værdi for en nøgle og kan bestå af enten en *expr* eller en *rng*.

```
1 listpart    : expr
2             | rng;
```

Listing 4.30: Grammatik for `listpart`.

Et eksempel på en liste, og opslag i denne liste, i Komet kan ses på listing 4.31 på næste side, hvor variablen *a* bliver tildelt værdien associeret med *key* 2. “sdsd”

```
define list := (key1, key2, true, string)[1,2,3,4];

define a := list["key2"];
```

Listing 4.31: Eksempel på liste i Komet.

4.11.1 Lister i Lua

Listing 4.32 viser Lua–kode tilsvarende Komet–koden vist i listing 4.31. Forskellen mellem Komet og Lua er, at i Komet opdeles key'sne og værdierne fra hinanden, hvorimod i Lua, er der et assign-tegn mellem key'en og værdien. Listeopslag i Komet og Lua er ens.

```
list = {key1 = 1, key2 = 2, true = 3, string = 4}

a = list[''key2'']
```

Listing 4.32: Liste i Lua.

4.12 *Foreign Functions Interface*

For at robotterne fra *OpenComputers* skal udføre deres opgaver, kaldes funktioner, som følger med moddet i koden. For at Komet skal kunne bruges til at kode robotter, er det nødvendigt at kunne bruge disse funktioner. Dette gøres i Komet ved hjælp af *Foreign Functions*.

Et *Foreign Functions Interface* (FFI) gør det muligt, for et program skrevet i et sprog, at kalde kode skrevet i et andet sprog[1]. I Komet vil FFI'et kunne bruges til at give programmøren mulighed for at kalde funktioner fra *OpenComputers* moddet.

Med et FFI er det muligt at bruge funktioner fra Lua defineret i andre filer. Dette gør Komet kompatibelt med OpenComputers funktioner, da disse er skrevet i Lua.

Komets FFI

I Komet bruges syntaksen fra listing 4.33 til at tilføje en funktion fra en anden fil.

```
            | 'ff' '['SLIT']' NAME'(' params=namelst? ')' ('changes' se=namelst )?';'
    #FFI
```

Listing 4.33: Grammatikken for inkludering af filer som resourcer i Komet.

Inkluderingssyntaksen består af et filnavn, skrevet i firkantede parenteser, efterfulgt af hvilken funktion der ønskes fra filen samt dennes parametre. Desuden skal der oplyses, hvilke variable funktionen ændrer, hvis den ændrer nogen. Herefter kaldes funktionen på samme måde som en funktion defineret i koden, hvilket er beskrevet i afsnit 4.7 på side 23.

I det nuværende FFI er det kun muligt at inkludere filer af typen `.lua`.

Til Komet er der desuden blevet udformet et lille bibliotek, Stdlib, med udvalgte funktioner der kunne være behjælpelige, samt et bibliotek, kaldet Robot, med funktioner til brug i *OpenComputers* moddet. Begge biblioteksfiler kan findes i bilaget i listing 13.2 på side 90 og listing 13.3 på side 90.

Kapitel 5

Kompileropbygning

5.1 Kompiler kompiler

En kompiler kompiler er, som en almindelig kompiler, et program, som tager en CFG og resulterer i en hel eller delvis *frontend*. Ofte skal der tilføjes bestemte ting til, eller redigeres i, en CFG, som en bestemt kompiler kompiler skal modtage.[12] Et eksempel på kompiler kompilere, som kræver modifikationer af CFG'en, er Coco/R. For at kunne fungere, kræver Coco/R, at startproduktionen i grammatikken har samme navn, som der er erklæret i toppen af CFG-dokumentet. Navnet i toppen af grammatikken er ligeledes et krav fra Coco/R.

Java er et populært valg blandt kompiler kompilere, men der findes også kompiler kompilere som kompilerer til C#, C/C++ og andre populære sprog[19].

Til valg af kompiler kompiler i dette projekt er der blevet testet seks kompiler kompilere. De udvalgte kompiler kompilere er blevet valgt efter forslag fra lektor Bent Thomsen fra Aalborg Universitet (AAU). Kompiler kompilerne vil ikke blive testet udelukkende ud fra deres features, da der lægges mest vægt på, hvor nemme de er at bruge. Dette giver subjektive resultater, hvilket der skal tages højde for, når der bliver valgt en kompiler kompiler. Først vil valget mellem kompiler kompiler eller selv at skrive lexer og parser blive diskuteret, derefter vil hver kompiler kompiler blive beskrevet og til sidst findes en konklusion med valget af kompiler kompiler.

Kompiler kompiler vs. selvskrevet lexer og parser

Som beskrevet ovenfor er en kompiler kompiler et program, som ud fra given en CFG danner lexer og parser.

Der kan også vælges manuelt at skrive en lexer og parser. Ved at skrive lexer og parser selv, er der mange scenarier at tage hensyn til. Der vil ikke blive gået i dybden med alle scenarier, men ved manuel skrivning af en lexer skal der f.eks. tages hensyn til reserverede ord: Må man bruge reserverede ord i bestemte kontekster o.l. En anden ting er ydelsen af lexeren: Hvor mange

tegn skal den kunne køre igennem i sekundet. Det kunne også være fejlforbedring: Hvad skal den gøre i tilfælde af fejl.[12]. I dannelse af en parser er der ligeledes scenarier, der skal tages i betragtning.

En kompiler kompiler klarer alt arbejdet for en, da i den lexer og parser, som genereres, er alle scenarier taget i betragtning. Det vurderes ikke at en selvskrevet lexer og parser vil give mærkbare fordele i forhold til genererede lexer og parser. Det vurderes også at kvaliteten af lexer og parser, som dannes af en kompiler kompiler, vil være svær at matche i en selvskrevet lexer og parser.

ANTLR

ANTLR er en kompiler kompiler, som tager en ANTLR-specifik EBNF-grammatik og kompilerer det til lexer og parser i Java. ANTLR kan som standard give et parsetræ, ud fra en givet EBNF-grammatik og tilhørende program.[28]

Ud fra test af ANTLR konkluderes det, at grammatikken som ANTLR kræver, er let at skrive, da den ligner en standard EBNF-grammatik. Det er også let at danne et AST i ANTLR, da ANTLR selv danner såkaldte *visitor*-klasser, som skal *overrides* med klasser, som definerer hvordan AST'et ser ud.

Fordele:

- Bruger skrive- og læsevenlig EBNF.
- Giver parse træ ud fra standard konfigurationer.
- Let at danne et AST ud fra *visitor*-klasser.
- Koden som danner AST'et er separeret fra selve ANTLR-parseren.

Coco/R

Coco/R er en kompiler kompiler, som tager EBNF-grammatik og danner lexer og parser ud fra dette. Lexer og parser, som dannes, kan være i Java, C#, C++, F#, VB.Net og andre sprog. Den version som er testet, er versionen, som danner lexer og parser i Java.[27]

Ud fra test af Coco/R konkluderes det, at det er let at deklarere tegn og tokens og de grammatiske regler skrives i en form af EBNF, som ofte ses i lærebøger omkring EBNF-notation. EBNF-grammatikken, som Coco/R danner til lexer og parser, opdeles i helt klare sektioner, hvilket gør det nemmere at læse.

Ved brug af Coco/R er det dog sværere at danne et syntakstræ, da man manuelt skal deklarere klasserne, som Coco/R skal bruge til dannelse af træet[34].

Fordele:

- Let deklaring af tegn og tokens.
- Modtager skrive- og læsevenlig EBNF.

- Præcise fejlbeskeder.

Ulemper:

- Besværligt at danne syntakstræ.

JavaCC

Kompiler kompileren JavaCC laver en parser og en lexer, skrevet i Java, ud fra en EBNF. JavaCC bruger en modificeret udgave af EBNF som input. Syntaksen for at skrive token definitioner og andre produktioner er forskellige, hvilket ikke stemmer overens med en almindelig EBNF.

Parseren, som JavaCC genererer, kan ikke automatisk generere et output, f.eks. et AST. For at opnå dette skal der tilføjes annotationer til den givne grammatik. Dette kan gøres i hånden eller der kan gøres brug af enten JJTree eller JBT, som er præprocessorer for JavaCC der tilføjer de annotationer, der skal bruges for at lave et AST. Grammatikken bliver rodet, når de nødvendige annotationer bliver tilføjet, hvilke kan gøre det svært at overskue grammatikken.

JavaCC er fleksibel, da programmøren bestemmer hvad den producerede parser skal give af output. Dette output kan f.eks. være et AST eller et tal, hvis der skrives en lommeregner. [36]

Fordele:

- Inputtet skrives i en modificeret EBNF, som er let at skrive og læse.
- Den er fleksibel, da der kan bestemmes output type for parseren.

Ulemper:

- Tilføjelse af annotationer gør grammatikken rodet.

SableCC

SableCC er en kompiler kompiler, som genererer lexer, parser og token–klasser i et objektorienteret framework, ud fra en given grammatik[4]. Udover lexer, parser og token–klasser genereres der desuden metoder til at traversere træer med.

Fordelen ved SableCC er, at den adskiller det genererede kode fra den del af koden, som programmøren selv skal skrive, hvilket gør koden lettere at overskue.

Ulempen ved SableCC er, at produktioner, med flere muligheder, eller flere af det samme ikke–terminale tegn, kræver annotationer i grammatikken. Dette gør at SableCC kan kende forskel på mulige produktioner og ikke–terminale tegn. Desuden skal der tilføjes ekstra annotationer, hvis der ønskes at generere et abstrakt syntaks træ. Disse annotationer gør grammatikken sværere at læse.

GOLD Parser Builder

GOLD Parser Builder er et værktøj til at generere kompilere med ud fra et sæt grammatiske regler.

En fordel ved GOLD Parser Builder er, at den kommer med et Integreret udviklingsmiljø (*eng.: Integrated Development Environment*) (IDE), komplet med syntaksfarvning af grammatikdefinitionerne, samt test af grammatikken. En anden fordel er, at filen med grammatikken er opdelt i fire tydeligt adskilte dele, således at læsbarheden af denne forbedres. Den typiske opdeling er:

- Metadata og indstillinger.
- Tegnsæt (altså tilladte tegn).
- Terminal–symboler (her *kan* der benyttes regulære udtryk).
- Ikke–terminal–symboler (den grammatiske sammensætning).

Ud fra denne fil opretter programmet tabeller til såvel lexeren som parseren, og disse kan så eksporteres til en fil, som kan benyttes af en såkaldt *engine*. En engine i forbindelse med GOLD Parser Builder–programmet fungerer ved at sammenkoble den eksporterede fil samt kode, som programmøren selv skriver. Dermed fås enten en kompiler eller en interpreter, alt efter, hvordan denne kode skrives.

Ulempen ved GOLD Parser Builder er, at den benytter BNF til at definere kernen i sprogets grammatik. I forhold til EBNF betyder dette en begrænsning i udformningen af grammatikken, der således ikke tillader brugen af eksempelvis regulære udtryk.

Bison

GNU Bison er en kompiler kompiler og en del af GNU projektet. Bison benytter en grammatikfil skrevet i BNF til at generere parseren. Den understøtter næsten alle typer af parsere, men er optimeret til *Read input in one direction reversed rightmost derivation (LR)*(1). Bison kan danne parsere i C, C++ eller Java. Java er på nuværende tidspunkt i et eksperimentalt stadie [11]. Der bliver i dette projekt brugt Java, derfor undersøges Java parseren.

Fordele:

- Den understøtter LR,
  Inadequacy Elimination LR (IELR) og Look-Ahead LR parser (LALR) parsere [11, s. 118].

Ulemper:

- Understøtter ikke EBNF.
- Dårlige fejlbeskeder.
- Java parseren er i en eksperimental udgave, og der er funktioner, som ikke er færdigudviklet endnu.

Valg af kompiler kompiler

Ud fra sammenligningerne af kompiler kompilerne vælges ANTLR. Valget blev taget ud fra, hvor let kompiler kompileren var at bruge. ANTLR vælges, da det er let at skrive EBNF-grammatik til ANTLR, som også er læsevenlig. Det vurderes som værende let at danne AST i ANTLR, hvilket der lægges stor vægt på, da AST'et er vigtig for den senere behandling i kompileren af programmeringssproget til styring af robotter i Minecraft.

Det vurderes ikke, at ANTLR mangler funktioner, som er kritiske for projektet.

5.2 Lexer — Leksikalsk analyse

En leksikalsk analyse er første fase for en compiler/interpreter, og værktøjet kaldes for en *scanner* eller en *lexer*. Lexeren får kildekode som input, som den gennemgår tegn for tegn, og danner leksikale elementer, *tokens*. Disse leksikale elementer kan eksempelvis være konstanter, såsom tal eller tekststrenge, operatorer, reserverede ord (*eng.: keywords*) eller variabelnavne.

Afhængigt af programmeringssprogets kompleksitet kan det være fordelagtigt at benytte et værktøj til at konstruere lexeren. Ved simple sprog kan det være hurtigere at skrive en simpel lexer selv, i forhold til at benytte et værktøj, men det kan øge risikoen for fejl i lexeren. Modsat kræver en lexer–generator en del forarbejde, da de leksikale elementer skal beskrives. Dette sker ofte i et format der minder om EBNF.

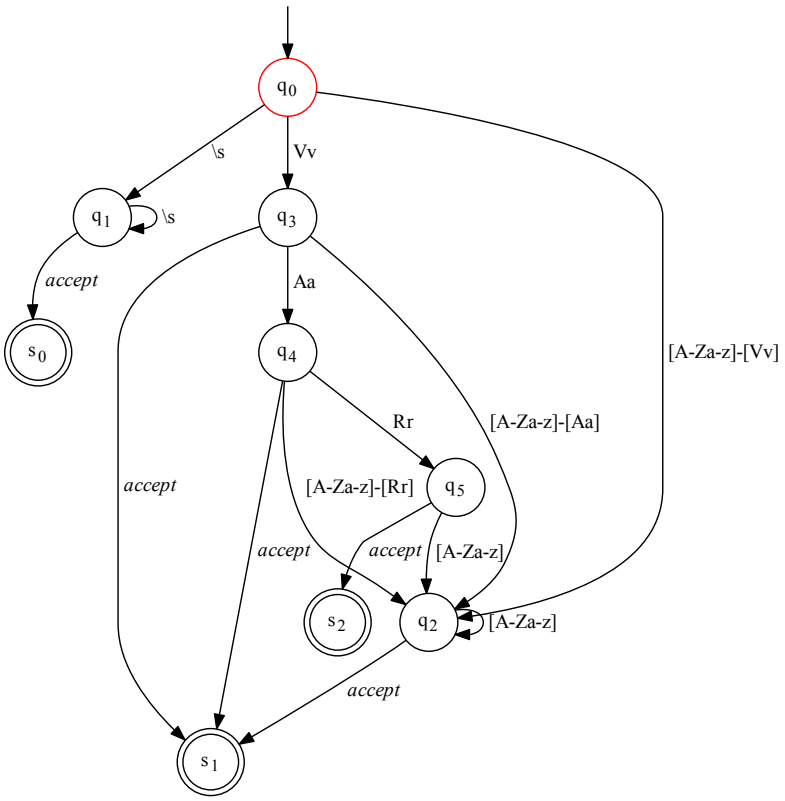
I den genererede kode anvendes typisk regulære udtryk (*eng.: regular expressions*) til at identificere de forskellige, leksikale elementer. Disse regulære udtryk defineres som en Deterministic Finite–state Automaton (DFA). Et simpelt eksempel på en sådan DFA kan ses på figur 5.1 på modstående side.

Eksemplet viser dannelsen af leksikale elementer i et sprog bestående af tre dele:

- Det reserverede ord `var` skrevet med vilkårlige store og små bogstaver.
- Variabelnavne bestående af store og/eller små bogstaver i en vilkårlig længde.
- "Støj" som er tegn lexeren ignorerer og typisk bruger til at adskille leksikale elementer med.

Fremgangsmåden for scanningen fremgår som nævnt af figur 5.1 på næste side, men beskrives også kort herunder.

0. Der startes i tilstanden q_0, hvorfra der er tre stier.
    - Hvis der mødes et støj–tegn, "\s", skiftes der til tilstanden q_1.
    - Mødes et bogstav som *ikke* er "v" eller "V", skiftes der til tilstanden q_2.
    - Mødes bogstavet "v" eller "V" skiftes der til tilstanden q_3.
1. I tilstanden q_1 ses der kontinuerligt på næste tegn indtil et ikke–støj–tegn forefindes.
    - Når sådan et tegn findes, accepterer vi det leksikale element som støj.



Figur 5.1: Simpel Deterministic Finite–state Automaton

2. I tilstanden q_2 læses bogstaver indtil et ikke–bogstav forekommer, hvorefter det godkendes som et variabelnavn.
3. I q_3–tilstanden er der igen tre stier.
   - Hvis næste tegn er enten "a" eller "A", skiftes til tilstanden q_4.
   - Hvis det er et andet bogstav end de to ovenstående, skiftes til tilstanden q_2.
   - Ellers accepteres det leksikale element som et variabelnavn.
4. Tilstanden q_4 minder meget om q_3, med undtagelse af, at der ledes efter "r" eller "R" i stedet for.
   - Bliver et af tegnende fundet på den pågældende plads, skiftes til tilstand q_5.

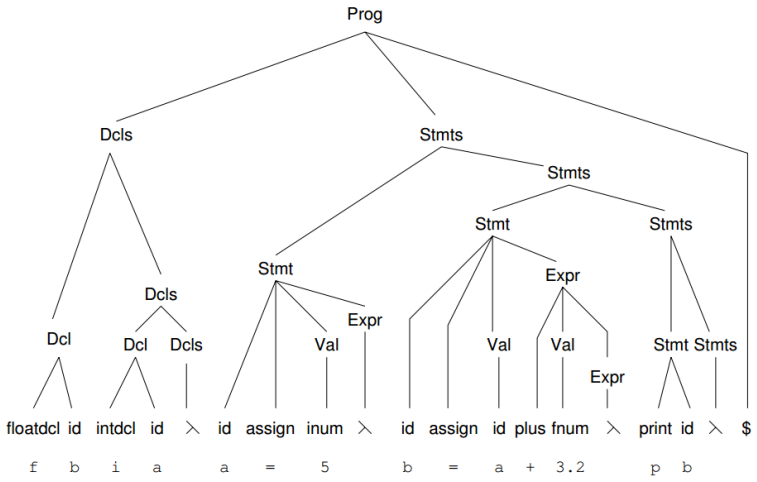
- Er det et andet bogstav skiftes til tilstand q_2.
- Ellers accepteres det leksikale element som et variabelnavn.

5. I tilstand q_5 kan der enten skiftes til tilstand q_2, hvis et bogstav efterfølger "R"–et, og ellers accepteres det reserverede ord `var`.

En lexer ser ikke på sammensætningen af de leksikale elementer, men overlader det til den syntaktiske analysefase.

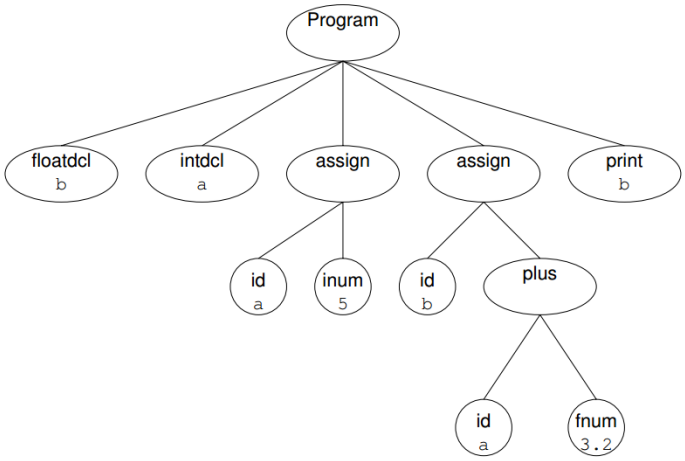
5.3 Parser

I en kompiler er det parserens ansvar at validere, at den sekvens af tokens, som lexeren returnerer til parseren, er korrekt ifølge den grammatik, som specificerer sproget. Ud fra kildekoden, som lexeren får som input, danner parseren en form for datastruktur, ofte enten et parsetræ eller et AST. Et parsetræ er en fuld repræsentation af kildekoden i en træstruktur. Hver node repræsenterer et ikke-terminal symbol fra CFG'en og hvert blad et terminalsymbol. Roden af træet er startproduktionen i CFG'en. Et eksempel på et parsetræ for et program skrevet i sproget AC kan ses på figur 5.2.



Figur 5.2: Parsetræ for et AC-program.[12, s. 37]

Et AST er en simplificeret version af et parsetræ. Disse simplificeringer afhænger af hvad AST'et skal bruges til, men ofte bliver syntaktiske elementer, som f.eks. ";", parenteser og reserverede ord, skåret fra. Et eksempel på et AST for parsetræet på figur 5.2, kan ses på figur 5.3 på modstående side.



Figur 5.3: AST for et AC-program.[12, s. 44]

5.3.1 Tvetydig grammatik

En grammatik er tvetydig, hvis der kan konstrueres to eller flere parsetræer ud fra grammatikken. Et eksempel på sådan en grammatik kan ses på listing 5.1.

```
1 Expr → Expr - Expr
2      |  id
```

Listing 5.1: Eksempel på en tvetydig grammatik.[12]

Med inputstrengen "id - id - id", så vil der kunne laves to forskellige parsetræer, som vises på figur 5.4 på den følgende side.

På figur 5.4(a) vises, at det sidste "id" subtraheres fra de to første.

På figur 5.4(b) vises, at det første "id" subtraheres fra de to sidste.

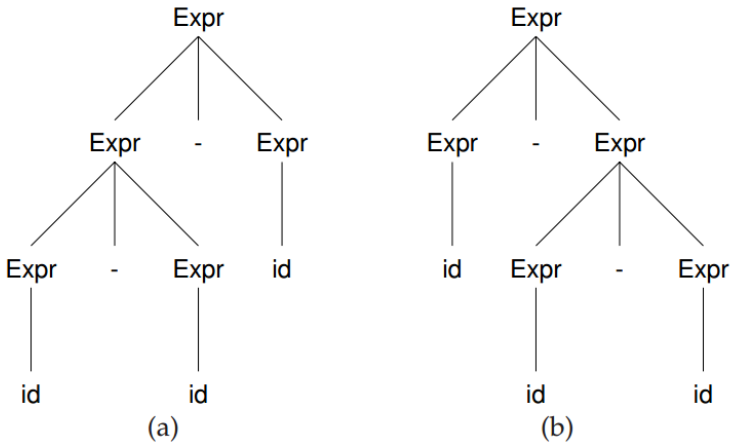
Hvis de tre "id"-symboler har værdierne 3, 2 og 1, vil figur 5.4(a) evalueres til 0, men hvor figur 5.4 vil evalueres til 2.

Tvetydige grammatikker undgås, da der ikke kan garanteres et unikt parsetræ, hvilket kan resultere i en forkert oversættelse af de givne program.

Generelt er der to fremgangsmåder ved parsing: "top-down" og "bottom-up". Disse to metoder vil blive beskrevet herunder.

5.3.2 Top-down parsing

En top-down parser genererer et parsetræ ved at starte i roden af træet og arbejde ud mod træets blade i dybden først (*eng.: depth-first*) traversering. Træet bliver udvidet med produktioner



Figur 5.4: Parsetræ for et AC-program.[12, s. 121]

fra CFG'en, efterhånden som parseren når igennem det givne program. Et parse svarer til at præorder traversere (*eng.: preorder traversal*) træet. Præorder traversere betyder, at der startes i roden (incl. roden) og traverseres i først venstre subtræ fra roden og derefter højre subtræ[14].

Når et input skal tjekkes op mod sprogets grammatik, læser en top-down parser grammatikken oppefra og ned. Den starter i startproduktionen og udfolder hver ikke-terminal, indtil den kun har terminale symboler tilbage. Parseren finder venstresiden i grammatikken og erstatter den med højresiden.

Top-down parsere er forudsigende, da de altid forudsiger produktionen, som skal sammenlignes før selve sammenligningen begynder.

Den mest udbredte top-down parsing strategi er LL(k)-parsing[12, s. 126]. Det første L betyder, at tokens læses fra venstre (*eng.: left*) mod højre. Det andet L betyder, at der finder en venstre afledt parsing sted. *K*'et angiver hvor mange symboler parseren "kigger fremad" (*eng.: lookahead*) for at bestemme hvilken produktion, den skal bruge, deraf er top-down parsing forudsigende. Hvis der ud fra en CFG kan konstrueres en LL(k) parser, er CFG'en en LL(k) grammatik. Der findes sprog, hvor det ikke er muligt at konstruere en LL(k) grammatik.

To metoder til konstruktion af top-down parsere, som vil blive beskrevet, er "recursive-descent parser" og tabeldrevet LL-parser.

Recursive-descent parser

En recursive-descent LL-parser er en LL-parser, som er opbygget af rekursive procedurer, der som regel implementerer en produktion af den givne grammatik. En sådan procedure kan implementeres som en *switch*, hvor der bliver valgt procedure alt efter, hvad inputtet er. Et eksempel for sådan en procedure kan ses på figur 5.5 på næste side, hvor *ts.peek* er token inputtet (*eng.: token stream*). Hvis inputtet består af {a,b,q,c,$}, bliver procedurerne, som svarer

til produktion A og C kaldt og der bliver kaldt `Match` på $ (`Match` tjekker om det rent faktisk er den kaldte token, i dette tilfælde $, som kommer).

Da en recursive-descent parser er afhængig af procedurekald og returværdier, som skaber overhead, kan det gøre parseren ineffektiv.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH($)
end
```

Figur 5.5: Udsnit af et eksempel på en recursive-descent parser.[12, s. 151]

Tabeldrevet LL-parser

En tabeldrevet LL-parser gør brug af en parsetabel. Parseren gør brug af en stakstruktur til at efterligne `Match`-proceduren fra tidligere og kald til ikke-terminale procedurer. Hvis toppen af stakken er en terminal, kaldes `Match` og hvis kaldet er succesfuldt, så *poppes* stakken og der gås videre til næste element. Hvis toppen af stakken er en ikke-terminal, bliver den egnede produktion valgt ud fra opslag i parsetabellen.

Parsetabellen er inddelt i ikke-terminaler som rækkerne og terminaler som kolonnerne. Hver ikke-blank indgang i tabellen er en produktion, som har rækkens ikke-terminal som venstre side i grammatikken. En produktion er typisk repræsenteret af dens regelnummer i grammatikken. Hvis der er en ikke-terminal i toppen af stakken, slås der op i tabellen og ud fra den ikke-terminal, bestemmes hvilken række i tabellen, som skal bruges. Det næste input-token bestemmer hvilken kolonne, som skal bruges. Den resulterende indgang i parsetabellen afgør, hvilken produktion i grammatikken som skal bruges. Et eksempel på en parsetabel kan ses på figur 5.6.

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figur 5.6: Udsnit af et eksempel på en parsetabel.[12, s. 154]

LL-grammatikker har nogle svagheder, bl.a. overfor produktioner som har samme præfiks og overfor venstrerekursive produktioner.

Hvis to ikke-terminale symboler deler samme præfikssymbol i produktionens højre side, ved

parseren ikke hvilken produktion i grammatikken, som skal vælges. Et eksempel på en grammatik, hvor to produktioner har samme præfikssymbol, kan ses på listing 5.2.

```
1 Stmt → if Expr then StmtList endif
2      |  if Expr then StmtList else StmtList endif
```

Listing 5.2: Eksempel på grammatik med samme præfikses.[12]

Parseren kan ikke se forskel på de to *if stmt*, da forskellen på dem er *else*, som kan komme vilkårligt langt bagefter *if*, så det vil ikke hjælpe at sætte parseren til at se flere symboler fremad. Problemet kan løses ved at restrukturere grammatikken, så de to *stmt*'s ikke har samme præfikssymbol.

En produktion er venstrerekursiv, hvis dens symbol på venstre side i grammatikken også er dens første symbol på højre side. Et generelt eksempel på en venstre rekursiv grammatik kan ses på listing 5.3.

```
1 StmtList → StmtList ; Stmt
2          |  Stmt
```

Listing 5.3: Eksempel på grammatik med samme præfikses.[12]

Grammatikker, som er venstrerekursive, kan aldrig blive LL(1). En venstrerekursiv grammatik, som f.eks. A = Aβ, vil i en LL(1)-parser skabe en uendelig løkke, da parseren vil kalde A, uden at avancere inputtet.[12]

5.3.3 Bottom-up parsing

Bottom-up parsere genererer et parsetræ ved at starte i bladene og arbejder sig ind mod træets rod, deraf bottom-up. En node bliver først indsat efter dens børn er blevet indsat. Et bottom-up parse svarer til at postorden traversere (*eng.: postorder traversal*) træet. Postorden traversere betyder, at venstre subtræ traverseres først, derefter højre subtræ, hvorefter roden traverseres[14].

Den mest udbredte bottom-up parsing strategi er LR-parsing[12, s. 126]. Som ved top-down, står det første L for, at tokens bliver læst fra venstre mod højre. R'et angiver, at der finder en højre afledt parsing sted. Hvis der står LR(k), så angiver k'et, hvor mange tegn som læses fremad, som ved en LL(k)-parser.

Når en LR-parser skal vurdere et input, bliver den grammatik, som definerer sproget, læst nedefra og op. Grammatikken bliver læst omvendt i forhold til en top-down parser. Det vil også sige, at en LR-parser starter med den udfoldede grammatik og reducerer hver produktion til en ikke-terminal, indtil parseren til sidst ender med startproduktionen. Det vil sige, at den finder højresiden af en produktion og erstatter det med venstresiden.

Navnet LR er en smule tvetydigt, da det både dækker over den generiske bottom-up parsermotor og en konkret LR parser.

Shift reduce parser

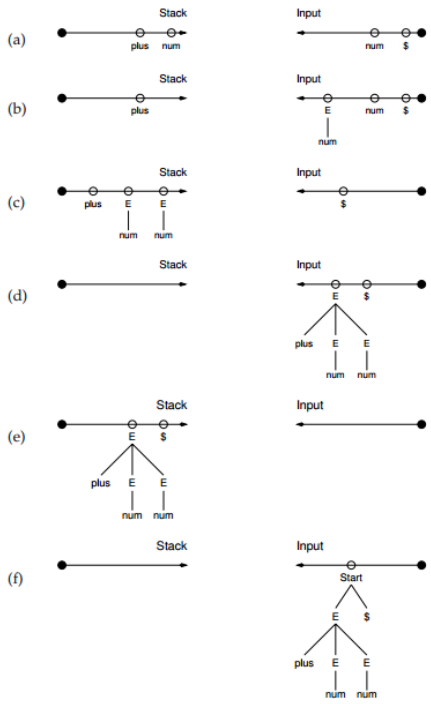
En shift reduce parser *shifter* og *reducer* inputsymbolerne. Et *shift* tager et symbol fra inputtet og flytter det over på en stak.

Reducering er når det "shiftede" input flyttes tilbage fra stakken for at reducere det. Reducering finder sted, når der er en sekvens af symboler på stakken, som kan reduceres til en ikke-terminal. Denne sekvens af symboler, som kan reduceres, kaldes for et *handle*, og erstatter symbolerne som bliver reduceret.

Et eksempel er bedst til at vise hvordan *shift reduce* virker. Ud fra grammatikken i listing 5.4 gives inputtet *plus*, *num*, *num* og *$*.

```
1 Start → E $
2 E     → plus E E
3       |  num
```

Listing 5.4: Grammatik til shift reduce.[12]



Figur 5.7: Et eksempel på shift reduce.[12, s. 183]

På figur 5.7 kan man se stakken og inputtet (venstre nål og højre nål respektiv).

På figur 5.7(a) er *plus* og *num* allerede blevet shiftet på stakken.

På figur 5.7(b) bliver *num* reduceret til *E*.

På figur 5.7(c) bliver *E* og *num* shiftet, hvorefter *num* bliver reduceret til *E* og dernæst bliver shiftet igen.

På figur 5.7(d) bliver begge *E* samt *plus* reduceret til *E*.

På figur 5.7(e) bliver *E* og *$* shiftet, hvor det på figur 5.7(f) bliver reduceret til *Start*.

Hvad der ikke bliver vist på figuren er, at *Start* bliver shiftet på stakken, hvorefter parset er accepteret og slutter.

Alle typer af bottom-up parsere har nogle fælles træk, bl.a. ved at de består af en løkke, som gør følgende:

- Prøver at finde noden længst til venstre i parsetræet, som endnu ikke er blevet konstrueret, men hvor børnene allerede er blevet konstrueret.
  - Disse børn er det såkaldte *handle*.
- Konstruere en ny parsetrænode.

Når der bliver reduceret, dannes der en ny node på parsetræet.

LR-parser

En LR-parser består af en parsermotor, som er drevet af en tabel. Det er tabellen, som afgør om der skal *shiftes*, *reduces*, om parset skal accepteres, eller om der er en syntaksfejl. Tabellen holder styr på det hele ved at hver række er en såkaldt *state* og kolonnerne er de mulige inputsymboler.

Så længe parset ikke er accepteret, slår parsermotoren op i tabellen med parsets *state* og det næste uprocesserede tegn. Hvis opslaget i parsetabellen vælger en blank celle, er der en syntaksfejl. Hvis opslaget vælger en fyldt celle, afgør det felt hvilken handling parsermotoren skal udføre, om det er *shift* eller *reduce*. Cellen angiver også hvilken *state*, der skal skiftes til i tabellen. Hvis opslaget vælger cellen, som indeholder *Accept*, er parset færdigt.

Antallet af rækker i parsetabellen stiger sammen med dens *lookahead*. Generelt har en LR(k)-parser n^k kolonner, hvor n er størrelsen på alfabetet i sproget og k er antallet af tegn som læses fremad. For at holde parsetabellens størrelse nede, er mange parsergeneratorer begrænset til ét tegn *lookahead* (parsergeneratorer bliver beskrevet i afsnit 5.1 på side 34)[12, s. 189]. En LR(0)-parsers parsetabel holder sig lille i forhold til LR(k)-parsere, men til gengæld er LR(0) svagere i forhold til den grammatik den kan bruge.

Der eksisterer flere typer af bottom-up parsere som bygger på LR-parseren. Forskellen mellem de forskellige typer er hvordan de finder et *handle*. En populær parser er LALR, som gør brug af en mindre parsetabel end LR(k)-parsere, dog er den ikke så svag som LR(0). Ofte gør parsergeneratorer brug af LALR netop pga. dens mindre parsetabel.[12]

I næste afsnit vil kompiler kompileren ANTLR blive beskrevet samt dens brug.

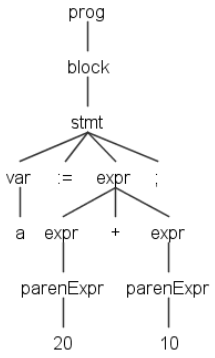
5.4 ANTLR lexer og parser

I afsnit 5.1 på side 34, hvor mulige værktøjer til automatisk at danne dele af kompileren blev undersøgt, faldt valget på kompiler kompileren ANTLR. Det blev desuden valgt at bruge den nyeste version af ANTLR, version 4. Parseren, som genereres af version 4, giver kun et parsetræ som output, hvorimod ANTLR 3 har mulighed for at outputte et AST.

Parseren, der genereres af ANTLR 4, er af typen LL(*), hvilket betyder at parserens lookahead ikke er begrænset til et bestemt K. Parseren kan derimod på parsetidspunktet, baseret på de følgende tokens, forøge lookaheded hvis dette er nødvendigt[3]. Fordelen ved LL(*) parserer er, at de kan håndtere mere indviklede grammatikker end LL(k) parserne[35].

ANTLR 4 kan generere *visitor*– og *listener*–klasser, som kan bruges til at bygge AST'et ud fra parsetræet.

For at kunne gøre sit arbejde, skal ANTLR have en grammatik. Ud fra den givne grammatik genererer ANTLR lexeren og parseren til kompileren. Desuden kan der fra kommandolinjen, efter lexeren og parseren er blevet genereret, bruges "-gui" argumentet sammen med den genererede parser og et stykke kode for at teste parseren. Outputtet af testen bliver en billedfil, som viser parsetræet for koden. På figur 5.8 ses det genererede billede af parsetræet for en simpel variabel definition, *a := 20 + 10*.



Figur 5.8: Parsetræ for simpel variabel definition, a := 20 + 10.

Kompilerens AST

Til genereringen af AST'et i dette projekt er der blevet gjort brug af den genererede *visitor*–klasse, hvilken kan genereres med "–visitor" argumentet fra kommandolinjen. Projektets *ASTGenerator*–klasse arver fra den genererede *BaseVisitor*–klasse, hvori *visitor*–metoderne til de forskellige produktioner i grammatikken findes. I grammatikken specificeres hvilket navn de forskellige produktioners *visitors* skal have. Hvis en ikke–terminal produktion har flere mulige produktioner, skal hver enkelt navngives for at generere en specifik *visitor* til hver mulighed, ellers

genereres der kun en *visitor* for alle mulighederne. Et eksempel på dette ville være, som på listing 5.5, hvor *expr* ikke specificerer hvilke *visitors* der skal oprettes, og derfor genererer ANTLR kun *exprVisitor*.

```
expr : expr + expr
     | expr - expr
     | NLIT
     ;
```

Listing 5.5: Eksempel på grammatik uden *visitor* specificering.

Havde der derimod stået ligesom på listing 5.6, havde ANTLR genereret tre *visitors*, én for hver navngivet produktion.

```
expr : expr + expr     # Addition
     | expr - expr      # Subtraction
     | NLIT             # Number
     ;
```

Listing 5.6: Eksempel på grammatik med *visitor* specificering.

Begge eksempler er skrevet i en syntaks som ANTLR kompiler kompileren forstår.

I grammatikfilen til Komet–sproget, som ANTLR genererer parser og lexer til, er der *visitor* specificering for hver produktion hvor der er flere muligheder. Dette er gjort for at gøre det mere overskueligt at skrive *visitor*–metoder til de forskellige produktionsmuligheder.

Desuden er der blevet tilføjet identifikationer i grammatikken, til produktioner, hvor to af den samme produktion optræder på den højre side. Disse identifikationer kan bruges til at identificere, hvilken del af produktionen der arbejdes med. Et eksempel på dette kan ses på listing 5.7, hvor en *expr* kan være en *expr* lagt sammen med en anden *expr*. Syntaksen for identifikation er *"id ="* efterfulgt af den produktion, id'et gives til. Senere kan denne identifikation bruges i koden i *visitor*–metoderne, som gør det lettere at se hvilken *expr* der arbejdes med. Uden disse identifikationer ville programmøren stå med to *expr*–felter, uden viden om hvilket felt der arbejdes med.

```
expr: left = expr + right = expr
    | NLIT
```

Listing 5.7: Eksempel på grammatik med identifikationer af produktioner

Hele grammatikken til Komet, som er bliver brugt af ANTLR til at generere lexer og parser, findes i listing 13.1 på side 84.

ASTGenerator–klassen redefinerer de generiske *visitor*–metoder fra *BaseVisitor*–klassen, til at oprette de korrekte noder til AST'et. På listing 5.8 ses koden som genererer en *assignNode* til AST'et.

```
    @Override
    public IAbstractNode visitAssign(ULOBParser.AssignContext ctx) {
        VariableNode left = (VariableNode)visit(ctx.var());
        exprDepth++;
        ExprNode right = (ExprNode)visit(ctx.expr());
```

```
6           exprDepth--;
7           AssignNode assign = new AssignNode(left, right);
8           assign.getPositionInformation(ctx);
9           return assign;
10      }
```

Listing 5.8: *Visitor*–metode for *assign statements*.

Noderne til AST'et er ikke genereret af ANTLR, men skal derimod skrives af programmøren. På listing 5.9 ses klassen for nodetypen *assign*, som består af to andre noder, en venstre side og en højre side. De to børn besøges fra *visitAssign*–metoden, og de noder der returneres gemmes i *assign*–noden.

```
1  public class AssignNode extends StmtNode {
2      VariableNode left;
3      ExprNode right;
4
5      public AssignNode(VariableNode leftNode, ExprNode rightNode) {
6          left = leftNode;
7          right = rightNode;
8      }
```

Listing 5.9: *AssignNode*–klassen.

På samme måde som ved *assign*–statementet, genereres de andre noder til AST'et med *visitor*–metoder til de andre noder fra parsetræet.

Fra den genererede lexer og parser er AST'et blevet dannet gennem *visitor*–metoder for produktionerne i grammatikken. Træet kan nu dekoreres under den semantiske analyse af koden, inden kodegenerering. I det næste afsnit beskrives teorien bag den semantiske analyse, og derefter den konkrete brug af det i projektet.

Kapitel 6

Semantisk teori

Semantik dækker over mening og pragmatik af et sprog. I en computervidenskabelig kontekst dækker semantik over et programs interne adfærd, når det kører.

Et computerprograms semantik kan matematisk formelt beskrives ved brug af flere forskellige tilgange. Tre af disse tilgange er denotational semantik, strukturel operationel semantik og aksiomatisk semantik.

I denotational semantik beskrives et programs opførsel ved at definere en funktion, som tildeler mening til hver konstruktion i sproget. En sprogkonstruktions mening er kaldet dets *denotation*. Typisk i et imperativt program vil *denotationen* være en tilstandstransformation, som er en funktion, der beskriver, hvordan de endelige værdier af variabler i et program er fundet fra deres startværdier.

I strukturel operationel semantik beskrives et programs opførsel ved definering af et overgangssystem (*eng.: transition system*), hvor overgangene beskriver evalueringstrinene af et program. Det centrale ved strukturel operationel semantik er, at evalueringstrinene af en syntaktisk entitet kan beskrives på en struktureret måde. En fordel ved strukturel operationel semantik er, at parallelle programmer kan beskrives simpelt.

I aksiomatisk semantik beskrives semantikken ved brug af matematisk logik. Helt konkret bliver der formuleret nogle regler, som skal beskrive udsagnet, som skal holde før og efter sprogkonstruktionen har kørt.

Fremadrettet vil der blive brugt strukturel operationel semantik til beskrivelse af semantikken i Komet.

6.1 Abstrakt syntaks

Til beskrivelse af semantik, ved brug af strukturel operationel semantik, gøres der brug af abstrakt syntaks. Abstrakt syntaks bruges ikke til at beskrive selve syntaksen af et sprog, men hjælper med en notation, som gør det nemmere at beskrive essentielle strukturer ved et program.

$$\frac{a_1 \rightarrow v_1 \;\; a_2 \rightarrow v_2}{a_1 + a_2 \rightarrow v} \text{ hvor } v = v_1 + v_2$$

Figur 6.1: Eksempel på big-step semantik for addition.[21]

Abstrakt syntaks deler et sprog ind i syntaktiske kategorier, som hver har et endeligt sæt af produktioner i BNF-format. Et eksempel på en abstrakt syntaks er Bims fra [21, s. 29] og kan ses på listing 6.1.

```
Syntaktiske kategorier:
n ∈ Num  Talord
x ∈ Var  Variabler
a ∈ Aexp  Aritmetiske udtryk
b ∈ Bexp  Boolean udtryk
S ∈ Stm  Statements
Produktionsregler:
S ::= x:=a | skip | S1; S2 | if b then S1 else S2 | while b do S
b ::= a1 = a2 | a1 < a2 | ¬b1 | b1 ∧ b2 | (b1)
a ::= n | x | a1+a2 | a1*a2 | a1-a2 | (a1)
```

Listing 6.1: Abstrakt syntaks af Bims.[21]

6.2 Strukturel operationel semantik

Som tidligere nævnt består strukturel operationel semantik af et overgangssystem. Et overgangssystem er en bestemt form for orienteret graf. Noderne i grafen repræsenterer konfigurationerne, hvilket er et øjebliksbillede af programmet og dets nuværende stadie. Siderne i grafen repræsenterer overgangene, hvilket er overgangene fra hvert trin i et program. Konfigurationer, som ikke har en overgang ledende væk fra sig, kaldes for slutkonfigurationer. En overgang fra en konfiguration til en anden angives ved $\gamma \rightarrow \gamma'$.

Formelt består et overgangssystem af $(\Gamma, \rightarrow, T)$. Γ er et sæt af systemets konfigurationer, $\rightarrow$ er overgangsrelationen, som er et subsæt af $\Gamma \times \Gamma$ og $T \subseteq \Gamma$ er et sæt af slutkonfigurationer.

Der findes to typer af operationel semantik, som beskriver semantikken på forskellige måder. Begge typer vil blive beskrevet i de følgende afsnit.

6.2.1 Big-step semantik

I big-step semantik beskriver en enkelt overgang, $\gamma \rightarrow \gamma'$, en komplet udregning, som starter i γ og hvor γ' er altid en slutkonfiguration. En overgang for aritmetiske udtryk (Aexp) fra listing 6.1 kan skrives som: $a \rightarrow v$, hvor $a \in \mathbf{Aexp}$ og $v \in \mathbb{Z}$.

Et eksempel på big-step semantik for addition fra Aexp kan ses på figur 6.1.

Eksemplet på big-step semantik for Aexp på figur 6.1 skal læses som: "Hvis $a_1 \rightarrow v_1$ og $a_2 \rightarrow v_2$ (påstandene), så er $a_1 + a_2 \rightarrow v$ (konklusionen), hvor $v = v_1 + v_2$ (sidebetingelsen)."

$$\frac{a_1 \Rightarrow a_1'}{a_1 + a_2 \Rightarrow a_1' + a_2}$$

$$\frac{a_2 \Rightarrow a_2'}{a_1 + a_2 \Rightarrow a_1 + a_2'}$$

$$v_1 + v_2 \Rightarrow v \text{ hvor } v = v_1 + v_2$$

Figur 6.2: Eksempel på small-step semantik for addition.[21]

Big-step semantik kan ikke bruges til at vise parallelle programmer, da en overgang skal gå til et slutstadie, hvor der mellem start og slut kan ske noget, som big-step semantik ikke kan beskrive. Dog kan big-step semantik ofte være mere simpelt at læse end small-step semantik.

6.2.2 Small-step semantik

I small-step semantik beskriver en enkelt overgang, $\gamma \Rightarrow \gamma'$, et trin i en større udregning. γ' behøver ikke være en slutkonfiguration. For Aexp kan en small-step overgang derfor være enten $a \Rightarrow a'$, hvor a' er en mellemliggende konfiguration, eller $a \Rightarrow v$, hvor v er en slutkonfiguration.

En small-step semantisk overgang for addition fra Aexp fra Bims på listing 6.1 på side 51, kan ses på figur 6.2.

På figur 6.2 kan man se den tilsvarende small-step semantik for big-step semantikken på figur 6.1 på side 51. Til forskel fra big-step semantikken, har small-step semantikken tre trin, som viser udregning af hver enkelt del af regnestykket.

Small-step semantik er god til at beskrive parallelle programmer, da der kan være brug for at vise de mellemliggende konfigurationer, før semantikken giver mening i konteksten.

6.2.3 Environment-store model

Environment-store modellen er en model til at vise, at variabler bliver bundet til hukommelsesceller, når de bliver erklæret. Hukommelsescellens indhold er værdien af variablen.

Environment-store modellen består af to dele:

- Variabelmiljøet, env_V, er en partiel funktion, som for hver variabel fortæller, hvilken plads i hukommelsen variablen er bundet til.
- Store, *sto*, er en partiel funktion, som for hver plads i hukommelsen fortæller, hvilken værdi er fundet i den enkelte hukommelsescelle.

I environment-store modellen bliver hukommelsesceller kaldt for lokationer og sættet af lokationer skrives som **Loc**.

Definitionen på sættet af variabelmiljøer er sættet af partielle funktioner fra variabler til lokationer.

$$\mathbf{EnvV} = \mathbf{Var} \cup \{\text{next}\} \rightharpoonup \mathbf{Loc}.$$

Figur 6.3: Definition på variabelmiljøet EnvV.[21]

EnvV er et vilkårligt variabelmiljø, **Var** er variabler og next peger til næste frie lokation og returnerer det.

Definitionen på sættet af *stores* er sættet af partielle funktioner fra lokationer til variabler.

$$\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbb{Z}$$

Figur 6.4: Definition på store.[21]

Den aritmetiske big-step semantik fra figur 6.1 på side 51 kan modificeres således, at en overgang gør brug af environment-store modellen, og derved kan der også indgå variabler i big-step semantikken.

$$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 + a_2 \rightarrow_a v} \text{ hvor } v = v_1 + v_2$$

Figur 6.5: Eksempel på big-step semantik for addition med environment-store modellen.[21]

$env_V, sto \vdash a \rightarrow_a v$ skal læses som: "Givet variablerne kendt af env_V og hukommelsesindholdet af *sto*, evaluerer det aritmetiske udtryk a til værdien v". Resten af udtrykket skal læses på samme måde som figur 6.1 på side 51.

For statements med environment-store modellen er generelle overgange i formen:

$env_V, env_P \vdash \langle S, sto \rangle \rightarrow sto'$

Dette skal læses som: "Givet variabelbindingen env_V og procedurebindingen env_P, vil udførelsen af statement S modificere hukommelsesindholdet *sto* til det nye hukommelsesindhold *sto*'".

6.2.4 Scope regler

I denne sektion vil semantikken omkring scopes i et programmeringssprog blive beskrevet. Der vil blive uddybet på tidligere generel information omkring scopes fra afsnit 4.8 på side 25.

Semantisk set fortæller scopes hvilke bindinger, der er i effekt under eksekveringen af et procedurekald.

For dynamiske scopes er det kun bindingerne, som er kendt på tidspunktet hvor de kaldes, som er vigtige. Så når en procedure er deklareret, skal der kun huskes på kroppen af procedureren. Dette leder ud i den følgende definition for proceduremiljøet **EnvP** med dynamiske scopes:

$$\mathbf{EnvP} = \mathbf{Pnames} \rightharpoonup \mathbf{Stm}$$

Figur 6.6: Definition på EnvP med dynamiske scopes.[21]

Pnames er kategorien af procedurenavne og **Stm** er statements.

For statiske scopes kan der, som tidligere nævnt i afsnit 4.8 på side 25, ved kald af procedure p, gøres brug af variabelbindingen og procedurebindingen, som var kendt da p blev erklæret. Dette leder ud til følgende definition på statiske scopes:

$$\mathbf{EnvP} = \mathbf{Pnames} \rightharpoonup \mathbf{Stm} \times \mathbf{EnvV} \times \mathbf{EnvP}$$

Figur 6.7: Definition på EnvP med statisk scopes.[21]

EnvV er variabelmiljøet og **EnvP** er proceduremiljøet.

6.2.5 Procedurer med parametre

Til definering og kald af procedurer med parametre opdateres definition af EnvP, da den skal holde på den formelle parameter af en procedure. Den opdaterede definitionen på **EnvP** med statisk scope er:

$$\mathbf{EnvP} = \mathbf{Pnames} \rightharpoonup \mathbf{Stm} \times \mathbf{Var} \times \mathbf{EnvV} \times \mathbf{EnvP}$$

Figur 6.8: Definitionen på EnvP med parametre og statisk scope.[21]

Her repræsenterer **Var** den formelle parameter.

Definitionen af **EnvP** er uafhængig af valget af parametermekanisme (Call by name, call by value o.l.) og reflekterer kun valget af scoperegler, og at navnet på den formelle parameter skal huskes. Til gengæld er statements afhængige af parametermekanismen.

6.3 Komets semantik

I dette afsnit vil Komets semantik blive forklaret. Først vil Komets abstrakte syntaks blive defineret, hvorefter selve overgangsreglerne for de forskellige sprogkonfigurationer vil blive defineret og forklaret.

6.3.1 Komets abstrakte syntaks

I listing 6.2 er Komets abstrakte syntaks. Det er denne syntaks, som der tages udgangspunkt i i den semantiske analyse. Som nævnt i afsnit 6.1 på side 50 er det en abstraktion af Komets syntaks fra afsnit 13.2 på side 84.

```
1   Syntaktiske kategorier:
2   n ∈ ℤ - Talord
3   l ∈ {A-Z} ∪ {a-z} - Bogstaver
4   x ∈ l - Variabelnavne
5   z ∈ n ∪ {true,false} ∪ l ∪ a ∪ b ∪ listInit - Variabler
6   e ∈ {k, p} - Listeelementer
7   k ∈ z - Keys i liste
8   p ∈ z - Values i liste
9   b ∈ Bexp - Booleanudtryk
10  a ∈ Aexp - Aritmetiske udtryk
11  S ∈ Stm - Statements
12  c ∈ Csb - Caseblock
13  cC ∈ CsC - Case kontrol
14  p ∈ l - Pname - Procedurenavn
15  D_V ∈ DecV - Deklaration af variabel
16  D_P ∈ DecP - Deklaration af procedure
17
18  Produktionsregler:
19  S ::= a | b | x := z | 'if' b 'do' S_1 'else' S_2 'end' | 'for' x 'in' 'listInit' 'do' a 'end' |
     'forever' 'do' S 'end' | 'break' | 'case' 'do' c 'anywherein' S 'end' | p(z⃗) | D_V | D_P | D_L
20
21  c ::= b S cC
22
23  cC ::= 'break' | 'continue'
24
25  a ::= n | x | a_1+a_2 | a_1-a_2 | a_1*a_2 | a_1/a_2 | a_1 mod a_2 | (a_1)
26
27  b ::= a_1 = a_2 | a_1 > a_2 | a_1 < a_2 | a_1 <= a_2 | a_1 >= a_2 | a_1 != a_2 | b_1 and b_2 | b_1 or b_2 | not b_1
     | (b_1)
28
29  listInit ::= list | rng
30  list ::= e⃗
31  e⃗ ::= k, p
32  rng ::= n_1 .. n_2 | n_1 .. n_2, n_3
33
34  D_V ::= 'define' x := z; D_V | ϵ
35  D_P ::= 'func' p(x⃗) 'do' S 'end' D_P | ϵ
36  D_L ::= x = listInit; D_L | ϵ
```

Listing 6.2: Abstrakt syntaks af Komet.[21]

6.3.2 Overgangsregler

I denne sektion vil der blive vist og forklaret overgangsregler for udvalgte sprogkonstruktioner i Komet. De resterende overgangsregler kan findes i bilag i afsnit 13.3 på side 86. I overgangsreglerne vil der tages udgangspunkt i, at Komet gør brug af fuldt statiske scopes, som tidligere nævnt i afsnit 4.8 på side 25.

Aritmetiske overgange

På figur 6.9 kan man se tre udvalgte aritmetiske overgange i Komet. [ADD] beskriver hvordan to elementer, som er kendt af env_V og sto, adderes. [NUM] beskriver hvordan elementer bliver konverteret fra talord til den matematiske værdi med funktionen $\mathcal{N}$. Dens inverse funktion, $\mathcal{N}^-$, konverterer en værdi tilbage til dens tekstuelle repræsentation. [VAR] beskriver, hvordan en variabels værdi findes ved først at finde dens lokation og derefter dens værdi.

$$\text{[ADD]} \quad \frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 + a_2 \rightarrow_a v} \quad \text{hvor } v = v_1 + v_2$$

$$\text{[NUM]} \quad env_V, sto \vdash n \rightarrow_a v \text{ hvis } \mathcal{N}[n] = v$$

$$\text{[VAR]} \quad env_V, sto \vdash x \rightarrow_a v \text{ hvis } env_V\ x = l \text{ og } sto\ l = v$$

Figur 6.9: Aritmetiske overgange for Komet.

Statements

I denne sektion vil overgangene for Komets statements gennemgås.

På figur 6.10 kan overgangen ses for, hvordan Komet beregner statements i sekvens. Først beregnes S_1 i sto, som resulterer i sto'', hvor beregningen af S_2 starter i sto'' og resulterer i sto', som er slutstadiet.

$$\text{[COMP]} \quad \frac{env_V, env_P \vdash \langle S_1, sto \rangle \rightarrow sto'' \quad env_V, env_P \vdash \langle S_2, sto'' \rangle \rightarrow sto'}{env_V, env_P \vdash \langle S_1; S_2, sto \rangle \rightarrow sto'}$$

Figur 6.10: Big-step semantik for rækkefølgen af beregning af statements.

På figur 6.11 på næste side er der overgange for både deklaration og opdatering af variabler.

[VAR_DECL] angiver deklaration af variabler relativ til env_V og sto. Først findes værdien v af z og derefter bliver variabelbindingen, som er fundet i env_V, opdateret med bindingen x til l, hvor l er den næste tilgængelige lokation og binder next-pointeren til lokationen efter l. Derefter bliver sto opdateret ved at lade lokation l indeholde værdien v.

[VAR_DECL_TOM] viser, at en tom deklaration ikke vil ændre på hverken env_V eller sto.

[VAR_UPDATE] viser, hvordan variabler bliver opdateret. Først bliver $x := z$ eksekveret ved at evaluere z til dens værdi v, og ved opslag i env_V findes lokationen associeret med x. Til sidst bliver indholdet i l opdateret til v.

$$\text{[VAR\_DECL]} \quad \frac{\langle D_V, env''_V, sto[l \mapsto v]\rangle \rightarrow_{DV} (env'_V, sto')}{\langle \text{define } x{:=}z; D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')}$$
$$\text{hvor } env_V, sto \vdash z \rightarrow v$$
$$\text{og } l = env_V \text{ next}$$
$$\text{og } env''_V = env_V[x \mapsto l][\text{next} \mapsto \text{new } l]$$

$$\text{[VAR\_DECL\_TOM]} \quad \langle \epsilon, env_V, sto\rangle \rightarrow_{DV} (env_V, sto)$$

$$\text{[VAR\_UPDATE]} \quad env_V, env_P \vdash \langle x := z, sto\rangle \rightarrow sto[l \mapsto v]$$
$$\text{hvor } env_V, sto \vdash z \rightarrow v \text{ og } env_V\ x = l$$

Figur 6.11: Big-step semantik for deklaration og opdatering af variabel.

På figur 6.12 kan der ses overgangen for deklaration af lister i Komet. I deklarationen bliver der på samme tid tilføjet elementerne e_1 til e_m til selve listen. I selve overgangen ændrer env_V, så x bliver bundet til den første ledige lokation l, og next bliver opdateret til at pege på næste ledige lokation. *sto* ændres ligeledes til at indeholde elementerne e_1 til e_m i lokationen for x.

$$\text{[LIST\_DECL]} \quad \frac{\langle D_L, env''_V, sto[l \mapsto v]\rangle \rightarrow_{DL} (env'_V, sto')}{\langle x{:=}e_1, e_2..e_m; D_L, env_V, sto\rangle \rightarrow_{DL} (env'_V, sto')}$$
$$\text{hvor } env_V, sto \vdash e_i \rightarrow v_i \text{ for } 1 \leq i < m$$
$$\text{og } l = env_V \text{ next}$$
$$\text{og } env'_V = env_V[x \mapsto (l, m)][\text{next} \mapsto (l + v_0)]$$
$$\text{og } sto' = sto[l \mapsto v_0]...[l + (m-1) \mapsto v_m]$$

Figur 6.12: Big-step semantik for for konstruktion med list.

På figur 6.13 kan overgangene for *if-else* konstruktionen i Komet ses. [IF_SAND] viser, at i *sto* skal boolean-udtrykket b beregnes, og hvis resultatet er `true`, ved evalueringen af if b do S_1 else S_2, bliver S_1 eksekveret i *sto*. Ligeledes for [IF_FALSK], hvis b i *sto* beregnes til `false`, så bliver S_2 eksekveret i *sto*.

$$\text{[IF\_SAND]} \quad \frac{env_V, env_P \vdash \langle S_1, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{if } b \text{ do } S_1 \text{ else } S_2, sto\rangle \rightarrow sto'}$$
$$\text{hvis } env_V, sto \vdash b \rightarrow_b \texttt{true}$$

$$\text{[IF\_FALSK]} \quad \frac{env_V, env_P \vdash \langle S_2, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{if } b \text{ do } S_1 \text{ else } S_2, sto\rangle \rightarrow sto'}$$
$$\text{hvis } env_V, sto \vdash b \rightarrow_b \texttt{false}$$

Figur 6.13: Big-step semantik for if-else konstruktion.

I Komet er der tre forskellige typer af for-løkker. De to første [FOR_RNG] og [FOR_RNG_STEP]

er almindelige for-løkke, hvor forskellen er, at [FOR_RNG_STEP] har mulighed for at angive intervallet af løkken.

I [FOR_RNG_1], på figur 6.14, bliver x først sat til v_1, som er værdien af n_1. Hvis $v_1 \leq v_2$, hvor v_2 er værdien af n_2, bliver S eksekveret og værdien af x bliver adderet med én. Dette kører indtil $v_1 > v_2$, hvor direkte efter løkken er stoppet, har x værdien $v_2 + 1$. [FOR_RNG_2] er overgangen, for når løkken stopper.

$$\text{[FOR\_RNG\_1]} \quad \frac{env_V, env_P \vdash \langle S, sto[x \mapsto v_1] \rangle \rightarrow sto'' \quad env_V, env_P \vdash \langle \text{for } x := n_1' \text{ .. } n_2 \text{ do } S, sto'' \rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{for } n_1 \text{ .. } n_2 \text{ do } S, sto \rangle \rightarrow sto'}$$
$$\text{hvis } v_1 \leq v_2 \text{ hvor } v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$$
$$\text{og } n_1' = \mathcal{N}^{-1}(v_1 + 1)$$

$$\text{[FOR\_RNG\_2]} \quad env_V, env_P \vdash \langle \text{for } x := n_1 \text{ .. } n_2 \text{ do } S, sto \rangle \rightarrow sto[x \mapsto v_1]$$
$$\text{hvis } v_1 > v_2 \text{ hvor } v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$$

Figur 6.14: Big-step semantik for for konstruktion med rng.

På figur 6.15 er overgangene for for-løkke i Komet med tæller (step). [FOR_RNG_STEP_1] evalueres stort set som [FOR_RNG_1]. Den største forskel er, at når løkken er kørt igennem, så bliver x adderet med v_2 og v_3, hvor v_3 er værdien af n_3.

$$\text{[FOR\_RNG\_STEP\_1]} \quad \frac{env_V, env_P \vdash \langle S, sto[x \mapsto v_1] \rangle \rightarrow sto'' \quad env_V, env_P \vdash \langle \text{for } x := n_1' \text{ .. } n_2, n_3 \text{ do } S, sto'' \rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{for } n_1 \text{ .. } n_2, n_3 \text{ do } S, sto \rangle \rightarrow sto'}$$
$$\text{hvis } v_1 \leq v_2 \text{ hvor } v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2] \text{ og } v_3 = \mathcal{N}[n_3]$$
$$\text{og } n_1' = \mathcal{N}^{-1}(v_1 + v_3)$$

$$\text{[FOR\_RNG\_STEP\_2]} \quad env_V, env_P \vdash \langle \text{for } x := n_1 \text{ .. } n_2, n_3 \text{ do } S, sto \rangle \rightarrow sto[x \mapsto v_1]$$
$$\text{hvis } v_1 > v_2 \text{ hvor } v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2] \text{ og } v_3 = \mathcal{N}[n_3]$$

Figur 6.15: Big-step semantik for for konstruktion med rng og step.

På figur 6.16 ses overgangen for en for-løkke med en liste. I overgangen er værdien m antallet af elementer i listen, så løkken kører fra $1..m$. x bliver tildelt elementet som løkken er nået til. $list[i]$ er ikke en notation, som findes i Komet, men den skal vise, at element i i $list$ tilgås.

$$\text{[FOR\_LIST\_1]} \quad \frac{env_V, env_P \vdash \langle \text{for } x := 1 \text{ .. } m, sto[l \mapsto v_i] \rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{for } x \text{ in } list \text{ do } S \text{ end}, sto \rangle \rightarrow sto'}$$
$$\text{hvor } env_V\ list = (l, m) \text{ og } env_V x_0 = l$$
$$\text{og } env_V, env_P, sto \vdash list[i] \rightarrow v_i \text{ for } 1 \leq i \leq v$$

Figur 6.16: Big-step semantik for for konstruktion med list.

Forever-løkken, som er beskrevet i afsnit 4.10 på side 28, kører for evigt. Det vises ved brug af small-step semantik, da uendelige løkker ikke giver mulighed for en overgang i big-step semantik[21]. På figur 6.17 på modstående side vises der, at en overgang i small-step semantik giver en ny S og sto, men overgangen vil aldrig ende i slutovergangen sto'. En forever-løkke kan dog kontrolleres med *break*, som vil stoppe løkken.

$$\text{[FOREVER]} \quad env_V, env_P \vdash \langle \text{forever do } S \text{ end}, sto\rangle \Rightarrow^1 env_V, env_P \vdash \langle \text{forever do } S_1 \text{ end}, sto_1\rangle$$

Figur 6.17: Small-step semantik for forever løkke.

På figur 6.18 kan man se semantik for *break*. *Break* tager en *sto* og lader den gå i slutstadiet *sto'*, hvorved eksekveringen afsluttes. På den måde kan forever-løkken kontrolleres med *break*.

$$\text{[BREAK]} \quad env_V, env_P \vdash \langle \text{break }, sto\rangle \rightarrow sto'$$

Figur 6.18: Big-step semantik for break.

På figur 6.19 kan big-step semantikken for *case anywherein* ses. I [CASE_1] bliver der antaget, at b i sto'' bliver beregnet til `true`. Dette betyder, at efter S_2 er blevet beregnet, bliver S_1 derefter beregnet i sto''. I [CASE_2] bliver der antaget, at b i sto' bliver beregnet til `false`, hvorved S_2 ikke bliver beregnet.

$$\text{[CASE\_1]} \quad \frac{env_V,env_P\vdash\langle S_2,sto\rangle\rightarrow sto'' \quad env_V,env_P\vdash\langle S_1,sto''\rangle\rightarrow sto'}{env_V,env_P\vdash\langle \text{case do } b\ S_1\ cC \text{ anywherein } S_2 \text{ end},sto\rangle\rightarrow sto'} \text{ hvis } env_V, sto'' \vdash b \rightarrow_b \texttt{true}$$

$$\text{[CASE\_2]} \quad \frac{env_V,env_P\vdash\langle S_2,sto\rangle\rightarrow sto'}{env_V,env_P\vdash\langle \text{case do } b\ S_1\ cC \text{ anywherein } S_2 \text{ end},sto\rangle\rightarrow sto'} \text{ hvis } env_V, sto' \vdash b \rightarrow_b \texttt{false}$$

Figur 6.19: Big-step semantik for case anywherein.

På figur 6.20 på næste side kan overgangene for proceduredeklaration samt procedurekald ses. [PROC_DECL] er proceduredeklaration i Komet og er afhængig af, hvilke scoperegler som Komet bruger. Da Komet gør brug af statiske scopes, som nævnt i afsnit 4.8 på side 25, skal der gøres brug af proceduremiljøet og variabelmiljøet ved procedurekald. Når procedure p kaldes, bliver S i p eksekveret ved brug af variabelbindingerne og proceduredeklarationerne, som er kendt da p blev deklareret. [PROC_DECL_TOM] viser en tom proceduredeklaration. Notationen $\vec{x}$ viser, at x kan være en række af parametre[26].

Til procedurekald gør Komet brug af *Call by value*, som ses i [PROC_CALL]. Ved opslag i env_P ses der, at p er bundet til $(S, \vec{x}, env_V, env_P)$. Først skal den aktuelle parameter, $\vec{z}$, beregnes til værdien v, hvorefter den næste tilgængelige lokation l findes. l vil derefter blive tildelt den formelle parameter x, hvori værdien v vil blive placeret. Til sidst kan S eksekveres ved brug af bindingen, som var kendt ved deklarationen af p.

[PROC_DECL] $$\frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, \vec{x}, env_V, env_P)] \rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \text{func } p(\vec{x}) \text{ do } S \text{ end } D_P, env_P \rangle \rightarrow_{DP} env'_P}$$

[PROC_DECL_TOM] $env_V \vdash \langle \epsilon, env_P \rangle \rightarrow_{DP} env_P$

[PROC_CALL] $$\frac{env'_V[\vec{z} \mapsto l][\text{next} \mapsto \text{new } l], env'_P \vdash \langle S, sto[l \mapsto v] \rangle \rightarrow sto'}{env_V, env_P \vdash \langle p(\vec{z}), sto \rangle \rightarrow sto'}$$

hvor $env_P(p) = \langle S, \vec{z}, env'_V, env'_P \rangle$
$env_V, sto \vdash z \rightarrow v$
$env_V(\text{next}) = l$

Figur 6.20: Big-step semantik for erklæring og kald af procedure.

Kapitel 7

Semantisk analyse

7.1 Semantisk analyse teori

Den semantiske analyse består i at dekorere AST'et med information. Analysen tjekker den semantiske korrekthed og dekorerer træet med informationer, som senere processer benytter.

For at holde styr på elementerne og deres deklarationer i kildekoden, f.eks. variablerne, kan der konstrueres en eller flere symboltabeller. En symboltabel konstrueres ved at gennemløbe AST'et, hvorved alle elementerne og deres typer bliver fundet, samt andet relevant information, såsom hvilket scope de er i og hvilke adgangsmodifikationer de har.

Under den semantiske analyse tjekkes der, om typerne bruges korrekt, denne aktivitet kaldes typetjekket *eng.: type checking*. Under typetjekket tilføjes typen til den pågældende node, der undersøges i AST'et. Dette kan f.eks. bruges til sikre, at der ikke adderes en streng til en heltalsvariabel. Hvis dette sker, skal kompileren give besked om, at der er en typefejl.

I de fleste programmeringssprog kan der anvendes det samme navn for to forskellige variabler, hvis ikke de er i samme scope. Der sikres, at den rigtige variabel bruges i det rigtige scope under *scope checking*. Desuden sørger scopetjekket for, at der ikke findes to variabler med det samme navn indenfor det samme scope.

Symboltabel

Under den semantiske analyse dannes en symboltabel til at holde styr på de forskellige symboler i sproget. Nogle kompilere begynder med at danne en symboltabel under den leksikalske analyse, og udfylder den under den semantiske analyse, andre venter til den semantiske analyse. Dette skyldes, at der indtil den semantiske analyse ikke har været nok information omkring hvert symbol til at beskrive det. [22]

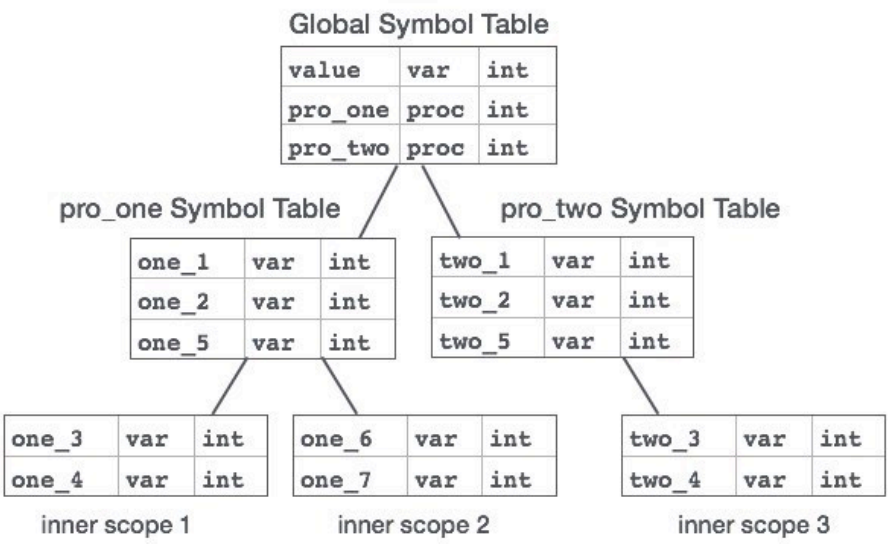
En symboltabel indeholder informationer om en given identifikation, dens tilknyttede symbol,

hvor oplysningerne der er forbundet med symbolet og hvordan information tilgås i hukommelsen.

En symboltabel kan blandt andet bruges til:

- At gemme navnene på alle enheder et sted i en struktureret form [30].
- At bekræfte at variabler er blevet deklareret [30].
- Typekontrol ved at bekræfte at variablers tildeling i kildekoden sker semantisk korrekt [30].
- At bestemme et symbols scope (se figur 7.1).

Scopet kan bl.a. håndteres ved at have en global symboltabel, der kan tilgås af alle. Hvert eneste scope har sin egen lokale symboltabel, et eksempel kan ses på figur 7.1. Figuren viser, hvordan de forskellige symboltabeller kan tilgå hinanden, hver boks kan ses som værende et scopeniveau.



Figur 7.1: Eksempel på symboltabel og dets scope [30]

Visitormønsteret

Når *software* udvikles, kan et *visitor pattern* benyttes til at adskille funktionalitet fra objektstrukturen der arbejdes på. I praksis betyder dette, at man placerer funktioner, der skal udføre ensartede handlinger på en række forskellige objekttyper, i en separat klasse. Dette er i modsætning til at tilføje denne funktionalitet til hver af objektklasserne. En fordel ved denne model er, at et eventuelt behov for at ændre hvilke handlinger der skal udføres på en række objekttyper, kun kræver ændringer i en enkelt fil. [12]

En sådan samling af ensartede funktioner til et bestemt formål kaldes en *visitor*, fordi den "besøger" de forskellige objekttyper. En metode i det besøgte objekt kalder den korrekte metode

i visitoren, hvormed den pågældende kodestump kun skal indsættes én gang, hvis et interface benyttes til at bestemme visitorernes metoder.

I praksis implementeres en visitor, ved at have en visit–metode for hver objekttype den skal kunne håndtere. Normalt gøres dette, ved at tilføje objekttypens navn til ordet "visit", så man eksempelvis får metoden `visitIfNode`. Hvis det valgte sprog, hvori implementationen skrives, understøtter metodeoverloading, kan alle metoderne have samme navn og blot kaldes med forskellige parametertyper. Ydermere er det muligt, hvis implementationssproget understøtter reflektion, at benytte et reflektivt visitormønster.

Reflektionen benyttes til at finde den bedst egnede metode til håndteringen af en bestemt objekttype. Dette gør det muligt at have en standardmetode, der kaldes, hvis der ikke er tilføjet en visit–metode for den pågældende objekttype. Dette kan være praktisk i opbygningen af en kompiler, hvor en visitor ikke nødvendigvis skal lave noget ved alle nodetyper. Det er ligeledes en fordel, at det med reflektion ikke er nødvendigt at skrive kode i hver objekttype, men kan nøjes med få linjer kode i en klasse, objektklasserne nedarver fra.

I Komet vil reflektive visitors blive brugt til at gennemløbe kodens AST i de forskellige faser, eksempelvis i forbindelse med dekorering af træet.

7.2 Implementation af den semantiske analyse

Som nævnt i afsnit 7.1 på side 61 er der primært to ting at analysere i den semantiske analysefase: Typetjek og scopetjek. I dette afsnit beskrives kun scopetjekket.

Til gennemløb af det genererede AST blev det reflektive visitormønster implementeret for at sikre, at ændringer til visitors foretages ét sted, i stedet for i hver node-klasse.

7.2.1 Symboltabel og scopes

Til Komet blev der udviklet en multidimensionel symboltabel, implementeret som en stak af individuelle symboltabeller, hvor oprettelsen af et nyt scope tilsvarer et nyt element på stakken. Søgning efter tilstedeværelsen af et symbol tillader gennemløb ned gennem stakken, således at symboler i scopes der ligger længere ude, kan tilgås. Dette er forsøgt illustreret i figur 7.2 på den følgende side, hvor variablen "currentLocation" ikke kan tilgås udenfor "ContinueMining"–funktionen.

Nye scopes oprettes når en kodeblok starter, hvor programmet som helhed også tolkes som en kodeblok, hvilket udgør det globale scope.

Scopes, med undtagelse af det første, tildeles et forælder–scope. Dermed dannes et træ, hvor hvert element kender til det tidligere element i træet, så det er muligt at gå til tidligere elementer. Dette gør det også muligt for scopes at dele forældre–scopes, så eksempelvist det globale scope vil have en del scopes under sig, uden dog at kende til deres eksistens. Når et scope afsluttes ved slutningen af en kodeblok, bliver dette scope tilknyttet den pågældende kodebloks node i ASTet, hvormed disse informationer kan tilgås af andre elementer, såsom visitors. Eksempelvis vil det dermed være muligt for kodegeneratoren at finde ud af, hvilket symbol der henvises til

Figur 7.2: Symboltabeller i flere niveauer.

fra en bestemt kodeblok, uden at skulle implementere sin egen scopetjekker. Afslutningen af et scope kan ses på figur 7.3, hvor "If–struktur2"–scopet netop er afsluttet, så det ikke længere er på stakken, men det har stadig en pointer til "For–løkke1"-scopet, der nu er det aktive scope.

Figur 7.3: Aktive og afsluttede scopes.

Variabler, funktioner og andre symboler skal være definerede før de kan bruges. Det er derfor ikke muligt at kalde en funktion, der er defineret længere fremme i filen.

7.2.2 Implementationsdetaljer

Variabler fra tidligere scopes kan overskrives i de dybere scopes. Dermed kan en variabel, der er defineret i det globale scope, overskrives, så den kan benyttes som en lokal variabel i en funktion

eller en hvilken som helst anden kodeblok. Dette påvirker blandt andet case-anywherein–konstruktionen, som foretager tjek på bestemte variabler. Hvis disse variabler overskrives inde i anywherein–kodeblokken, eventuelt i en under–kodeblok såsom en kontrolstruktur, så vil case–sætningerne ikke fange ændringer i værdien, fordi der er tale om et nyt symbol.

Det er ikke muligt at have to funktioner med samme navn i den nuværende implementation af Komet, hvilket essentielt forhindrer funktionsoverloading. Dette skyldes, at det er besværligt, og i nogle tilfælde umuligt, at lave funktionsoverloading i et sprog med et dynamisk typesystem.

Kapitel 8

Kodegenerering

8.1 Kodegenererings teori

Den sidste del af kompileringen af et program er kodegenereringen. Kodegenerering er en aktivitet, hvor koden bliver oversat til en anden repræsentation. Et høj-niveau sprog kan oversættes til et lav-niveau sprog, f.eks. fra C til maskinkode.

8.1.1 Intermediate representation

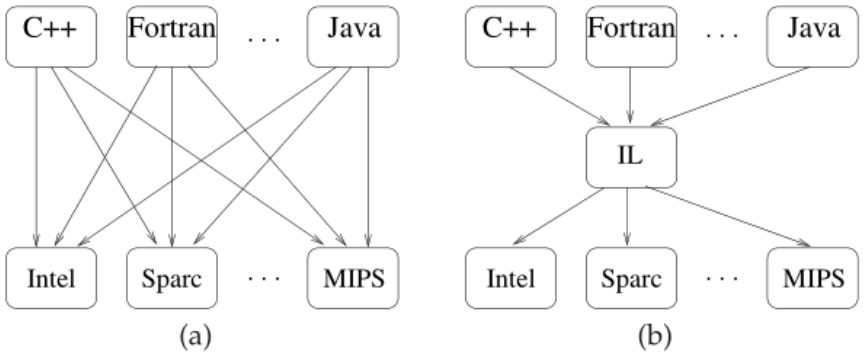
Når et sprog oversættes kan der gøres brug af et mellemled, kaldet *intermediate representation (IR)*. Java Bytecode er et eksempel på en IR. Kildekoden programmet skrives i, oversættes til en IR, hvorefter der oversættes til et *target langugage.* Det er især en fordel at bruge en IR, når der ønskes at kunne oversætte et sprog til flere forskellige *target languages*, som illustreret på figur 8.1 på modstående side. Ved brug af en IR kan allerede udviklede kompilere genbruges, så der kun behøves at udvikle en kompiler, der oversætter sproget til en IR, hvorefter de allerede udviklede kompilere kan oversætte IR'en til de ønskede *target languages*.

8.1.2 Optimering

Under kodegeneringen kan der gøres brug af en optimeringsmetode kaldet *peephole optimization*. Optimering af koden går ud på at lave om på koden, så det bliver så effektivt som muligt, samtidig med at meningen bibeholdes. *Peephole optimization* kan bruges på både en IR og på et *target language*. I det genererede kode ledes der efter mønstre i små kodestykker af gangen, og hvis et mønster genkendes, som noget der kan gøres mere effektivt, udskiftes det.

Når koden optimeres, undersøges følgende ting:

- Overflødige instruktioner



Figur 8.1: Brug af IR. [12, s. 395]

  - F.eks. hvis der lægges to tal sammen i to linjer, hvorefter de lægges sammen i den tredje, kan dette skrives om til en linje.

- Uopnåelig kode
  - F.eks. hvis programmøren er kommet til at skrive instruktioner efter et return-statement i en kodeblok, kan instruktionerne aldrig nåes og derfor kan disse slettes.
- Kontrolflow
  - F.eks. hvis der bruges GOTO til at hoppe til en instruktion, og denne instruktion bruger GOTO til at hoppe til en anden instruktion, laves koden så der startes med at hoppe til den anden instruktion.
- Algebraiske udtryk
  - F.eks. vil udtrykket `a = a + 1`, blive lavet om til inkrementering af `a`.
- Reducering af operatorers styrke
  - Nogle operatorer optager mere tid og plads når de skal implementeres, f.eks. er a^2 mere effektiv end `a * a`.

8.1.3 Generering af kode

Til kodegenerering bruges det dekorerede AST fra den semantiske analysefase som input. Træet traverseres ved hjælp af *visitor*-mønsteret, for at få de oplysninger om AST'ets noder, der skal bruges til genereringen af koden.

Følgende noder er nyttige til at organisere kodegenereringen[12, s. 418]:

- TopVisitor
  - Behandler klasse- og metodedeklarationer og starter behandlingen af metoders indhold.

- MethodBodyVisitor
  - Genererer kode for metoders indhold. Denne *visitor* står for det meste af kodegenereringen.
- LHSVistor
  - Genererer kode de steder, hvor et variabelnavn betyder variablens adresse frem for værdien af variablen. Dette er typisk på venstre side af et assignment-statement.
- Signaturevisitor
  - Laver signaturen for en metode, da *visitoren*, der skal generere koden, ellers ville generere kode, som vil kalde metoden i stedet for kode, der definerer metoden.

Visitorne bruges ved at de få en node som parameter, f.eks. en node for et *forloop*, hvorefter *visitoren* fortælles hvordan et *forloop* er konstrueret i det sprog der oversættes til.

8.2 Implementering af kodegenerering

I dette afsnit vil aktiviteterne, der bliver udført i dette projekts kodegenerering, blive forklaret. Disse aktiviteter er som følger:

- Håndtering af funktionskald som udtryk
- Tilføjelse af compiletime og runtime tjeks.
- Udskrivning af Lua-kode.

8.2.1 Håndtering af funktionskald som udtryk

I Komet kan returværdien fra et funktionskald tildeles til en variabel og et funktionskald kan bruges som parameter osv. Da Komet har dynamiske typer, kan det dog ikke vides, om en funktion overhovedet returnerer noget. Pga. dette skal der tilføjes tjeks for dette i den genererede Lua-kode. Dette gøres ved hjælp af metoden `fCallReplace` vist i listing 8.1 på næste side. Metoden tilføjer en *assign*-node før det statement, hvor funktionskaldet bruges i. I denne node bliver værdien fra funktionskaldet tildelt til en variabel. Der kan herefter tilføjes typetjek og `null` tjek til denne variabel, ligesom der bliver til alle andre variabler.

Listing 8.3 viser hvordan det generede Lua–kode ser ud for Komet–koden vist i listing 8.2, når `fCallReplace` er blevet anvendt.

```
1 var := a + testFunc();
```

Listing 8.2: Funktionskald som parameter i Komet.

```
1 funcVar = testFunc()
2 var = a + funcVar
```

Listing 8.3: Værdien fra funktionskald lægges i en variabel og variablen bruges som parameter i Lua.

```
1     public ExprNode fCallReplace(FCallExprNode fCallNode){
2         VariableNode fCallVar = new VariableNode("_func_" + fCallNameCount);
3 
4         DefineNode fCallVarAssign = new DefineNode()
5                 .setLeft(fCallVar)
6                 .setRight(fCallNode);
7         addBefore(fCallVarAssign);
8 
9         fCallNameCount++;
10 
11        return fCallVar;
12    }
```

Listing 8.1: Udskifting af funktionskald.

8.2.2 Compiletime og Runtime tjeks

Da Komet har dynamiske typer, bør der tilføjes typetjek til det genererede kode. Dette bør gøres, fordi det genererede kode ikke bør kunne give en runtime-fejl i runtime interpreteren, da dette ikke vil give en fejlbesked, som kan bruges til at debugge det oprindelige kode. For at kunne give ordentlige fejlbeskeder bliver der derfor tilføjet `if`-statements, som kalder funktionen `error`, hvis der er fejl i koden. Denne `error` funktion har oplysninger om, hvor fejlen sker i Komet koden og hvilken type fejl det er, hvilket gør det lettere for programmøren at debugge koden.

Listing 8.4 viser metoden `ensureType`, som står for typetjeks for tosidede udtryk, som f.eks en `plus`-node. På linje 10 tjekkes der for, om venstre side af en node er af typen `RUNTIME`. Når en node har typen `RUNTIME` betyder det, at der kun kan tjekkes for, om det er den rigtige type, når programmet kører. Den føromtalte funktion `error` vil her blive brugt, for at give en ordentlig fejlbesked. Noder, der har typen `RUNTIME`, er f.eks. variabler og funktionskald. Hvis en node har typen `RUNTIME`, bliver der tilføjet en "assumption", på linje 11, om at værdien af den ikke er `null`, hvilket er fælles for alle tosidede udtryk. På linje 12 tilføjes typetjek for noden og i det tilfælde, at det er en `plus`-node, tilføjes der et tjek for, om typen af udtrykket noden indeholder, er af typen `NUMBER`.

```
1     private AssignTypes ensureType(ITwoSidedExpr node, AssignTypes[] acceptedTypes, AssignTypes
      resultType) {
2         AssignTypes left = dispatch(node.getLeft());
3         AssignTypes right = dispatch(node.getRight());
4 
5         if(left == AssignTypes.ERROR || right == AssignTypes.ERROR) {
6             node.setType(AssignTypes.ERROR).getType();
7             throw new TypeException("Type error", node.getNode());
8         }
9 
10        if(left == AssignTypes.RUNTIME){
11            assumptionTable.addAssumption(currentStatement, new NotNullAssumption(node.getLeft()));
12            assumptionTable.addAssumption(currentStatement, new TypeAssumption(node.getLeft(),
13                    acceptedTypes));
14        }else{
15            boolean isAccepted = false;
16            for(AssignTypes type : acceptedTypes)
17                if(left == type)
```

```
18                  isAccepted = true;
19              if(!isAccepted){
20                  throw new TypeException("Tried to apply binary " + node.getName() + " to left side
    of type " + node.getLeft().getType(), node.getNode());
21                  /*return node.setType(AssignTypes.ERROR).getType();*/
22              }
23          }
```

Listing 8.4: Udsnit af klassen der håndterer tilføjelse af assumptions til en tabel.

I kompileren bliver der også udført statiske typetjeks, hvor det er muligt. Dette gøres, hvis et tosidet udtryks noder ikke har typen `RUNTIME`, som vist på linje 14-23 i listing 8.4 på side 69. Hvis noden ikke har en type, som accepteres af det tosidede udtryk der tjekkes på, sættes det tosidede udtryks type til `ERROR`, for at kunne fortælle *parent*-noden, at udtrykket ikke vil kunne køres. Dette sker f.eks. ved en `plus`-node, hvis en af siderne er af typen `STRING`, altså en streng tekst, hvilket ikke kan adderes.

8.2.3 Udskrivning af Lua kode

Der er to essentielle klasser for kodegenereringsfasen, hvilket er `CodeGenEmitter` og `CodeGenVisitor`. I `CodeGenVisitor` bruges `visit`-metoden på alle noderne, hvori `CodeGenEmitter` bruges til skrive den tilsvarende Lua-kode til en fil.

I listing 8.5 ses kodegenerering for en `function`-node. De reserverede ord for en funktion, som er nødvendige for at funktionen bliver skrevet ud som en rigtig Lua funktion, bliver håndteret i denne metode. Parametrene bliver desuden udskrevet i denne metode. Dette gøres da en funktions parametre er repræsenteret af strenge i en liste og derfor ikke er noder i sig selv som der skal kaldes `dispatch` fra. De resterende elementer i en `function`-node bliver metoden `dispatch` kaldt på. `Dispatch` sørger for, at `visit`-metoderne, for de respektive noder, bliver kaldt. Disse noder er: Noden der indeholder funktionens navn og en node, som indeholder alle *statements*, der er i funktionen.

```
1   public void visit(FunctionNode node){
2       emitter.write("function ");
3       dispatch(node.getName());
4       emitter.write("(");
5       SymbolTable fTable = sTable.get(node.getBlock());
6       boolean first = true;
7       if(node.getInCase()) {
8           emitter.write("caseTable");
9           first = false;
10      }
11      for (String param : node.getParms()){
12          if(!first)
13              emitter.write(", ");
14          Symbol paramSy = fTable.getSymbol(param);
15          emitter.write(paramSy.getUniqueName());
16          first = false;
17      }
18      emitter.write(")\n");
19      funcDepth++;
20      dispatch(node.getBlock());
```

```
21          funcDepth--;
22          emitter.write("end");
23      }
```

Listing 8.5: Kodegenerering for `FunctionNode`.

8.3 Præprocessor

For at gøre Komet mere effektivt at skrive, er der blevet udviklet en måde, hvorpå det er muligt at skrive ét funktionskald, som kan ekspanderes til flere kald med forskellige parametre, som beskrevet i afsnit 4.7 på side 23. For at kunne implementere funktioner, hvori der kan anvendes sekventielle kald af den samme funktion med forskellige parametre, skal der udvikles en præprocessor. I det følgende afsnit vil præprocessorer blive beskrevet, og derefter implementeringen af præprocessoren til Komet.

8.3.1 Teori

En præprocessor er et stykke software, der arbejder på et input og leverer et output til en anden proces. Typisk ses dette i forbindelse med kompilere, hvor en præprocessor arbejder på kildekoden, og derefter overleverer dens output til kompileren.

Præprocessorer findes i to kategorier: Leksikalske og syntaktiske.[8]

Leksikalsk:

De leksikalske præprocessorer er de mest basale af de to typer af præprocessorer. Deres arbejde består i at udskifte bestemte tokens i kildekoden, med andre prædifinerede tokens.

Et af de mere kendte eksempler på en leksikalsk præprocessor, er præprocessoren fra C. Den søger efter linjer, i kildekoden, der begynder med #–symbolet og finder derefter, ud fra de definerede regler, ud af hvad der skal udskiftes med. Dette kunne eksempelvis være en global definering af en variabel eller inkludering af en headerfil.

Syntaktisk:

De syntaktiske præprocessorer arbejder med syntakstræet i stedet for med kildekoden. Præprocessoren ændrer i syntakstræet ud fra en række brugerbestemte regler.

Syntaktiske præprocessorer bruges ofte til at tilføje nye primitive typer, eller indlejre et domænespecifikt sprog inde i et alment sprog.

Et eksempel på dette er et projekt i et sprog fra Lisp familien. I et stort projekt kunne et modul være skrevet i en dialekt af SQL for databasehåndtering, eller et andet sprog til håndtering af den grafiske brugergrænseflade. Dette ville være indlejring af et domænespecifikt sprog i et mere generelt sprog.

8.3.2 Implementering

Til Komet, i dets nuværende stadie, er det blevet vurderet, at en leksikalsk præprocessor vil være tilstrækkelig, da det eneste præprocessoren skal håndtere er ekspanderingen af funktioner med sekventielt kaldte parametre.

For at kunne identificere hvilke steder præprocessoren skal udskifte grupperingen af sekventielle parametre, med hver parameter i grupperingen, skal der bruges en speciel syntaks. Til dette formål er syntaksen et sæt af #–symboler, hvori mellem parametrene findes, adskilt af kommaer. Syntaksen kan ses på listing 8.6.

```
1  # parameter1, parameter2, parameter3 #
```

Listing 8.6: Syntaksen for sekventielle parametre.

Præprocessorens første opgave, når kildefilen er blevet indlæst, er at finde ud af hvor mange gange kildefilen skal gennemløbes. Dette gøres, da præprocessoren kun udskifter ét sæt sekventielle parametre pr. linje pr. gennemløb. For at kunne sikre at alle parametre bliver udskiftet, finder præprocessoren den linje, i kildekoden, hvori der findes flest sæt af sekventielle parametre.

Herefter gennemløbes kildekoden det antal gange, som linjen med flest sæt af sekventielle parametre havde sæt.

I hvert gennemløb finder præprocessoren linjer med #–sættet og tager parametrene og udvider linjen det antal gange, der er parametre i sættet. De nye linjer bliver indsat lige efter den fundne linje, så funktionskaldene bliver sekventielle. For at finde de rigtige linjer, og ignorere linjer, som er udkommenterede eller hvor #–sættene står inde i en tekstreng, bruges der tre seperate regulære udtryk. De tre udtryk kan ses på listing 8.7.

```
1  Pattern paramPattern = Pattern.compile("\\#.*?\\#");
2  Pattern avoidStringPattern = Pattern.compile(".*\\\".*\\#.*?\\#.*\\\".*");
3  Pattern avoidCommentPattern = Pattern.compile(".*\\/\\/.*\\#.*?\\#.*");
```

Listing 8.7: Regex's fra præprocessoren.

Til sidst skriver præprocessoren den nye kildekode ud til en fil, hvorfra kompileren kan overtage. Præprocessoren kan dog i stedet for at udskrive sit output til en fil, overføre det direkte til kompileren.

Kapitel 9

Test

For at teste Komet–kompileren er det blevet besluttet at lave modultest på den producerede kode, i stedet for kompilerens individuelle dele. I dette afsnit forklares kort, hvorfor dette er valgt.

De enkelte dele af kompileren er afhængige af alle tidligere dele. Dette gør det mere kompliceret at opsætte test for de senere dele, hvorfor det er vurderet, at brugbarheden af de skrevne modultest ikke opvejer kompleksiteten i opsætning af disse. Tilstrækkelig isolering af senere dele, til brug i test, kræver definition og konfigurering af tidligere deles output, såsom et fuldt dekoreret AST.

Dette ligger til grund for valget om modultest til den genererede kode, der dermed fungerer som integrationstest for kompileren. Til disse test benyttes systemet "Busted", skrevet i Lua, til test af Lua–kode. Med "Busted" kan test indlejres i andre test, hvormed en logisk struktur opnås. Der er stor frihed, så strukturen kan tilpasses et vilkårligt projekt.

Der er skrevet test til sprogkonstruktionerne *multidimensiel for–løkke*, *case* og *funktioner*. De skrevne test dækker gyldigt såvel som ugyldigt kode, ligesom det sikres, at den genererede Lua–kode eksekveres korrekt. Dette kan eksempelvis involvere at teste, at en funktion kaldes et bestemt antal gange fra koden i listing 9.1.

```
1 ff ["outlib"] out(out);
2
3 for i in (x, y, z)[1..10, 1..10, 1..10] do
4   out(i);
5 end
```

Listing 9.1: Komet–program til test.

I listing 9.2 på næste side ses et udsnit af den Busted–test, som tester den multidimensionelle `for`–løkke. I eksemplet ses også de indlejrede test, som Busted understøtter. Først testes der om koden kan kompileres, hvorefter det sikres, at den resulterende Lua–kode er gyldig. Dette gøres ved at kalde `load`–metoden i Lua, med stien til den genererede kode som parameter. Endeligt testes

det i linjerne 15–24, at funktionen `out` kaldes 1000 gange. Denne funktion er implementeret via Komets FFI, beskrevet i afsnit 4.12 på side 32, kombineret med kodeinjicering. Kodeinjiceringen er skrevet til Komets testsystem og gør det muligt at simulere et vilkårligt antal metoder. Dette system initialiseres, når `load`–funktionen kaldes og nulstilles i kaldet til `prog:run()`.

```
1  [...]
2    describe("multi-dimensional forloop #multidimlist", function()
3      describe("straight forward", function()
4        local prog
5
6        it("should compile", function()
7          assert.True(compile("multidimlist/iterations"))
8        end)
9
10
11       it("should load", function()
12         prog = load("multidimlist/iterations")
13       end)
14
15       it("should call print 1000 times", function()
16         local outlib = {}
17         stub(outlib, "out")
18
19         prog:inject(outlib, "outlib")
20
21         prog:run()
22
23         assert.stub(outlib.out).called(1000)
24       end)
25 [...]
```

Listing 9.2: Udsnit af testkoden.

Kapitel 10

Diskussion

For at kunne vurdere hvor godt det udviklede sprog er, sammenlignes Komet med de underpunkter, der blev stillet op sammen med problemformuleringen i kapitel 2 på side 6.

Punkterne der blev opstillet er:

- Sproget skal kunne interagere med OpenComputers i Minecraft.
- Sproget skal have specifikke sprogkonstruktioner til robotter.
- Der skal være en fordel ved brug af det udviklede sprog til robotter i Minecraft end allerede eksisterende sprog.
- Erfarende brugere skal kunne starte på det udviklede sprog uden at have problemer med syntaksen.

Komet kan interagere med OpenComputers moddet ved brug af FFI'et i Komet. I bilaget `robot`, listing 13.3 på side 90, findes udvalgte funktioner, som kan kaldes via FFI'et, hvilket fungerer inde i Minecraft. Komet opfylder derfor det første underpunkt til problemformuleringen, dog er antallet af funktioner i biblioteket begrænset til de mest essentielle. Det anses dog som værende trivielt at udvide bilioteket med flere funktioner til kontrolleringen af robotter. Det samme gælder `stdlib`, hvilket indeholder standard funktioner, som kunne forventes at findes i et standard bibliotek.

I Komet er der blevet udviklet en speciel feature, *Case-Anywherein*, som skal gøre det lettere for programmøren at udføre mange tjeks igennem en kodeblok. Featuren anses som værende god til at kode robotter, da en robot skal kunne håndtere forskellige situationer, hvor særlige situationer kræver særlige håndteringer. Ved brug af *Case-Anywherein* mindskes det antal tjeks, programmøren skal skrive i forhold til, hvis hvert tjek skulle skrives manuelt. *Case-Anywherein* konstruktionen er kun en fordel for større kodeblokke, hvor der skal udføres flere tjeks.

Case-Anywherein har størst fordel ved brug sammen med en større kodeblok, hvor der skal udføres flere tjeks, kan dette anses som værende en fordel for programmøren, som kun behøver skrive hvert tjek én gang, og derefter tager kompileren sig af, at tjekket bliver indsat de korrekte steder. Derved skal programmøren ikke gentage de samme linjer kode flere gange, hvilket

desuden vil gøre koden lettere at læse, da der ikke fyldes tjeks ind i kodeblokken. Ved korte blokke er det ikke en fordel at anvende *Case-anywherein*.

Syntaksen i Komet minder på mange punkter om syntaksen fra Lua og C. Dette skyldes, at Komets syntaks skulle være let at gå til for erfarende programmører. Ved at designe Komets syntaks til at efterligne to kendte programmeringssprog, skulle programmører, med erfaringer indenfor mindst ét ad disse sprog, ikke have problemer med at sætte sig ind syntaksen for Komet.

Det er desuden værd at bemærke, at Komet er et Turing–komplet sprog, da det både er muligt at lave løkker og betingede konstruktioner deri [6].

Som beskrevet tidligere, i kapitel 9 på side 73, er der blevet skrevet tests til Komet. Disse tests er ikke dækkende for sproget, og skal disse kunne bruges til vurdering, om hvorvidt sproget opfører sig korrekt, skal der skrives test til flere dele af sproget. På nuværende tidspunkt er der kun løst testet *Case-anywherein*, funktioner, og for-løkker med multidimensionelle lister. Disse tests kan dog ikke bruges til at konkludere, om de testede funktionaliteter opfører sig korrekt under forskellige forudsætninger.

Kapitel 11

Konklusion

Det vurderes, at Komet kan bruges til at kode robotter i Minecraft, for erfarende programmører. Komet gør det lettere for programmøren at kode robotter i Minecraft, ved brug af *Case-anywherein* konstruktionen.

Bibliotekerne, *Robot* og *Stdlib*, kan udvides til at indeholde flere funktioner, hvilket ville gøre Komet brugbart i flere programmeringssituationer.

Implementationen af FFI'et gør desuden Komet kompatibel med andre robotmods til Minecraft, som benytter Lua–kode, f.eks. ComputerCraft, da der kun skal tilføjes et bibliotek, som forbinder funktionskaldene i Komet til de funktionskald moddet bruger.

Komet er i sit nuværende stadie ikke gennemtestet, og der kan derfor opstå fejl under kompilering og kørsel af programmer.

Kapitel 12

Perspektivering

Det var tiltænkt at Komet, skulle have specielle funktionaliteter til håndtering af vektorer. Dette blev ikke implementeret, men visionen var at kunne have en smart måde at håndtere vektorerne, evt. i en datatype, som skulle kunne holde styr på x,y,z koordinaterne i spillet. Specielle beregninger kunne så foretages på koordinatsættene. Specielt swizzling[5] kunne være brugbart i forhold til koordinater, og specielt Minecrafts koodinater, hvor y og z aksen er byttet rundt.

Komets kobling til Minecraft blev ikke så stærk, som forventet. Dette skyldes tildels at alle Minecraft specifikke konstruktioner ikke blev implementeret, specielt *prop*, *is* og håndteringen af vektorer.

Sproget blev ikke tilstrækkeligt testet, og dette kunne forbedres hvis der skulle arbejdes videre med Komet, da der kan gemme sig fejl i kompileren.

Til mere avancerede robotter ville det brugbart at kunne dele sit kode op i moduler, og at kunne kalde eksterne funktioner fra andre filer skrevet Komet.

Case-anywherein–konstruktionen kunne udbygges til at kunne undersøge returværdier fra funktionskald.

Anonyme funktioner og first class funktioner kunne tilføjes til Komet. I forbindelse med anonyme funktioner kunne closures ligeledes tilføjes til sproget.

Litteratur

[1] Foreign function interface. URL: en.wikipedia.org/wiki/Foreign_function_interface [cited 2015-05-09].

[2] Gnu compiler collection. URL: https://gcc.gnu.org/ [cited 2015-02-27].

[3] Ll parsers. URL: http://en.wikipedia.org/wiki/LL_parser [cited 2015-03-31].

[4] Sablecc. URL: http://sablecc.org/ [cited 2015-03-15].

[5] Swizzling. URL: http://en.wikipedia.org/wiki/Swizzling_%28computer_graphics%29 [cited 2015-05-26].

[6] Turing completeness. URL: http://en.wikipedia.org/wiki/Turing_completeness [cited 2015-05-26].

[7] Interpreter, Januar 1995. URL: http://foldoc.org/interpreted [cited 2015-03-03].

[8] Preprocessor, Okt 2014. URL: http://en.wikipedia.org/wiki/Preprocessor [cited 2015-04-22].

[9] Opencomputers, Februar 2015. URL: http://ocdoc.cil.li/ [cited 20-2 2015].

[10] Mike Banahan. 8.4. const and volatile, 1993. URL: http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html [cited 2015-03-04].

[11] Charles Donnelly and Richard Stallman. Bison, Jan 2015. URL: http://www.gnu.org/software/bison/manual/bison.pdf [cited 2015-03-16].

[12] Charles N. Fisher, Ron K. Cytron, and Richard J. LeBlanc, Jr. Crafting a compiler, 2009.

[13] Christopher Zorn, Emiko Charbonneau, Chadwick Wingrave, and Joseph J. LaViola Jr. Exploring Minecraft as a Conduit for Increasing Interest in Programming. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.381.4045&rep=rep1&type=pdf.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[15] Inc. Cunningham & Cunningham. Dynamic scoping, Jun 2013. URL: http://c2.com/cgi/wiki?DynamicScoping [cited 2015-04-14].

[16] Stack exchange users. What are the pros and cons of incorporating lua into a c++ game?, oct 2011. URL: http://gamedev.stackexchange.com/questions/18285/what-are-the-pros-and-cons-of-incorporating-lua-into-a-c-game [cited 2015-05-03].

[17] Will Freeman. Revealed: The countries that love minecraft most, August 2011. URL: http://www.develop-online.net/news/revealed-the-countries-that-love-minecraft-most/0109967 [cited 9-2 2015].

[18] Nick Gamberini. Spockbot. URL: https://github.com/SpockBotMC/SpockBot [cited 20-02-2015].

[19] Hanspeter Mössenböck and Markus Löberbauer. The compiler generator coco/r, Dec 2014. URL: http://www.ssw.uni-linz.ac.at/Coco/ [cited 2015-03-17].

[20] http://www.minecraftforum.net/. Survival vs creative, Aug 2012. URL: http://www.minecraftforum.net/forums/minecraft-discussion/survival-mode/287912-survival-vs-creative [cited 2015-04-13].

[21] Hans Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[22] Worcester Polytechnic Institute. Compiler design - symbol table, 2003. URL: http://web.cs.wpi.edu/~kal/courses/cs4533/module5/myst.html [cited 2015-04-05].

[23] Andrew Kelley. Mineflayer, November 2014. URL: https://github.com/andrewrk/mineflayer [cited 9-2 2015].

[24] Mike G mikkey. 6. conditionals, Jul 2000. URL: http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-6.html [cited 2015-03-01].

[25] Mojang. Mincraft. URL: https://minecraft.net/ [cited 20-12-2015].

[26] Peter Muller. Semantics of programming languages, Apr 2004. URL: https://www1.ethz.ch/pminf/education/courses/semantik/archive/2004/lectures/sps2.1-bigstep_semantics.pdf [cited 2015-05-20].

[27] Hanspeter Mössenböck. The compiler generator coco/r. URL: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Doc/UserManual.pdf [cited 2015-03-09].

[28] Terence Parr. *The Definitive ANTLR 4 Reference*, volume 2013. The Pragmatic Bookshelf, 1 edition, 2013.

[29] Markus Persson, Februar 2014. URL: https://twitter.com/notch/statuses/438444097141882880 [cited 9-2 2014].

[30] Tutorials Point. Compiler design - symbol table, 2015. URL: http://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm [cited 2015-04-05].

[31] Mark Prigg. American hackers 'bomb' minecraft version of denmark and raise stars and stripes in cyber-attack, Maj 2014. URL: http://www.dailymail.co.uk/sciencetech/article-2623697/American-hackers-bomb-Minecraft-version-Denmark-raise-stars-stripes-cyber-attack-educatior
html [cited 2015-02-23].

[32] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10 edition.

[33] Tali Garsiel and Paul Irish. How browsers work: Behind the scenes of modern web browsers, Aug 2011. URL: http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Parsing_general [cited 2015-03-31].

[34] Henrik Teinelund. Coco/r parser creating syntax tree - part 3. URL: http://structured-parsing.wikidot.com/coco-r-parser-creating-syntax-tree-part-3 [cited 2015-03-09].

[35] Terence Parrand Kathleen S. Fisher. Ll(*). URL: http://www.antlr.org/papers/LL-star-PLDI11.pdf [cited 2015-03-31].

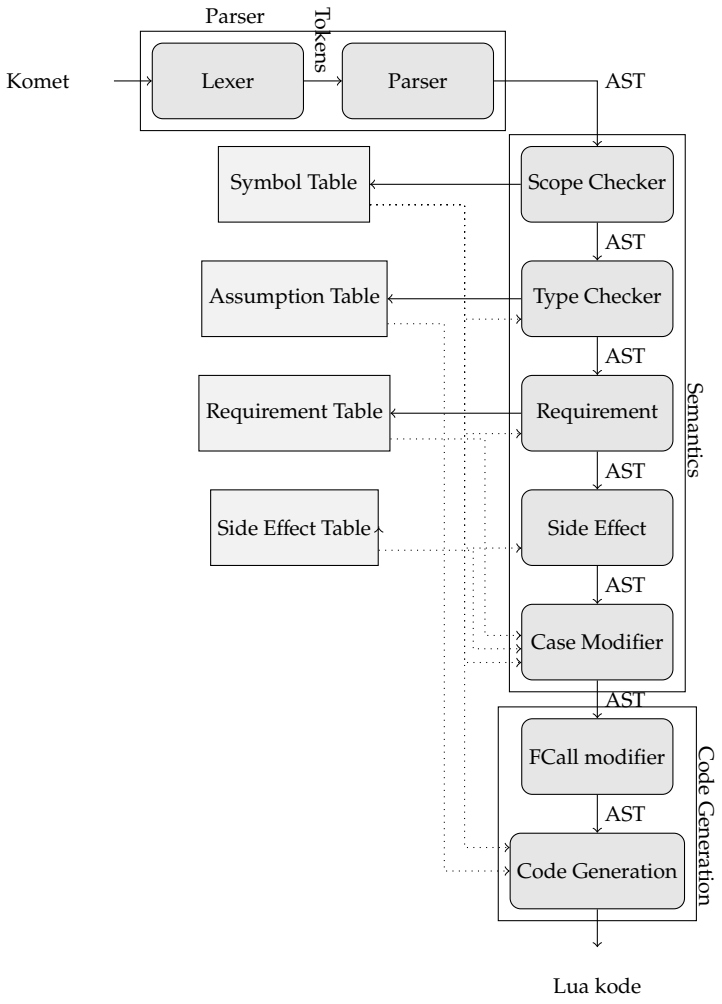
[36] Theodore S. Norvell. The JavaCC FAQ. URL: http://www.engr.mun.ca/~theo/JavaCC-FAQ.

[37] tutorialspoint. Python basic syntax. URL: http://www.tutorialspoint.com/python/python_basic_syntax.htm [cited 2015-03-01].

Kapitel 13

Bilag

13.1 Kompilerflowdiagram



13.2 Komets grammatik

```
grammar ULOB;
WS          : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
COMMENT     : '//' ~('\r' | '\n')* -> skip;
NLIT        : ('0'..'9')|('1' .. '9')('0' .. '9')+;
SLIT        : '"' (.)*? '"';
ADDSUBOP    : '+' | '-';
MULTDIVOP   : '*' | '/';
MODOP       : 'mod';
COMPOP      : '=' | '<'|'>'| '<='|'>='|'!=';
OROP        : 'or';
ANDOP       : 'and';
NAME        : (('A' .. 'Z')|('a' .. 'z'))+;

var         : NAME                                                  #varName
            | listLook                                              #varlistLook
            ;
listLook    : NAME ('[' expr ']')+;

prog        : proplst? block;

proplst     : prop+;
prop        : 'prop' id=NAME ':=' '{' (NAME (',' NAME)*)? '}' ('or' NAME)* ';';

block       : stmt* ;
stmt        : var ':=' expr ';'                                             #Assign
            | 'define' var ':=' expr ';'                                    #Define
            | fcall ';'                                                     #FCallStmt
            | 'if' expr 'do' ifb=block ('else' elseb=block)? 'end'          #If
            | 'for' var 'in' listVar 'do' block 'end'                       #Forloop
            | 'forever' 'do' block 'end'                                    #Forever
            | 'func' NAME '(' namelst? ')' 'do' block 'end'                 #Function
            | 'return' expr ';'                                             #Return
            | 'break' ';'                                                   #Break
            | cse                                                           #Case
            | 'ff' '['SLIT']' NAME'(' params=namelst? ')' ('changes' se=namelst )?';'
    #FFI
            ;

fcall       : NAME paramlst;
paramlst    : '(' exprlst? ')';

expr        : ADDSUBOP parenExpr
    #AddSubParenthesesExpr
            | parenExpr                                                     #ParentExpr
            | left=expr MULTDIVOP right=expr                                #MulDivExpr
            | left=expr MODOP right=expr                                    #ModOpExpr
            | left=expr ADDSUBOP right=expr                                 #AddSubExpr
            | left=expr COMPOP right=expr                                   #ComOPExpr
```

```
47              | left=expr ANDOP right=expr                                  #AndExpr
48              | left=expr OROP right=expr                                  #OrExpr
49              | listInit                                                   #ListInitExpr
50              | SLIT                                                       #StringLiteralExpr
51              | variable=var 'is' property=NAME                            #IsExpr
52              | 'not'  expr                                                #NotExpr
53              | 'true'                                                     #trueExpr
54              | 'false'                                                    #falseExpr
55              ;
56  parenExpr   : fcall                                                      #FCallParenExpr
57              | '(' expr ')'                                               #ParenthesesExpr
58              | var                                                        #VarParenthesesExpr
59              | NLIT
        #NumericalLiteralExpr
60              ;
61  exprlst     : expr (',' expr)*;
62  namelst     : NAME (',' NAME)*;
63
64  cse         : 'case' 'do' caseblk+ 'anywherein' block 'end';
65  caseblk     : ('(' expr ')' block casecontrol);
66  casecontrol : ('break' | 'continue') ';';
67
68  rng         : NLIT '..' NLIT (',' NLIT)?;
69  list        : ('(' namelst? ')')? '[' listpart (',' listpart)* ']';
70
71  listpart    : expr
72              | rng;
73
74  listInit    : rng                                                        #RngListInit
75              | list                                                       #ListListInit
76              ;
77
78  listVar     : listInit                                                   #ListInitListVar
79              | var                                                        #VarListVar
80              ;
```

Listing 13.1: Grammatikken for Komet.

13.3 Komets overgangsregler

13.3.1 Aritmetiske udtryk

[AEXP_ADD] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 + a_2 \rightarrow_a v}$ hvor $v = v_1 + v_2$

[AEXP_SUB] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 - a_2 \rightarrow_a v}$ hvor $v = v_1 - v_2$

[AEXP_MULT] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 * a_2 \rightarrow_a v}$ hvor $v = v_1 * v_2$

[AEXP_DIV] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 / a_2 \rightarrow_a v}$ hvor $v = v_1 / v_2$

[AEXP_MOD] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 \bmod a_2 \rightarrow_a v}$ hvor $v = v_1 \bmod v_2$

[AEXP_PARENTES] $\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1}{env_V, sto \vdash (a_1) \rightarrow_a v_1}$

[AEXP_NUM] $env_V, sto \vdash n \rightarrow_a v$ hvis $\mathcal{N}[n] = v$

[AEXP_VAR] $env_V, sto \vdash x \rightarrow_a v$ hvis $env_V\ x = l$ og $sto\ l = v$

Figur 13.1: Big-step semantik for aritmetiske udtryk.

13.3.2 Boolean udtryk

[BEXP_LIG_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 = v_2$$

[BEXP_LIG_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 \neq v_2$$

[BEXP_IKKE_LIG_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 != a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 \neq v_2$$

[BEXP_IKKE_LIG_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 != a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 = v_2$$

[BEXP_STØRRE_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 > v_2$$

[BEXP_STØRRE_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 \not> v_2$$

[BEXP_MINDRE_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 < v_2$$

[BEXP_MINDRE_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 \not< v_2$$

[BEXP_STØRRE_LIG_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 >= a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 \geq v_2$$

[BEXP_STØRRE_LIG_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 >= a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 \not\geq v_2$$

[BEXP_MINDRE_LIG_1] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 <= a_2 \rightarrow_b \texttt{true}} \text{ hvis } v_1 \leq v_2$$

[BEXP_MINDRE_LIG_2] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 <= a_2 \rightarrow_b \texttt{false}} \text{ hvis } v_1 \not\leq v_2$$

[BEXP_OG_1] $$\frac{env_V, sto \vdash b_1 \rightarrow_b \texttt{true} \quad env_V, sto \vdash b_2 \rightarrow_b \texttt{true}}{env_V, sto \vdash b_1 \text{ and } b_2 \rightarrow_b \texttt{true}}$$

[BEXP_OG_2] $$\frac{env_V, sto \vdash b_i \rightarrow_b \texttt{false}}{env_V, sto \vdash b_1 \text{ and } b_2 \rightarrow_b \texttt{false}} \quad i \in \{1, 2\}$$

[BEXP_ELLER_1] $$\frac{env_V, sto \vdash b_i \rightarrow_b \texttt{true}}{env_V, sto \vdash b_1 \text{ or } b_2 \rightarrow_b \texttt{true}} \quad i \in \{1, 2\}$$

[BEXP_ELLER_2] $$\frac{env_V, sto \vdash b_1 \rightarrow_b \texttt{false} \quad env_V, sto \vdash b_2 \rightarrow_b \texttt{false}}{env_V, sto \vdash b_1 \text{ or } b_2 \rightarrow_b \texttt{false}}$$

[BEXP_IKKE_1] $$\frac{env_V, sto \vdash b \rightarrow_b \texttt{true}}{env_V, sto \vdash \text{ not } b \rightarrow_b \texttt{false}}$$

[BEXP_IKKE_2] $$\frac{env_V, sto \vdash b \rightarrow_b \texttt{false}}{env_V, sto \vdash \text{ not } b \rightarrow_b \texttt{true}}$$

[BEXP_PARANTES] $$\frac{env_V, sto \vdash b_1 \rightarrow_b v}{env_V, sto \vdash (b_1) \rightarrow_b v}$$

Figur 13.2: Big-step semantik for boolean udtryk.

13.3.3 Statements

[COMP]
$$\frac{env_V,env_P \vdash \langle S_1,sto\rangle \rightarrow sto'' \quad env_V,env_P \vdash \langle S_2,sto''\rangle \rightarrow sto'}{env_V,env_P \vdash \langle S_1;S_2,sto\rangle \rightarrow sto'}$$

[VAR_DECL]
$$\frac{\langle D_V,env_V'',sto[l \mapsto v]\rangle \rightarrow_{DV} (env_V',sto')}{\langle \text{define } x:=z;D_V,env_V,sto\rangle \rightarrow_{DV} (env_V',sto')}$$
hvor $env_V, sto \vdash z \rightarrow v$
og $l = env_V$ next
og $env_V'' = env_V[x \mapsto l][\text{next} \mapsto \text{new } l]$

[VAR_DECL_TOM] $\langle \epsilon, env_V, sto\rangle \rightarrow_{DV} (env_V, sto)$

[VAR_UPDATE] $env_V, env_P \vdash \langle x := z, sto\rangle \rightarrow sto[l \mapsto v]$
hvor $env_V, sto \vdash z \rightarrow v$ og $env_V\ x = l$

[LIST_DECL]
$$\frac{\langle D_L,env_V'',sto[l \mapsto v]\rangle \rightarrow_{DL} (env_V',sto')}{\langle x:=e_1,e_2..e_m;D_L,env_V,sto\rangle \rightarrow_{DL} (env_V',sto')}$$
hvor $env_V, sto \vdash e_i \rightarrow v_i$ for $1 \leq i < m$
og $l = env_V$ next
og $env_V' = env_V[x \mapsto (l, m)][\text{next} \mapsto (l + v_0)]$
og $sto' = sto[l \mapsto v_0]...[l + (m - 1) \mapsto v_m]$

[IF_SAND]
$$\frac{env_V,env_P \vdash \langle S_1,sto\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{if } b \text{ do } S_1 \text{ else } S_2,sto\rangle \rightarrow sto'}$$
hvis $env_V, sto \vdash b \rightarrow_b \texttt{true}$

[IF_FALSK]
$$\frac{env_V,env_P \vdash \langle S_2,sto\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{if } b \text{ do } S_1 \text{ else } S_2,sto\rangle \rightarrow sto'}$$
hvis $env_V, sto \vdash b \rightarrow_b \texttt{false}$

[FOR_RNG_1]
$$\frac{env_V,env_P \vdash \langle S,sto[x \mapsto v_1]\rangle \rightarrow sto'' \quad env_V,env_P \vdash \langle \text{for } x:=n_1' .. n_2 \text{ do } S,sto''\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{for } x:=n_1 .. n_2 \text{ do } S,sto\rangle \rightarrow sto'}$$
hvis $v_1 \leq v_2$ hvor $v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$
og $n_1' = \mathcal{N}^{-1}(v_1 + 1)$

[FOR_RNG_2] $env_V, env_P \vdash \langle \text{for } := n_1 .. n_2 \text{ do } S, sto\rangle \rightarrow sto[x \mapsto v_1]$
hvis $v_1 > v_2$ hvor $v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$

[FOR_RNG_STEP_1]
$$\frac{env_V,env_P \vdash \langle S,sto[x \mapsto v_1]\rangle \rightarrow sto'' \quad env_V,env_P \vdash \langle \text{for } x:=n_1' .. n_2,n_3 \text{ do } S,sto''\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{for } x:=n_1 .. n_2,n_3 \text{ do } S,sto\rangle \rightarrow sto'}$$
hvis $v_1 \leq v_2$ hvor $v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$ og $v_3 = \mathcal{N}[n_3]$
og $n_1' = \mathcal{N}^{-1}(v_1 + v_3)$

[FOR_RNG_STEP_2] $env_V, env_P \vdash \langle \text{for } := n_1 .. n_2, n_3 \text{ do } S, sto\rangle \rightarrow sto[x \mapsto v_1]$
hvis $v_1 > v_2$ hvor $v_1 = \mathcal{N}[n_1], v_2 = \mathcal{N}[n_2]$ og $v_3 = \mathcal{N}[n_3]$

[FOR_LIST_1]
$$\frac{env_V,env_P \vdash \langle \text{for } x:=1 .. m,sto[l \mapsto v_i]\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{for } x \text{ in } list \text{ do } S \text{ end},sto\rangle \rightarrow sto'}$$
hvor $env_V\ list = (l, m)$ og $env_V x_0 = l$
og $env_V, env_P, sto \vdash list[i] \rightarrow v_i$ for $1 \leq i \leq v$

[FOREVER] $env_V, env_P \vdash \langle \text{forever do } S \text{ end}, sto\rangle \Rightarrow^1 env_V, env_P \vdash \langle \text{forever do } S_1 \text{ end}, sto_1\rangle$

[BREAK] $env_V, env_P \vdash \langle \text{break }, sto\rangle \rightarrow sto'$

Figur 13.3: Semantik for statements i Komet 1.

[CASE_1]
$$\frac{env_V,env_P \vdash \langle S_2,sto\rangle \rightarrow sto'' \quad env_V,env_P \vdash \langle S_1,sto''\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{case do } b\ S_1\ cC \text{ anywherein } S_2 \text{ end},sto\rangle \rightarrow sto'}$$
hvis $env_V, sto'' \vdash b \rightarrow_b \texttt{true}$

[CASE_2]
$$\frac{env_V,env_P \vdash \langle S_2,sto\rangle \rightarrow sto'}{env_V,env_P \vdash \langle \text{case do } b\ S_1\ cC \text{ anywherein } S_2 \text{ end},sto\rangle \rightarrow sto'}$$
hvis $env_V, sto' \vdash b \rightarrow_b \texttt{false}$

[PROC_DECL]
$$\frac{env_V \vdash \langle D_P,env_P[p \mapsto (S,\vec{x},env_V,env_P)]\rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \text{func } p(\vec{x}) \text{ do } S \text{ end } D_P,env_P\rangle \rightarrow_{DP} env'_P}$$

[PROC_DECL_TOM] $env_V \vdash \langle \epsilon, env_P\rangle \rightarrow_{DP} env_P$

[PROC_CALL]
$$\frac{env'_V[\vec{z} \mapsto l][\text{next} \mapsto \text{new } l],env'_P \vdash \langle S,sto[l \mapsto v]\rangle \rightarrow sto'}{env_V,env_P \vdash \langle p(\vec{z}),sto\rangle \rightarrow sto'}$$
hvor $env_P(p) = \langle S, \vec{z}, env'_V, env'_P\rangle$
$env_V, sto \vdash z \rightarrow v$
$env_V(\text{next}) = l$

Figur 13.4: Semantik for statements i Komet 2.

13.4 Biblioteker

13.4.1 Standard funktionsbibliotek

```
local oldG = _G
function print(...)
  local parameters = { ... }
  for _,v in pairs(parameters) do
    if type(v) == "table" then
      printTable(v)
    else
      oldG.print(v)
    end
  end
end

function printTable(table, depth)
  depth = depth or 0
  for k,v in pairs(table) do
    local prefix = ""
    for i=0,depth-1 do
      prefix = prefix .. "\t"
    end
    if type(v) == "table" then
      oldG.print(string.format("%s[%s]={ ", prefix, k))
      printTable(v, depth+1)
      oldG.print(prefix .. "}")
    else
      oldG.print(string.format("%s[%s]= %s", prefix, k, v))
    end
  end
end
```

Listing 13.2: Standard funktionsbibliotek til Komet

13.4.2 Robotfunktionsbibliotek

```
local robot = require("robot")

function forward(length)
  for i=0, length do
    robot.forward()
  end
end

function move( direction, length, keepPrevDirection )
  if direction == nil then direction = "forward" end
  if keepPrevDirection == nil then
    keepPrevDirection = false
  end
  print("Something works")
  if direction == "forward" then
    forward(length)
```

```
  elseif direction == "back" then
    turn(direction)
    forward(length)
    if keepPrevDirection then
      turn(around)
    end
  elseif direction == "left" then
    turn(direction)
    forward(length)
    if keepPrevDirection then
      turn("right")
    end
  elseif direction == "right" then
    turn(direction)
    forward(length)
    if keepPrevDirection then
      turn(left)
    end
  elseif direction == "up" then
    for i=0, lenght do
      robot.up()
    end
  elseif direction == "down" then
    for i=0, length do
      robot.down()
    end
  end
end

function turn(direction)
  if direction == "left" then robot.turnLeft()
  elseif direction == "right" then robot.turnRight()
  elseif direction == "around" then robot.turnAround()
  end
end

function swing(direction)
  if direction == nil then direction = "forward" end

  if direction == "forward" then
    robot.swing()
  elseif direction == "left" then
    turn(direction)
    robot.swing()
    turn("right")
  elseif direction == "right" then
    turn(direction)
    robot.swing()
    turn("left")
  elseif direction == "up" then
    robot.swingUp()
  elseif direction == "down" then
    robot.swingDown()
  end
end
```

Listing 13.3: Robotfunktionsbibliotek til Komet.