P4 - PROJEKT

Ruben Mensink
Palle Thillemann
Søren Flensborg
Frederik Østerby Hansen

Et domæne-specifikt programmeringsprog til bioteknologistuderende

Supervisor:
Anders Mariegaard

Department of Computer Science
Aalborg University
Denmark

29/5-2017

AALBORG UNIVERSITET
STUDENTERRAPPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://www.cs.aau.dk

Title:

Et domæne-specifikt programmerings-prog til bioteknologistuderende

Theme:

Design, Definition and

Implementation of Programming

Languages

Project Period:

6/2-2017 - 29/5-2017

Project Group:

D404F17

Authors:

Frederik Østerby Hansen
Søren A. Flensborg
Palle Thillemann
Ruben Mensink

Supervisor:

Anders Mariegaard

Total Pages: 82 + frontpage
Appendix: 4
Completion Date: 29/5-2017

Abstract:

The first part of the report contains an analysis of a problem domain, which lead to a list of use cases and a list of criteria, which the implemented programming language should support/fulfill. The use cases and criteria served as a basis for the design of the programming language. The scope of this project has been relatively limited, and therefore the amount of features the implemented language supports is also relatively limited. The overall purpose was therefore to design a simple programming language, for some specific domain.

Forord

Som led i udarbejdelsen af dette projekt, har vi benyttet os af personer, som vi ønsker at takke. Vi har adspurgt flere bio-relaterede personer på AAU, omkring problematikkerne i deres arbejde, med henhold til beregningsmæssigt arbejde på sekvenser. Her ønsker vi at takke Kamilla T. Larsen fra Institut for Kemi og Biovidenskab på AAU, for god respons omkring de eksisterende værktøjer, samt programmeringsmæssig erfaring.

Endvidere vil vi benytte lejligheden til at takke vores vejleder, Anders Mariegaard, der gennem projektet har bidraget med god og målrettet vejledning, samt megen konstruktiv kritik, som vi i projektgruppen har haft stor gavn af.

Indhold

Kapitel 1

Introduktion

Dette kapitel vil omhandle projektets problemstilling, hvilket ender ud i en problemformulering der har banet vejen for projektets videre arbejde.
I det følgende stykke tekst vil være en introduktion til dette projekt. Efterfølende bliver vores initierende problem i afsnit 1.3 præsenteret. Herefter har vi foretaget vores problemanalyse i afsnit 1.4, som havde til formål at bane vejen for projektafgræsningen i afsnit 1.6. Ud fra afgrænsningen af projektets omfang har vi i afsnit 1.7 specificeret en problemformulering og ud fra denne lavet nogle løsningskriterier i afsnit 1.8.

Formålet med dette projekt er, at designe og implementere et programmeringssprog, samt at konstruere en oversætter, for dette programmeringssprog. Dertil skal dette nye sprog gerne løse et problem. At finde sådan en type af problem, der kan løses vha. et programmeringssprog, er nærmest et problem i sig selv, da der er mange flere muligheder, som skal overvejes. Til gengæld, går der mange af de samme ting igen; at man skal have et udgangspunkt.

Udgangspunktet for projektet, er det aktuelle problem: at børn skal lære at programmere i folkeskolen, da der i fremtiden er et behov for mennesker med disse kompetencer [1].

Et tilsvarende initierende problem kunne være: *Hvordan kan man få børn til at lære mere omkring programmering?*

Til gengæld så er der i forvejen mange bud på, hvordan dette problem kan løses, hvor det ifølge en artikel om et visuelt algoritme læringsværktøj RAPTOR, er bedst at designe et visuelt programmeringssprog, da de fleste lærer bedst ved hjælp af noget visuelt[2]. Et visuelt sprog, er typisk implementeret ved at man kan "drag and drop"forskellige konstruktioner, og derved opnå funktionalitet, som det er i eks. Scratch [3]. Det vil kræve en editor, for at programmere i sådanne sprog, og dermed vil det være udenfor dette projekts omfang. Til gengæld så er problemstillingen, der omhandler at der er brug for flere programmører, ikke begrænset til en aldersgruppe. Det kunne eksempelvis være studerende, som læser på videnskabelige studier. Denne gruppe har oftest brug for, at kunne foretage beregninger, udtrykke algoritmer, behandle og præsentere data.

1.1 Biologi, bioteknologi og bioinformatik

For at begrænse projektet yderligere, vælger vi at tage udgangspunkt i biologien. Der findes en bred vifte af beregningsmæssige elementer indenfor biologien. Men dette felt er enormt bredt, og vi vil derfor definere hovedgrupperne herindenfor, og kort beskrive deres arbejdsfelter. I dette afsnit beskriver vi hvad vi overordnet set mener med biologi, bioteknologi og bioinformatik. Alle er indenfor faget, biologi. Men de har hver deres interesseområde, og de benytter vidt forskellige værktøjer i deres arbejde.

Biologi Biologien interesserer sig for det grundlæggende og det generelle. Dette er med til at gøre biologi til det bredeste fag, og grundstenene for bioteknologi og bioinformatik. Eksempler på hvilke emner biologien beskæftiger sig med er populationer, bakterier, DNA, osv. Biologer kan arbejde med et hav af forskellige ting. Bioteknologi og bioinformatik er tværfaglige udspringere af dette fag[4].

Bioteknologi Bioteknologien anvender levende organismer indenfor et teknologisk område, for at producere produkter. Her arbejdes der med at lære fra biologiens verden, og udvikle nye, eller forbedre eksisterende teknologier. Bioteknologisk arbejde bliver blandt andet brugt i landbrugs-, fødevare- og medicinalindustrien[5].

Bioinformatik Bioinformatikken arbejder med at kunne bearbejde data fra biologiens verden. Dette sker oftest enten i en form af programmering- eller beregningssoftware. Bioinformatik arbejder bl.a. indenfor forskning af DNA, proteiner, samt vedligeholdelse af biologiske databaser. Bioinformatik har til formål at analysere, automatisere, vedligeholde og præsentere data[6].

1.2 Specificering af målgruppe

Vi specificerer vores målgruppe til at være bioteknologi studerende. Dette betyder at vores målgruppe har minimal eller begrænset programmeringsmæssig viden og erfaring. Det betyder endvidere at de er under uddannelse i faget, og derfor endnu ikke nødvendigvis har en dybdegående forståelse for al fagstof.

1.3 Initierende problem

For at komme i gang med projektet har vi i dette afsnit opstillet det initierende problem for vores projekt, som har til formål at styre projektets retning fremadrettet. Det initierende problem er det følgende:

Hvordan kan man designe og implementere et programmeringssprog til brug inden for et område i bioteknologi, der vil være intuitivt og anvendeligt?

Det initierende problem giver anledning til nogle underspørgsmål. Disse underspørgsmål vil vi i en analyse undersøge og begrænse for at arbejde frem mod den endelige problemformulering. underspørgsmålene, som vi vil undersøge, er følgende:

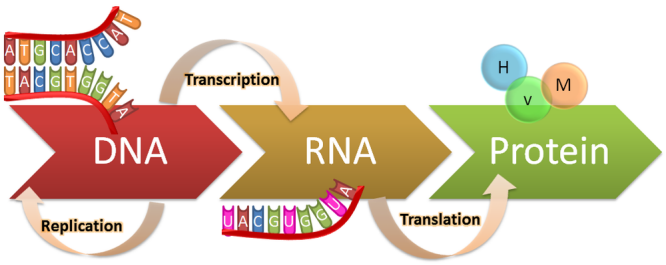
- Hvilke felter indenfor bioteknologi kunne have fordel af et dedikeret sprog? Er der specifikke termer, regler eller fremgangsmåder, som der skal tages højde for?
- Hvilke værktøjer/sprog, bruger man indenfor bioteknologi? Hvorfor?

1.4 Problemanalyse

Dette afsnit vil bestå af vores problemanalyse for dette projekt. I problemanalysen vil vi analysere det initierende problem med henblik på at finde frem til en konkret og specifik problemformulering, der senere vil bane vej for design og implementation af et programmeringssprog.

1.4.1 Det centrale dogme

Vi undersøger første underproblem fra afsnit 1.3, hvor vi har fundet frem til et fagligt område, som er relevant for bioteknologer, dvs. noget fagligt som de kender til, og hvis ikke så er det stadig noget centralt indenfor bioteknologien. Dette faglige område, kaldes for det centrale dogme[7]. Figur 1.1 viser hvad det centrale dogme går ud på.



Figur 1.1: Det centrale dogme beskriver, at DNA kan blive replikeret, at det kan *transskriberes* til RNA, hvor RNA kan *translateres* til aminosyrer. En sekvens af aminosyrer udgør tilsammen et protein.[8]

1.4.2 Terminologi

For at få et overblik over dette fagområde, præsenterer vi en kort beskrivelse af de forskellige nødvendige termer.

Vi opfatter *DNA* (deoxyribonucleic acid), som en sekvens af fire forskellige baser A, T, G, C. Disse symboler repræsenterer *baserne*: *adenin*, *thymin*, *guanin* og *cytosin*.

Basen A kan bindes til basen T og ligeledes for G og C. Ved disse regler kan den *komplementære* sekvens, til enhver DNA sekvens findes. Dette illustreres på figur 1.2, som viser at et DNA *molekyle* er dobbeltstrenget. Dvs. at et DNA molekyle består af en sekvens og en komplementær sekvens.

Komplementær sekvens

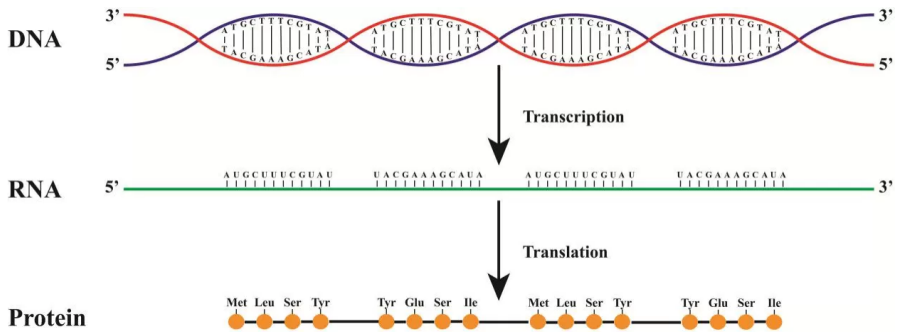
Den komplementære sekvens, kan beregnes ved at erstatte en base, med den base, som den kan bindes til. Hvis en DNA sekvens består af følgende baser $ATTCCG$ så vil den komplementære sekvens være $TAAGGC$, da hver forekomst af A erstattes med T, og omvendt, og det samme gøres for G og C.

Reglerne kan opsummeres således

- $A \rightarrow T$
- $T \rightarrow A$
- $G \rightarrow C$
- $C \rightarrow G$

Transskription

DNA kan *transskriberes* til *mRNA* (messenger ribonucleic acid). mRNA er en sekvens af fire forskellige baser A, U, G, C. Symbolet U repræsenterer basen *uracil*. Uracil er en erstatning for thymin i mRNA.



Figur 1.2: Mere detaljeret illustration af det centrale dogme, hvor DNA transskriberes til RNA, og efterfølgende translateres til en sekvens af aminosyrer (protein).[9]

Før vi kan transskribere DNA til mRNA, skal vi vide hvilken *retning* sekvensen vender. Dette har ikke noget at gøre med læseretningen af sekvensen, som desuden altid er fra venstre mod højre. Det har at gøre med om sekvensen går fra $5'$ mod $3'$ enden eller $'3$ mod $5'$ enden. Dette kan ses på figur 1.2. Hvis en DNA sekvens er $5'$ mod $3'$, så kaldes

den for den kodende sekvens. Dens komplementære sekvens vil så være $3'$ mod $5'$ og kaldes for skabelon sekvensen. Sekvenserne beskrives, som at være *antiparallelle* [10].

Vi har givet en DNA sekvens bestående af følgende baser: $CGT\ CCA\ TTC$ [1]. Hvis vi siger at denne sekvens er den kodende sekvens, så er det eneste man skal foretage, er at erstatte basen T med basen U. mRNA sekvensen bliver dermed følgende: $CGU\ CCA\ UUC$. Hvis man derimod har skabelon sekvensen, så kan man gøre det, at man først finder den komplementære sekvens (den kodende sekvens), og så transskriberer.

Translation

mRNA kan *translateres* til en sekvens af *aminosyrer*, som tilsammen udgør et protein, som vist på figur 1.2. Tre mRNA baser kaldes for et *codon*. Et codon oversættes til én bestemt aminosyre. Der er 20 mulige aminosyrer, som kan ses på figur 1.4. Der findes én bestemt 3-tupel af mRNA baser (codon), som kaldes for et *startcodon*. Det er tuplen A, U, G. Et startcodon er hvorfra translationen begynder. En translation af en mRNA sekvens slutter, når et *stopcodon* ses i mRNA sekvensen. Der findes tre stopcodon, som kan ses på figur 1.3.

First Base	Second Base				Third Base
	U	C	A	G	
U	UUU Phenylalanine (Phe)	UCU Serine (Ser)	UAU Tyrosine (Tyr)	UGU Cysteine (Cys)	U
	UUC	UCC	UAC	UGC	C
	UUA Leucine (Leu)	UCA	UAA Stop	UGA – Stop	A
	UUG	UCG	UAG	UGG – Tryptophan (Trp)	G
C	CUU Leucine (Leu)	CCU Proline (Pro)	CAU Histidine (His)	CGU Arginine (Arg)	U
	CUC	CCC	CAC	CGC	C
	CUA	CCA	CAA Glutamine (Glu)	CGA	A
	CUG	CCG	CAG	CGG	G
A	AUU Isoleucine (Ile)	ACU Threonine (Thr)	AAU Asparagine (Asn)	AGU Serine (Ser)	U
	AUC	ACC	AAC	AGC	C
	AUA	ACA	AAA Lysine (Lys)	AGA Arginine (Arg)	A
	AUG – Start Methionine (Met)	ACG	AAG	AGG	G
G	GUU Valine (Val)	GCU Alanine (Ala)	GAU Aspartic Acid (Asp)	GGU Glycine (Gly)	U
	GUC	GCC	GAC	GGC	C
	GUA	GCA	GAA Glutamic Acid (Glu)	GGA	A
	GUG	GCG	GAG	GGG	G

Figur 1.3: Figur viser en tabel over de forskellige aminosyrer og hvilke codon, der koder for hvert bestemt aminosyre. Flere forskellige codon kan lede til den samme aminosyre.[11]

En aminosyre sekvens kan således repræsenteres på flere måder, som kan ses på figur 1.4.

[1] Formatteringen af sekvenserne er for øget læsbarhed.

Alanine	Ala	A	Methionine	Met	M
Cysteine	Cys	C	AsparagiNe	Asn	N
Aspartic Aci**D**	Asp	D	**P**roline	Pro	P
Glutamic Acid	Glu	E	Glutamine	Gln	Q
Phenylalanine	Phe	F	A**R**ginine	Arg	R
Glycine	Gly	G	**S**erine	Ser	S
Histidine	His	H	**T**hreonine	Thr	T
Isoleucine	Ile	I	**V**aline	Val	V
Lysine	Lys	K	Tryptophan	Trp	W
Leucine	Leu	L	TYrosine	Tyr	Y

Figur 1.4: Oversigt over de 20 naturlige forekomster af aminosyrer, og henholdsvis 3-bogstavs og 1-bogstavs identifikation af hver aminosyre.[12]

At translatere mRNA sekvensen fra tidligere $CGU\ CCA\ UUC$ til en sekvens af aminosyrer, er i princippet ikke muligt, da denne sekvens ikke indeholder et startcodon, eller nogen stopcodon. Derfor indsætter vi et startcodon og en af de tre mulige stopcodon. mRNA sekvensen kunne dermed se ud således: $AUG\ CGU\ CCA\ UUC\ UAA$.

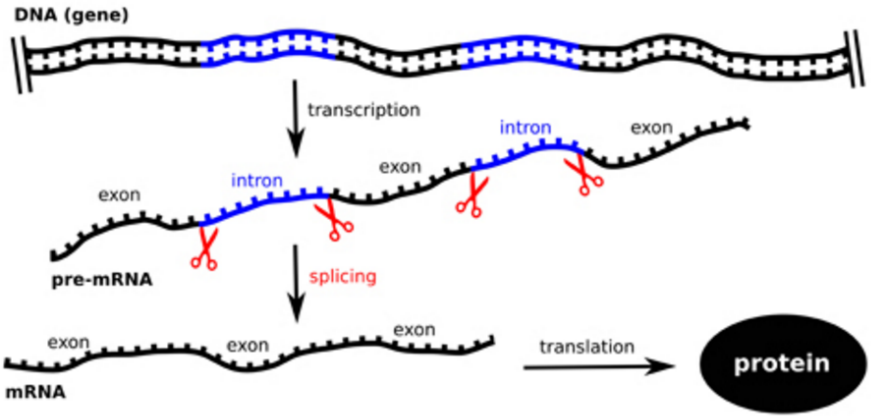
For at translatere mRNA sekvensen anvendes figur 1.3 ved at slå op i tabellen, med tre baser ad gangen. Dermed svarer første codon til aminosyren Methionin, andet codon til Arginin, tredje codon til Prolin og fjerde codon til fenylalanin. Femte codon er et stopcodon, som ikke svarer til en aminosyre og derfor ikke er med i aminosyresekvensen. 3-bogstavs identifikation for de tre aminosyrer, som følge af figur 1.4 er følgende $Met\ Arg$, Pro og Phe. Dermed er sekvensen af aminosyrer $N - Met - Arg - Pro - Phe - C$.[2]

1.4.3 Splicing

Transskriptions processen er lidt mere kompliceret end hidtil beskrevet. Når DNA bliver transskriberet, bliver der først dannet det, der kaldes for *pre-mRNA*[3]. Det er nødvendigt, fordi DNA består af *ikke kodende-*, og *kodende dele*. Ikke kodende dele kaldes for *introns* og kodende dele for *exons*. Figur 1.5 illustrerer, hvordan dette foregår.

[2] N og C anvendes som terminaler eller retninger for sekvensen. Hvor henholdsvis N svarer til $5'$ og C for $3'$, for en aminosyre sekvens.

[3] Dette foregår, som hidtil beskrevet vha. erstatsningsreglerne (transskription)



Figur 1.5: En DNA sekvens transskriberes til pre-mRNA. Pre-mRNA behandles i en process der kaldes for *splicing*, ved at fjerne introns (delsekvenser). Herefter bliver exon sekvenserne konkateneret, som danner det endelige mRNA. Herefter translateres mRNA på sædvanlig vis. [13]

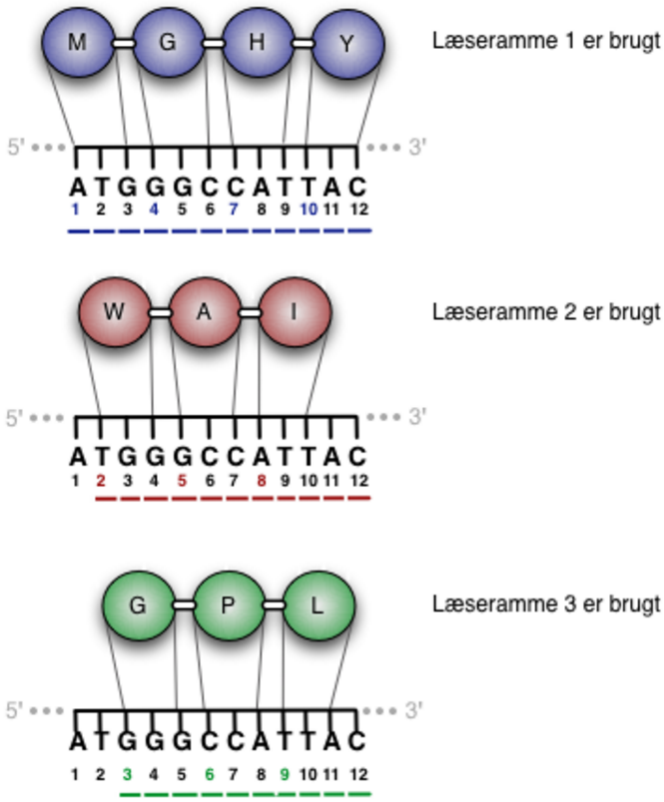
.

Selve mekanismen, som udvælger hvilke delsekvenser der skal fjernes under splicing processen, er rettere kompliceret. Splicing gør at det er muligt at have den samme pre-mRNA sekvens, men mange forskellige mRNA sekvenser, da ikke på forhånd er afgjort, hvilke delsekvenser der fjernes. Fordelen ved dette er, at én DNA sekvens kan udtrykke flere proteiner[13].

1.4.4 Læserammer

Hvis man har givet en DNA sekvens, som fx $GAT\ GCG\ GTA\ AGT$, så vil man opdage at sekvensen, hvis transskriberet til pre-mRNA, ikke indeholder et startcodon eller et stopcodon. Alligevel indeholder sekvensen begge disse codon. De kan findes, ved at ændre *læserammen* for sekvensen. Det man gør, er at man starter med at læse sekvensen, fra symbol nummer 2 i stedet for symbol nummer 1. Hermed får man sekvensen $ATG\ CGG\ TAA$, som indeholder både start-, og stopcodon, når transskriberet til pre-mRNA[4]. Idéen med læserammer, er at hvis man ændrer læserammen for en sekvens, så ændrer man også hvilke aminosyrer den translaterede sekvens består af. Det er muligt at anvende forskellige læserammer, fordi sekvenser aflæses som 3-tupler [14]. Dette illustreres yderligere på figur 1.6. Desuden så har ét DNA molekyle, seks forskellige læserammer, da en DNA sekvens har en komplementær sekvens.

[4]Det første symbol bliver ikke taget med, og heller ikke de to sidste, da disse ikke kan translateres til codon



Figur 1.6: Figuren viser, hvordan den samme DNA sekvens, aflæses ved hjælp af de tre forskellige læserammer, og hvordan det resulterer i tre forskellige sekvenser af aminosyrer, hvis DNA sekvensen blev transskriberet og oversat med henholdsvis læseramme 1, 2 og 3. [14]

.

En *åben* læseramme betegner en delsekvens, som har muligheden for at blive oversat. Hvis man har givet en DNA sekvens, som fx $TTG\ ATG\ CGG\ TAA\ CGT$. Den åbne læse ramme ville så være, $ATG\ CGG\ TAA$, da denne har mulighed for at blive translateret til et protein, som blev vist tidligere. Delsekvensen kan også kaldes for *den kodende sekvens*, hvis delsekvensen er en åben læseramme og hvis den bliver transskriberet og translateret [5][15].

1.4.5 Afrunding af det centrale dogme

Idéen ved at tage udgangspunkt i et fagområde, er at programmeringssproget vha. teorien fra fagområdet kan designes sådan, at det kan anvendes til at lære om det centrale dogme, og samtidigt kan sproget være mere intuitivt for dem, som i forvejen har god kendskab til DNA, dens replikation og hvordan proteiner syntetiseres på baggrund af DNA og RNA.

[5] Engelsk: *the coding sequence, CDS*

1.4.6 Sekvens operationer

Dette afsnit er en kort opsummering af forrige afsnit 1.4.1, med henblik på at påpege nogle af de interessante ting, som man skal kunne beregne eller specificere.

DNA

Det som er interessant ved en DNA, er at kunne finde dens komplementære sekvens, da det kan være at, den aminosyre sekvens man leder efter, befinder sig i den komplementære sekvens. Samt at kunne transskribere DNA til mRNA. I selve transskriptions processen, ville det være nyttigt, at kunne specificere introns og exons, og at transskriptionen herefter kun konkatenerer exons fra pre-mRNA. Hvis man har en DNA sekvens, at det så vil være muligt, at finde åbne læserammer, ved at specificere længde, eller kun den længste.

mRNA

At man kan finde bestemte codon i en mRNA sekvens, ved at specificere en aminosyre eller 3-tupel, som koder for den pågældende aminosyre. At translatere mRNA til en sekvens af aminosyrer.

Generelt

At kunne manipulere sekvenser, ved at indsætte delsekvenser, eller fjerne dem. Dette vil dermed understøtte splicing processen.

1.4.7 Værktøjer og sprog

Indenfor bioteknologi, biologi og bioinformatik er der et antal programmeringssprog, som typisk anvendes: R, Python, Perl og MatLab. Der lader til at være en konsensus indenfor faget, at disse sprog tjener hvert deres formål. Og dermed at der ikke findes ét sprog, som kan klare en bred vifte af opgaver optimalt. Alle disse sprog er 4. generations sprog, dvs. de er højniveau sprog. Sprogene er multiparadigme og er derfor generelt både imperative, objekt-orienterede, procedure og/eller funktionelle. Alle sprog kan dog bruges, fx er Java og C# også benyttede. Der findes en mængde af værktøjer/frameworks til sprog, som oftest bliver brugt. (BioPython, BioPerl, osv.) [16].

For at få mere information omkring, hvilke sprog målgruppen anvender, har vi taget kontakt til flere ph.d. studerende indenfor bioteknologi, dog har vi kun fået ét respons. Det kan findes i bilag A. Til gengæld stemmer det meget godt overens, med det information vi har fundet tidligere, hvilket er at målgruppen bliver undervist i at anvende værktøjet Matlab, samt at sproget R bruges til statiske beregninger. Fra vores ene respons har vi ikke fået mere information omkring brugen af sprogene Perl eller Python, men til gengæld anvender målgruppen et antal værktøjer.

Værktøjer, som ph.d. studerende anvender

- Pymol
- chirascan
- CCP4i
- CLC
- PEAQ-ITC
- mMass
- coot
- GraphPad prism

En oversigt over disse værktøjer, og hvad de kan anvendes til, kan findes i bilag B.

Det er her, at det er vigtigt at differentiere mellem de forskellige studier, som vi beskrev i afsnit 1.1, da det er typisk for bioteknologer, at anvende værktøjer, hvorimod biologi og bioinformatik, til en hvis grad, har fokus på programmering i forvejen [17].

Det giver anledning til spørgsmålet: *Hvorfor overhovedet have fokus på en målgruppe, som bioteknologer, hvis der allerede findes værktøjer, som kan opfylde deres behov?*

Der er flere argumenter for, at det er en god idé at have fokus på at implementere et programmeringssprog, specifikt for denne målgruppe.

Det første argument omhandler, at man skal investere tid, for at lære at anvende et værktøj. Listen af værktøjer, som ph.d. studerende anvender er omfattende, og tiden som bliver brugt på at tilegne sig kendskab til nogle special værktøjer, kan også bruges på programmering.

Samtidigt kan man være begrænset af værktøjer, fx hvis man skal løse en opgave, men hvortil der ikke findes et værktøj, der kan hjælpe med at løse den specifikke opgave. Her ville man have en fordel, hvis man kunne programmere [17].

Det kan også være, at det tager lang tid at anvende eksisterende værktøjer, hvor man kan skrive små programmer, som kan løse meget manuelt arbejde, ved at automatisere en arbejdsprocess[17].

Hermed er der god grund til at designe et sprog, som kan støtte indlæring af programmering, som også er særdeles anvendeligt indenfor et specifikt fagområde, som ligger indenfor bioteknologien.

1.4.8 Analyse af værktøjer

I dette afsnit vil vi analysere nogle af de værktøjer, der bliver brugt inden for bioteknologi. I analysen vil vi have fokus på hvad disse værktøjer tilbyder, som kan være til gavn når man arbejder inden for bioteknologi.

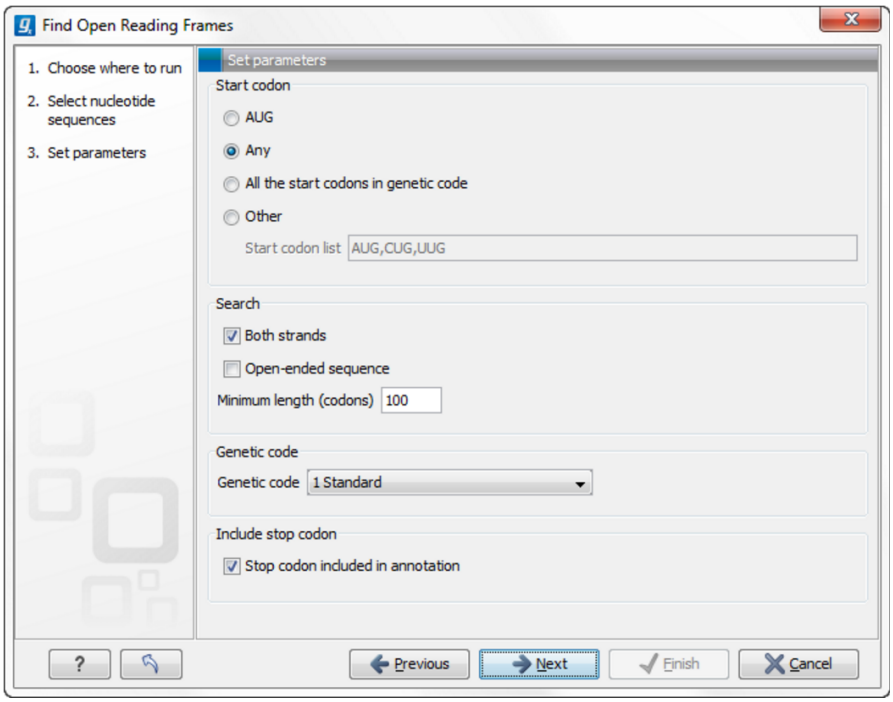
CLC Sequence Viewer

Indtil videre har vi nævnt et antal værktøjer, men der er et værktøj, som er mere interessant at kigge nærmere på, end dem der står i bilag B. Vi analyserer CLC Sequence Viewer, med henblik for at finde ud af, hvilke behov dette program opfylder, i forhold til det centrale dogme.
Nogle vigtige funktionaliteter i dette program er følgende[18]:

Nukleotid[6] analyse

- Konvertere DNA til RNA (transskription)
- Konvertere RNA til DNA (omvendt transskription)[7]
- Omvendt komplementære sekvens
- Omvendt sekvens
- Transskription af DNA til RNA
- Translation af RNA til protein
- Finde åbne læserammer

CLC Sequence Viewer er et GUI baseret program. Figur 1.7 viser hvordan man kan bruge programmet til at finde åbne læserammer for nukleotid sekvenser.



Figur 1.7: Parametre for søgning efter åbne læserammer inkluderer: specifikation af startcodon, om der skal søges i den ene sekvens eller også den komplementære sekvens, kun finde åbne læserammer af minimum længde og hvilken codon translations tabel der anvendes. [19]

Funktionerne nævnt tidligere, understøtter det der kaldes for *sekvens analyse*. Formålet med sekvens analyse er at udsætte en DNA sekvens, RNA sekvens eller en sekvens af aminosyrer, ud for en række analytiske metoder, for at forstå en organismes egenskaber,

[6] Nukleotid dækker over DNA og RNA
[7] Proces der finder en DNA sekvens ud fra en given RNA sekvens.

funktion, struktur og evolution [20]. Herunder er der en metodologi, som kaldes for sekvenssammenligning, som netop går ud på at sammenligne to sekvenser af DNA, RNA eller aminosyrer, og på denne måde beregne, hvor ens eller uens de to sekvenser er [21].

Biopython

Biopython[22] har vi valgt at analysere, grundet at det kombinerer kodning med operationer, der er interessante for Bioteknologi. Biopython er en række værktøjer, som er i form af klasser, moduler og fortolkere[23], der er skrevet i programmeringssproget Python[24]. Målet med disse værktøjer er, at gøre det så nemt som muligt, at hente data fra hjemmesider og fortolke data fra forskellige filformater, og på dette data gøre brug af algoritmer, der bliver brugt inden for felter som bioteknologi. Alt dette ved brug af Python. For at bruge disse værktøjer bør man have kendskab til Python, da dette kan anses for at være en udvidelse hertil. Nedenfor vil der være eksempler på, hvad værktøjerne i Biopython kan bruges til, eksemplerne er fra guiden til brug af Biopython[23], der findes selvfølgelig mange flere operationer og beregningsmuligheder end der bliver vist i dette afsnit.

Et eksempel på en af de simple ting man kan i Biopython, kan ses i listing 1.1. I dette eksempel fra linje 1 til 3 importeres modulerne *Seq* for i dette tilfælde at kunne deklarere sekvenser, *IUPAC* for at kunne bruge en række alfabeter og *GC()*, som er en metode for at kunne finde ud af hvor stor en del af den givne sekvens består af *G* og *C*. Efter at modulerne er importeret bliver der fra linje 4 til 6 lavet en sekvens, *my_seq*, hvorefter metoden *GC* bliver brugt på sekvensen og giver output der kan ses på sidste linje 6.

```
1  >>> from Bio.Seq import Seq
2  >>> from Bio.Alphabet import IUPAC
3  >>> from Bio.SeqUtils import GC
4  >>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
       IUPAC.unambiguous_dna)
5  >>> GC(my_seq)
6  46.875
```

Listing 1.1: Eksempel på brug af sekvens objekt

Biopython understøtter også mulighed for at finde komplementære sekvenser. Et eksempel på dette kan ses i listing 1.2. Her bliver der på linje 3 deklareret en sekvens *my_seq* hvorpå der på linje 6 og linje 8 bliver brugt metoderne *complement()* og *reverse_conplement()* for at give produktionerne set på linje 7 og 9.

```
1  >>> from Bio.Seq import Seq
2  >>> from Bio.Alphabet import IUPAC
```

```
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC",
    IUPAC.unambiguous_dna)
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
```

Listing 1.2: Komplementær sekvenser i Biopython

Ud over Biopython findes der også andre lignende værktøjer til andre sprog, her kan der eksempelvis være tale om BioPerl[25] til sproget Perl, og BioJava[26] til sproget Java. Disse værktøjer har også til indsigt at gøre det nemmere at foretage beregninger indenfor områder som bioteknologi. Forskellen ligger i hvad for et programmeringssprog der bruges.

Ud fra dette afsnit skulle det gerne fremgå at Biopython og dermed også programmeringssproget Python, som dette værktøj er bygget på, er nyttigt inden for felter som bioteknologi.

1.5 Det centrale dogme i kode

I dette afsnit vil vi se på operationer inden for det centrale dogme, som vi beskrev i afsnit 1.4.1, implementeret i eksisterende programmeringssprog. I afsnit 1.5.1 vil vi se på implementationen af funktioner der konverterer DNA til mRNA i C, og i afsnit 1.5.2 vil vi se på implementationen af KMP algoritmen i nogle udvalgte sprog.

1.5.1 Transskription af DNA

I dette afsnit vil vi præsentere implementationer af to funktioner i programmeringssproget C, da det er et sprog som vi har erfaring med. Den ene funktion i listing 1.3 viser en implementation af en funktion, der transskriberer DNA til mRNA. Den anden funktion i listing 1.4 viser en implementation af en funktion, der finder en komplementær sekvens, for en DNA sekvens. Vi har udvalgt netop disse funktioner fordi de er simple og fordi de er en fundamental del af det centrale dogme, som beskrevet i afsnit 1.4.1.

```
void TranscribeDNATomRNA(const char DNA[], char mRNA[]){
int counter = 0;
while(DNA[counter] != '\0'){
    if(DNA[counter] == 'T'){
        mRNA[counter] = 'U';
    }
    else{
```

```
            mRNA[counter] = DNA[counter];
        }
        counter++;
    }
    mRNA[counter+1] = '\0';
}
```

Listing 1.3: Funktion der tager en DNA sekvens ($5'$ mod $3'$) som input, og konverterer denne til RNA.

```
void FindComplementarySequence(const char DNA[], char compDNA[]) {
    int counter = 0;
    while(DNA[counter] != '\0') {
        if(DNA[counter] == 'A') {
            compDNA[counter] = 'T';
        }
        else if(DNA[counter] == 'T') {
            compDNA[counter] = 'A';
        }
        else if(DNA[counter] == 'G') {
            compDNA[counter] = 'C';
        }
        else if(DNA[counter] == 'C') {
            compDNA[counter] = 'G';
        }
        counter++;
    }
    compDNA[counter+1] = '\0';
}
```

Listing 1.4: Funktion der tager en DNA sekvens som input, og konverterer denne til mRNA.

Ud fra de ovenstående eksempler, så er det at skulle programmere nogle basale funktioner, som kun udfører transskriptioner, relativt omfattende i C. Tilmed er der ikke nogen simpel syntaks for konverteringer mellem DNA og RNA, da der hver gang man ønsker at konvertere, skal kaldes en funktion. Desuden så er der ikke nogen specifikke typer i C, som kan udføre typecheck, på en DNA variabel, da DNA vil være lagret, som et char array. Hertil kunne dedikerede typer rapportere bedre fejlmeddelelser, hvis en DNA type fx indeholdte et symbol som 'K', som ikke er en valid base i en DNA sekvens.

1.5.2 KMP algoritme

Vi har implementeret en algoritme, der er relevant for bioteknologer. Vi har her valgt Knuth-Morris-Prat algoritmen [27], som implementeres i nogle udvalgte sprog, som vi ved der bruges inden for området på baggrund af information fra afsnit 1.4.7 og 1.4.8. KMP algoritmen kan finde en specificeret delstreng i en givet streng og returnerer

det sted i strengen hvor delstrengen starter, dette kan anses for at være en relevant algoritme for bioteknologer, fordi den kan bruges på en sekvens og angive det sted hvor en delsekvens starter, disse delsekvenser kunne eksempelvis være start- og stop-codon for at finde ud af hvor en translation starter og slutter, som beskrevet i afsnit 1.4.2. Vi ønsker at se hvor krævende det er at implementere algoritmen i de udvalgte sprog samt se hvilke konstruktioner der er nødvendige i de forskellige sprog for at en algoritme som KMP kan implementeres. I figur 1.8 ses KMP algoritmen.

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an integer (the zero-based position in S at which W is found)

    define variables:
        an integer, m ← 0 (the beginning of the current match in S)
        an integer, i ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    while m + i < length(S) do
        if W[i] = S[m + i] then
            if i = length(W) - 1 then
                return m
            let i ← i + 1
        else
            if T[i] > -1 then
                let m ← m + i - T[i], i ← T[i]
            else
                let m ← m + 1, i ← 0

    (if we reach here, we have searched all of S unsuccessfully)
    return the length of S
```

Figur 1.8: Knuth-Morris-Prat algoritme (KMP) [27]

R

I dette afsnit undersøger vi hvorvidt R lever op til de tidligere definerede kriterier. I listing 1.5 ses implementationen af KMP algoritmen [27] i R. Der medfølger en hjælpefunktion i listing 1.6.

```r
kmp_search <- function(S = S, W = W){
  m <- 0
  i <- 0
  T <- kmp_table(S)
```

```
5
6     while ((m + i) < nchar(S)) {
7       if (substr(W, i, i) == substr(S, m + i, m + i)) {
8         if (i == (nchar(W) - 1)) {
9           return(m)
10        }
11        i <- i + 1
12      }
13      else {
14        if (T[i] > -1) {
15          m <- m + i - T[i]
16        }
17        else {
18          m <- m + 1
19          i <- 0
20        }
21      }
22    }
23  }
```

Listing 1.5: KMP funktion i R

```
1  kmp_table <- function(S = S){
2    pos <- 2
3    cnd <- 0
4    T <- integer(nchar(S))
5    T[0] <- -1
6    T[1] <- 0
7
8    while (pos < nchar(S)) {
9      if (substr(S, pos -1, pos -1) == substr(S, cnd, cnd)) {
10       T[pos] <- cnd + 1
11       cnd <- cnd + 1
12       pos <- pos + 1
13     }else if (cnd > 0){
14       cnd <- T[cnd]
15     } else {
16       T[pos] <- 0
17       pos <- pos + 1
18     }
19   }
20   return(T)
21 }
```

Listing 1.6: Hjælpefunktion - Generer et array T, som hovedfunktionen skal bruge

Implementationen af denne algoritme i sproget R adskiller sig meget fra de andre sprog i det at operatorer som $+=$ og $++$ ikke bruges. Endvidere bruges $<-$ for at tilskrive en variabel frem for $=$ som det gør sig gældende for mange andre sprog. Dog ligner denne

implementation den der er angivet i figur 1.8 mere end nogle af implementationenerne i de andre sprog og vi vil derfor antage at algoritmen vil være intuitiv for bioteknologer at implementere i R.

Python

I dette afsnit vil vi se på implementationen af KMP algoritmen i sproget Python. Implementationen kan ses i listing 1.7. Denne implementation er fra [28].

```
1  def KnuthMorrisPratt(text, pattern):
2      # allow indexing into pattern and protect against change during
           yield
3      pattern = list(pattern)
4
5      # build table of shift amounts
6      shifts = [1] * (len(pattern) + 1)
7      shift = 1
8      for pos in range(len(pattern)):
9          while shift <= pos and pattern[pos] != pattern[pos-shift]:
10             shift += shifts[pos-shift]
11         shifts[pos+1] = shift
12
13     # do the actual search
14     startPos = 0
15     matchLen = 0
16     for c in text:
17         while matchLen == len(pattern) or \
18               matchLen >= 0 and pattern[matchLen] != c:
19             startPos += shifts[matchLen]
20             matchLen -= shifts[matchLen]
21         matchLen += 1
22         if matchLen == len(pattern):
23             yield startPos
```

Listing 1.7: KMP algoritmen i Python

I forhold til implementationen i Python kan det tydeligt ses at den ikke fylder mange linjer kode, sammenlignet med eksempelvis implementationen i PERL, afsnit 1.5.2. Dog mener vi at for målgruppen vil læsbarheden af denne implementation være lav grundet den måde operationerne er sat op på, eksempelvis som i linje 6 hvor et *array* bliver sat til længden af *pattern*. Vi mener derfor ikke at det er specielt intuitivt for en bioteknolog, som ikke er vant til at skrive i Python, at forstå implementationen af denne algoritme i sproget.

Selv om læsbarheden er lav mener vi at udtryksfuldheden i implementationen er høj grundet at typerne er bestemt ud fra udtrykket og skal ikke skrives manuelt, samt un-

derstøttelse af korte og præcise udtryk som + = og >= og undladelse af brugen af {} omkring løkker og lignende.

Perl

I dette afsnit vil vi se på implementationen af KMP algoritmen i sproget Perl. Denne implementationen kan ses i listing 1.8. Implementationen af algoritmen i Perl er fundet fra [29].

```
sub knuth_morris_pratt {
    my ( $T, $P ) = @_; # Text and pattern.
    use integer;
    my $m = knuth_morris_pratt_next( \$P );
    my ( $n, $i, $j ) = ( length($T), 0, 0 );
    my @next;

    while ( $i < $n ) {
        while ( $j > -1 &&
                substr( $P, $J, 1 ) ne substr( $T, $i, 1 ) ) {
            $j = $next[ $j ];
        }
        $i++;
        $j++;
        return $i - $j if $j >= $m; # Match.
    }
    return -1; #Mismatch.
}
sub knuth_morris_pratt_next {
    my ( $P ) = @_; # The pattern.
    use integer;
    my ( $m, $i, $j ) = ( length $P, 0, -1 );
    my @next;
    for ($next[0] = -1; $i < $m; ) {
        #Note that this while() is skipped during the first for()
            pass.
        while ( $j > -1 &&
                substr( $P, $i, 1 ) ne substr( $P, $j, 1 ) ) {
            $j = $next[ $j ];
        }
        $i++;
        $j++;
        $next[ $i ] =
            substr( $P, $j, 1 ) eq substr( $P, $i, 1 ) ?
                $next[ $j ] : $j;
        }
    return ( $m, @next ); # Length of pattern and prefix function.
```

```
37 }
```

Listing 1.8: KMP algoritme formuleret i Perl

I Perl ser vi at det ikke er muligt at indekse en streng på normal vis, såsom Javas $String.charAt(i)$ metode, i stedet bruges funktionen *substr()*, som egentlig burde bruges til at returnere en delstreng, og ikke bare et enkelt tegn. Det specielle ord *my* er en måde at erklære scope af variabler. Variabler er som udgangspunkt globale, medmindre *my* bliver defineret. I eksemplet ser vi at dette skal gøres for alle variabler.

Det har lykkedes at implementere, eller finde implementationer af algoritmen i de udvalgte sprog. Kodeeksemplerne tyder på at implementationen af KMP algoritmen, afviger meget fra sprog til sprog. Dog er der nogle af implementationerne, der minder mere om den givne algoritme, som fx figur 1.8, end andre, som fx R. Mens andre i mindre grad ligner algoritmen i figuren, såsom Python. Dette er primært grundet at eks. Python implementationen benytter sig af mange indbyggede funktioner, som gør at formuleringen bliver mere kortfattet.

1.5.3 Vurdering af sprog

Vi vil se på hvordan man vurderer programmeringssprog, og dermed definere hvorledes vi vil kvantificere et sprog, og holde det op imod nogle krav. Vi giver derfor en række karakteristika, som vi bruger til at kigge på sprog, med henblik på at kunne drage nogle konklusioner af disse. Dette skal videre bruges som grundlag for design- og implemenations-mæssige beslutninger i vores eget sprog. Vi vil benytte dette afsnit til at teste hvorvidt vores eget sprog lever op til kriterierne - om i hvilken grad sproget er læsbart, skrivbart og pålidelig.

Kriterier

Vi ønsker at kvantificere sprogene ud fra målgruppens synspunkt. Vi tager udgangspunkt i kriterierne beskrevet I *Concepts of Programming Languages* [30], set på Figur 1.9, men indrager også vore egne. Vi ser på sprogene med perspektivet af vores målgruppe (bioteknologi studerende). I Figuren 1.9 ses karakteristika, samt hvilket kriterier de falder ind under (læsbarhed, skrivbarhed eller pålidelighed).

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Figur 1.9: Kriterier for evaluering af sprog, Concepts of Programming Languages [30]

Vi benytter definitionerne fra bogen *Concepts of Programming Languages* [30]. Karakteristika definereres således:

Data typer

Vi bedømmer dette ud fra om hvorvidt et sprog har de fornødne data typer, til at evaluere en given konstruktion. Dvs. har et sprog ikke understøttelse for Boolean værdier, bliver vi nødt til at benytte en integer type og bruge værdierne 1 og 0 eller lignende. Dette er ikke godt når vi bedømmer et sprogs indbyggede datatyper. Men hvis et sprog understøtter at kunne implementere sine egne data typer (eks. som klasser), er dette et plus, da sproget kan udvides i den specifikke situation. Desto bedre mening vi kan formulere med de indbyggede data typer, desto højere vil vi vurdere dette kriterie.

Syntaks design

Syntaks design betyder hvorvidt sprogets syntaks er let læselig, og effektiv at skrive - Altså sproget kan formulere sig kort og præcist. Der er mange faktorer som har indflydelse på dette. Primært sprogets specielle ord (*while, if, class, for, ...*), har stor vigtighed for dette, da de giver mulighed for at højne læsbarheden markant.

Udtryksfuldhed

Her bedømmes et sprogs muligheder for at udtrykke sig kort og præcist. Muligheden for at formulere *[i = i + 1]*, som *[i++]*, giver en enorm sammenfattet og præcis formulering af udtrykket. Generelt betyder dette at sprogets brug af specielle ord eller konstruktioner, medføre at man kan kortfattet formulere beregninger eller strukturere, hvilket højner programmørens evne til at udtrykke sig effektivt i sproget.

Krav

Sprogets fokus er på bearbejdning af DNA, RNA og aminosyrer. Til det har vi et behov for at opstille nogle krav. Disse krav beskriver hvad vi ønsker fra vores sprog.

Sproget skal kunne:

- Understøtte sekvens-operationer (DNA/RNA)
- Understøtte muligheden for at implementere egne algoritmer og datatyper
- Være let tilgængeligt (være let at gå til, uden tidligere programmeringserfaring)

For senere reference, anvender vi disse krav på eksisterende sprog. Vi ønsker at vise hvorledes de eksisterende sprog ikke lever op til disse krav. Hvorved vi så har et udgangspunkt for vort eget sprog.

Analyse

Vi analyserer sprogene R, Python og Perl. Vi kigger på deres syntaks og design, med udgangspunkt i de karakteristika som vi definerede tidligere. Vi slutter med at opsummere hvad vi kan drage af dette afsnit til sidst. De kodeeksempler som vi tager udgangspunkt i, er KMP implementationerne (se Afsnit 1.5.2).

R

R er er et programmeringssprog med fokus på statistik og data-analyse. R er et interpreteret sprog, ofte benyttet gennem et command-line interface. R kan også skrives i scripts, og køres deraf. R er multiparadigme og understøtter: Array, objektorienteret, imperativ, funktionel, procedure og reflektiv programmering. R er dynamisk typed, og tilgængeligt på mange operativsystemer.

Data typer R indeholder en række data typer - dog ikke de typiske typer. Da R er rettet imod matematisk og statistik benyttelse, indeholder R data typer som er relevante for dette felt. *Vectors, lists, matrices, arrays, factors, data frames, logical, numeric, integer, complex, character* and *raw* er de indbyggede data typer i R. Dette kan gøre det svært for typiske programmører at bruge R, udenfor et matematisk scope. Men det er ikke umuligt. R understøtter brugerdefinerede klasser, via *setClassmethods*. Dette gør det muligt at implementere egne typer, hvis det ønskes.

Syntaks design R's syntaks ligner meget en typisk matematisk pseudokode syntaks (se KMP pseudokode 1.8). Bl.a. benytter R, '<-' som assignment-operator. Dette vil være familiært for matematikere. R er derfor yderst læsbar for brugere indenfor faget - hvorimod typiske programmører skal vende sig til det.

Udtryksfuldhed Da R følger en meget specifik syntaks, er der ikke megen rum til at kunne udtrykke sig kort. F.eks. findes der ingen incrementer ($i++$) funktionalitet i R. Man er derfor tvunget til at skrive '*var* <- *var* + *1*'. Dette er gennemgående i R's syntaks. R tillader derfor ikke megen udtryksfuldhed.

Python

Python er et højniveau generelt sprog.Python er, som R, også et fortolket sprog og dynamisk typed. Endvidere har Python og R begge tilfældes at de er multiparadigme sprog, og er tilgængelige på mange operativsystemer.

Data typer Python indeholder mange data typer. Bl.a. har Python indbyggede typer såsom *dict, list, set, tuple, unicode, frozenset, string, int, float, long, complex*, samt flere. Endvidere har Python et relativt stort standard bibliotek. Python tillader derudover at definere egne typer. Python har derfor en meget stærk side, ift. data typer.

Syntaks design Pythons design er meget fokuseret på læsbarhed. Dette ses ved brugen af whitespace som erstatning til de traditionelle start/slut-tegn eller -ord. Python benyttet sig af whitespace til at adskille konstruktioner. Der findes dermed ingen *"{...}"* eller *"Begin... End"* syntaks i Python. Dette gør at Pythons syntaks minder mere om et naturligt sprog, og dermed er let læseligt sammenlignet med sprog som R.

Udtryksfuldhed Python er designet med målet om at skulle skrive så få linjer kode som muligt. Det er muligt et udtrykke sig kortfattet i Python. Sammenlignet med Java eller C++, er Python et meget effektivt sprog at formulere sig kort i.

Perl

Perl er et højniveau generelt sprog. Ligesom R og Python, er Perl et dynamisk typed, interpreteret sprog. Perl blev udviklet med fokus på at være et UNIX scripting sprog. Det er derfor udbredt blandt sysadmin-, netværks- og finansverden. Perl er også brugt indenfor bioinformatik.

Data typer Perl har de basale datatyper såsom diverse tal typer, string og reference. Disse faldet under klassifikationen *skalar* - disse bliver alle deklareret ved et $-tegn. Derudover findes der arrays, some understøtter alle skalar typer. Disse bliver deklareret ved @-tegn. Derudover har Perl indbygget type for hashing. deklaret ved %-tegn. Perl er meget speciel, med måden man benytter disse tegn til at deklarere forskellige typer.

Syntaks design Perl er ofte omtalt som et grimt sprog, eller et skrivbart-fokuseret sprog. Dvs. Perl ses som et sprog der er effektivt at skrive, med har en lav læsbarhed - specielt hvis læseren ikke er erfaren med syntaksen. Så på den ene side er syntaksen utrolig effektiv, men kun for skriveren. Det er derfor et svært sprog for folk uden den store programmeringserfaring.

Udtryksfuldhed Pga. Perls anderledes syntaks design er det meget effektivt at formulere sig i. Syntaksen gør det muligt at skrive meget kort kode. Dette højner udtryksfuldhed, gør det derimod udfordrende at læse, hvis man ikke er erfaren Perl bruger, som nævnt tidligere.

Opsummering

Vi vil gennemgå hvad vi har lært af at kigge på sprogene. Vi sammendrager hvad vi kan lære af deres design og generelle opbygning.

R lægger sig meget tæt op af matematiske udtryk og almen syntaks, da R er designet med statistisk arbejde i fokus. R er derfor ikke optimalt til langt de fleste generelle områder. R er et meget specialiseret sprog, men som dog stadig kan benyttes udenfor dets scope.

Python er et meget generelt sprog. Dette ses i dets syntaks design, samt det enorme standard bibliotek, og indbyggede funktionalitet. Python er derudover også multiparadigmt, og er tilgængeligt på et hav af platforme. Dette gør Python ekstremt brugbart i mange senarier. Men ender også med at være 'jack of all trades, master of none'. Hvis man ønsker at bruge Python indenfor specifikke fagområder findes der dog et hav af udvidelser, såsom BioPython.

Perl blev udviklet som et sprog til at indgå i større systemer, men kan sagtens stå på egne ben. Perl har en lidt speciel syntaks, som gør det effektivt at skrive i men, samtidig udgør dette en udfordring for nye brugere. Perl er derfor knapt så bruger venlig, men kan være ekstremt effektiv hvis man benytter det i den rette kontekst.

Perl og Python har hvert deres udvidelses bibliotek, specifikt til bioinformatik. Men dette ser vi bort fra her, da vi udelukkende har ønsket at kigge på sprogene selv. Vi vælger udelukkende at kigge på sprogenes syntaks, da det projektet sigter efter et nyt sprog, og ikke et nyt bibliotek.

I tabel 1.1 ses det hvorledes vi har vurderet sprogene. Vi ser at de alle understøtter data typer, enten i højere eller mindre grad. Vi ser også at R og Python har et udemærket syntaks design. Vi har vurderet at R har et godt design, idet det ligner meget matematiske notationer, og dermed antager vi at det vil være oplagt for videnskabelige studerende generelt at kunne lære det hurtigt. Endvidere ser vi at mens Python og Perl gør det muligt at formulere sig kort, så er R ikke særlig god til dette.

Tabel 1.1: Karakteristika af sprog

Karakteristika	R	Python	Perl
Data typer	✓	✓	✓
Syntaks design	✓	✓	%
Udtryksfuldhed	%	✓	✓

I tabel 1.2 ses det hvorledes vi vurderer at sprogene lever på til vore krav. Vi ser at ingen af sprogene har indbygget funktionalitet til at håndtere sekvensoperationer. Det er derfor dette som vores eget sprog skal kunne, for at have en fordel over disse eksisterende sprog.

Tabel 1.2: Krav til sprog

Krav	R	Python	Perl
Sekvens-operationer	%	%	%
Algoritmer og datatyper	✓	✓	✓
Let tilgængeligt	%	✓	%

Vi har nu kigget på de eksisterende sprog. Vi vil nu fortsætte med at specificere løsningskriterierne for vort eget sprog. Dette tager udgangspunkt i de tidligere nævnte, men er dog en konkretisering for kriterierne som vort eget sprog skal bygges op omkring.

1.6 Problemafgrænsning

I dette afsnit vil vi afgrænse omfanget af projektet. Dette har til formål at bane vejen for at vi kan formulere det endelige problem. Vil vi også argumentere for at det sprog vi laver skal have specifikke typer, på baggrund af teorien fra afsnit 1.4.1.

I analysen har vi implementeret nogle simple funktioner fra det centrale dogme. Såsom at transskribere DNA til mRNA vist i afsnit 1.5.1, og implementeret en algoritme, der er relevant for det centrale dogme, hvilket var KMP algoritmen vist i afsnit 1.5.2, som er relativt omfattende at implementere. I afsnit 1.4.8 analyserede vi Biopython, hvor det er simpelt at udføre operationer, som at finde den komplementære sekvens, men hvor der er begrænsninger i forhold til at ændre allerede indlæste sekvenser, eller hvor det ikke er muligt at konkatenere en sekvens af DNA til en protein sekvens, uden at skulle specificere, at det skal transskriberes og translateres inden konkatenering. Der skrives mange linjer kode for at kunne transskribere DNA til mRNA, og implementationer af KMP algoritmen er meget forskellige fra sprog til sprog.

Vi afgrænser derfor til at designe og implementere et nyt sprog, hvor fokus vil være på at understøtte transskription mellem DNA, RNA og proteiner. Endvidere skal sproget understøtte notation og konstruktioner, der gør det muligt at implementere algoritmer i sproget. Ud over dette skal KMP algoritmen være en del af sproget pga. nyttigheden af at kunne finde delsekvenser i sekvenser. Denne afgrænsning medfører derfor at understøttelse af transskription og translation, samt mulighed for implementation af algoritmer og den direkte tilgang til KMP algoritmen i sproget vil være de vigtige elementer i vores programmeringssprog. Disse elementer vil derfor være prioriteret.

Vi afgrænser også til at sproget skal være et statisk typet sprog, som indeholder typer der er relevante for det centrale dogme Her er der specifikt tale om typer til at repræsentere DNA, RNA, Codon og Proteiner. Ud over disse typer skal sproget også indeholde typer, som *integers* og *booleans* grundet at der skal kunne implementeres algoritmer i sproget, som benytter iterationer og boolske udtryk. Grunden til at det er blevet valgt at sproget skal være statisk typet er fordi, som nævnt i bogen *Types and programming languages* [31], at det bliver hurtigere at opdage og rette fejl i den skrevne kode. Dette skyldes, at når det skrevne kode oversættes, vil type checkeren mælde fejl, hvis typer bliver brugt på en forkert måde i forhold til hvad der er lovligt i sproget. Typerne bliver således bestemt på oversættelsestidspunktet. At fejlene bliver hurtigere at opdage og rette medfører også at det bliver hurtigere at skrive fungerende kode. Endvidere udvider brug af typer i et statisk typet sprog også læsbarheden af den skrevne kode, dette sker eksempelvis gennem typen på en funktion eller metode, her er det nemt at se hvilken type af data, som den vil returnere når den bliver udført og det bliver nemt at se ud fra typen hvad variablen må bestå af. At typen er tydelig kan også anses, som en form for dokumentation, og til forskel fra kommentarer der er skrevet til kode, bliver denne dokumentation aldrig forældet siden den bliver tjekket hver gang koden kompileres.

Brugsmønstre

Ud fra afgrænsningen kan vi nu opstille nogle brugsmønstre. Brugsmønstrene fra dette afsnit vil være eksempler på hvad en bruger skal kunne anvende programmeringssproget til, som man ikke kan udføre på samme måde i andre programmeringssprog. Disse brugsmønstre vil senere blive brugt til at lave eksempler på programmer skrevet i sproget.

- Brugeren vil gerne selv definere en splicing proces, hvor brugerdefinerede delsekvenser skal fjernes (introns), og de resterende delsekvenser konkateneres (exons).
- Brugeren vil gerne finde et codon i en delsekvens af DNA eller mRNA ved at specificere det codon som brugeren leder efter.

- Brugeren vil gerne deklarere en DNA sekvens af typen DNA og efterfølgende transskriberer sekvensen til mRNA ved at tilskrive DNA sekvensen til en ny sekvens af typen mRNA.
- Brugeren vil gerne kunne finde den omvendte komplementære sekvens, af en given DNA sekvens, ved brug af indbyggede funktioner i sproget.
- Brugeren skal have mulighed for at kunne formatere output. Dvs. farve-highlighting, og separering af karakterer i en sekvens.

1.7 Problemformulering

I det følgende afsnit definerer vi en problemformulering for dette projekt. Problemformuleringen vil være udgangspunktet for det videre arbejde. Problemet er følgende:

Studerende som arbejder med bioteknologi, anvender som udgangspunkt værktøjer, som CLC Sequence Viewer, eller benytter omfattende biblioteker, som Biopython. Samtidigt er det omfattende at implementere lignende funktionalitet, som erstatning i eksisterende sprog, som C og R, for relativt meget simple processor, som transskriptionen.

Hvordan kan man designe og implementere et sprog hvorigennem man kan manipulere sekvenser af DNA, RNA og aminosyrer gennem begreber fra det centrale dogme i form af typer, operationer og funktionskald?

Formålet med sproget er dermed, at det skal være muligt for bioteknologistuderende, at arbejde med det centrale dogme ved at kunne skrive programmer i vores sprog.

1.8 Løsningskriterier

Vi opsætter løsningskriterier, med henblik på at dette er målene vi ønsker at opnå i vores sprog. Dette er kriterier som vi ønsker at opfylde med et nyt sprog. For at kunne være en løsning til dette projekts problem, skal sproget vi designer og implementerer opfylde de følgende kriterier:

Bio typer

Sproget skal indeholde typer for DNA, RNA, proteiner og codon, disse skal kun kunne indeholde gyldige tegn og der skal kunne laves variabler af disse typer gennem deklaration. Dette er med til at sikre, at fx en sekvens af aminosyrer ikke kan konverteres til DNA. At begrænse typernes tegn, er med til at kunne give brugeren bedre fejlbeskeder.

Operationer

Sproget skal understøtte basale sekvens operationer, som konkatenering af DNA, RNA og aminosyre sekvenser, fjerne delsekvenser og finde komplementære- og omvendte-sekvenser for DNA samt finde startpositioner for delsekvenser i sekvenser. Endvidere skal sproget også understøtte konvertering mellem DNA, RNA og Proteiner.

Funktionalitet fra imperativ programmering

Ud over biotyperne og de dertilhørende operationer skal sproget også indeholde kontrolstrukturer som *for*, *while* og *if else*, samt typerne *int* og *bool* og datastrukturer som *arrays*. Grunden til at dette tages med er for at der også gives mulighed for at udtrykke algoritmer og behandle data på flere måder, frem for kun at kunne udfører operationer på biotyperne.

Kapitel 2

Design af sprog og syntaks

Med vores problemformulering og løsningskriterier på plads kan vi nu påbegynde designet af vores sprogs syntaks, samt vælge scoperegler og parametermekanismer for vores sprog. Dette kapitel vil dermed omhandle design af vores programmeringssprog. Som det første vil vi i afsnit 2.1 analysere forskellige parsergeneratorer og derefter vælge et værktøj, som vi vil bruge i til at genererer vores lexer og parser. Efterfølgende vil vi i afsnit 2.2 give en specifikation af tokens for vores sprog og herigennem angive hvad de må bestå af. For at finde ud af hvordan vores syntaks for sproget skal være vil vi i afsnit 2.3 præsenterer en kontekstfri grammatik for vores programmeringssprog. Efterfølgende vil vi i afsnit 2.4 præsenterer nogle eksempelprogrammer ud fra den kontekstfrie grammatik. De efterfølgende afsnit vil omhandle vores scoperegler i 2.5 og valg af parametermekanismer i 2.6.

2.1 Parserværktøjer

I dette afsnit kigger vi på, og sammenligner populære parserværktøjer. Vi kan enten vælge at skrive vores skanner/parser selv, eller vi kan benytte et parserværktøj, som vi i dette afsnit vil referere til som værende et værktøj. Et parserværktøj tager et input, oftest bestående af en grammatik og en specifikation af tokens - og oftest *action code* for at generere et *abstrakt syntaks træ* - for at kunne generere en parser til en given grammatik. Vi vælger at benytte sådan et værktøj, da dette giver os muligheden for lettere at kunne revidere vores grammatik mm. senere i udviklingsprocessen. I tilfælde af vi ønsker at foretage ændringer i grammatikken, kan dette foretages på en praktisk og effektiv måde. Skriver vi vores egen parser, kræver det at vi skal tilbage og håndskrive ændringer - dette ønsker vi at undgå. Vi skal derfor vælge et værktøj, som er passende til vores ønsker. I tabel 2.1 ses de værktøjer vi sammenligner. Alle disse værktøjer kan producere en lexer og en parser.

Tabel 2.1: Sammenligning af parserværktøjer

	Parsing algoritme	Sprog	Grammatik notation
ANTLR4	LL(*) (AdaptiveLL(*))	C#, Java, Python, JavaScript, C++, Swift, Go	EBNF
GOLD	LALR(1)	x86 assembly, ANSI C, C#, D, Java, Pascal, Object Pascal, Python, VB6, VB.NET, Visual C++	BNF
YACC	LALR(1)	ANSI C, C++	BNF
SableCC	LALR(1)	C, C++, Java, OCaml, Python	Egen notation

Vi ønsker at benytte et værktøj, hvis grammatiske notation ligger så tæt på EBNF som muligt. Vi ønsker også at holde action code til et absolut minimum - altså programdefinerende kode i grammatiken. ANTLR4 benytter ingen action code i dets grammatik, og derved separere grammatik og action code. I stedet giver man grammatikelementer labels, således at action code kan håndteres seperat fra grammatikken. Alle værktøjer benytter notation som ligger tæt op ad BNF, undtagen SableCC, som benytter sig af en meget modificeret notation. Da vi ikke selv skal skrive parseren, er det knapt så relevant for os at kigge på hvilken type algoritme værktøjerne benytter. Men vi ser at ANTLR (per version 4), benytter en speciel form for LL(*), nemlig ALL(*) [32]. Hvor det bliver interessant, er værktøjernes forskellige sprog - altså hvilke sprog som de formulerer deres output i. Vi programmeringserfaring med C# og C, og vil derved foretrække at arbejde med disse sprog, el.lign. Alle værktøjerne understøtter enten C#, Java eller C. Et af vores mål med vores sprog, er at det skal være tilgængelig på så mange platforme som muligt. Vi er derfor beredt på at benytte Java, som et alternativ til C#. Og da vi ønsker at benytte et objekt orienteret sprog, er Java et oplagt sprog at benytte. Alt i alt, er værktøjerne meget ens, der er ikke meget i specifikationerne som adskiller dem, eller brugen af dem.
Men, vi ønsker at bruge et moderne, non-legacy, værktøj. Dvs. et værktøj som er moderne, og stadig vedligeholdt værktøj, med solid dokumentering og en bred brugerbase - således at vi kan drage på disse resurser, skulle vi møde problemer. Det er vores indtryk at ANTLR er det absolut mest udbredte moderne værktøj. ANTLR har god dokumentation, samt en bog, skrevet af udvikleren. Mange af disse værktøjer er meget ens, men vi vurderer at ANTLR er et rigtigt valg her. Vi går derfor videre, med at benytte ANTLR (v4) som vores parsergenerator.

2.2 Specifikation af tokens

I dette afsnit beskriver vi de tokens, som indgår i sproget og hvad de må bestå af. Hertil anvendes regulære udtryk. Terminalerne i tabel 2.2, som ikke er en del af de regulære udtryk bliver brugt i produktionsreglerne i den kontekstfrie gramamtik i afsnit 2.3.

Terminal		Regulært udtryk
letter	$\longrightarrow$	['a'-'z' | 'A'-'Z']
digit	$\longrightarrow$	['0'-'9']
bool	$\longrightarrow$	'true' | 'false'
int	$\longrightarrow$	$(\text{digit})^+$
identifier	$\longrightarrow$	letter $(\text{letter} \mid \text{digit})^*$
dnasymbol	$\longrightarrow$	'A' | 'T' | 'G' | 'C'
rnasymbol	$\longrightarrow$	'A' | 'U' | 'G' | 'C'
dna	$\longrightarrow$	$(\text{dnasymbol})^+$
rna	$\longrightarrow$	$(\text{rnasymbol})^+$
codon	$\longrightarrow$	'('dna',' dna ',' dna')' | '('rna',' rna',' rna')'
protein	$\longrightarrow$	$(\text{aminoacidsingle})^+ \mid (\text{aminoacidtriple})^+$
aminoacidsingle	$\longrightarrow$	'A' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'K' | 'L' |
	$\longrightarrow$	'M' | 'N' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'V' | 'W' | 'Y'
aminoacidtriple	$\longrightarrow$	'Ala' | 'Cys' | 'Asp' | 'Glu' | 'Phe' | 'Gly' | 'His' | 'Ile' | 'Lys' | 'Leu' |
	$\longrightarrow$	'Met' | 'Asn' | 'Pro' | 'Gln' | 'Arg' | 'Ser' | 'Thr' | 'Val' | 'Trp' | 'Tyr'

Tabel 2.2: Tabel bestående af terminaler og regulære udtryk

2.3 Kontekst-fri grammatik

I dette afsnit vil vi definere den kontekstfrie grammatik for vores programmeringssprog. Den kontekstfrie grammatik er angivet på udvidet Backus-Naur form (EBNF). Grammatikken der er givet i dette afsnit er givet på et format der skulle være læsbart, og nemt for en læser at forstå. I tabel 2.3 nedenfor er en liste af notationerne der bliver brugt i EBNF grammatikken. Den kontekstfrie grammatik på EBNF kan ses i listing 2.1 efter tabellen.

Notation	Brug
:	Definition
|	Alternativ
;	Terminering
’...’	Terminal streng
...*	Repitation
[...]	Optionel
(...)	Gruppering

Tabel 2.3: Notationer i EBNF

I den følgende kontekstfrie grammatik bliver der gjort brug af vores token specifikation fra afsnit 2.2 til at angive hvilke symboler der er lovlige at fylde non-terminalerne i grammatikken ud med.

```
prog
        : declarations* functions* statements* EOF
        ;
declarations
        : declaration ';'
        ;
declaration
        : type assignment
        ;

type
        : 'dna' | 'rna' | 'codon' | 'protein' | 'int' | 'bool'
        ;
statements
        : statement
        ;
block
        : declarations* statements*
        ;
statement
        : assignment ';' | compoundstatement | expression ';'
        | printstatement
        ;
compoundstatement
        : iteration | selection | jump
        ;
selection
        : 'if' '('expression')' '{' block '}'
        | 'if' '('expression')' '{' block '}' 'else' '{' block '}'
        ;
iteration
```

```
32          : 'while' '('expression')' '{' block '}'
33          | 'for' '('assignment ';' expression ';' assignment')' '{'
                block '}'
34          ;
35  functions
36          : functiondeclaration
37          ;
38  functiondeclaration
39          : type identifier '(' declaration (',' declaration)* ')' '{'
                block '}'
40          ;
41  assignment
42          : identifier '=' expression
43          ;
44  expression
45          : '(' expression ')'
46          | expression 'as' type
47          | '!' expression
48          | expression 'contains' expression
49          | ('position of' | 'count' ) expression 'in' expression
50          | 'remove' expression 'from' expression
51          | expression ('*' | '/' | '%') expression
52          | expression ('+' | '-') expression
53          | expression ('<' | '>' | '<=' | '>=') expression
54          | expression ('==' | '!=') expression
55          | expression '&&' expression
56          | expression '||' expression
57          | functioncall
58          | expression '['expression']'
59          | identifier
60          | dna | rna | codon | protein | int | bool
61          ;
62  functioncall
63          : ('comp:' | 'rev:' | 'len:' ) '(' expression ')'
64          | identifier '(' (expression) (',' (expression))* ')'
65          ;
66  jump
67          : 'break' ';' | 'return' expression ';'
68          ;
69  printstatement
70          :'Print' '(' expression ')' ';'
71          |'Print' '(' expression ',' expression ')' ';'
72          |'Print' '(' expression ',' expression ',' expression ')' ';'
73          ;
```

Listing 2.1: Grammatik for vores sprog på EBNF

Det er tydeligt at den ovenstående grammatik for vores sprog er tvetydig og er både dirikte-venstre- og højre-rekursiv. Dette er specielt tydeligt i produktionsreglen for

udtryk, `expression`. I forhold til tvetydigheden bliver denne fjernet ved brug af precedence og associativitet for de forskellige operatorer i vores sprog, dette er givet i afsnit 2.3.1. I forhold til den dirikte-venste- og højre-rekursion bliver dette håndteret af vores valgte parserværktøj: ANTLR.

2.3.1 Precedence for operatorer

For bedre at kunne forstå den kontekstfrie grammatik for vores sprog har vi brug for en liste der angiver precedence for de forskellige operatorer i vores sprog. Operatorer med den højeste precedence er i toppen af listen og operatorer med den laveste precedence er i bunden. Dette medfører at den parser vi laver vil splitte udtryk op ved den laveste precedence, som det første og efterfølgende de højere niveauer af precedence. Listen for precedence kan ses i tabel 2.4.

Precedence	**Operator**	**Beskrivelse**	Associativitet
1	() comp:() len:() rev:() contains position of in count in remove from as []	Udtryk indkapslet i parenteser, funktionskald Komplementær sekvens Længde af sekvens og omvendt sekvens Test af om delsekvens er i sekvens Position af delsekvens i sekvens Antallet af delsekvenser i sekvenser Fjerne delsekvens fra sekvens Typeskift Hente delsekvens af sekvens	left to right
2	!	Logisk negation	right to left
3	* / %	Multiplikation, division og modulo	left to right
4	+ - +	Addition og subtraktion Addition af sekvenser	left to right
5	<><= >=	Relation	left to right
6	== !=	Lighed	left to right
7	&&	Betinget AND	left to right
8	||	Betinget OR	left to right
9	=	Tilskrivning	right to left

Tabel 2.4: Precedence liste for operatorer i vores sprog

Når et udtryk bliver parset og det bruger flere operatorer, vil den operator der har det laveste precedence niveau blive bundet til udtrykket med en højere prioritet, ligesom hvis der blev brugt parenteser. Eksempelvis vil udtrykket $a + b * c$ blive parset som $a + (b * c)$ frem for $(a + b) * c$. Hvis der er et udtryk med flere operatorer der har samme precedence vil udtrykket blive evalueret i en given retning, denne retning er for hver niveau af precedence givet under kolonnen 'Associativitet'. Heraf kan vi se at

udtrykket $a * b * c$ vil blive parset som $(a * b) * c$, og udtrykket $a = b = c$, vil blive parset som $a = (b = c)$.

2.4 Eksempel på programmer

For at få en ide om, hvordan programmer i vores sprog kunne se ud, vil vi i dette afsnit præsenterer nogle programmer i vores, indtil videre, hypotetiske sprog. Med andre ord vil disse programeksempler være eksempler på hvordan vi ønsker at vores sprog skal se ud når der er implementeret.

```
dna minDnaSekvens = ATGGTC;

//transskription
rna minRnaSekvens = minDnaSekvens as rna;

//oversaettelse
protein mitProtein = minRnaSekvens as protein;

Print(mitProtein);

> MV
```

Listing 2.2: Program, hvori en DNA sekvens bliver transskriberet, translateret og printet, ved at assigne til nye typer

```
dna seq1 = ATGCCATTGGGTGTTGCCAGTTTGAAAGCGTGGCTAA;
dna seq2 = CATT;
codon cod = (A,C,G);

seq1 = (remove seq2 from seq1) + cod as dna + CGGGCTAA;

if(seq1 contains cod){
    protein proteinSeq = seq1 as protein;
    Print(proteinSeq, 1);
}
else{
    Print(cod);
}

> M R V L P V S T O P K R G S T O P T R A
```

Listing 2.3: Programeksempel på hvor en DNA sekvens kun printes, hvis den indeholder et bestemt codon

2.5 Valg af scoperegler

Scoperegler er noget, som er defineret for et programmeringssprog, og det valg, om hvilke scoperegler vores sprog skal overholde, er noget vi vil diskutere i dette afsnit. Før en diskussion af scoperegler kan give mening, forklares der først hvad blokke er for programmeringssprog.

2.5.1 Blokke

Listing 2.4 er et eksempel på hvordan blokke implementeres i C-baserede sprog[30].

Listing 2.4: Et mindre kodeeksempel, som viser en blok, hvilket angives i C-baserede sprog med { og } [30]

```
1 { int temp;
2   temp = list[upper];
3   list[upper] = list[lower];
4   list[lower] = temp;
5 }
```

En blok indgår i et større program, hvor variabel erklæringer inde i en blok, ikke kan findes i den resterende del af et program. Der kan kun refereres til variablen *temp*, fra listing 2.4, inde i den del, som står indenfor for { og }. Fordelen ved at have blokke er, at man kan genbruge variabel navne forskellige steder i programmet[30].

2.5.1.1 Syntaks for blokke

I den Kontekst-frie grammatik har vi defineret, at blokke starter med { og slutter med }, hvilket svarer til blokke i C-baserede sprog. Alternative måder at indikere en ny blok på, kunne være som i Python, hvor indrykning svarer til at bevæge sig ind i en ny blok. I forhold til målgruppen, har vi vurderet, at den bedste tilgang var at være eksplicit omkring erkæring af nye blokke, og derfor har vi valgt den C-baserede tilgang 2.3.

2.5.2 Forskellen på statiske og dynamiske scoperegler

Figur 2.1 illustrer de forskellige scoperegler.

Static scoping

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

Dynamic scoping

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

Figur 2.1: Et eksempel på, hvad retur værdien er for to metoder, som umiddelbart er ens, men hvor den ene metode har statiske scoperegler, hvor den anden har dynamiske scoperegler [33].

Grunden til at *bar()* returnerer 10 i static scoping, er fordi at vi husker hvilken værdi b havde på det tidspunkt, hvor *bar()* blev erklæret, hvilket var $b = 5$. Med dynamiske scoperegler, så returnerer *bar()* værdien 7, fordi her finder vi den b, som vi kender når *bar()* bliver kaldt, og det er $b = 2$.

Evaluering af statiske scoperegler

Statiske scoperegler giver programmøren en let måde at anvende ikke-lokale variabler, som fungerer i mange situationer. Dette skyldes at ved statiske scoperegler, anvendes de bindinger der er kendt ved erklæringen af funktionen [34].

Dog er der nogle ulemper ved brugen af statisk scope, da det ofte giver mere adgang til funktioner og variabler, end nødvendigt. Dette skyldes at en funktion kan ændre på andre værdier selvom de ikke er lokale, eller givet som en reference til funktionen. Endvidere hvis et program allerede har en effektiv struktur, og programmet skal udvides, fører det ofte til en større rekonstruering af program strukturen. Dette medfører at programmørerne skal slås med at bibeholde restriktionen af variabler samt funktioner, hvilket kan medføre at strukturen af det originale program ændres fuldstændigt[30].

Evaluering af dynamiske scoperegler

Dynamisk scoping resulterer i mindre pålidelige programmer end statisk scoping, da der er ingen måde at beskytte lokale variabler for at være tilgængelig til resten af programmet. Endvidere er programmer med dynamisk scope meget svære at læse, fordi at en reference til en ikke-lokal variabel, afhænger af kald-sekvensen, som kan være svær at bestemme før runtime [30].

Valgte scoperegler

Til vores programmeringssprog, har vi valgt at anvende fuld statiske scoperegler, da vi ser at der er langt flere fordele, i det at det er nemmere at anvende og læse, mere pålideligt samt at det eksekvere hurtigere([30] side 246). Endvidere gør sproget R, som vi ved målgruppen bruger fra afsnit 1.4.7, brug af statiske scope regler.

2.6 Valg af parametermekanisme

Vi vil i dette afsnit vælge den parametermekanisme, som vores sprog skal gøre brug af. For at træffe dette valg vil vi her betragte to former for parametermekanismer. Disse er henholdsvis *pass-by-reference* og *pass-by-value*, og efterfølgende vælge en af disse.

Pass-by-reference

En reference indeholder ikke noget data, men indeholder derimod en reference til en variabels data. Med disse referencer, kan flere referencer referere til det samme data eller objekt. Dette medfører, at hvis der udføres en operation på en variabel, kan resultatet aflæses på de andre referencer, da det er samme data de alle refererer til. Ved at anvende pass-by-reference, opnås en større effektivitet af programmet, da der ikke skal laves en kopi af argumenternes data[30].

En negativ side ved pass-by-reference, er at det kræver et større overblik fra programmørens side, da han skal holde styr på hvilke data ændrer sig og hvornår.

Pass-by-value

Pass-by-value, foregår ved at der bliver lavet en kopi af variablens data, som så bliver gjort tilgængelig for funktionen. Denne data, i modsætningen til når pass-by-reference bliver anvendt, eksisterer kun inde i scopet for funktionen, og mistes når funktionen er færdig med at eksekvere[30]. Det er derfor vigtigt at, hvis funktionen skal ændre værdien som blev givet til funktionen, så skal den blive returneret.

Pass-by-value er ofte mere sikker end pass-by-reference, fordi sideeffekter ikke forekommer, når man anvender pass-by-reference. Dette resulterer i et lettere sprog, da programmøren ikke skal bekymre sig om de variabler der bliver givet til en funktion[30], hvilket er grunden til at vi vælger pass-by-value parametermekanismen for vores sprog.

2.7 Target sprog

Vi har valgt at oversætte til Java kode, fremfor JVM bytecode, da dette tillader os at implementere de metoder, der svarer til big-step transitionsreglerne fra tabel 3.3, på en mere overskuelig måde end hvis det skulle gøres i JVM bytecode. Endnu en fordel ved Java, er at dets runtime er crossplatform, dvs. at Java programmer kan køre på en lang række systemer, uden at tage højde for system- eller maskin-specifikke detaljer. Java kan derfor anvendes både på Windows, MacOS samt Linux, udenfra samme kode. Dette gør at brugere kan benytte vores sprog, på flere platforme. Grunden til dette er en fordel, er at målgruppen benytter sig af Python, bl.a. i form af BioPython som blev beskrevet i afsnit 1.4.7, og en af grundene til at Python var et attraktivt sprog, er fordi det kan anvendes på alle platforme [35].

Kapitel 3

Strukturel operationel semantik og typesystem

Med indblik i vores sprogs syntaks fra den kontekst-frie grammatik, samt viden omkring vores scoperegler og parametermekanismer fra kapitel 2 er næste skridt at definere hvad de forskellige udtryk i vores sprog egentlig betyder gennem en strukturel operationel semantik. Hertil vil vi også give et typesystem, der definerer, hvilke typer der er lovlige at bruge, i de forskellige konstruktioner i vores sprog. Dette kapitel vil derfor omhandle den operationelle semantik og typesystemet for vores programmeringssprog. I afsnit 3.1 vil vi præsentere den abstrakte syntaks for sproget hvis syntaks vil blive brugt i de efterfølgende afsnit. Vi vil i afsnit 3.2 præsentere den strukturelle operationelle semantik for vores sprog, med fokus på opbygningsregelerne, der bearbejder bio udtryk, som præsenteret i den abstrakte syntaks. Da vores sprog gør brug af typer bliver vi nødt til at have et typesystem. Reglerne for dette vil blive gennemgået i afsnit 3.3. Sproget vil også, som nævnt i løsningskriterierne i afsnit 1.8, understøtte konvertering mellem udvalgte bio typer, disse konverteringer vil der blive opstillet typeregler for i afsnit 3.3.4.

3.1 Abstrakt syntaks

I dette afsnit vil vi præsentere den abstrakte syntaks for vores programmeringssprog. Formålet med den abstrakte syntaks er at give en formel definition af vores programmeringssprog hvorigennem der kan laves semantik for sproget og endvidere opstilles programeksempler. Den abstrakte syntaks for vores sprog vil følge standarderne og konventionerne, som angivet i bogen *Transitions and Trees* [34]. De følgende metavariabler bruges i opbygningsreglerne for den abstrakte syntaks:

$S \in$ **Kom** - Kommandoer
$e \in$ **Udtr** - Udtryk
$T \in$ **Typer** - Typer

$x \in$ **Var** - Variabler
$n \in$ **Num** - Numeraler
$g \in$ **Seq** - Mængde af lovlige sekvenser
$f \in$ **Fnavne** - Funktionsnavne
$c \in$ **Blok** - Blokke
$D_v \in$ **ErkV** - Variabelerklæringer
$D_f \in$ **ErkF** - Funktionserklæringer

Element, g, er et element i en mængde af lovlige sekvenser af DNA, RNA, proteiner og codon, de lovlige sekvenser kan være sekvenser af typerne der bliver præsenteret i typesystemet i afsnit 3.3. Opbygningsregelerne der udgør den abstrakte syntaks for vores sprog kan ses herunder. I disse opbygningsregler optræder $\overrightarrow{T\ x}$ og $\overrightarrow{e}$ disse bruges i dette tilfælde som notation for henholdsvis en mængde af formelle og aktuelle parametre til funktioner, disse mængder kan også være tomme og skal forstås som følgende:

$$\overrightarrow{e} \quad = \quad e_1,\ e_2,\ \ldots e_n \mid \varepsilon$$

$$\overrightarrow{T\ x} \quad = \quad T_1\ x_1,\ T_2\ x_2,\ \ldots T_n\ x_n \mid \varepsilon$$

I den abstrakte syntaks bliver metavariablen T også brugt, vi definerer elementerne i denne kategori i afsnit 3.3 Hver opbygningsregel er nummereret og der vil efter den abstrakte syntaks være en uformel semantik på dansk for de interessante dele af opbygningsreglerne i den abstrakte syntaks:

$prog$	::=	$D_v\ D_f\ S$	(1)
S	::=	$x = e;$ | $S_1 S_2$	(2)
	|	if $(e)\ c_1$ else c_2	(3)
	|	while (e) do c	(4)
	|	for $(x = e_1; e_2; x = e_3)\ c$	(5)
	|	Print(e);	(6)
	|	Print(e_1, e_2);	(7)
	|	Print(e_1, e_2, e_3);	(8)
	|	break;	(9)
	|	return e;	(10)
e	::=	n | g | x | e_1 + e_2 | e_1 - e_2 | $e_1 * e_2$ | $e_1 \div e_2$ | (e)	(11)
	|	e_1 == e_2 | $e_1 < e_2$ | $!e$ | e_1 && e_2 | $e_1 \mid\mid e_2$	(12)
	|	e as T | comp:(e) | rev:(e)	(13)
	|	len:(e)	(14)
	|	position of e_1 in e_2	(15)
	|	count e_1 in e_2	(16)
	|	e_1 contains e_2	(17)
	|	remove e_1 from e_2	(18)
	|	$e_1[e_2]$	(19)
	|	$f(\overrightarrow{e})$	(20)
c	::=	$\{D_v\ S\}$	(21)
D_v	::=	$T\ x$ = e; D_v | ε	(22)
D_f	::=	$T\ f(\overrightarrow{T\ x})\ c\ D_f$ | ε	(23)

Tabel 3.1: Abstrakt syntaks for vores sprog

Beskrivelse af abstrakt syntaks

Gennem den følgende tekst vil vi beskrive de interessante dele af den abstrakte syntaks for at give et bedre indblik i hvad de forskellige elementer i opbygningsreglerne betyder. Dette kan således anses for at være en uformel semantik for de interessante elementer. Beskrivelsen er som følger:

1 Et programs opbygning. Et program består af variabelerklæringer efterfulgt af funktionsdefinitioner og til sidst kommandoer.

6 Print funktion, der printer værdien af et udtryk e.

7 Denne print funktion har en sekundær parameter, som angiver et mellemrum i den udskrevne sekvens for hver gang der har været en følge af længden svarende til e_2 i det udskrevne. Eksempelvis vil der være mellemrum mellem hver tredje symbol hvis e_2 er har værdien tre. Således kan en sekvens printes ud i enkelte Codon.

8 En tredje printfunktion, som har endnu en ekstra parameter. Den tredje parameter e_3 bestemmer om start- og stop-codon i det udskrevne skal være fremhævet, dette bliver gjort ud fra en bool.

11 Plus udtrykket har forskellig opførsel alt efter hvilke typer der indgår i udtrykket. Mere om dette vil følge i afsnit 3.2 om sprogets semantik.

13 Udtryk til brug på bio typerne, her er mulighed for at konvertere fra en bio type til en anden, gennem et udtryk, samt mulighed for at finde komplementær sekvenser og omvendte sekvenser.

14 Aritmetisk Udtryk der finder længden af e.

15 Aritmetisk Udtryk der finder den første forekomst af sekvens e_1 i sekvens e_2 og giver start positionen for sekvens e_1, dette kan således siges at være en implementation af KMP-algoritmen [27] i sproget.

16 Aritmetisk udtryk der tæller forekomster af sekvens e_1 i sekvens e_2.

17 Boolsk udtryk der giver en sandhedsværdi ud fra om e_1 indeholder e_2.

18 Bio Udtryk der fjerner alle forekomster af sekvens e_1 fra sekvens e_2 hvis der er nogle.

19 Bio udtryk der giver det symbol fra sekvens e_1 der er på plads e_2.

20 Funktionskald med en mængde af aktuelle parametre.

21 Blokke med nyt scope, disse består af variabelerklæringer efterfulgt af kommandoer.

23 Funktionserklæringer, her angives typen af funktionen samt dens navn, formelle parametre og krop.

Sammenhæng mellem løsningskriterier og abstrakt syntaks

I dette underafsnit vil vi beskrive sammenhængen mellem vores løsningskriterier og de forskellige elementer i opbygningsreglerne for den abstrakte syntaks. Vi kan se ud fra vores løsningskriterier i afsnit 1.8 at vi skal have typer i form af DNA, RNA, codon, proteiner, int og bool, som vi kan skelne mellem. I vores abstrakte syntaks betegnes

disse typer med metavariablen T og det fremgår gennem den abstrakte syntaks at vi ønsker at det skal være muligt at erklære variabler og funktioner af disse typer. De basale sekvensoperationer, der omtales i afsnittet om løsningskriterier, svarer i den abstrakte syntaks til elementer i opbygningsreglen for udtryk e. Her ser vi at for konkatenation af sekvenser har vi udtrykket $e_1 + e_2$, for at fjerne delsekvenser fra sekvenser er der `remove` e_1 `from` e_2, for at finde komplementære- og omvendte-sekvenser er der udtrykkene `comp`(e) og `rev`(e), samt flere udtryk der repræsenterer sekvens operationer. I forhold til funktionalitet fra imperative sprog har sproget, som det kan ses ud fra opbygningsreglen for kommandoer S, kontrolstrukturer som for- og while-løkker, og if-else.

3.2 Semantikken for sproget

I dette afsnit vil vi, ud fra opbygningsreglerne fra den abstrakte syntaks, give en strukturel operationel semantik i form af en *big-step-semantik*. I en *big-step-semantik* vil en transition fra $\gamma \rightarrow \gamma'$ beskrive hele beregningen, som starter i γ, hvor γ' altid vil være slutkonfigurationen. I kontrast til en *big-step-semantik* har vi en *small-step-semantik*, her vil transitionen $\gamma \rightarrow \gamma'$ beskrive et enkelt skridt i en beregning, dette betyder at γ' ikke nødvendigvis behøver at være en slutkonfiguration. Grunden til at det er blevet valgt at lave en *big-step-semmantik* for sproget er fordi det ikke indeholder nogen parallelitet, som en *small-step-semantik* er god til at beskrive, og derfor er en *small-step-semantik* ikke nødvendig. Når '*big-step-semmantik*' efterfølgende bruges i transitionsreglerne, vil det blive refereret til som *BSS*.

Brug af strenge

I transitionsreglerne for udtryk i dette afsnit bliver der gjort brug af strenge, vi vil i dette underafsnit præsentere strenge, som de bliver opfattet i vores semantik. Her vil vi introducere formen af strenge, hvad de består af, samt de operationer der er mulige at lave på dem.

Når strenge optræder i transitionsregler betegner vi dem med w. En streng er opbygget af et eller flere elementer, disse elementer betegner vi med a således kan opbygningen af en streng defineres på følgende måde:

$$w = a_1 a_2 \dots a_n$$

Med denne definition af en streng er det nu muligt at angive hvilke symboler en streng kan bestå af, hertil kan vi sige at en streng kan bestå af symbolerne der er givet i alfabetet Σ, hvor Σ er en endelig mængde defineret, som følgende:

$$\Sigma = \{A, T, C, G, U, D, E, F, H, I, K, L, M, N, P, Q, R, S, V, W, Y\}$$

For at en streng er gyldig skal det gælde at $w \in \Sigma^*$. Her er Σ^* repræsentationen for mængden af alle strenge over alfabetet Σ. De symboler som indgår i alfabetet Σ, er de samme symboler som er tilladt i DNA og RNA, der som bekendt er: A, T, C, G og U, samt alle symboler, som et protein kan bestå af, som kan findes i afsnit 1.4.2.

I forhold til operationer på strenge gør vi i transitionsreglerne brug af nogle af dem, som angivet i bogen *Introduction to the Theory of Computation* [36] samt flere. Her vil der specifikt blive gjort brug af de følgende operationer:

$w_1 \circ w_2$	w^R	$\lvert w \rvert$	$w[i, j]$
(a)	**(b)**	**(c)**	**(d)**

$$w_1 \backslash w_2$$

(e)

Tabel 3.2: Operationer på strenge

Det ovenstående udtryk i tabel 3.2a gør brug af $\circ$ operatoren, som bruges til at angive konkatenation af strenge således at $w_1 \circ w_2 = w_1 w_2$, altså at w_1 konkateneret med w_2 giver strengen w_1 efterfulgt af w_2. Udtrykket i tabel 3.2b gør brug af R operatoren som angiver den omvendte streng, i dette tilfælde er det den omvendte streng w, således at strengen $w^R = a_n a_{n-1} \cdots a_1$. Udtrykket i tabel 3.2c angiver længden af strengen w og er således det samme som antallet af symboler, som w indeholder. Udtrykket i tabel 3.2d betegner en delstreng af w, denne delstreng er delstrengen der går fra w_i til og med w_j. Udtrykket i tabel 3.2e anvender vi operatoren $\backslash$ på to operander, som begge er strenge og skal læses som w_1 uden forekomster af delstreng w_2. Definitionen af hvad operatoren udtrykker er følgende:

$$\begin{aligned} &Lad\ w, w_n \in \Sigma^*\ hvor\ n \in \mathbb{Z}^+ \\ &w \quad\ = w_1 w_2 w_3 \cdots \\ &w \backslash w_2 = w_1 w_3 \cdots \end{aligned}$$

Environment-store-modellen

I transitionsreglerne i dette afsnit, vil der blive gjort brug af environment-store-modellen, som beskrevet i [34]. Denne model beskriver lageradresserne, som variabler og funktioner, er bundet til og værdierne der er på de forskellige lageradresser. Dette gøres gennem et *variabel-environment* for variabler, et *funktions-environment* for funktioner, og et *store*, der beskriver værdier, som ligger på de enkelte lageradresser. Mængden af variabel-environments er defineret som følger:

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

I modellen betegner vi lageradresser, som *lokationer* hvor mængden af mulige lokationer betegnes med **Loc** hvoraf et element, altså en lokation, betegnes l. I ovenstående angiver $next$ det næste ledige element i **Loc**. Her lader vi env_V betegne et element i $\mathbf{EnvV}$. Mængden af stores er defineret, som følger:

$$\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbb{Z} \cup \{tt,\ ff\} \cup \Sigma^*$$

Herfra ser vi at værdierne som stores kan være heltal, men også sandhedsværdier og strenge over alfabetet Σ. Her betegner vi et vilkårligt element i $\mathbf{Sto}$, som sto. For funktionsenvironments antager vi helt statiske scoperegler, derfor er mængden af funktionsenvironments defineret, som følger:

$$\mathbf{EnvF} = \mathbf{Fnavne} \rightharpoonup \mathbf{Kom} \times \mathcal{P}(\mathbf{Var}) \times \mathbf{EnvV} \times \mathbf{EnvF}$$

Vi lader her env_f betegne et element i $\mathbf{EnvF}$.

3.2.1 Big-step-semantik for udtryk

I dette underafsnit vil vi give en big-step-semantik for udtryk i sproget. Hertil vil vi definere transitionssystemet, hvor hver transition svarer til en hel beregning i semantikken. Dette betyder at transitioner går fra udtryk til deres værdier. Transitionssystemet for udtryk er en udvidelse af det der kan findes i [34] og er følgende:

$$((\mathbf{Udtr} \times \mathbf{Sto}) \cup \mathbf{Udtr} \cup \mathbb{Z} \cup \{tt,\ ff\} \cup \Sigma^* \cup \mathbf{Sto}, \rightarrow_e, \mathbb{Z} \cup \{tt,\ ff\} \cup \Sigma^* \cup \mathbf{Sto})$$

Transitionsrelationen '$\rightarrow_e$' er defineret ved de efterfølgende transitionsregler. I de viste regler er en samling af bio udtryk, boolske udtryk og aritmetiske udtryk. Der er dog boolske udtryk og aritmetiske udtryk der ikke er vist i de følgende transitionsregler, men som stadig er med til at definere transitionsrelationen. Transitionsregler for de manglende operationer kan findes i [34].

I transitionsreglerne der gives i dette afsnit vil der blive gjort brug af nogle semantiske funktioner. Den første af disse funktioner er G, som kan ses i nedenfor:

$$G : \mathbf{Seq} \rightarrow \Sigma^*$$

Funktionen tager en parameter g og returnerer den tilsvarende streng $w \in \Sigma^*$ Funktionen G bliver brugt i transitionsreglen $[Seq_{BSS}]$, denne transitionsregel kan også betegnes, som et aksiom da den ikke har nogle præmisser. Endvidere har vi også brug for endnu en funktion:

$$C : \Sigma^* \rightarrow \Sigma^*$$

Enhver streng w bestående af symbolerne i Σ^* giver en streng w', som svarer til den komplementære sekvens af w. Her ved vi fra afsnit 1.4.1 at om komplementære sekvenser gælder det:

$$A \to T,\ T \to A,\ G \to C,\ C \to G$$

Heraf kan vi også se at det kun er muligt at finde den komplementære sekvens for en DNA-sekvens. Funktionen C bliver brugt i transitionsreglen $[Comp_{BSS}]$. Der findes også en semantisk funktion $\mathcal{N}$ der for ethvert numeral $\mathbf{Num}$ giver det tilsvarende heltal, definitionen af denne er som følger:

$$\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$$

Denne funktion er fra bogen [34]. Den semantiske funktion $\mathcal{T}$ bliver brugt i transitionsreglen for $[as_{exp}]$ og er defineret på følgende måde:

$$\mathcal{T} : \Sigma^* \to \Sigma^*$$

Funktionen $\mathcal{T}$ erstatter symbolerne i en streng w med de symboler, som svarer til dem der er gyldige i typen, som strengen w bliver konverteret til. Vi vil her også nævne at transitionsreglen $[Call_{bss}]$ er fra bogen [34].

$[Concat_{BSS}]$

$$\frac{env_V, sto \vdash e_1 \to_e w_1 \quad env_V, sto \vdash e_2 \to_e w_2}{env_V, sto \vdash e_1 + e_2 \to_e w}$$

$\texttt{hvor} \quad w = w_1 \circ w_2$

$[As_{BSS}]$

$$\frac{env_V, sto \vdash e \to_e w_1}{env_V, sto \vdash e\ \texttt{as}\ T \to_e w_2}$$

$\texttt{hvor} \quad w_2 = \mathcal{T}[\![w_1]\!]$

$[Comp_{BSS}]$

$$\frac{env_V, sto \vdash e \to_e w_1}{env_V, sto \vdash \texttt{comp}\ :(e) \to_e w_2}$$

$\texttt{hvor} \quad w_2 = C[\![w_1]\!]$

$[Rev_{BSS}]$

$$\frac{env_V, sto \vdash e \to_e w_1}{env_V, sto \vdash \texttt{rev}\ :(e) \to_e w_2}$$

$\texttt{hvor} \quad w_2 = w_1^R$

$[Len_{BSS}]$

$$\frac{env_V, sto \vdash e \to_e w}{env_V, sto \vdash \texttt{len}\ :(e) \to v}$$

$\texttt{hvor} \quad v = |w|$

$[Pos_{BSS}]$

$$\frac{env_V, sto \vdash e_1 \to_e w_1 \quad env_V, sto \vdash e_2 \to_e w_2}{env_V, sto \vdash \texttt{position of}\ e_1\ \texttt{in}\ e_2 \to_e v}$$

$$\texttt{hvor} \quad v = i$$

$$\texttt{hvis} \quad \exists i.j.i \leq j \leq |w_2|.w_2[i,j] = w_1$$

$$\texttt{og} \quad env_V, sto \vdash e_1 \; contains \; e_2 \rightarrow_e tt$$

$$[Count_{BSS}] \quad \frac{env_V, sto \vdash e_1 \rightarrow_e w_1 \quad env_V, sto \vdash e_2 \rightarrow_e w_2}{env_V, sto \vdash \texttt{count}\; e_1 \;\texttt{in}\; e_2 \rightarrow_e v}$$

$$\texttt{hvor} \quad v = 1 + (env_V, sto \vdash \texttt{count}\; e_1 \;\texttt{in}\; e_2[j, |w_2|])$$

$$\texttt{hvis} \quad \exists i.j.i \leq j \leq |w_2|.w_2[i,j] = w_1$$

$$\texttt{og} \quad env_V, sto \vdash e_1 \; contains \; e_2 \rightarrow_e tt$$

$$[Contain{-}tt_{BSS}] \quad \frac{env_V, sto \vdash e_1 \rightarrow_e w_1 \quad env_V, sto \vdash e_2 \rightarrow_e w_2}{env_V, sto \vdash e_1 \;\texttt{contains}\; e_2 \rightarrow_e tt}$$

$$\texttt{hvor} \quad \exists i.j.w_2 = w_1[i,j]$$

$$\texttt{hvis} \quad i \leq j \leq |w_1|$$

$$[Contain{-}ff_{BSS}] \quad env_V, sto \vdash e_1 \;\texttt{contains}\; e_2 \rightarrow_e ff$$

$$\texttt{hvis} \quad env_V, sto \vdash \neg(e_1 \; contains \; e_2) \rightarrow_e tt$$

$$[Remove_{BSS}] \quad \frac{env_V, sto \vdash e_1 \rightarrow_e w_1 \quad env_V, sto \vdash e_2 \rightarrow_e w_2}{env_V, sto \vdash \texttt{remove}\; e_1 \;\texttt{from}\; e_2 \rightarrow_e w}$$

$$\texttt{hvor} \quad w = w_2 \backslash w_1$$

$$[Get_{BSS}] \quad \frac{env_V, sto \vdash e_1 \rightarrow_e w \quad env_V, sto \vdash e_2 \rightarrow_e v}{env_V, sto \vdash e_1[e_2] \rightarrow_e w'}$$

$$\texttt{hvor} \quad w' = w[v, |w|]$$

$$[Call_{BSS}] \quad \frac{env'_V[x \mapsto l][next \mapsto new\; l], env'_f \vdash \langle S, sto[l \mapsto v]\rangle \rightarrow_e sto'}{env_V, env_f \vdash \langle f(\overrightarrow{e}), sto\rangle \rightarrow_e sto'}$$

$$\texttt{hvor} \quad env_f \; f = (S, x, env'_V, env'_f), env_V, sto \vdash \overrightarrow{e} \rightarrow_e v$$

$$\texttt{og} \quad l = env_V \; next$$

$$[Seq_{BSS}] \quad env_V, sto \vdash g \rightarrow_e w$$

$$\texttt{hvis} \quad w = G[\![g]\!]$$

$$[Call_{BSS}] \quad \frac{\begin{array}{c} env'_V[x_1 \mapsto l_1][next \mapsto new\; l_1][x_2 \mapsto l_2][next \mapsto new\; l_2][\cdots] \\ [x_n \mapsto l_n][next \mapsto new\; l_n] \\ env'_f \vdash \langle S, sto[l_1 \mapsto v_1][l_2 \mapsto v_2][\cdots][l_n \mapsto v_n]\rangle \rightarrow_e sto' \end{array}}{env_V, env_f \vdash \langle f(\overrightarrow{e}), sto\rangle \rightarrow_e sto'}$$

$$\texttt{hvor} \quad env_f \; f = (S, \overrightarrow{T\; x}, env'_V, env'_f), env_V, sto \vdash \overrightarrow{e} \rightarrow_e v_1, v_2 \cdots v_n$$

$$\texttt{og} \quad l_1 = env_V \; next, l_2 = env_V \; next, \cdots l_n = env_V \; next$$

$$\texttt{hvor} \quad n = |\overrightarrow{e}|$$

Tabel 3.3: Big-step-transitionsregler for **Udtr**

3.2.2 Big-step-semantik for blokke

I dette underafsnit vil vi give en big-step-semantik for blokke i vores sprog. Transitionssystemet for blokke er følgende:

$$((\mathbf{Blok} \times \mathbf{Sto}) \cup \mathbf{Sto}, \rightarrow_{BL}, \mathbf{Sto})$$

Transitionsrelationen '$\rightarrow_{BL}$' er defineret ud fra den følgende transitionsregl, som er en modifikation af transitionsreglen fra bogen [34]:

$$[Blok_{BSS}] \quad \frac{\langle D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto'') \quad env'_V, env_f \vdash \langle S, sto''\rangle \rightarrow_{kom} sto'}{env_V, env_f \vdash \langle \{D_V, S\}, sto\rangle \rightarrow_{BL} sto'}$$

Tabel 3.4: Big-step-transitionsregl for **Blok**

3.2.3 Big-step-semantik for erklæringer

I dette underafsnit vil vi give en big-step-semantik for erklæringer af variabler og funktioner i vores sprog. Transitionssystemet for henholdsvis variabelerklæringer og funktionserklæringer er følgende:

$$((\mathbf{ErkV} \times \mathbf{EnvV} \times \mathbf{Sto}) \cup \mathbf{EnvV} \times \mathbf{Sto}, \rightarrow_{DV}, \mathbf{EnvV} \times \mathbf{Sto})$$
$$((\mathbf{ErkF} \times \mathbf{EnvF}) \cup \mathbf{EnvF}, \rightarrow_{DF}, \mathbf{EnvF})$$

De ovenstående transitionssystemer er fra bogen [34] og gør sig også gældende for vores sprog. Transitionsrelationerne '$\rightarrow_{DV}$' og '$\rightarrow_{DF}$' er defineret gennem de følgende transitionsregler:

$$[Var{-}erkl_{BSS}] \quad \frac{\langle D_v, env_V[x \mapsto l][next \mapsto new\ l], sto[l \mapsto v]\rangle \rightarrow_{DV} (env'_V, sto')}{\langle T\ x = e; D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')}$$

$$\texttt{hvor} \quad env_V, sto \vdash e \rightarrow_e v$$

$$\texttt{og} \quad l = env_V\ next$$

$[Tom{-}Var{-}erkl_{BSS}]$ $\quad \langle \varepsilon, env_V, sto\rangle \rightarrow_{DV} (env_V, sto)$

$[Funk{-}erkl_{BSS}]$ $\quad \dfrac{env_V \vdash \langle D_f, env_f[f \mapsto (c, env_V, env_f)]\rangle \rightarrow_{DF} env_f}{env_V \vdash \langle T\ f(\overrightarrow{T\ x})\ c; D_f, env_f\rangle \rightarrow_{DP} env'_f}$

$[Funk{-}erkl_{BSS}]$ $\quad \dfrac{env_V \vdash \langle D_f, env_f[f \mapsto (c, \overrightarrow{T\ x}, env_V, env_f)]\rangle \rightarrow_{DF} env'_f}{env_V \vdash \langle T\ f(\overrightarrow{T\ x})\ c; D_f, env_f\rangle \rightarrow_{DP} env'_f}$

$[Tom{-}Funk{-}erkl_{BSS}]$ $\quad env_V \vdash \langle \varepsilon, env_f\rangle \rightarrow_{DF} env_f$

Tabel 3.5: Big-step-transitionsregler for **ErkV** og **ErkF**

3.2.4 Big-step-semantik for kommandoer

I dette underafsnit vil vi give en big-step-semantik for kommandoer i vores sprog. I sproget indgår kommandoerne: `Print`(e); `break`; og `return`(e); også, men disse vil der ikke blive givet transitionsregler for. Her vil vi i stedet give en uformel semantik som siger at `Print`(e); udskriver et udtryk, `break`; bryder det nuværende scope og `return`(e) returnerer et udtryk. Når `return`(e) bruges af en funktion vil værdien af udtrykket e være værdien af kaldet af funktionen. Da vi laver en big-step-semantik for kommandoer er transitionssystemet det følgende:

$$((\mathbf{Kom} \times \mathbf{Sto}) \cup \mathbf{Sto}, \rightarrow_{kom}, \mathbf{Sto})$$

Det ovenstående transitionssytem er fra bogen [34] og gør sig også gældende for kommandoer i vores sprog.

Vi ved at en kommando kan resultere i et ændret sto da vi gennem kommandoer har mulighed for at ændre værdierne af eksisterende variabler, dette kan gøres gennem tilskrivninger. En kommando kan derimod ikke ændre i vores variabel-environment env_V da vi gennem kommandoer ikke har mulighed for at lave variabel- og funktions-erklæringer. Transitionsrelationen '$\rightarrow_{kom}$' er defineret ved de efterfølgende transitionsregler. Det bør nævnes at der er kommandoer, som ikke er angivet her, men stadig er med til at definere transitionsrelationen.

I de følgende transitionsregler antager vi også, at der er en funktion $\mathcal{N}^{-1} : \mathbb{Z} \rightarrow \mathbf{Num}$, som for for ethvert heltal giver et tilsvarende numeral, denne funktion er også fra bogen [34].

$$env_V, env_f \vdash \langle c, sto[l \mapsto v]]\rangle \rightarrow_{kom} sto''$$

$$[For{-}tt_{BSS}] \quad \frac{env_V, env_f \vdash \langle \texttt{for}(x = e_4; e_2; x = e_3)\ c, sto'' \rangle \rightarrow_{kom} sto'}{env_V, env_f \vdash \langle \texttt{for}(x = e_1; e_2; x = e_3)\ c, sto \rangle \rightarrow_{kom} sto'}$$

$\texttt{hvor} \quad e_4 = \mathcal{N}^{-1}(sto(env_V\ x))$

$\texttt{og} \quad env_V\ x = l$

$\texttt{hvis} \quad env_V, sto \vdash e_2 \rightarrow_e tt$

$$[For{-}ff_{BSS}] \quad env_V, env_f \vdash \langle \texttt{for}(x = e_1; e_2; x = e_3)\ c, sto \rangle \rightarrow_{kom} sto$$

$\texttt{hvis} \quad env_V, sto \vdash e_2 \rightarrow_e ff$

Tabel 3.6: Big-step-transitionsregler for **Kom**

3.3 Typesystem

En klar fordel ved at have et typesystem er, at man kan bestemme om programmer er veltypede og derigennem kan undgå at bestemte køretidsfejl opstår. Derfor er det blevet valgt at der for sproget skal være et typesystem. Vi vil i dette afsnit beskrive typesystemet for sproget, ved først at give en definition af *mængden af typer* i sproget, samt give en definition for hvordan *typedomme* ser ud for de syntaktiske kategorier i sproget. Efter dette vil der blive givet *typeregler* for de forskellige syntaktiske kategorier, disse regler vil tilsammen definere hvad der er gyldige typedomme. Da konvertering mellem bio typerne udgør en stor del af sproget vil der i afsnit 3.3.4 blive givet typeregler med fokus på konvertering mellem disse typer. Opbygningen af typesystemet og dets regler vil være på den form som er præsenteret i bogen *Transitions and Trees* [34]. Endvidere vil der i dette afsnit være mest fokus på at beskrive den del af typesytemet der bruger bio typerne.

3.3.1 Typer i sproget

For **Typer** har vi opbygningsreglerne som er vist herunder. Som bekendt betegnes elementer i **Typer** med metavariablen T. En delmængde heraf er *basistyperne*, som er delt op i to kategorier, disse betegnes med metavariablerne $B1$, $B2$ og $B3$, og er de typer som er angivet i basistypernes opbygningsregler.

T ::= $B1 \mid B2 \mid B3 \mid B4 \mid f : T_1 \rightarrow T_2 \mid \text{ok}$

$B1$ ::= `Int` | `Bool`

$B2$ ::= $B3$ | `Protein`

$B3$ ::= `Dna` | `Rna` | `Codon`

$B4$ ::= `Dna` | `Rna` | `Protein`

De ovenstående opbygningsregler skal læses som at T kan være en basistype $B1$, $B2$, $B3$ eller $B4$, der er til for at beskrive typen af et udtryk ($B3$ er en delmængde af typerne i $B2$, og er nødvendig da nogle udtryk ikke kan gøre brug af alle typerne i $B2$). T kan også være den sammensatte type $f : T_1 \rightarrow T_2$ der beskriver typen af funktionsnavne og skal forstås, som at hvis de formelle parametre har en type af T_1, som ikke nødvendigvis er samme type, da vil returtypen være T_2. T kan også være typen 'ok', som bliver brugt ved erklæringer, kommandoer og blokke, for at sige at disse ikke indeholder typefejl. Hvordan vi giver typen for udtryk, erklæringer, kommandoer og blokke bliver bestemt gennem typedomme i afsnit 3.3.2 om typedomme.

3.3.2 Typedomme

I dette afsnit vil vi beskrive de typedomme der gør sig gældende for udtryk, erklæringer, kommandoer og blokke, da alle disse kan indeholde variabler har vi brug for et *typeenvironment*, således bliver E betegnelsen for typeenvironment. Nedenfor vil være en definition af det omtalte typeenvironment efterfulgt af typedommene:

En definition af typeenvironment E er det følgende:

$$E : \mathbf{Var} \cup \mathbf{Fnavne} \rightharpoonup \mathbf{Typer}$$

Det ovenstående skal læses således: Typeenvironment E er en partiel funktion fra mængden af variabler og funktionsnavne ind i mængden af typer.

$$E \vdash e : \texttt{T}$$

Det ovenstående viser formatet for typedomme for udtryk, dette skal læses som: e har type T, givet typebingingerne i typeenvironment E, hvis e ikke indeholder typefejl. Det vil ud fra typereglerne der bliver vist i afsnit 3.3.3 altid vise sig at være en basistype.

$$E \vdash D_v : \texttt{ok}$$

Den ovenstående er formatet for typedommen for variabelerklæringer, her ser vi at D_v har type `ok` givet typebindingerne i typeenvironment E, hvis D_v ikke indeholder nogle typefejl.

$$E \vdash D_f : \texttt{ok}$$

Den ovenstående typedom er for funktionserklæringer, her ser vi at D_f har type `ok` givet typebindingerne i typeenvironment E, hvis erklæringen af en funktion har en basistype som stemmer overens med typen af det udtryk der bliver returneret af funktionen.

$$E \vdash S : \texttt{ok}$$

Det ovenstående viser formatet for typedomme for kommandoer, her ser vi at S har type `ok` givet typebindingerne i typeenvironment E, hvis der intet sted i S kan forekomme nogle bestemte fejl på køretid.

$$E \vdash c : \texttt{ok}$$

Det ovenstående er formatet for typedomme for nye blokke, her ser vi at c har type `ok` givet typebindingerne i typeenvironment E, hvis der i variabelerklæringerne ikke er typefejl og der i kommandoerne ikke kan forekomme fejl på køretid.

3.3.3 Typeregler

I dette afsnit vil vi opstille typeregler for udtryk, blokke, erklæringer af variabler og funktioner, og kommandoer. Her vil vi opstille typeregler for indholdet i de før nævnte kategorier, som der ikke er typeregler for i bogen [34], eller på anden vis er interessante. I de følgende typeregler optræder også notationerne $\overrightarrow{x}$ for parametre til funktioner, og deres typer $\overrightarrow{T_n}$, indekset n er her en indikator og har ikke noget med betydningen af notationerne at gøre, definitionen for notationerne er de følgende:

$$\overrightarrow{T} = T_1,\ T_2,\ \ldots T_n \mid \varepsilon$$

$$\overrightarrow{x} = x_1,\ x_2,\ \ldots x_n \mid \varepsilon$$

Disse notationer skal forstås, som at samlingen af typer $\overrightarrow{T}$ kan være en mængde af typer og behøver ikke kun at være af en type. Endvidere er $\overrightarrow{x}$ en mængde af parametre, og vi antager at forholdet mellem parametre og deres typer er en til en når notationen $\overrightarrow{x} : \overrightarrow{T}$ bruges.

Typeregler for udtryk

Typereglerne for udtryk kan ses nedefor, her er det blevet valgt kun at medtage udvalgte typeregler hvori bio typerne indgår:

$$[concat_{exp}] \qquad \frac{E \vdash e_1 : B4 \quad E \vdash e_2 : B4}{E \vdash e_1 + e_2 : B4}$$

$$[as_{exp}] \qquad \frac{E \vdash e : B3}{E \vdash e\ \texttt{as}\ B4 : B4}$$

$[comp_{exp}]$	$\dfrac{E \vdash e : \texttt{Dna}}{E \vdash \texttt{comp} :(e) : \texttt{Dna}}$
$[rev_{exp}]$	$\dfrac{E \vdash e : B2}{E \vdash \texttt{rev} :(e) : B2}$
$[len_{exp}]$	$\dfrac{E \vdash e : B2}{E \vdash \texttt{len} :(e) : \texttt{Int}}$
$[pos_{exp}]$	$\dfrac{E \vdash e_1 : B2 \quad E \vdash e_2 : B2}{E \vdash \texttt{position of } e_1 \texttt{ in } e_2 : \texttt{Int}}$
$[count_{exp}]$	$\dfrac{E \vdash e_1 : B2 \quad E \vdash e_2 : B2}{E \vdash \texttt{count } e_1 \texttt{ in } e_2 : \texttt{Int}}$
$[contain_{exp}]$	$\dfrac{E \vdash e_1 : B2 \quad E \vdash e_2 : B2}{E \vdash e_1 \texttt{ contains } e_2 : \texttt{Bool}}$
$[remove_{exp}]$	$\dfrac{E \vdash e_1 : B2 \quad E \vdash e_2 : B2}{E \vdash \texttt{remove } e_1 \texttt{ from } e_2 : B2}$
$[get_{exp}]$	$\dfrac{E \vdash e_1 : B2 \quad E \vdash e_2 : \texttt{Int}}{E \vdash e_1[e_2] : B2}$
$[call_{exp}]$	$\dfrac{E(f) = (\overrightarrow{x} : \overrightarrow{T_2}) \to T_1 \quad E \vdash \overrightarrow{e} : \overrightarrow{T_2}}{E \vdash f(\overrightarrow{e}) : T_1}$

Tabel 3.7: Typeregler for udtryk

I de ovenstående typeregler for $[comp_{exp}]$, $[rev_{exp}]$ og $[len_{exp}]$ er der, i reglernes konklusioner, et kolon mellem navnet og udtrykket indkapslet i parenteser. Dette kolon er en del af syntaksen for udtrykkene og har altså ikke noget at gøre med hvad disse udtryk evaluerer til.

En ting der hurtigt bliver tydeligt er at i størstedelen af disse typeregler bliver der brugt metavariablen $B2$, dette skyldes at alle typerne der hører til $B2$ kan bruges i disse udtryk, med nogle få undtagelser, men dette vil der være mere om i afsnit 3.3.4 der omhandler konvertering mellem typer. For at give et eksempel på hvordan en typeregel skal læses beskriver vi her udtrykket for funktionskald, $[call_{exp}]$. I denne regel antager vi at funktionen f tager parametre $\overrightarrow{x}$ af typerne $\overrightarrow{T_2}$, da vil funktionen være af typen T_1. Efterfølgende tjekkes det om de aktuelle parametre, $\overrightarrow{e}$, til funktionen også har typerne $\overrightarrow{T_2}$. Hvis dette er tilfældet vil typen af funktionen være T_1, hvis funktionskaldet er uden typefejl.

Typeregl for blokke

Det følgende er typereglen for blokke, i denne typeregl får vi brug for en hjælpefunktion, $E(D_v, E)$, der som værdi giver det opdaterede typeenvironment vi får fra de nye variabelerklæringer, D_v, i blokken. Definitionen kan findes i bogen [34] og er den følgende:

$$E(\varepsilon, E) = E$$

$$E(T\ x;\ D_v, E) = E(D_v, E, x : T)$$

Definitionen skal forstås som, at hvis vi udvider et typeenvironment med en tom erklæring ε, hvilket betyder at der ikke er nogle nye variabelerklæringer, så vil typeenvironment E forblive uændret. Hvis vi udvider E med en følge af nye variabelerklæringer skal E udvides med de resterende erklæringer D_v hvor vi nu ved at vi har et typeenvironment hvor vi kender typen T for x. Med definitionen af funktionen på plads, kan vi nu give typereglen for blokke:

$$[blok] \qquad \frac{E \vdash D_v : \texttt{ok} \quad E' \vdash S : \texttt{ok}}{E \vdash \{D_v\ S\} : \texttt{ok}} \quad hvor \quad E' = E(D_v, E)$$

Tabel 3.8: Typeregl for blokke

I stedet for at skrive c er det her i stedet $\{D_v\ S\}$ dette er gjort for at gøre det mere tydeligt hvad en bloks bestanddele er, men i andre typeregler der benytter blokke vil c blive brugt i stedet. I reglen for blokke bliver variabelerklæringerne i blokken `ok` hvis de ikke indeholder typefejl. Efterfølgende får S i blokken typen `ok` hvis de ikke indeholder typefejl og blokken vil således være `ok`.

Typeregler for erklæringer

Det følgende er typeregler for erklæringer af variabler og funktioner:

$$[tom_{dec}] \qquad E \vdash \varepsilon : ok$$

$$[var_{dec}] \qquad \frac{E \vdash e : T \quad E,\ x : T \vdash D_v : ok}{E \vdash T\ x = e;\ D_v : \texttt{ok}}$$

$[func_{dec}]$	$\dfrac{E,\ \overrightarrow{x} : \overrightarrow{T_2} \vdash c : \texttt{ok} \quad E,\ f : (\overrightarrow{x} : \overrightarrow{T_2}) \to T_1 \vdash D_f : \texttt{ok}}{E \vdash T_1\ f(\overrightarrow{T_2\ x})\ c\ D_f : \texttt{ok}}$

Tabel 3.9: Typeregler for erklæringer

Den øverste af typereglerne er for tomme erklæringer, denne vil få typen `ok` da det er lovligt i den abstrakte syntaks, og også sproget, ikke at have nogle variabelerklæringer eller funktionserklæringer, dette vil tilmed heller ikke give anledning til nogle fejl på køretid. Den næste af typereglerne er for variabelerklæringer, i denne regel bliver typen af udtrykket e tjekket ved at den skal have samme type T som variablen x. Efter dette tjekkes de efterfølgende variabelerklæringer D_v, ud fra antagelsen af at variablen x har type T. Grunden til at typen af x skal være kendt er fordi at denne variabel kan indgå i erklæringen af de følgende variabler. Den sidste af typereglerne er for funktionserklæringer, i denne regel tjekkes det først om blokken i funktionen, c, har typen `ok` hvor det bliver antaget at de formelle parametre har de rigtige typer. Dernæst kan det antages at funktionen f tager parametre af typerne $\overrightarrow{T_2}$ og har typen T_1. Med denne viden om funktionen kan de efterfølgende funktionserklæringer typetjekkes.

Typeregler for kommandoer

Det følgende er typeregler for kommandoer, her er det blevet valgt at medtage kommandoer fra vores sprog, som der ikke har typeregler i [34]:

$[for_{kom}]$	$\dfrac{E \vdash x : \texttt{Int} \quad E \vdash e_1 : \texttt{Int} E \vdash e_2 : \texttt{Bool} \quad E \vdash e_3 : \texttt{Int} \quad E \vdash c : \texttt{ok}}{E \vdash \texttt{for}(x = e_1; e_2; x = e_3)\ c : \texttt{ok}}$
$[print - 1_{kom}]$	$\dfrac{E \vdash e : T}{E \vdash \texttt{Print}(e) : \texttt{ok}}$
$[print - 2_{kom}]$	$\dfrac{E \vdash e_1 : T \quad E \vdash e_2 : \texttt{Int}}{E \vdash \texttt{Print}(e_1, e_2) : \texttt{ok}}$
$[print - 3_{kom}]$	$\dfrac{E \vdash e_1 : T \quad E \vdash e_2 : \texttt{Int} \quad E \vdash e_3 : \texttt{Bool}}{E \vdash \texttt{Print}(e_1, e_2, e_3) : \texttt{ok}}$
$[break_{kom}]$	$E \vdash \texttt{break} : \texttt{ok}$
$[return_{kom}]$	$\dfrac{E \vdash e : \texttt{T}}{E \vdash \texttt{return}(e) : \texttt{ok}}$

Tabel 3.10: Typeregler for kommandoer

For at give et eksempel på hvordan disse typeregler skal læses tager vi udgangspunkt i $[for_{kom}]$. Her ser vi at en for-løkke har type `ok` hvis typen af x og e_1 er `int`, typen af e_2 er `bool`, typen af e_3 er `int` og typen af c er `ok`.

3.3.4 Konvertering mellem typer

I dette afsnit vil vi kigge på de konverteringer mellem typer, som er mulige i vores sprog. Konvertering mellem typer i vores sprog foregår gennem et udtryk nemlig $[as_{exp}]$, som er givet i afsnit 3.3.3 omkring typeregler. Dette udtryk giver bl.a. mulighed for at transskribere en variabel af typen `Dna` og tilskrive den til en variabel af typen `Rna` osv. For at demonstrere brugen af dette udtryk vil der nedenfor blive givet nogle typeregler hvori $[as_{exp}]$ indgår:

$[DnaToRna]$	$\frac{E \vdash e_1 : \texttt{Rna} \quad E \vdash e_2 : \texttt{Dna}}{E \vdash e_1 + e_2 \texttt{ as Rna} : \texttt{Rna}}$
$[RnaToDna]$	$\frac{E \vdash e_1 : \texttt{Dna} \quad E \vdash e_2 : \texttt{Rna}}{E \vdash e_1 + e_2 \texttt{ as Dna} : \texttt{Dna}}$
$[CodonToRna]$	$\frac{E \vdash e_1 : \texttt{Rna} \quad E \vdash e_2 : \texttt{Codon}}{E \vdash e_1 + e_2 \texttt{ as Rna} : \texttt{Rna}}$
$[CodonToDna]$	$\frac{E \vdash e_1 : \texttt{Dna} \quad E \vdash e_2 : \texttt{Codon}}{E \vdash e_1 + e_2 \texttt{ as Dna} : \texttt{Dna}}$
$[RnaToProtein]$	$\frac{E \vdash e_1 : \texttt{Protein} \quad E \vdash e_2 : \texttt{Rna}}{E \vdash e_1 + e_2 \texttt{ as Protein} : \texttt{Protein}}$
$[DnaToProtein]$	$\frac{E \vdash e_1 : \texttt{Protein} \quad E \vdash e_2 : \texttt{Dna}}{E \vdash e_1 + e_2 \texttt{ as Protein} : \texttt{Protein}}$
$[CodonToProtein]$	$\frac{E \vdash e_1 : \texttt{Protein} \quad E \vdash e_2 : \texttt{Codon}}{E \vdash e_1 + e_2 \texttt{ as Protein} : \texttt{Protein}}$

Tabel 3.11: Typeregler for konvertering mellem typer

I det ovenstående ser vi at e_2 i alle eksemplerne skifter type. Det gælder for alle de ovenstående eksempler at ingen af dem vil give anledning til typefejl da der her kun vises lovlige udtryk. Eksemplerne udgør tilsammen de lovlige måder at bruge $[as_{exp}]$ på, for at skifte type. Det bør her også nævnes at man ikke nødvendigvis behøver at bruge $[concat_{exp}]$, som gjort her, dette kunne ligeså godt have været et andet udtryk hvori der benyttes $[as_{exp}]$ såfremt udtrykket overholder typereglerne for udtryk. De ovenstående

eksempler viser også at der er måder at sammensætte typer på, som ikke er lovlige. Her er det typen `Codon` der skal tages hensyn til. Vi ved fra afsnit 1.4.1 at et `Codon` består af tre baser, således kan et `Codon` ikke lægges til et andet `Codon` og stadig være af typen `Codon`. Omvendt kan et `Codon` heller ikke være kortere end tre baser, hvilket medfører at et `Codon` ikke kan trækkes fra et andet `Codon`. Endvidere er der også begrænsninger på brugen af typen `Protein`. Variabler af typerne `Dna`, `Rna` og `Codon` kan omskrives til at være af typen `Protein` ved brug af $[as_{exp}]$, men ikke omvendt. Dette skyldes at vi ved fra afsnit 1.4.1 figur 1.3 at flere forskellige codon kan udgøre den samme aminosyre.

Med en formel definition, i form af en strukturel operationel semantik for alle udtryk, kommandoer mv. og et typesystem er det nu muligt at påbegynde implementationen af selve oversætteren til sproget, som gør at sproget kan bruges til at skrive kørende programmer.

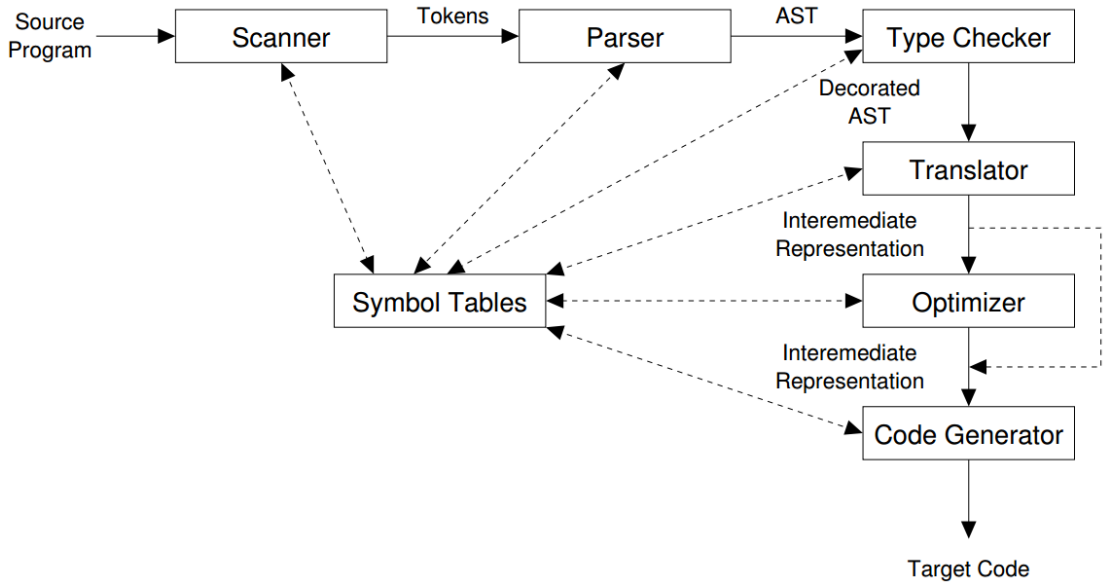
Kapitel 4

Implementation af oversætter

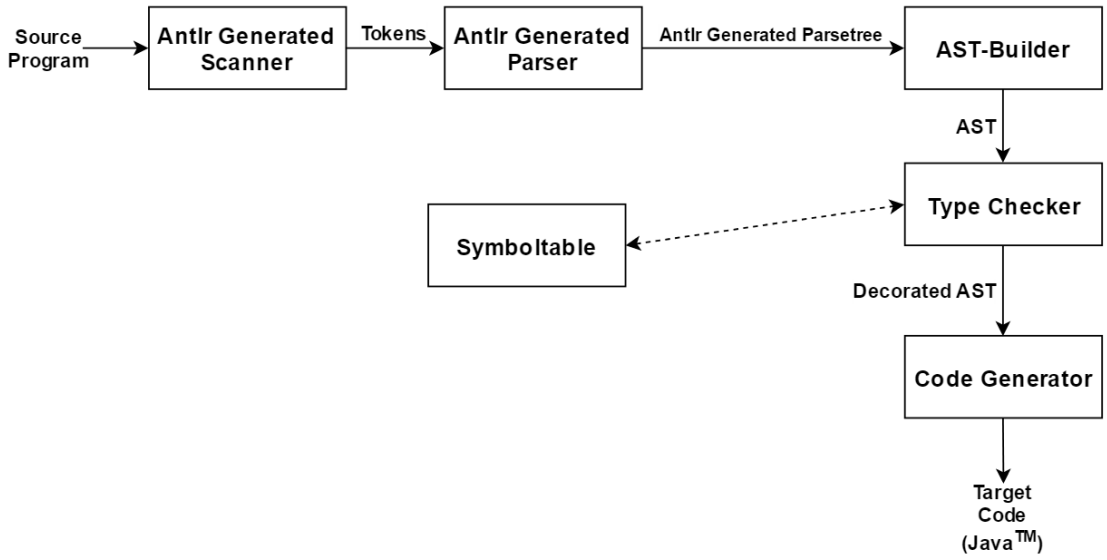
Nu hvor vi har givet en specifikation for vores sprog, er det muligt implementere sproget, hvilket gøres ved at konstruere en oversætter. I dette kapitel starter vi med at give et overblik over den implementerede oversætter, som sammenlignes med en general oversætter, for at få et udgangspunkt. Herefter beskriver vi implementationen af hver enkelt komponent for sig og hvordan vi har konstrueret det. Dette enten er blevet gjort ved hjælp af værktøjer, specifikke programmerings mønstre eller datastrukturer.

4.1 Oversætterens komponenter

I dette afsnit gennemgår vi faserne af en oversætter, med udgangspunkt i vores implementation. De komponenter, som findes i vores oversætter, kan ses på figur 4.2, hvor figur 4.1 viser en mere general oversætter.



Figur 4.1: Dette er en figur taget fra [37], som viser en hvordan de forskellige komponenter er opdelt, i en general syntaks dirigeret oversætter.



Figur 4.2: Komponenter som indgår i den implementerede oversætter. Rektanglerne repræsenterer de forskellige komponenter i oversætteren, og pilene indikerer flow og input/output for hver komponent.

Umiddelbart, så er der væsentlige forskelle på vores implementerede oversætter og en typisk oversætter.

Symboltabel

Figur 4.2 viser, at vores skanner, parser, og kodegenerator ikke interagerer med symboltabellen, hvilket ellers er en mulighed ifølge figur 4.1. Vores skanner og parser er genereret af ANTLR, og dermed er det ikke nødvendigt, at disse komponenter skal interagere med symboltabellen. I Kodegenerations fasen har vi ikke brug for symboltabellen, da typetjekker fasen dekorerer vores *abstrakte syntaks træ*, med alt det nødvendige information, som kodegeneratoren har brug for.

Translator og Optimizer

En typisk oversætter kan indeholder komponenter, som en *translator* og en *optimizer*, men disse typer af komponenter indgår ikke i vores oversætter. Det er fordi at vi kodegenerer til Java, som er et højniveau sprog, hvor optimering indgår i Java's virtuelle maskine, på runtime (JIT).

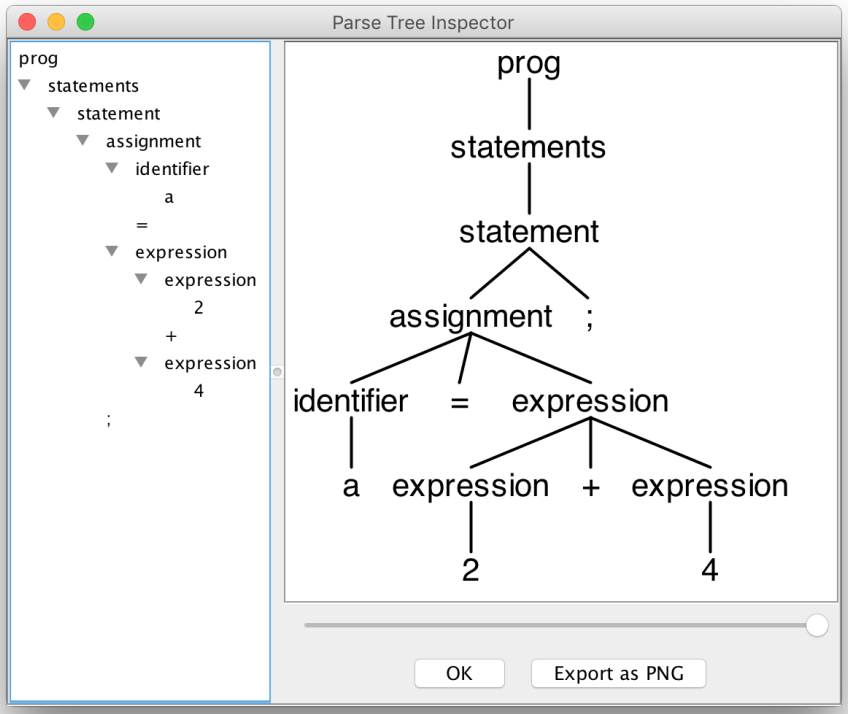
4.2 Skanner

Første fase i en oversætter er skanning. Skannerens input er selve program teksten, hvor output af skanneren er en følge af tokens. Tokens kan være typer, reserverede ord, samt symboler. Skanneren kan enten være håndskrevet, eller autogenereret, af et værktøj, som i vores tilfælde er ANTLR. Disse tokens bliver brugt i næste fase, som er parsing.

Vores tokens er specificeret i afsnit 2.2. ANTLR kan dermed generere en skanner på baggrund af en tokenspecifikation.

4.3 Parser

Parseren er også et produkt af ANTLR's genererede kode. På baggrund af vores kontekstfrie grammatik i afsnit 2.3 producerer ANTLR en parser klasse (Java klasse). Parserens funktion er at tage tokens fra skanneren, og lave et *parsetræ*, også kaldet et *konkret syntakstræ* (CST). Dette træ indeholder al information omkring input teksten, baseret på vores grammatik. Dette træ indeholder derfor parenteser, semikolon, og tomme knuder (terminaler).
Figur 4.2 viser at outputtet fra vores parser komponent er et parsetræ, hvilket er hvad ANTLR genererer. Et typisk output fra parseren er et *abstrakt syntaks træ*, som også er tilfældet i figur 4.1, fremfor et CST. I afsnit 4.4 beskriver vi hvordan vi anvender ANTLR's parsetræ til at konstruere et AST.



Figur 4.3: ANTLR's Parse Tree Inspector

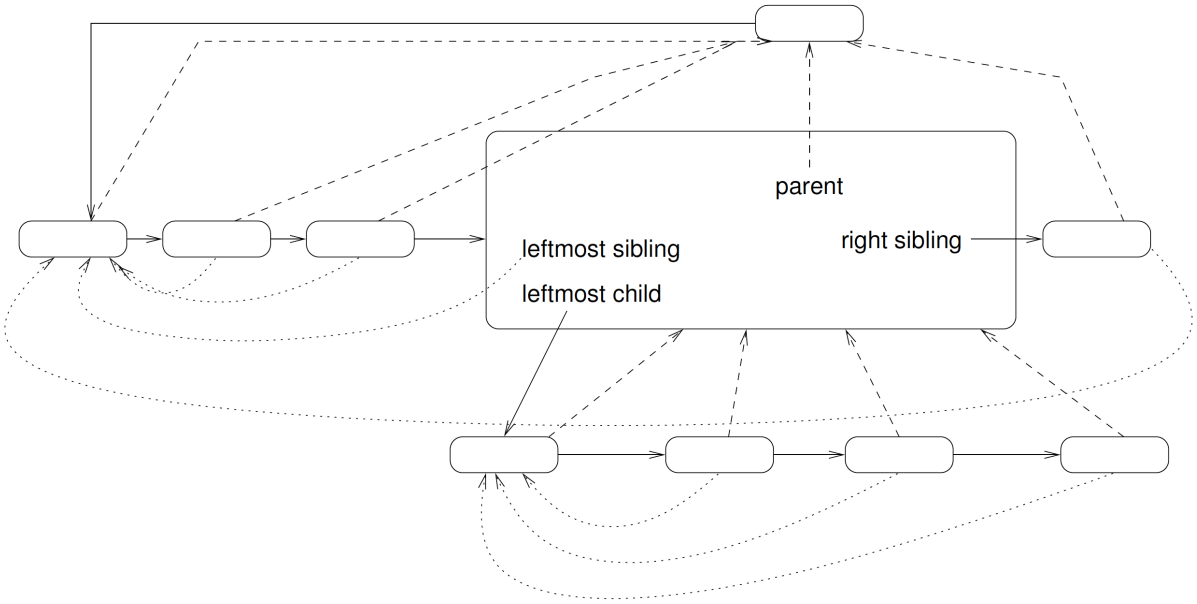
ANTLR kan desuden visualisere et parsetræ, som set i figur 4.3. Dette har været en stor hjælp, i løbet af vores arbejde med grammatikken.

4.4 Abstrakt syntaks træ

Et abstrakt syntaks træ er en repræsentation af input programmet, som de efterfølgende faser af oversætteren anvender. I vores AST indgår der knuder, som repræsenteres af instantierede java klasser, som baserer sig på den abstrakte syntaks, fra afsnit 3.1. Umiddelbart, så er der kun én knude, som findes som java klasse, men som ikke findes i den abstrakte syntaks, hvilket er *BaseNode*. Dette er en abstrakt klasse, som alle andre knude klasser arver fra. Herefter har vi ydereligere fem abstrakte klasser, som repræsenterer kommandoer (S), udtryk (e), blokke (c), erklæring af variable (D_v) og erklæring af funktioner (D_f). Herefter har vi en klasse for enhver kommando, udtryk osv. En klasse der repræsenterer en while kommando, ville være *WhileCommandNode*, som arver fra *CommandNode*, som arver fra *BaseNode*. Dette er strukturen for vores knuder, som er baseret på den abstrakte syntaks, og som ender med at udgøre AST'et.

4.4.1 Navigering af det abstrakte syntaks træ

For at vi kan navigere træet, skal hver knude have referencer til de andre knuder i træet. Alle de referencer, som enhver knude har tilgængelig, illustreres på figur 4.4.



Figur 4.4: AST struktur. Linjer er referencer til venstre barn, stiplede linjer er referencer til forældren, og prikkede linjer er referencer til mest venstre søskende.[37]

Dermed har vi gjort sådan, at *BaseNode* klassen indeholder fire felter, til henholdsvis *parent*, *leftmost sibling*, *leftmost child* og *right sibling*. Med disse fire referencer, som enhver knude har tilgængelig, er det muligt at navigere det abstrakte syntaks træ.

4.4.2 Konstruktion af det abstrakte syntaks træ

For at konstruere det abstrakte syntaks træ, anvender vi en metode *AddChild(BaseNode node)*, som enhver knude har har tilgængelig.
Som nævnt i afsnit 4.3, så genererer ANTLR et parsetræ, efter at have parset en følge af tokens. ANTLR genererer også en abstrakt superklasse, som kan bruges til at navigere parsetræet. Vores *AST-Builder* komponent, fra figur 4.2, arver fra ANTLR's superklasse, og efterfølgende er det den, som konstruerer AST'et ud fra parsetræet, ved at besøge de relevante knuder i parsetræet.

```
1  @Override
2  public BaseNode
       visitAssignment(LanguageParser.AssignmentContext ctx) {
3      BaseNode node;
4      switch(ctx.op.getType()) {
5          case LanguageLexer.EQUAL:
6              node = new AssignCommandNode();
7              break;
8          default:
9              node = new NullNode();
10             break;
11     }
12     node.AddChild(visit(ctx.left));
13     node.AddChild(visit(ctx.right));
14     node.line = ctx.getStart().getLine();
15     node.pos = ctx.getStart().getCharPositionInLine();
16
17     return node;
18 }
```

Listing 4.1: Metode fra *ASTBuilder.java* komponenten

Listing 4.1 viser er et eksempel på en metode fra AST-Builder komponenten. Denne metode bliver kaldt, når man navigerer parsetræet og besøger en assignment knude. Herefter bliver der switchet på typen af operatoren, hvor der i vores sprog kun er muligt at lave et simpelt assignment, hvor der så bliver instantieret en ny assignment knude, som kommer til at indgå i vores AST. Herefter besøger man henholdsvis venstre- og højre barn, hvor begge *visit()* metoder returnerer instantierede AST knuder, hvor disse bliver tilføjet til *node*, som var vores assignment knude. Til sidst gemmer metoden linje- og kolonne-data, i tilfælde af senere fejlfinding, sådan at vi kan vise brugeren hvor fejlen opstod.

4.5 Symboltabel

Datastrukturen som vi benytter for vores symboltabel, er en stack af hashtables. Hvert hashtable repræsenterer et scope. Det vi gemmer i hvert hashtable er hele knuder. Grunden til dette, er at hvis vi gemmer en *IdentifierNode*, så istedet for kun at gemme navnet på knuden, så gemmes hele knuden, som indeholder navn, samt linjenummer og karakternummer, som bruges til at give bedre fejlbeskeder. For at administrere symboltabellen, anvender vi nogle metoder, som er følgende: *OpenScope(), CloseScope(), EnterSymbol(String name, BaseNode node), RetrieveSymbol(String name)* samt *DeclaredLocally(String name)*.

4.5.1 Konstruktion af symboltabellen

Metoden som konstruerer vores symboltabel er *ProcessNode* og en mindre del af metoden kan findes i 4.2.

```
1  public static void ProcessNode(BaseNode node) {
2      switch(node.getClass().getSimpleName()) {
3          case "BlockNode":
4              BaseNode temp = node.getParent();
5              if(!temp.getClass().getSimpleName().
6              equals("DeclareFunctionNode")) {
7                  ProgNode.OpenScope();
8              }
9              break;
10         case "DeclareVarNode":
11             if(!ProgNode.DeclaredLocally(node.spelling){
12             ProgNode.EnterSymbol(node.spelling.toString(),node;
13             }
14             else {
15                 errorList.add(new Error("Identifier
                       \""+node.spelling+"\""+" already used",
                       node.line, node.pos));
16             }
17             break;
18         (...)
```

Listing 4.2: Første del af metoden *ProcessNode(BaseNode node)* der konstruerer symboltabellen

Metoden tager som parameter en knude, og efterfølgende switches der på knudens type (navn). Hvis knuden er en *Blocknode*, så åbner vi et nyt scope, ved at pushe et nyt hashtable på stacken. Hvis knuden er en erklæring af en ny variabel, så bliver knuden

gemt i symboltabellen, hvis ikke den allerede er erklæret. Igennem denne konstruering af symboltabellen, bliver der implicit lavet *scopechecking*, hvor der fx på linje 15 bliver instantieret en ny *Error*, hvis et variabel navn allerede er blevet brugt i det nuværende scope.

```
1  (...)
2  for (BaseNode item : GetListOfChildren(node)) {
3          ProgNode.ProcessNode(item);
4  }
5  if (node.getClass().getSimpleName().equals("BlockNode") ||
      node.getClass().getSimpleName().equals("ProgNode")) {
6          TypeChecker typeChecker = new TypeChecker();
7          node.Accept(typeChecker);
8          if(node.getClass().getSimpleName().equals("BlockNode")){
9                  BlockNode temp2 = (BlockNode) node;
10                 temp2.HasNotBeenChecked = false;
11         }
12 ProgNode.CloseScope();
13 }
```

Listing 4.3: Sidste del af metoden *ProcessNode(BaseNode node)*

I den sidste del af *ProcessNode*, bliver metoden kaldt rekursivt på hvert barn, på linje 2 i listing 4.3, for den pågældende knude. På denne måde bliver AST'et traverseret *depth-first*, og enhver knude vil blive besøgt.
Når vi så har besøgt alle børn, for fx en *BlockNode*, så vil vi gerne lukke et scope, fordi så ved vi, at vi har besøgt alt det, som der måtte være i det scope, og dermed har vi ikke brug for de lokale variabler mere. Inden vi lukker scopet, med *CloseScope()* på linje 15 i listing 4.3, så kunne vi jo typetjekke det scope, da symboltabellen indeholder alt fra det nuværende scope, og de ydre scopes. Derfor instantierer vi en ny typetjekker, og starter et typetjek ved at kalde *node.accept(typeChecker)*, lige inden et scope lukkes.

4.5.2 Visitor pattern

Måden hvorpå vi håndterer typetjekker fasen og kodegenererings fasen er ved at implementere et *visitor pattern*. I figur 4.2 svarer komponenten *Type Checker* og *Code generator* til en specialiseret visitor, som er baseret på en abstract klasse *visitor*.

4.6 Typetjekker

Vores *Type Checker* komponent er den implementerede version af det type system, som er blevet defineret i afsnit 3.3.3. Et eksempel på denne implementation, kunne være for

reglen $[as_{exp}]$ fra afsnit 3.3.3. Den tilsvarende metode for denne regel kan ses i listing 4.4

```
1  public void Visit(ConvertNode node) {
2  visitChildren(node);
3  String type1 = node.getLeftmostchild().type;
4  String type2 = node.content.toString();
5  if(type1.equals(INTTYPE) || type1.equals(BOOLTYPE)) {
6      ProgNode.errorList.add(new Error("Cannot convert "+type1,
           node.line, node.pos));
7      node.type = DEFAULTTYPE;
8      }
9  else if(type1.equals(type2)) { node.type = type1; }
10 else if(type1.equals(DNATYPE) && (type2.equals(RNATYPE) ||
       type2.equals(PROTEINTYPE))) { node.type = type2; }
11 else if(type1.equals(RNATYPE) && (type2.equals(PROTEINTYPE) ||
       type2.equals(DNATYPE))) { node.type = type2; }
12 else if(type1.equals(CODONTYPE) && (type2.equals(PROTEINTYPE)
       || type2.equals(DNATYPE) || type2.equals(RNATYPE))) {
       node.type = type2; }
13 else {
14     ProgNode.errorList.add(new Error("Cannot convert "+type1 +"
           to "+type2));
15     node.type = DEFAULTTYPE;
16     }
17 }
```

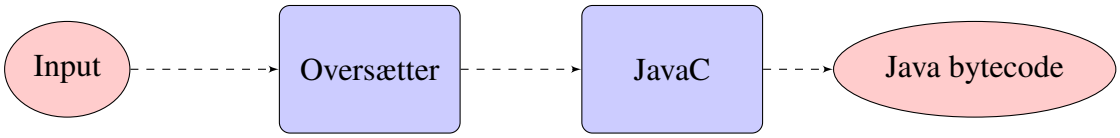
Listing 4.4: Metoden der typetjekker *[as_exp]*

Metoden er opbygget således, at børnene for knuden bliver besøgt først, da man ellers ikke kender typerne af børnene, som indgår i dette konverteringsudtryk. Herefter finder vi typen af venstre barn (*type1*), som er det udtryk der skal konverteres. Herefter finder vi typen af højre barn (*type2*), som er typen, som type1 skal konverteres til. Herefter følger if-else kæden direkte typereglen fra afsnit 3.3.3, hvor *type1* må være $B2$ og *type2* må være $B4$. I alle andre tilfælde gives der en error. Da vi anvender visitor pattern, findes der en metode for enhver knude i typetjekker visitoren, som følger samme struktur, som metoden i listing 4.4.

4.7 Kodegenerering

Kodegenerering er det sidste skridt i vores oversætter. Her oversættes det bearbejdede AST til et *target* sprog - det sprog som vi ønsker at oversætte til. Her kan vi enten over-

sætte til direkte eksekverbar kode, fx forskellige former for maskinkode, herunder assembly eller JVM bytecode. Vi har også muligheden for at oversætte til et ikke-direkte-eksekverbart target, eks. C#-, Java- eller C-kode. Herfra vil de respektive oversættere oversætte vores output, til noget eksekverbart.
På figur 4.5 ses hvordan vi, fra vores oversætter, benytter en Java oversætter (JavaC), som giver os det endelige eksekverbare program.



Figur 4.5: Oversætter struktur

Kodegenerering fungerer principielt på samme måde som vores typetjekker. Vi traverserer vores AST, og ved brug af visitor pattern udsender vi kode til en tekstfil. Dvs. når vi møder en *AssignmentNode* i AST'et, så printer vi den tilsvarende Java kode til en tekstfil. Denne tekstfil vil til sidst indeholde alt som vores AST beskriver.

4.7.1 Indpakning

Vores oversætter skal gerne være brugbar, og skal derfor have en form for indpakning. Dvs. oversætteren skal kunne køres som et program, som vi kan give argumenter, som en input-fil, samt evt. oversætterflag. Vi lægger derfor et kontrollag ovenpå oversætteren, som håndterer input af kildeprogram, oversætterflag, samt visning af det kørende program. Brugeren skal have mulighed for at vælge hvad de ønsker vi skal outputte. Dvs. Hvis brugeren ønsker at få Java-tekstfilen som vores oversætter producerer, skal de have lov til det. Vi skal derfor give brugeren mulighed for at bruge oversætterflag (compile flags).

Kapitel 5

Test og vurdering af sprog

I dette kapitel vil vi teste og vurdere vores sprog. Dette vil vi gøre ved at skrive testprogrammer i vores sprog, og derved se om disse giver det resultat, som vi forventer. Vi har ikke udført systematiske automatiserede tests, og dermed ikke udført unit-testing. Vi har testet funktionalitet, efterhånden som de er blevet implementeret. Vi har derfor lavet ustruktureret integrationstest.

Formålet med dette kapitel er således at se om implementationen af vores sprog overholder de transitionsregler, givet i afsnit 3.2, samt typereglerne givet i 3.3. Vi tager udgangspunkt i brugsmønstrene 1.6.

5.1 Testprogrammer

Brugeren ønsker at definere en splicing proces. Brugeren har en DNA sekvens, hvor man ønsker at fjerne en eller flere delsekvenser. Her benyttes *remove x from y*, som finder og fjerner en delsekvens i en sekvens.

```
1 dna seq = ATTGTCTTGCA;
2
3 seq = remove GT from seq;
4 seq = remove CA from seq;
5
6 Print(seq);
7
8 > ATTCTTG
```

Listing 5.1: Testprogram 1

Brugeren har en DNA sekvens. Her ønsker man at finde position for et givet codon, i sekvensen. Her benyttes *position of x in y*, til at finde positionen af den første forekomst af *x* i *y*. Endvidere benyttes *as*-ordet, til at konvertere codon til DNA, således de kan accepteres af metoden.

```
dna seq = ATTGTCTTGCA;
codon cod = (G,T,C);
int posInSeq = 0;

posInSeq = position of (cod as dna) in seq;

Print(posInSeq);

> 4
```

Listing 5.2: Testprogram 2

Brugeren har en DNA sekvens. Man ønsker at finde den komplementære sekvens, samt den tilsvarende RNA sekvens. Her benyttes det indbyggede metodekald *comp:()*, for at finde den komplementære sekvens. Dernæst benyttes *as*-ordet, til at konvertere DNA sekvensen til dens tilsvarende RNA sekvens.

```
dna seq = ATTGTCTTGCA;
dna complSeq = comp:(seq);

Print(complSeq, 3);
Print(seq as rna, 3);

> TAACAGAACGT
> AUUGUCUUGCA
```

Listing 5.3: Testprogram 3

Disse små programeksempler, 5.15.25.3 viser hvorledes sproget kan bruges, som beskrevet i brugsmønstrene 1.6. Sproget mangler stadig funktionalitet - såsom yderligere indbyggede funktioner til håndtering af typerne, samt en implementation af arrays.

Vi kan ud fra testing, konkludere at oversætteren fungerer som beregnet. Da vi ikke har benyttet automatiseret systematisk tests, kan vi ikke sige at oversætteren er fejlfri, men vi kan drage at ved normalt brugt opfører den sig forudsigeligt.

I kodeeksemplerne 5.5 og 5.4 har vi implementeret følgende opgave:

- Fjern introns fra DNA sekvens 1. Introns findes fra starten af sekvens 1 til base 38, og 5 baser af slutningen på sekvensen.
- Den nye delsekvens konkateneres med sekvens 2, og vi får en ny sekvens 3.
- Sekvens 3 skal nu omvendes og translateres til protein.
- Protein sekvensen skal printes ud på skærmen.

- Tilsidst skal der printes antallet af baseparet "SC"i protein sekvensen.

I kodeeksempel 5.4 ser vi en implementation af ovenstående opgave i BioPython. I dette eksempel har undladt at anvende BioPythons funktion til at importere DNA sekvenser, fordi at vores sprog DNALang ikke understøtter sådan en funktionalitet.
Udfra kodeeksemplet kan vi se at BioPython har nogle sekvensoperationer til at hjælpe med at transkribere og translatere sekvenserne, såvel som at tælle antal af en bestemt base(r). Dette gør arbejdsprocessen markant lettere, end hvis en med i forvejen lav erfaring med programmering, selv skulle implementere følgende funktioner, for at kunne løse opgaverne.

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
# https://www.ncbi.nlm.nih.gov/nuccore/2
my_seq = Seq('aattcatgcgtccggacttctgcctcgagccgccgtacactg
ggccctgcaaagctcgtatcatccgttacttctacaatgcaaaggcaggcctgtgt
cagaccttcgtatacggcggttgccgtgctaagcgtaacaacttcaaatccgcgga
agactgcgaacgtacttgcggtggtccttagtaaagcttg',IUPAC.unambiguous_dna)
# https://www.ncbi.nlm.nih.gov/nuccore/3
part_seq = Seq('caagctttactaaggaccaccgcaagtacgttcgcagtct
tccgcggatttgaagttgttacgcttagcacggcaaccgccgtatacgaaggtctg
acacaggcctgcctttgcattgtagaagtaacggatgatacgagctttgcagggcc
cagtgtacggcggctcgaggcagaagtccggacgcatgaatt',
IUPAC.unambiguous_dna)

count = len(my_seq)
no_introns = my_seq[38:count-5]

complete_seq = no_introns + part_seq
reverse_seq = complete_seq[::-1]
protein_seq = reverse_seq.transcribe().translate()

print(protein_seq)
amount_of_basepair = protein_seq.count("SC")
print(amount_of_basepair)


> LSTQA*RRSSAACDPGRFEHSRQ*RCYVSVRTQSGSICRQRHDSHC*SLGAF*
RLHERHQESFRTNDSWWRSCKRQKAPKLQQCESCRWRHMLPDCVRTET*HLHCLL
CSKRPGS
> 2
```

Listing 5.4: Kodeeksempel i BioPython

I kodeeksempel 5.5 ser vi samme opgave som bliver løst vedhjælp af vores sprog. Vi kan se at syntaksen er meget anderledes end BioPython i det at vi aktivt angiver hvilken

type vores variabler har. På samme måde som i BioPython har vi også implementeret specifikke funktioner til at simplificere arbejdet med sekvenser. Hos BioPython kan vi se på linje 119, at der er mulighed for at både klippe foran og bagfra en sekvens. Dette er ikke muligt i vores sprog, men vi har kommet udenom dette problem ved at vende sekvensen, og derefter klippe 5 tegn fra.

```
1  dna mySeq = AATTCATGCGTCCGGACTTCTGCCTCGAGCCGCCGTACACTGGGCC
2      CTGCAAAGCTCGTATCATCCGTTACTTCTACAATGCAAAGGCAGGCCTGTGTCAGACC
3      TTCGTATACGGCGGTTGCCGTGCTAAGCGTAACAACTTCAAATCCGCGGAAGACTGCG
4      AACGTACTTGCGGTGGTCCTTAGTAAAGCTTG;
5  dna partSeq = CAAGCTTTACTAAGGACCACCGCAAGTACGTTCGCAGTCTTCCG
6      CGGATTTGAAGTTGTTACGCTTAGCACGGCAACCGCCGTATACGAAGGTCTGACACAG
7      GCCTGCCTTTGCATTGTAGAAGTAACGGATGATACGAGCTTTGCAGGGCCCAGTGTAC
8      GGCGGCTCGAGGCAGAAGTCCGGACGCATGAATT;
9
10 int counter = len:(mySeq);
11 dna noIntrons = mySeq[38];
12
13 int amountOfBasepair = 0;
14 dna completeSeq = A;
15 dna reverseSeq = A;
16 protein proteinSeq = "A";
17
18 noIntrons = rev:(noIntrons);
19 noIntrons = noIntrons[5];
20 noIntrons = rev:(noIntrons);
21
22 completeSeq = noIntrons + partSeq;
23 reverseSeq = rev:(completeSeq);
24
25 proteinSeq = ((reverseSeq as rna) as protein);
26 Print(proteinSeq, 1, true);
27
28 amountOfBasepair = count "SC" in proteinSeq;
29 Print(amountOfBasepair);
30
31 >L S T Q A STOP R R S S A A C D P G R F E H S R Q STOP R C Y V S V R
      T Q S G S I C R Q R H D S H C STOP S L G A F STOP R L H E R H Q E
      S F R T N D S W W R S C K R Q K A P K L Q Q C E S C R W R H M L P
      D C V R T E T STOP H L H C L L C S K R P G S
32 > 2
```

Listing 5.5: Kodeeksempel i vores sprog

5.1.1 Evaluering af vores sprog

På baggrund af disse programmer, vil vi nu kigge på hvilke forskelle der findes mellem Python (m. BioPython) og vores løsning. Endvidere evaluerer vi vores sprog, efter kriterierne fra afsnit 1.5.3.

Datatyper

Vores sprog har de relevante datatyper, til at håndtere det grundlæggende indenfor bioinformatiske sekvensoperationer. Altså, vi har datatyperne DNA, RNA, codon og protein, hvilket udgør de essentielle dele i det centrale dogme. Men da vores sprog ikke tillader brugerdefinerede datatyper, eller brug af decimaltal samt negative tal, er sproget begrænset i hvilke beregninger kan laves, og med hvilken præcision de kan udføres. F.eks. ønsker brugeren af lave en funktion som finder et procenttal, kan brugeren kun finde procenttal som afrundede heltal. I sprog såsom (Bio)Perl, er sekvenser håndteret som strenge, og ikke i dedikerede datatyper. Det at vi gør dette i vores sprog, er en kæmpe styrke.

Syntaksdesign

Vores sprog benytter sig af nogle naturlige sprogkonstruktioner, f.eks. *remove x from y*, som funktionskald til nogle af de indbyggede funktioner. Dette gør vores sprog ekstremt let læseligt, samt let at skrive, for vores målgruppe. På den anden side er der mange undtagelser for hvad disse funktionskald kan håndtere, mellem typer. Dette giver os en dårlig ortogonalitet i sproget. Endvidere benytter vores sprog sig af typiske konstruktioner såsom for- og while-løkker. Alt i alt vurderer vi at vores syntaks design er en forbedring over de eksisterende sprog, som Python og Perl.

Udtryksfuldhed

I forhold til vores sprogs mulighed for at udtrykke sig kort og præcist, bidrager vores naturlige sprog konstruktioner enormt til dette. Vi kan udtrykke os præcist i vores sprog, hvilket for læsbarheden af sproget. Men sproget er dog ikke kortfattet. Vi har ikke muligheden for at anvende en dedikeret inkremerings operator, og derved er brugeren nødt til at beskrive det som $i = i + 1$, i stedet for $i{+}{+}$, som man kunne forvente.

5.1.2 Opsummering

Vi har nu testet vores sprog, samt evalueret dets evner. Vi kan dermed drage at vores sprog er effektivt i dets felt, ved at formulere sig relativt kortfattet og præcist - samtidig med at det er både læsbar og skrivbart. Dette er godt, med henhold til vores målgruppe, da de ikke er erfarne programmører, som bare ønsker at udføre sekvensoperationer. Dog har vores sprog nogle mangler, som vi beskriver i diskussionen 6.1.

Kapitel 6

Afslutning

I dette kapitel vil vi konkludere vores projekt, samt diskutere den valgte løsning og reflektere til hvad der videre kunne arbejdes med i forhold til sproget, hvis projektet skulle fortsættes. Kapitlet vil inddrage vores brugsmønstre, problemformulering og løsningskriterier, som blev præsenteret i afsnit 1.6, 1.7 og 1.8. Vi vil med vores viden på dette tidspunkt bruge de nævnte afsnit til at vurdere om projektet har nået sit mål og vi derved har løst vores problem.

6.1 Diskussion

I dette afsnit vil vi diskutere vores løsning. Vi går over de emner som vi enten ikke har implementeret, eller kunne have implementeret anderledes. Vi kommer også omkring alternative løsninger, som andre target sprog eller at lave en fortolker.

Eksterne sekvenskilder

Vores sprog er fokuseret på sekvenser. Disse sekvenser kan ske at være ekstremt lange - men brugeren har kun mulighed for at definere sekvenser i programfilen. Dette kan resultere i programfiler, som er fyldt med sekvensdefinitioner, og meget lidt programlogik. Derfor ville det være optimalt hvis brugeren kunne hente sekvens-data fra eksterne kilder - f.eks. tekstfiler, webadresser eller databaser - og tilskrive til variabler. Da vores sprog er bygget på Java, er der mulighed for at implementere dette, med relativt lille besvær.

Arrays

Vores sprog har ikke en implementation af arrays. Dette er et problem, da mange algoritmer benytter arrays. Et af målene med vores sprog, var at brugeren kunne, hvis nødvendigt, implementere sin egen algoritme. Hvis denne algoritme er afhængig af arrays, er dette ikke muligt. Sproget burde derfor indeholde arrays.

Fortolker

Vi har implementeret en oversætter, men sproget kunne lige så godt have væ-

ret implementeret som en fortolker. Vores brugsmønstre ligger op til at sproget bruges som et konverterings/udregnings-værktøj, og ikke til traditionelle kompilerede programmer. Programmer kan ikke indgå med andre programmer, eller i et større system. Programmer i vores sprog, er derfor meget isolerede, og kan derfor, som udgangspunkt kun blive brugt i et begrænset omfang. En fortolker kunne udføre det nødvendige arbejde for brugeren, og fortolke sproget i et fortolker-miljø.

Return problemer

Vi har et stort problem, med henhold til return-kommandoer og if-else kæder. Vores oversætter kræver at funktioner har mindst ét return, i roden af funktionen. Hvis return-kommandoer står indeni if-else konstruktioner, bliver de ikke talt med af typetjekkeren. Derfor skal alle funktioner have et return i slutningen af en funktionserklæring. Men, dette medvirker at den genererede Javakode indeholder utilgængelig kode (unreachable code), og derved vil Java oversætteren ikke godtage vores output, som et lovligt Java program. Da det ikke er muligt at ignorere denne fejl i Java oversætteren, er der ikke noget vi kan gøre ved dette. Dette stammer fra en mangel i vores typetjekker, som burde have et komponent, som foretager *semantic analysis*, som kan detekere *unreachable code*. Brugeren vil derfor sidde i dødvande mellem vores oversætter og Java oversætteren, da det ene senarie bliver accepteret af den ene oversætter, men ikke af den anden, og omvendt. I listing 6.1 ses et eksempel på dette. Fjernes den sidste return-kommando, vil vores typetjekker give en fejl - omvendt, hvis man forsøger at oversætte dette program, vil Java oversætteren give fejl, da den sidste return-kommando, ikke er muligt at nå.

Oversætter target

Vores oversætter er implementeret ved at oversætte til Java-kode. Alternativt kunne vi have oversat til et lavere niveau - VM bytecode, et *intermediate language*, eller sågar assembly. Der er ikke noget i vores sprog som kræver at vi benytter Java. Vi har været begrænset af regler og strukturer i Java - f.eks. Javas scoperegler kan vi ikke undgå; derved har begrænsninger i Java været med til at forme vores sprog og dets regler. Havde vi oversat til f.eks. Java bytecode, ville vi have kunnet definere vores egne kontrolstrukturer, scoperegler, osv. Dette vil have været et større stykke arbejde, men vil dog også give os mulighed for at definere hver enkel del af sproget. Vi har mødt nogle problemer mellem vores sprog, og Java som vi ikke har kunnet håndtere. Her tænkes primært på return-problematikken, som beskrives senere. I bakspejlet, burde vi have overvejet vores target sprog nærmere. Eller have valgt et sprog, som ikke har de begrænsninger som vi møder i Java - f.eks. C.

```
1 dna function(int a){
2         if(a > 1){
```

```
3                       return ATT;
4               }
5               else{
6                       return AGG;
7               }
8               return ATG;
9       }
```

Listing 6.1: Eksempel på return-problematik

Brugerdefinerede datatyper

Vi beskrev i afsnit 1.5.3 hvordan vi ønskede at brugere skulle have mulighed for at definere egne datatyper. Dette er desværre ikke muligt i vores implementation. Brugeren er derfor begrænset til at benytte de indbyggede datatyper.

Deklarationer med tvunget tilskrivning

Ifølge vores semantik er det ikke lovligt at deklarere et variabel, uden samtidig at tilskrive en værdi. Dette er ikke et problem med tal, da vi kan sige *int a = 0;*, som en substitut for *int a;*. Men med hensyn til vores datayper, DNA, RNA, codon og protein har vi et problem. Hvordan skal vi tilskrive en tom værdi? Her burde vi have haft et tegn (token) til at beskrive en tom sekvens, således at vi kunne sige *dna seq = ø;*

6.2 Konklusion

I dette afsnit vil vi konkludere på projekt og tage stilling til hvor meget der blev opnået gennem forløbet, og herigennem vurdere om projektet nåede sit mål. Til at besvare dette tages der udgangspunkt i vores løsningskriterier, som blev lavet ud fra problemformuleringen i afsnit 1.7. Løsningskriterierne var som følger:

Bio typer

Sproget skal indeholde typer for DNA, RNA, proteiner og codon, disse skal kun kunne indeholde gyldige tegn og der skal kunne laves variabler af disse typer gennem deklaration. Dette er med til at sikre, at fx en sekvens af aminosyrer ikke kan konverteres til DNA. At begrænse typernes tegn, er med til at kunne give brugeren bedre fejlbeskeder.

Operationer

Sproget skal understøtte basale sekvens operationer, som konkatenering af DNA, RNA og aminosyre sekvenser, fjerne delsekvenser og finde komplementære- og omvendte-sekvenser for DNA samt finde startpositioner for delsekvenser i sekvenser. Endvidere skal sproget også understøtte konvertering mellem DNA, RNA og Proteiner.

Funktionalitet fra imperativ programmering

Ud over biotyperne og de dertilhørende operationer skal sproget også indeholde kontrolstrukturer som *for*, *while* og *if else*, samt typerne *int* og *bool* og datastrukturer som *arrays*. Grunden til at dette tages med er for at der også gives mulighed for at udtrykke algoritmer og behandle data på flere måder, frem for kun at kunne udfører operationer på biotyperne.

Til det første løsningskriterie, har vi i vores sprog typerne der bliver forslået her. Variabler der bliver erklæret af disse typer i vores sprog kan også kun indeholde de gyldige tegn for hver enkelt, hvis en erklæring bliver lavet, som indeholder ugyldige tegn, vil vores typetjekker, lavet ud fra vores typesystem i afsnit 3.3, melde fejl og programmet vil ikke blive oversat.

Til det andet løsningskriterie, kan vi i vores sprog udføre de basale sekvensoperationer, som bliver nævnt i løsningskriteriet, samt flere operationer der fremgår i afsnit 1.4.1 om det centrale dogme. For at beskrive hvordan disse operationer skulle fungere når de er implementeret i oversætteren er der i afsnit 3.2 givet en strukturel operationel semantik for de enkelte operationer, samt de andre konstruktioner i sproget. Specifikt til konvertering mellem DNA, RNA og Proteiner er dette i sproget lavet som et udtryk, as, der konverterer en sekvens af en type, om til en sekvens af en anden type.

Til det tredje løsningskriterie er der i sproget også mulighed for at erklære variabler af typerne int og $bool$. Endvidere indeholder sproget også de konstruktioner som er angivet i løsningskriteriet. Der er i sproget, som det er nu, ikke mulighed for at lave $arrays$, og derfor kan der være nogle algoritmer, der anvender denne datastruktur, som ikke er mulige at implementere i sproget. Ud over dette er det, som nævnt i diskussionen i afsnit 6.1, også nødvendigt, i en variabelerklæring, at tilskrive variablen en værdi. Dette er ikke altid hensigtsmæssigt hvis det er meningen at variablen først senere i koden skal tilskrives sin egentlige værdi.

Vi kan ud fra de ovenstående dokumentation af løsningskriterierne konkludere at projektet har nået sit mål, med blot nogle mangler. Manglerne bliver hovedsageligt udgjort af at der i det nuværende sprog ikke er mulighed for at erklære og bruge arrays. Dette har den konsekvens at der er nogle algoritmer, som gør brug af denne datastruktur, eksempelvis KMP (selv om den i forvejen er en del af sproget), som ikke er mulige at implementere i vores sprog.

Bibliografi

[1] *Programmering kommer til folkeskolen uden plan for efteruddannelse*. https://www.version2.dk/artikel/programmering-traeder-ind-i-fysiklokalet-68296. 2014. (Sidst set 13.02.2017).

[2] Martin C. Carlisle m.fl. „RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving“. I: *SIGCSE Bull.* 37.1 (feb. 2005), s. 176–180. ISSN: 0097-8418. DOI: 10.1145/1047124.1047411. URL: http://doi.acm.org/10.1145/1047124.1047411.

[3] *Scratch*. https://scratch.mit.edu/projects/editor/?tip_bar=getStarted. unknown. (Sidst set 13.02.2017).

[4] *Biology*. https://en.wikipedia.org/wiki/Biology. (Sidst set 08.03.2017).

[5] *Biotechnology*. https://en.wikipedia.org/wiki/Biotechnology. (Sidst set 08.03.2017).

[6] *Bioinformatics*. https://en.wikipedia.org/wiki/Bioinformatics. (Sidst set 08.03.2017).

[7] *Det centrale dogme*. http://denstoredanske.dk/Natur_og_milj%C3%B8/Genetik_og_evolution/Genetik/det_centrale_dogme. (Sidst set 12.03.2017).

[8] *The Central Dogma*. https://genius.com/Biology-genius-the-central-dogma-annotated. (Sidst set 12.03.2017).

[9] *DNA, RNA and protein – the Central Dogma*. http://science-explained.com/theory/dna-rna-and-protein/. (Sidst set 12.03.2017).

[10] Kim Bruun, Karen Helmig og Pia Birgitte Geertsen. *Grundbog i bioteknologi 1*. Gyldendal, 2010.

[11] *DNA to Amino Acids-Translation*. https://students.ga.desire2learn.com/d2l/lor/viewer/viewFile.d2lfile/1798/12708/dna-rna13.html. (Sidst set 15.03.2017).

[12] *Amino Acids*. http://virology.wisc.edu/acp/Classes/DropFolders/Drop660_lectures/SingleLetterCode.html. (Sidst set 15.03.2017).

[13] *How can one gene code for several proteins?* http://www.iecb.u-bordeaux.fr/index.php/en/news/99-plus-de-100-000-genes-dans-le-ble-environ-30-000-chez-lhomme-nouveaux-elements-pour-comprendre-comment-un-gene-peut-coder-pour-plusieurs-proteines. (Sidst set 18.03.2017).

[14] *Codons og læserammer*. http://www.biotechacademy.dk/Undervisningsprojekter/Gymnasiale-projekter/Bioinfo/1-Teori2/1-Codons. (Sidst set 15.03.2017).

[15] *Translation and Open Reading Frame Search*. http://bioweb.uwlax.edu/genweb/molecular/seq_anal/translation/translation.html. (Sidst set 16.03.2017).

[16] Mathieu Fourment og Michael R. Gillings. „A comparison of common programming languages used in bioinformatics“. I: *BMC Bioinformatics* 9.1 (2008), s. 82. ISSN: 1471-2105. DOI: 10.1186/1471-2105-9-82.

[17] James Tisdall. *Beginning Perl for bioinformatics*. "O'Reilly Media, Inc.", 2001.

[18] *Convert DNA to RNA*. http://resources.qiagenbioinformatics.com/manuals/clcmainworkbench/current/index.php?manual=Convert_DNA_RNA.html. (Sidst set 12.03.2017).

[19] *Open reading frame parameters*. http://resources.qiagenbioinformatics.com/manuals/clcsequenceviewer/current/index.php?manual=Open_reading_frame_parameters.html. (Sidst set 20.03.2017).

[20] *Sequence analysis*. https://en.wikipedia.org/wiki/Sequence_analysis. (Sidst set 13.03.2017).

[21] *Sequence alignment*. https://en.wikipedia.org/wiki/Sequence_alignment. (Sidst set 13.03.2017).

[22] *Biopython*. http://biopython.org/wiki/Biopython. (Sidst set 23.02.2017).

[23] *Introduction to Biopython*. http://biopython.org/DIST/docs/tutorial/Tutorial.html. (Sidst set 23.02.2017).

[24] *Python*. https://www.python.org/. (Sidst set 23.02.2017).

[25] *BioPerl*. http://bioperl.org/howtos/index.html. (Sidst set 24.02.2017).

[26] *BioJava*. http://biojava.org/wiki/BioJava:CookBook4.0/. (Sidst set 24.02.2017).

[27] *Knuth-Morris-Prat algorithm*. https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm. (Sidst set 12.03.2017).

[28] *Knuth-Morris-Pratt String Matching (Python Recipe)*. http://code.activestate.com/recipes/117214/. (Sidst set 13.03.2017).

[29] Jon Orwant, Jarkko Hietaniemi og John Macdonald. *Mastering Algorithms with Perl*. O'Reilly Media, 2011. ISBN: 978-1-4493-8587-3.

[30] Robert W. Sebesta. *Concepts of Programming Languages*. 11. udg. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2016. ISBN: 1-292-10055-9.

[31] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002. ISBN: 0-262-16209-1.

[32] *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*. http://www.antlr.org/papers/allstar-techreport.pdf. (Sidst set 26.05.2017).

[33] *Static vs. Dynamic Scoping*. https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/. (Sidst set 27.05.2017).

[34] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010. ISBN: 9780521197465.

[35] Dr. Ethan White. *Why Python?* This website is from an instructor teaching a course in programming for biologists. URL: http://www.programmingforbiologists.org/ (sidst set 20.02.2017).

[36] Michael Sipser. *Introduction to the Theory of Computation, First Edition*. 1997.

[37] Charles N Fischer, Ronald K Cytron og Richard J LeBlanc. *Crafting a compiler*. Addison-Wesley Publishing Company, 2009.

[38] *Incentive Pymol Software Package*. https://www.pymol.org/pymol. (Sidst set 12.03.2017).

[39] *MicroCal PEAQ-ITC*. http://www.malvern.com/en/products/product-range/microcal-range/microcal-itc-range/microcal-peaq-itc-range/microcal-peaq-itc/. (Sidst set 12.03.2017).

[40] *mMass - Open Source Mass Spectrometry Tool*. http://www.mmass.org/. (Sidst set 12.03.2017).

[41] *Automated experimental phasing with Crank*. http://ccp4wiki.org/~ccp4wiki/wiki/index.php?title=Automated_experimental_phasing_with_Crank. (Sidst set 12.03.2017).

Bilag

A Svar på spørgsmål

1. Hvilke programmer/værktøjer bliver de studerende undervist i at bruge til at løse bioteknologiske opgaver med og/eller hvilke er du selv blevet undervist i at bruge, hvis nogle?
   Vi er blevet undervist i MATLAB og Excel til at lave beregninger på reaktorer. Herudover har vi fået undervisning i Pymol, som er et program til visualisering af proteinstrukturer.

2. Hvilke programmer/værktøjer ender de studerende med at bruge og hvilke bruger du selv i dit arbejde?
   Jeg anvender primært specialudviklet software til min databehandling, heriblandt PEAQ-ITC, chirascan, mMass, CCP4i (en samlet pakke med programmer til proteinstrukturløsning), coot (et program til visualisering og justering af proteinstrukturer), CLC (et program til visualisering af DNA og proteinsekvenser). Herudover bruger jeg et program som hedder GraphPad prism til at samle og plotte mine data. Der er ikke nogen af de ovennævnte programmer der involverer programmering, og de har alle en GUI. Jeg har desuden anvendt programmet latex en del til projektskrivning.

3. Hvor meget tid bruger de studerende på at lære at programmere, hvis nogle af programmerne/værktøjerne indeholder brug af programmering? Hvis ja, anbefaler man studerende til selv at bruge tid, på at lære at programmere?
   Programmering er ikke umiddelbart en del af uddannelsen.

4. Kender du nogle specifikke områder inden for bioteknologi hvor brugen af programmering er særdeles nyttig, hvis der er nogle? Eksempelvis ved løsning af opgaver ved brug af Python, R eller MATLAB
   Der er en del som anvender R til statistiske beregning og databehandling, heriblandt beregning af gennemsnittet for flere datasæt, jeg har dog aldrig anvendt det.

5. Kender du nogle specifikke områder inden for bioteknologi hvor de eksisterende programmer/værktøjer ikke er tilstrækkeligt gode til at løse de opgaver der måtte

være, eller hvor du tænker at det kunne gøres lettere?
Nej jeg har været godt tilfreds med de programmer jeg anvender i mit arbejde.

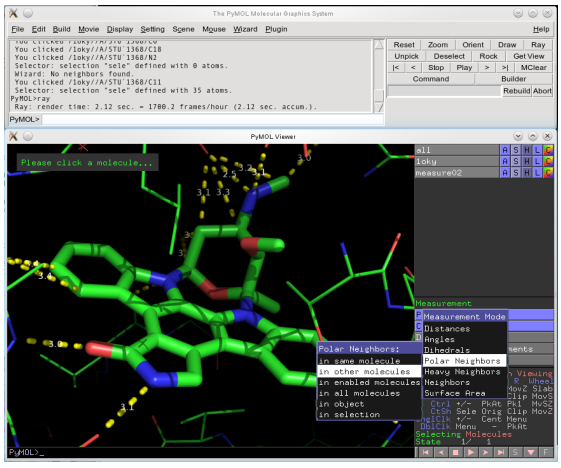
6. Hvordan repræsenterer man typisk data i en bioteknologisk sammenhæng? Er det eksempelvis i form af lister, tabeller, diagrammer eller andet?
   Data repræsenteres oftest i form af grafer og tabeller.

7. Gøres der meget brug af algoritmer inden for bioteknologi? Hvis ja hvordan kommer de forskellige algoritmer til udtryk i programmerne/værktøjerne, skriver i dem selv eller kan de kaldes ved navn og gives input? Eksempelvis ligesom x = Add(2, 2); x = 4
   Jeg har ingen erfaring med brug af algoritmer, udover at jeg mindes at de bruges i MATLAB.

8. Ved du hvilket programmerings paradigme (objekt-orienteret, imperativ, funktionel, osv.), som er populært, eller specielt brugbart indenfor faget?
   Jeg er ikke bekendt med fænomenet programmerings paradigme, så det kan jeg desværre ikke svare på.

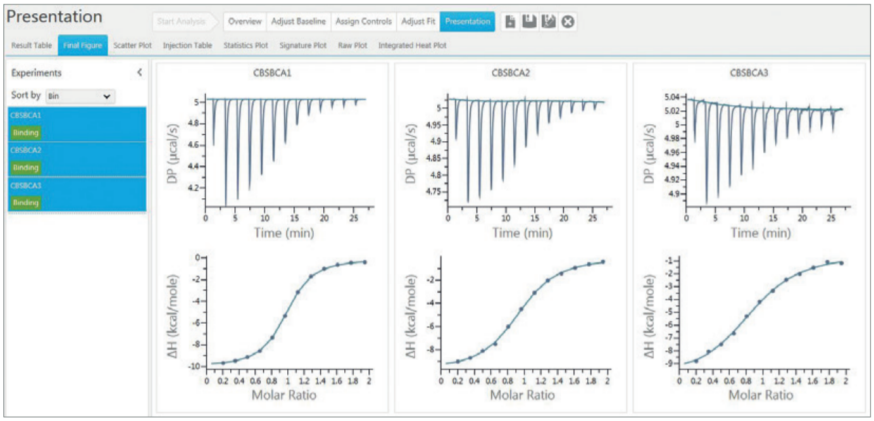
B Overblik over anvendte værktøjer

Pymol bruges til at visualisere og animere molekyler [38]. Et eksempel på Pymols interface, kan ses på figur 1.



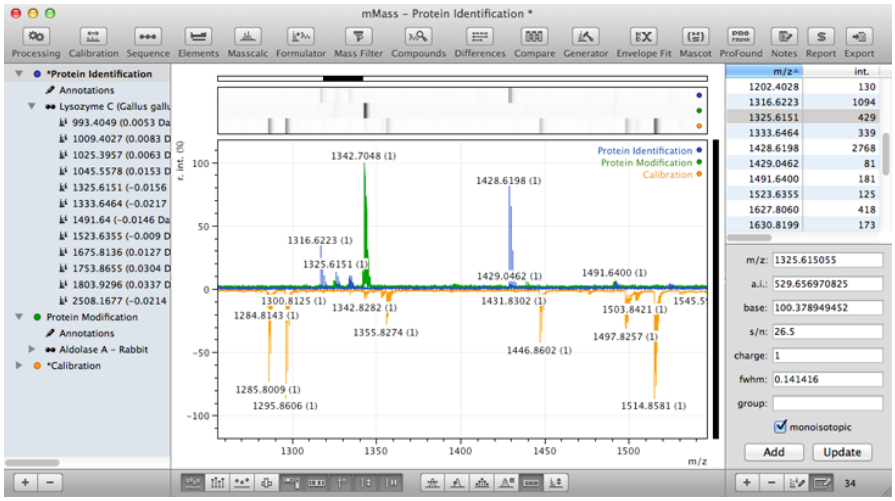
Figur 1: Interface for værktøjet Pymol.[38]

PEAQ-ITC er et fysisk måleværktøj, som har dertil hørende software[39]. Et eksempel på et interface, for det dertil hørende software, kan ses på figur 2.



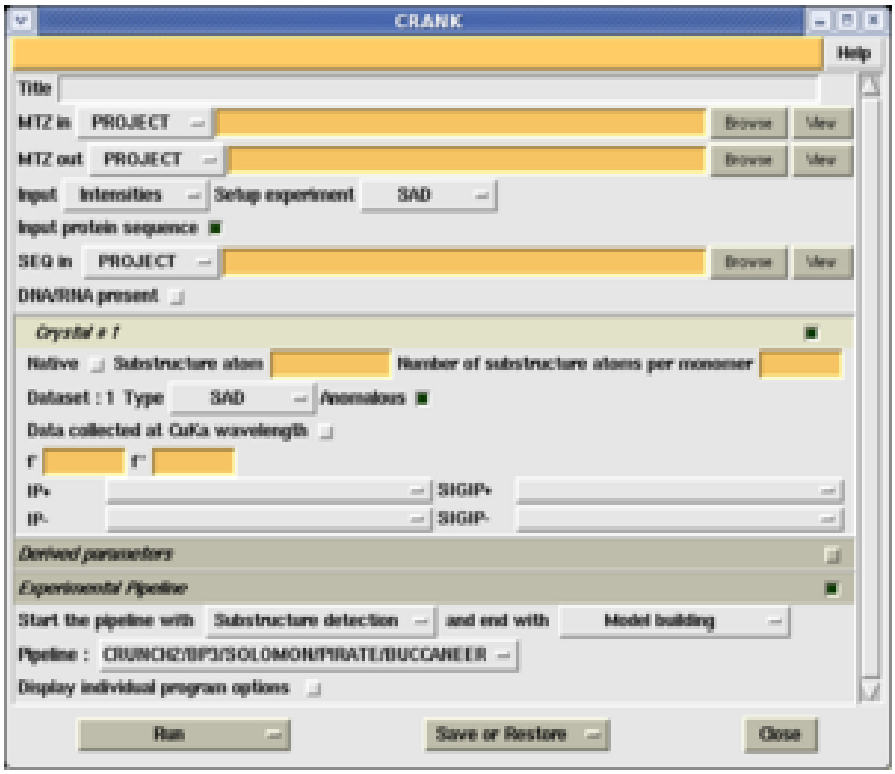
Figur 2: Interface for PEAQ-ITC.[39]

mMass er et OpenSource værktøj, som kan bruges til at mange forskellige ting. Bl.a. at processere bestemte data og korrigere det, generere molekylere ligninger og benytte online resurser til at identificere protein sekvenser[40]. Et eksempel på værktøjet kan ses på figur 3.



Figur 3: Interface for mMass.[40]

CCP4 er ikke et værktøj i sig selv, men en suite af værktøjer, som er specialiserede indenfor et vist område. **Crank** er et eksempel på sådan et program, som også gør brug af andre programmer. Crank tager forskellige typer af input, som fx data omkring bølgelængder og anvender det på protein sekvenser, for at bestemme proteinets struktur[41].



Figur 4: Interface for Crank.[41]