The MTIDD Firewall Language

Projektgruppe F603a

10. november 2003

AALBORG UNIVERSITET

Institut for Datalogi

Titel: The MTIDD Firewall Language

Tema: Sprog og oversættelse

Projektperiode: 3. februar - 30. maj 2003

Semester: F6S 2003

Gruppe: F603a

Gruppemedlemmer:

Kim Algreen

Kim R. Bille

Anders Hansen

Mads A. Jakobsen

Jakob Kirkegaard

Martin Maach

Jakob Skov-Pedersen

Vejleder:

Mikkel Christiansen

Oplagstal: 9

Rapport sideantal: 76

Appendiks sideantal: 60

Total sideantal: 131

Afsluttet: 30. maj 2003

Synopsis:

Dette projekt beskæftiger sig med udvikling af et specifikationssprog til pakkefiltre, samt design og implementation af en kompiler, der kan omsætte filterspecifikationer til repræsentationsformen Multi Terminal Interval Decision Diagrams (MTIDD).
En række eksisterende firewall specifikationssprog analyseres, hvorved problemer omkring skalérbarhed, testbarhed og effektivitet identificeres. Med udgangspunkt i analysen defineres et nyt specifikationssprog.
Der designes og implementeres endvidere en kompiler, som udfra filtre beskrevet i det designede specifikationssprog, kan generere MTIDDer. Disse optimerbare, muliggør offline test af filtre og reducerer kompleksiteten i pakkefiltreringen fra at være lineær i antallet af filterregler til konstant.
Kompileren implementeres i OCaml uden brug af lexer og parser værktøjer. Der implementeres en lexer, en recursive-descent parser, der genererer et abstrakt syntaks træ, en semantisk analysator, der verificerer typer og scope samt en kodegenerator, der opbygger en MTIDD repræsentation beskrevet ved en XML fil.
Igennem projektet er det lykkedes at definere et brugbart specifikationssprog til firewalls, samt at konstruere en velfungerende kompiler, der kan generere filtre repræsenteret ved MTIDDer.

AALBORG UNIVERSITY

Department of Computer Science

Title: The MTIDD Firewall Language

Theme: Language and Compilation

Time period: 3rd of February - 30th of May 2003

Term: F6S 2003

Group: F603a

Members:

Kim Algreen

Kim R. Bille

Anders Hansen

Mads A. Jakobsen

Jakob Kirkegaard

Martin Maach

Jakob Skov-Pedersen

Instructor:

Mikkel Christiansen

Number printed: 9

Number of pages: 76

Appendix pages: 60

Total pagecount: 131

Ended: 30th of May 2003

Abstract:

In this project a package filter specification language is developed. Furthermore a compiler for translating the filter specifications into Multi Terminal Interval Decision Diagrams (MTIDD) is designed and implemented.
Existing firewall specification languages are analyzed, whereby problems concerning scalability, testability and effectivness are identified. Based on the analysis a new specification language is defined.
Furthermore a compiler is designed and implemented, which based on filters specified in the designed language, is able to generate MTIDDs. These are easy to optimize, allow offline testing and reduces the complexity of the package filtering algorithm from linear in the amount of filter rules to a constant value.
The compiler is implemented in OCaml the use of lexer and parser tools. The parts designed and implemented are a lexer, a recursive-descent parser, which generates an abstract syntax tree, a semantic analyzer which validates types and scope, and a codegenerator, which generates the MTIDD representation, described in an XML file.
During the project, a useful firewall specification language is designed and an operational compiler, which is able to generate MTIDDs, is created.

Forord

Dette projekt er udarbejdet af gruppe F603a ved Institut for Datalogi på Aalborg Universitet. Projektet er udarbejdet i F6S-perioden, der strækker sig fra den 3. februar til den 30. maj 2003.

Formålet med F6S-projektet er at få den tilførte tekniske viden fra PE-kurserne indført i den problemorienterede og projektorganiserede indlæringsform gennemført i grupper. Målgruppen for dette projekt er studerende og andre interesserede med et teknisk niveau svarende til gruppens eget. Projektets formål er mere specifikt beskrevet i studieordningen for F6S, hvilket er vist herunder.

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

I rapporten er alle litteraturhenvisninger udført sådan at forfatter og årstal står i kantet parentes - eksempelvis [Knu97]. Placering af henvisningen afgør, hvad der specifikt henvises til. Står kildehenvisningen før et punktum, refereres der til den foregående sætning - står henvisningen efter et punktum, refereres til hele det foregående afsnit. Referencer til den medfølgende cd-rom skrives som (◎ *hjemmesiden* `homepage/index.html`)

Rapporten indeholder mange engelske fagudtryk og forkortelser, som anvendes i forbindelse med danske ord. Sådanne danske og engelske ord, sammensættes ikke. Eksempelvis skrives "source sproget", og ikke "sourcesproget".

I rapporten er der lagt vægt på analyse og metodik af systemet, fremfor implementering. Dette gør, at det kun er de vigtige dele af implementeringen, der er fremhævet. For nærmere inspektion af kildekoden, henvises der til den vedlagte cd-rom, der indeholder rapport og kildekode.

Aalborg d. 30. maj 2003

Kim Algreen	Kim R. Bille	Anders Hansen
Mads A. Jakobsen	Jakob Kirkegaard	Martin Maach
	Jakob Skov-Pedersen	

Indhold

List of Corrections

KAPITEL 1

Indledning

Formålet med dette kapitel er kort at introducere problemstillingen, som behandles i projektet.

I takt med Internettets stadig stigende udbredelse er sikkerhed et aspekt, som nødvendigvis må tages alvorligt af alle, der har et behov for at benytte nettet. Selv små virksomheder og institutioner, der bliver koblet på Internettet, besidder indtil flere mekanismer til opretholdelse af en specifik sikkerhedspolitik[1]. Den primære mekanisme til kontrol af trafik mellem netværk er en firewall.

En firewall udgør en barriere mellem to eller flere netværk, og sørger for gensidig beskyttelse imellem disse. Dette sker igennem evaluering af netværkstrafikken imod en sikkerhedspolitik, der eksempelvis kunne beskrive hvilken kommunikation der må foregå "ud af huset", eller hvorledes virksomhedens samarbejdspartnere kan tilgå virksomhedens netværk.

Den første generation af firewalls blev udviklet i 1985 og bestod af simple pakkefiltre, der opererede på netværkslaget. I slutningen af 1980erne udvikledes anden generation af firewalls. Disse virkede som almindelige pakkefiltre, men kunne også filtrere på transportlaget. Dette muliggjorde, at pakker kunne valideres udfra, om de var forbindelsesforsøg eller dele af en eksisterende forbindelse (stateful inspektion). Tredie generation af firewalls blev udviklet i begyndelsen af 1990erne og introducerede muligheder for, at netværkstrafikken kunne valideres på baggrund af de applikationsprotokoller, der benyttedes. I 1994 blev det første kommercielle produkt frigivet, som baserede sig på dynamiske pakkefiltre. Disse filtre udemærkede sig ved at kunne ændre reglerne løbende, hvorved der kunne opnås større fleksibilitet. [Cis02, p. 3.1-3.2]

På trods af, at firewalls siden de første primitive pakkefiltre, har undergået en stor udvikling indenfor områder som stateful inspektion, gennemsigtighed og effektivitet, synes selve konfigureringsdelen af halte bagefter. Dels at implementere en funktionel firewall, samt at gennemskue virkemåden af en allerede implementeret firewall, kan være en krævende opgave.

Konfiguration af firewalls foregår i mange tilfælde i et regelbaseret scriptsprog, hvis læsbarhed er lav pga. manglen på abstraktionsniveauer. Dette udgør en potientiel sikkerhedsrisiko, idet en korrekt specifikation og implementation af denne, er den vigtigste faktor for en firewalls funktionalitet. Desuden er filterspecifikation i dens nuværende form heller ikke befordrende for hverken verifikation eller optimering.

Dette projekt beskæftiger sig derfor med analyse af de muligheder, der ligger i at udvikle et programmeringssprog til specifikation af en firewall, for derigennem at kunne opnå optimerede filtre, større sikkerhed som følge af gennemskuelig konfigurering samt portabilitet, så samme scripts kan benyttes på forskellige firewallsystemer. Dette leder frem til projektets initierende problem.

Hvorledes kan sikkerhed, effektivitet og skalérbarhed forbedres for eksisterende firewalls?

[1] En sikkerhedspolitik er et abstrakt udtryk for en række metoder, til at holde risiko og konsekvens af sikkerhedsrelaterede hændelser på et niveau, som er acceptabelt for den pågældende virksomhed eller institution.

KAPITEL 2

Problemanalyse

I dette kapitel vil der blive givet en kort beskrivelse af de forskellige typer af firewalls, samt gennemgået eksempler på den eksisterede syntaks for filterspecifikation. Dette sker med henblik på opstilling af en problemformulering for projektet.

2.1 Beskrivelse af firewalls

Netværksfirewalls bruges til at kontrollere adgangen til et netværk af computere, fra et andet netværk. Brugen af firewalls øges i takt med Internettets udbredelse og er idag et vigtigt værktøj for systemansvarlige, i de fleste større organisationer.

Indførsel af firewalls sker med henblik på at beskytte data (fortrolighed, integritet og tilgængelighed) og ressourcer (tilgængelighed og undgå misbrug) på et givent netværk.

Selvom en firewall ikke er den endegyldige løsning på alle netværksrelaterede sikkerhedsproblemer, er der sket en stor udbredelse af firewalls. Dette finder sin begrundelse i, at en firewall giver mulighed for central filtrering af trafikken til og fra det beskyttede netværk. Det er derigennem muligt at opretteholde en specifik sikkerhedspolitik.

I det følgende gennemgås kort de fire generationer af firewalls, som er blevet udviklet siden midten af 1980erne. Der vil løbende blive henvist til elementer fra TCP/IP protokolsuiten, som er nærmere beskrevet i appendiks A på side 77.

2.1.1 Pakkefiltre

En firewall, opbygget som et pakkefilter, analyserer trafik baseret hovedsageligt på netværks- og transportlaget. Hver pakke analyseres udfra dens header, uden hensyntagen til foregående pakker, som er modtaget (stateless filtrering). Filtreringen foregår ved at sammenligne hver enkelt header med en liste af regler, med henblik på at afgøre om pakken skal accepteres eller afvises. Et filter defineres i den sammenhæng, som en sekvens af regler.

Pakkefiltre varierer efter hvilke header informationer samt hvilke pakketyper de kan håndtere. I det følgende beskrives en række parametre, som typiske pakkefiltre tillader filtrering på.

Udfra netværkslaget, kan der filtreres på informationer omkring source og destination IP adresser, hvilken type service pakken kræver (Type of Service - eks. maximize throughput eller minimize delay) eller hvilken transportprotokol, der benyttes. Generelt kan alle felter i IP headeren danne udgangspunkt for filtrering.

Transportlaget giver mulighed for filtrering udfra felter, i de respektive transportprotokollers headere. Dette vil nærmere bestemt sige

Transmission Control Protokollen (TCP) kan filtreres på source og destination port, flag i headeren (eksempelvis SYN og FIN) og andre options (eksempelvis MSS, Maximum Segment Size).

Uniform Datagram Protokollen (UDP) kan, som TCP protokollen, filtreres på de benyttede porte. Derudover kan der filtreres på længden af UDP pakker.

Internet Control Message Protokollen (ICMP) kan filtreres på dens type og kode[1].

Der findes en række andre transportprotokoller, men som for TCP, UDP og ICMP gælder, at felter i headerne kan danne udgangspunkt for filtrering.

Pakkefiltre implementeres som regel i selve operativsystemet (kernelspace). Selvom det ikke er en nødvendighed, er det den typiske måde indenfor Unix og lignende systemer. [Ste94]

Principielt kan et pakkefilter også filtrere på informationer fra andre lag end netværks- og transportlaget, eksempelvis linklaget. Dette er således ikke de specifikke lag, der definerer et pakkefilter, men derimod at filtreringen sker uden hensyntagen til konteksten.

2.1.2 Stateful inspektion

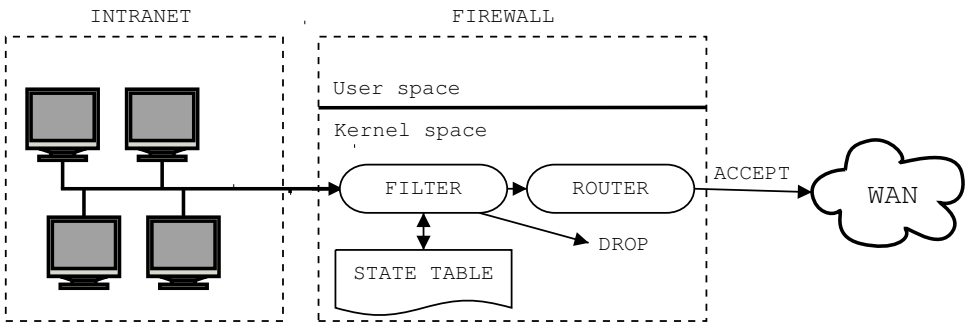
I foregående afsnit blev beskrevet en type firewalls, som ved filtering udelukkende betragter den enkelte pakke, uden hensyntagen til den givne kontekst af tidligere modtagne pakker. Denne type af firewalls kaldes stateless firewalls, idet der ikke eksisterer et tilstandsbegreb. I det følgende betragtes kategorien af firewalls, kaldet stateful firewalls, der udover at benytte headere i den given pakke, også vedligeholder informationer om tidligere modtagne pakker.

Tilstanden af en given forbindelse mellem to maskiner, vil typisk basere sig på informationer opsamlet på transportlaget. TCP protokollen, som er en forbindelsesorienteret transportprotokol, indeholder mange informationer omkring tilstanden af forbindelsen. Ligeledes eksisterer et veldefineret tilstandsdiagram for TCP protokollen, som beskriver de mulige tilstande forbindelsen kan være i [Pos81].

For TCP protokollen, vil en pakke typisk betragtes som en del af en eksisterende forbindelse, såfremt den følger et legitimt threeway handshake [Pos81], samt at pakkens sekvens- og ACK numre passer ind i den hidtidige række for forbindelsen.

For de forbindelsesløse transportprotokoller UDP og ICMP er det ikke i samme grad muligt at bestemme tilstanden af en forbindelse. I afsnit 2.1.4 på næste side, beskrives hvordan tilstande kan håndteres for disse protokoller.

Uafhængigt af den benyttede transportprotokol og hvorledes den vedligeholder tilstanden af forbindelsen, skal firewallen opbevare en række informationer om forbindelsen. Det sker typisk ved, at der vedligeholdes en tilstandstabel, der indeholder varierende information afhængig af den givne transportprotokol. Dette er vist på figur 2.1.



Figur 2.1: Princippet i et pakkefilter (her vist som stateful), hvor kommunikationen kun er vist i en retning. For en stateless firewall, benyttes ingen tilstandstabel, men ellers er princippet det samme.

De første udgaver af stateful firewalls havde mange af de samme fordele og ulemper, som et

[1] Kodefeltet uddyber beskeden givet af typefeltet

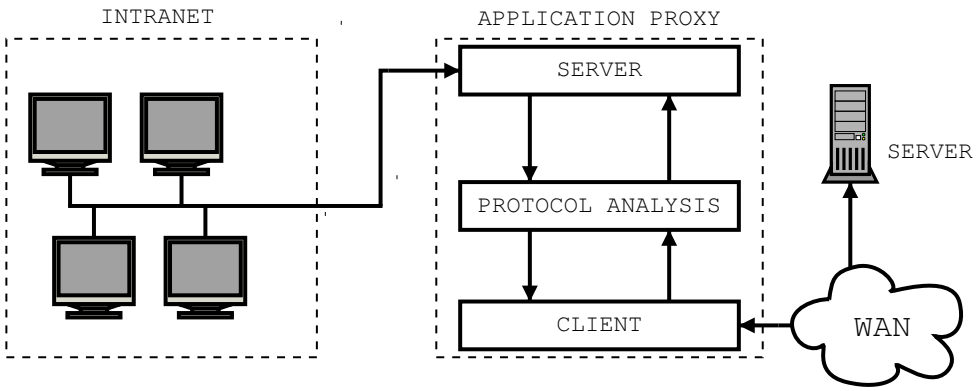
stateless pakkefilter. Selvom denne type firewall introducerede muligheden for validering af pakker udfra forbindelsens tilstand, var de dog kun marginalt mere sikre. Dette skyldes, at så snart en forbindelse var accepteret, blev der ikke foretaget yderligere valideringer af de pakker som udgjorde forbindelsen. [Cis02, p. 3-17]

Stateful firewalls benyttes idag, kombineret med pakkefiltre, i de fleste firewalls samt til NAT[2]. Som for stateless firewalls implementeres stateful firewalls også i selve operativsystemet (kernelspace). [Cis02, p. 3-16]

2.1.3 Applikationsproxy

Idéen i applikationsproxyteknologien er at validere kommunikationen på applikationslaget, før en forbindelse tillades. Udover at vedligeholde informationer omkring forbindelsestilstanden og sekvensinformation, kan firewallen også validere på data, som kun findes på applikationslaget - eksempelvis kodeord eller anmodninger om diverse services.

En proxy fungerer som et mellemled mellem en klient og en server, og indeholder således selv både en proxy klient (som kontakter den eksterne server) og en proxy server (der servicerer den interne klient). Dette er vist på figur 2.2.



Figur 2.2: Princippet i en applikationsproxy.

I modsætning til pakkefiltre og stateful firewalls, opererer applikationproxyen i userspace. På baggrund af dette, samt det faktum, at pakker skal traversere et større antal protokollag og flere protokolstakke, betragtes applikationsproxyer generelt som en langsommere type firewall end de tidligere nævnte typer. Afhængigt af formålet, kan dette dog opvejes, idet applikationsproxyen tilbyder en række valideringsmuligheder, som ikke findes for simple pakkefiltre.[Cis02, p. 3-22]

2.1.4 Dynamiske pakkefiltre

Dynamiske pakkefiltre er i stand til løbende at ændre reglerne. Dette er specielt velegnet til at give begrænset adgang til UDP og ICMP protokollerne. Dette sker ved, at firewallen opretholder en tabel med virtuelle forbindelser.

Når en UDP pakke sendes fra en intern maskine til en ekstern via en firewall, associeres pakken med en virtuel forbindelse. Hvis UDP pakken genererer et svar som sendes tilbage, oprettes

[2]Network Address Translation. Benyttes eks. til at få et helt lokalt intranet til at fremstå som en ekstern IP adresse.

en virtuel forbindelse, som kan benyttes i en kortere periode. For ICMP pakker kunne det eksempelvis være, at der i en kort periode efter en type 8 (echo) ICMP pakke, blev tilladt indgående type 0 (echo reply) pakker.

For applikationsprotokollerne kan et dynamiske pakkefilter benyttes til at give interne Domain Name Service (DNS) servere adgang til at kontakte eksterne DNS servere. Dette gøres nødvendigvis uden ukritisk at tillade UDP trafik på alle porte. For aktiv File Transfer Protokol (FTP) trafik, kunne der tillades indgående trafik på port 20 (FTP data) fra en given host, så længe der eksisterede en udgående forbindelse på port 21 (FTP control) til den givne host. [Cis02]

De dynamiske pakkefiltre betragtes undertiden som connection tracking firewalls.

2.1.5 Hybrid firewalls

Langt de fleste firewall produkter er kombinationer af de ovennævnte fire typer af firewalls. Dette skyldes, at de hver især indeholder en række fordele, som finder sin anvendelse i forskellige tilfælde. I det følgende afsnit betragtes specifikationen af en række firewalls, som kombinerer simpel pakkefiltrering med stateful inspektion.

2.2 Sprog til firewalls

For at kunne indføre en firewall i et netværk, skal det være muligt at kunne specificere dennes opførsel i forhold til trafikken gennem den. Denne konfiguration sker med nuværende firewalls oftest enten via et program, hvor regler indtastes en efter en til alle er opsat i firewallen, eller ved at lave en konfigurationsfil, som kan indlæses i firewallen. Sådan som situationen er, benytter hver type firewall sit eget sprog til konfiguration. Disse sprog er sjældent indbyrdes kompatible.

Indledningvis foretages en række teoretiske betragtninger omkring opgaven, som et pakkefilter skal løse, hvorefter der gennemgås tre eksempler på specifikationssprog for firewalls.

2.2.1 Specifikation af pakkefiltre

Det grundlæggende mål i forbindelse med specifikation af pakkefiltre er, at associere forskellige mængder af pakke headere med en eller flere handlinger (permit, deny etc.), samt definere hvad der som udgangspunkt skal være handlingen for pakker.

I det følgende betegner H en endelig mængde (størrelse n) af alle mulige headere, mens P betegner en endelig mængde (størrelse m) af alle mulige handlinger.

$$H = \{h_1, h_2, \ldots, h_n\} \tag{2.1}$$
$$P = \{p_1, p_2, \ldots, p_m\} \tag{2.2}$$

Betragtes eksempelvis den fiktive protokol X, hvor pakkerne har headere af længde tre og disse ønskes mappet til de to handlinger Permit og Deny, fås H_X og P_X.

$$H_X = \{000, 001, 010, 011, 100, 101, 110, 111\} \tag{2.3}$$
$$P_X = \{Permit, Deny\} \tag{2.4}$$

En regel r er i den forbindelse defineret ved en mængde, bestående af en delmængde af mulige headere η samt en eller flere handlinger π.

$$r = \{\eta, \pi\} \text{ , hvor } \eta \in \mathcal{P}(H) \text{ og } \pi \in \mathcal{P}(P) \tag{2.5}$$

En regel for et pakkefilter, der håndterer protokol X, kunne da være:

$$r_X = \{\{001, 011\}, \{Permit\}\} \tag{2.6}$$

Et filter defineres da som en endelig mængde (størrelse l) af regler:

$$f = \{r_1, r_2, \ldots, r_l\} \tag{2.7}$$

Et filter der filtrerer pakker af protokol X, kunne da være:

$$f_X = \{\{\{001, 011\}, \{Permit\}\}, \{\{100, 000\}, \{Deny\}\}\} \tag{2.8}$$

Mængden af regler, vist ved ligning 2.7, kan ligeledes betragtes som en partiel funktion, der mapper en header $\eta \in H$ til en række handlinger $\pi \in \mathcal{P}(P)$. Funktionen f, kan da beskrives ved forskriften $f : H \hookrightarrow \mathcal{P}(P)$. [CF02, p. 6]

Funktionen f er partiel, idet ikke alle header kombinationer nødvendigvis er specificeret i filtret. Derfor kræves en definition af en standardhandling, som benyttes i forbindelse med uspecificerede headere.

Det bemærkes endvidere, at funktion f ikke nødvendigvis er injektiv[3], idet der kan defineres forskellige handlinger for de samme header kombinationer. Det kaldes overlap og kan håndteres på forskellig vis.

Efter håndtering af standardhandling samt overlap, fås et endeligt filter, der således er en total og injektiv funktion, der mapper enhver header til en mængde af handlinger.

2.2.2 Specifikationssprog

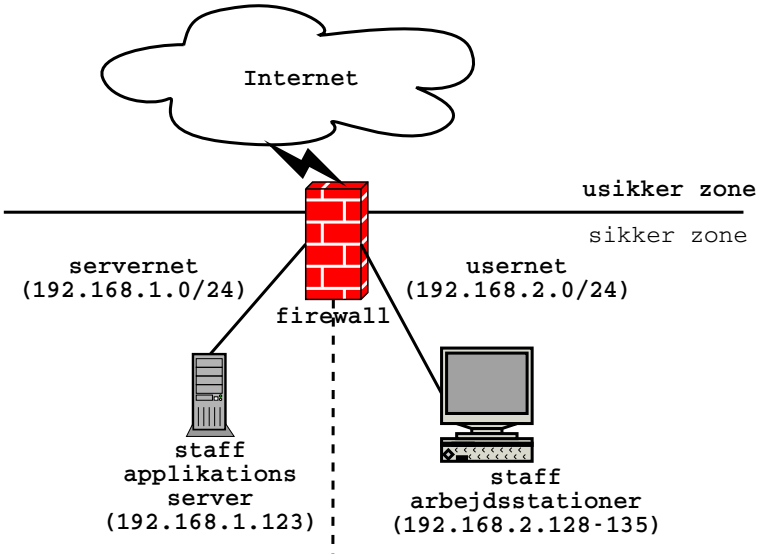
I forbindelse med at skulle designe et nyt sprog til at beskrive firewalls, undersøges fordele og ulemper ved de eksisterende sprog. For at gøre dette, beskrives i følgende afsnit tre forskellige sprog til specifikation af firewalls.

Eksempel 1 For at vise hvordan sprogene fungerer, er der benyttet et gennemgående eksempel, baseret på netværket vist på figur 2.3 på næste side. Netværket består af to interne net (usernet og servernet), samt adgang til Internet. Netværket skal sikres efter følgende politik:

- Det skal være muligt for alle arbejdsstationer i gruppen staff at arbejde på deres server på servernettet. Altså skal der kunne oprettes forbindelse fra staff til staff server.
- Det skal være muligt for staff arbejdsstationer at oprette forbindelser til webservere på Internettet (HTTP protokollen, port 80), samt at synkronisere filer med eksterne servere (rsync protokollen, port 873).
- Al anden trafik mellem de interne net og mellem Internet og de interne net skal filtreres bort. Det inkluderer også forsøg på at oprette forbindelser i modsat retning af ovenstående.

■

[3] En funktion er injektiv, såfremt ethvert element fra definitionsmængden højst mapper til et element i værdimængden.



Figur 2.3: Eksempelnet benyttet til sammenligning af firewall sprog. (/24 svarer til netmaske 255.255.255.0).

2.2.3 Cisco

Cisco IOS Access Control Lists (herefter Cisco) [Sed01] er et af de mest udbredte sprog til specifikation af firewalls. Cisco er den eneste af de analyserede sprog, der ikke tillader stateful filtrering.

Eksempel

Filtret fra eksempel 1, vil i Cisco tage sig ud som vist ved kildekode 2.1.

Kildekode 2.1 Eksempel på filterdefinition i Cisco.

```
1 access-list 101 permit ip 192.168.2.128 0.0.0.7 host 192.168.1.123
2 access-list 101 permit ip host 192.168.2.123 192.168.2.128 0.0.0.7 established
3 access-list 101 permit tcp 192.168.2.128 0.0.0.7 any eq www
4 access-list 101 permit ip 192.168.2.128 0.0.0.7 any eq 873
5 access-list 101 permit ip any gt 1024 192.168.2.128 0.0.0.7 established
```

Der skal her bemærkes, at idet dette sprog kun tillader beskrivelse af stateless filtrering, anvendes i stedet nøgleordet established. Established betyder, at en pakke enten skal have sat ACK eller RST. Dermed sikres, at der ikke kan oprettes en forbindelse i den pågældende retning. Generelt om opbygningen af regler i Cisco kan siges, at de altid begynder med access-list <listenummer>, her i eksemplet er valgt listenummeret 101. Herefter angives hvorledes pakken skal behandles hvis reglen opfyldes. Permit bruges, hvis pakken skal sendes videre, mens deny bruges, hvis den skal filtreres bort. Efter permit/deny angives hvilke protokoller, der er lovlige under reglen. I eksemplet godtager alle regler IP pakker, der kunne også have været krav om for eksempel UDP, TCP eller ICMP. Efter den valgte handling, angives IP adressen på afsenderen, source IPen, her anvendes en maske for at muliggøre permit af et antal source IPer samtidig. Hvis der er et krav til portnummeret, bliver dette krav skrevet lige efter masken. Til sidst skal destination IPen fastlægges, her bruges også en maske og eventuelt et krav til portnummeret. [Sed01, Kapitel 2]

I forbindelse med opstilling af regler i Cisco er det muligt at specificere regler, der overlapper. Det giver sig eksempelvis udtryk i, at den samme IP adresse tillades et sted i filtret, mens den afvises senere hen. Dette håndteres ved, at når en pakke skal passere en firewall eller router kodet i Cisco, løbes reglerne igennem fra toppen og ned, indtil der findes en regel der passer på pakken. Så snart der er fundet en regel, der matcher, behandles pakken efter den givne handling og resten af reglerne i filtret ignoreres.

Denne effekt kan give anledning til problemer, men kan også benyttes som et værktøj i forbindelse med filterspecifikation. Der kan opstå problemer, såfremt der overses en enkel regel i begyndelsen af filtret, som afviser en hel række regler længere nede i filtret. Omvendt kan overlap benyttes til at minimere filtre - et eksempel på dette er vist i kildekode 2.2.

Kildekode 2.2 Eksempel på overlap i Cisco, hvor 192.168.1.1 tillades og afvises.

```
1 access-list 102 permit ip host 192.168.1.1 any
2 access-list 102 deny   ip 192.168.1.0 0.0.0.255 any
```

Vedligeholdelse

Opbygning og vedligeholdelse af et pakkefilter i Cisco foregår enten ved indtastning af de enkelte regler én for én, eller ved upload af hele listen til firewallen ved hjælp af TFTP (Trivial FTP). Ved indtastning af enkelte regler, er der ikke mulighed for at slette en enkelt linie, hvorfor normal praksis er, at uploade hele listen. Under nogle versioner af Cisco slettes den gamle liste ikke, hvorved de nye regler appenderes.

Sammenfatning

Cisco er simpelt sprog, der er forholdsvis nemt at lære. Dette er en fordel ved mindre filtre. Ved filtre med mange regler, bliver det til gengæld hurtigt svært at bevare overblikket, over hvordan filtret faktisk filtrerer pakker. Dette skyldes, at rækkefølgen af reglerne er betydningsfuld, hvorved en enkelt forkert regel i begyndelsen af filtret kan afvise en hel række regler senere i filtret. Derudover er vedligeholdelse af et filter besværligt og sproget kan kun beskrive filtre til stateless firewalls.

2.2.4 IP Filter

IP Filter [nfp03] er et gratis firewall system, som følger med i flere af BSD operativsystemerne som standard, derudover fås IP Filter til en række andre UNIX implementationer.

Det specielle ved filtre skrevet i IP Filter er, at reglerne læses oppefra, men det er den sidste regel, der har størst prioritet. Kildekode 2.3 viser et simpelt filter, som tillader al indgående netværkstrafik.

Kildekode 2.3 Simpelt eksempel på en filterdefinition i IP Filter

```
1 block in  all
2 block in  all
3 pass  in  all
```

Eksempel

Filtret fra eksempel 1, vil i IP Filter se ud som i 2.4. Det ses, at al indkommende trafik, som

Kildekode 2.4 Eksempel på filterdefinition i IP Filter.

```
1 block in all
2 pass in from 192.168.2.128/29 to 192.168.1.123/32 keep state
3 pass in proto tcp from 192.168.2.128/29 to any port = 80 keep state
4 pass in proto tcp/udp from 192.168.2.128/29 to any port = 873 keep state
```

standard bliver blokeret. Derefter tillades al trafik mellem de to interne netværk, såfremt en computer på adressen 192.168.2.128/29 indleder forbindelsen. "Keep state" betyder, at firewallen tillader serveren at svare på klienternes forespørgsler, selv om der ikke er en eksplicit regel, der beskriver dette. De to sidste linier lader computere på 192.168.2.128/29 kommunikere med andre computere på port 80 og port 873.

IP Filter bliver altid læst helt igennem, med mindre man bruger det reserverede ord quick. Quick får IP Filter til at stoppe gennemlæsningen og bruge den nuværende regel på pakken, ifald den passer. Derved opnåes samme opførsel med hensyn til overlap, som et Ciscofilter beskrevet i afsnit 2.2.3 på side 13. Det er således muligt specifikt at vælge hvordan overlap skal håndteres.

Vedligeholdelse

Hvis det er et filter med mange regler, har IP Filter samme problem som filtrene beskrevet i afsnit 2.2 på side 11. Det er svært at få overblik over reglerne filtret benytter. Til gengæld er dette sprog lettere at forstå, da syntaksen ligger tæt på noget der ligner engelsk.

Sammenfatning

IP Filter er let at lære og giver brugeren mange muligheder for konfiguration af firewallen. Sproget er designet til brug for både stateless og stateful firewalls, og dermed er der mulighed for en bedre sikkerhed, end den beskrevet om Cisco i afsnit 2.2.3 på side 13. Desværre er det, af samme gund som for filtre skrevet i Cisco, besværligt at vedligeholde filtre skrevet i IP Filter

2.2.5 High Level Firewall Language

High Level Firewall Language (HLFL) [hlf03] er et generaliseret sprog til, på relativt højt niveau, at beskrive filtre. Siden kan det beskrevne filter konverteres til brug i en konkret firewall, f.eks. Cisco eller IP Filter. Da HLFL kompileres til den endelige syntaks, er det er muligt at lave abstraktioner såsom symbolsk udskiftning.

Sproget benytter ASCII tegn, til at gøre den tekstuelle repræsentation grafisk sigende. Kildekode 2.5 på den følgende side viser et simpelt filter i HLFL, som tillader TCP trafik mellem Peter og Per, men ikke nogen anden trafik. Symbolet "`<->`" betegner en simpel forbindelse, hvorimod "`<=>>`" betegner en stateful forbindelse, der kun kan initieres fra venstre mod højre.

Kildekode 2.5 Simpelt eksempel på en filterdefinition i HLFL

```
1  tcp (Peter) <-> (Per)
2  any (any) X (any)
```

Eksempel

Kildekode 2.6 viser filtret fra eksempel 1 implementeret i HLFL. Linier der starter med "%' eller "#" er blot kommentarer.

De første to definitioner er blot standard porte for to services i TCP/IP protokolsuiten (linie 2 og 3). De resterende defininitioner benyttes til beskrivelse af netværkets opbygning (linierne 5-17).

Kildekode 2.6 Eksempel på filter i High Level Firewall Language.

```
1  % Standard ports
2  define http 80
3  define rsync 873
4
5  % Outgoing port traffic to allow
6  define stafftcpout http,rsync
7  define staffudpout rsync
8
9  % Usernet
10 define usernet 192.168.2.0/24
11 % servernet
12 define servernet 192.168.1.0/24
13
14 % Staff computers on the usernet
15 define staff 192.168.2.128/29
16 % Internal application server for staff
17 define staffserver 192.168.1.123
18
19 # allow staff access to application server
20 tcp (staff) <=>> (staffserver) on any
21 # No other communication between usernet servernet
22 all (usernet) X (servernet) on any
23 # Allow staff access to internet using web and rsync
24 tcp (staff stafftcpout) <=>> (any)
25 udp (staff staffudpout) <=>> (any)
26 # Deny all else
27 all (any) X (any)
```

Efter beskrivelse af netværkets opsætning begynder beskrivelsen af reglerne. Første regel tillader TCP forbindelser, startet fra en brugercomputer til den interne server, uanset netværksinterface (linie 20). Næste regel blokerer al anden trafik mellem de to interne netværk (linie 22). De næste to regler, tillader brugercomputere at oprette, og bruge forbindelser ud på Internettet på visse porte (linie 24 og 25). Til sidst blokeres al resterende trafik (linie 27).

Eksemplet benytter definitioner, til både at lette forståelsen af reglerne, samt til at gøre det

nemmere, at vedligeholde filtret hvis opbygningen af netværket ændres.

Da HLFL i sidste ende blot er et lag ovenpå de eksisterende firewall sprog, vil overlap imellem regler blive håndteret, som de bliver i de underliggende sprog.

Vedligeholdelse

Abstraktionsmekanismerne i HLFL, samt muligheden for at skrive kommentarer i koden, gør at det er nemmere, at se hvad de enkelte regler gør, desuden gør definitioner det muligt at justere på netværksopsætningen af en opsat firewall. Ved firewalls med mange regler kan det dog være svært at bevare overblikket over koden, da det er vigtigt i hvilken rækkefølge de enkelte regler står.

Sammenfatning

Syntaksen af HLFL er meget nem at lære og forstå. Da sproget ikke er designet til en specifik platform, er der mulighed for at udvide det. Det er positivt, at der er mulighed for opdeling af filtre i flere forskellige filer, men da reglerne i et filter skal være i rækkefølge, kan disse to egenskaber gøre at et samlet filter bliver uoverskueligt. Da HLFL indeholder abstraktioner, er det dog mere overskueligt end sprogene der udelukkende anvender regler, hvorved det også bliver mere vedligeholdelsesvenligt.

2.3 Algoritmisk kompleksitet

Dette afsnit omhandler kompleksiteten i de filtre, som programmeres med Cisco, IP Filter og HLFL. Det antages i denne analyse, at mængden af informationer, der skal uddrages fra headere, er begrænset.

Algoritmisk kompleksitet i Cisco

Når en pakke skal passere en router kodet i Cisco, løbes reglerne igennem fra toppen og ned, indtil der findes en regel, der passer på pakken. Så snart der er fundet en regel der passer på pakken, kigges der ikke på flere regler, derfor kan rækkefølgen af reglerne betyde om visse pakker bliver videresendt eller stoppet. Hvis der ikke findes nogle regler i filtret, der passer på pakken, sendes den ikke videre. Denne sekventielle gennemløben af reglerne, betyder at worst case kompleksiteten af et filter skrevet i Cisco vil være $O(n)$, hvor n er antallet af regler.

Algoritmisk kompleksitet i IP Filter

IP Filter bliver altid læst helt igennem, med mindre man bruger det reserverede ord quick. Kompleksiteten for at læse alle regler igennem er $O(n)$, hvor n er antal regler. Hvis man bruger quick og sørger for at de mest brugte regler står øverst, bliver gennemsnitstiden for en filtrering en del mindre, men worst case kompleksiteten er stadig $O(n)$. For at modvirke skaléringsproblemer giver sproget mulighed for, at gruppere regler i undergrupper, baseret på hvilket interface pakken kom ind på. Men kompleksiteten forbliver $O(n)$, blot er n nu antal regler for et givet interface.

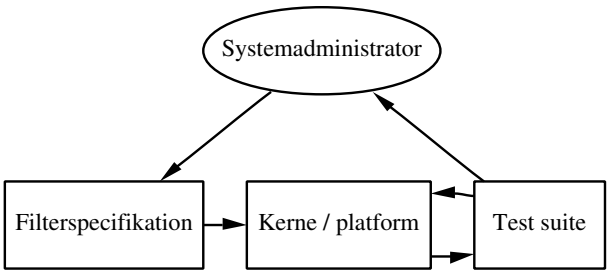
Algoritmisk kompleksitet i High Level Firewall Language

Kompleksiteten er afhængig af hvilken platform, der overføres til, men da de platforme, der kan overføres til, alle benytter sekventiel gennemløbning af en række regler, indtil der findes en der matcher, vil worst case kompleksitet blive $O(n)$, hvor n er antallet af regler.

2.4 Test af firewalls

For at finde ud af om en firewall lever op til de opstillede kriterier, er det nødvendigt at foretage nogle test. Dette er grundet, at det ikke altid er muligt at gennemskue den aktuelle kode, som måske indeholder flere tusinde regler.

Test af firewall filtre foregår som regel, når de er implementeret. Dette er grundet, at det vil være mere besværligt at checke flere forskellige traditionelle kodesprog med en generel test, i forhold til at checke firewalls når de er implementeret. Denne proces er skitseret på figur 2.4.



Figur 2.4: Test i forbindelse med firewalls.

Testteknikker

En mere generel og mere anvendelig metode til test af allerede implementerede firewalls, er programmet NMAP (Network Mapper) [nma03]. Programmet fungerer ved, at den giver mulighed for at scanne større eller mindre netværk, med henblik på at afgøre hvilke maskiner der kører, samt hvilke services disse maskiner tilbyder. Ved brug af NMAP fra en maskine på ydersiden af firewallen, skulle man således gerne - såfremt firewallen fungerer korrekt - kun kunne tilgå services, som tillades udefra.

Foruden de direkte testteknikker findes der også programmer, der løbende kan holde øje med en firewalls sikkerhed. En af disse programmer hedder logcheck. Programmet scanner periodisk loggede handlinger på et netværk og sammenligner disse med de opsatte regler. Såfremt der er overtrædelser eller usædvanlig aktivitet, kan programmet automatisk sende en e-mail med dette, til den der vedligeholder firewallen.

Der findes altså mange forskellige metoder til at teste og vedligeholde en firewall på, men fælles for dem alle er, at det foregår under runtime. Dette kan give nogle sikkerhedsmæssige huller i et netværk ved implementering af en ny, eller opdatering af en gammel firewall.

2.5 Problemformulering

I de foregående afsnit, analyseredes hvorledes eksisterende firewalls konfigureres og testes. Derudover blev det vist, at den algoritmiske kompleksitet for et filter, anvendt af eksisterende firewalls, er afhængig af antallet af regler i filtret. Følgelig skalerer disse dårligt. Igennem denne analyse er en række problemer blevet belyst, som medfører at specifikationen af firewalls i dens nuværende form, ikke er fremmende for hverken sikkerhed, effektivitet eller skalérbarhed. Dette leder frem til en problemformulering for projektet.

- Hvorledes udvikles et programmeringssprog med et større abstraktionsniveau og skalérbarhed end eksisterende firewall specifikationssprog, der tager højde for relevante parametre i forbindelse med specifikationen og som muliggør at filtre lettere kan testes offline og optimeres.
- Hvorledes konstrueres en kompiler, der oversætter imellem det udviklede firewall specifikationssprog og et stadie, som er brugbart i forbindelse med optimering og test af den specificerede firewall.

2.6 Projektafgrænsning

Projektet vil begrænse sig til udvikling af et programmeringssprog til pakkefiltre. Selvom sproget vil blive designet til at understøtte enkelte elementer fra stateful firewalls, vil kompileren kun understøtte simpel pakkefiltrering.

KAPITEL 3

Analyse

Formålet med dette kapitel er at analysere de elementer i en firewall beskrivelse, som kan medtages i det ønskede specifikationssprog. Desuden foretages overvejelser omkring hvilke programmeringssprog der vil være fremmende for implementation af kompileren, samt hvilken target model det vil være ønskeligt at benytte. Endeligt gives en kort opsummering af den valgte løsning.

3.1 Source sprog

Udgangspunktet for implementation af en firewall vil i de fleste tilfælde, være en sikkerhedspolitik, som eksempelvis beskriver hvilke ressourcer, der stilles tilrådighed på det interne net eller hvorledes interne brugere må tilgå det eksterne net. En sikkerhedspolitik er som regel formuleret i et naturligt sprog og sætter en række kriterier op for hvorledes krav til autentificering, adgangskontrol, integritet og konfidentialitet skal implementeres på det pågældende netværk. Eksempel 2 viser uddrag af en sikkerhedspolitik, hvor udvalgte dele af netværkstrafikken er beskrevet.

Eksempel 2 Eksempel på uddrag af en sikkerhedspolitik. [GB98, p. 17-18]

- FTP adgang skal tillades fra det interne til det eksterne net. Kryptering skal benyttes ved FTP tilgang fra det eksterne til det interne net.
- Telnet adgang skal tillades fra det interne til det eksterne net. Ved tilgang af Telnet fra eksterne net til det interne, skal benyttes autentificering.
- Alle HTTP servere, som skal kunne tilgås udefra, skal placeres udenfor firewallen ORG. Ingen indgående HTTP trafik tillades igennem firewallen ORG.
- Ingen ekstern adgang tillades til SMTP server.

■

Ud af sikkerhedspolitikken kan implicit læses en lang række tekniske detaljer, omkring hvorledes ORG og andre unavngivne firewalls skal opbygges i organisationen. Dels omkring hvilke net, der skal adskilles, hvilke protokoller, der tillades imellem dem og hvilke maskiner, der skal yde hvilke services.

Der er med nuværende firewall specifikationssprog, en stor afstand imellem sikkerhedspolitikkens beskrivelse og den egentlige implementation af firewallen. Dette skyldes, at det naturlige sprog, som sikkerhedspolitikken er formuleret i, har et andet abstraktionsniveau end filterspecifikationssprogene. Det kunne derfor være ønskeligt at designe source sproget, MFL, for filterspecifikationen, således elementer i sproget lagde sig tættere op ad det naturlige sprog, som sikkerhedspolitikken er formuleret i. Dette kunne ske igennem indførelse af en række abstraktioner i specifikationssproget.

Eksempelvis benyttes abstraktionen HTTP trafik i sikkerhedspolitikken. Dette dækker over en lang række tekniske detaljer, såsom hvilke protokoller der skal benyttes på flere forskellige lag, hvilke porte der må benyttes samt hvilke data der må udveksles.

3.1.1 Abstraktioner

I det følgende beskrives en række abstraktioner, som kunne implementeres i det ønskede filterspecifikationssprog. Med abstraktion menes, at der kan indføres en række mekanismer som muliggør, at uvæsentlige detaljer skjules og det væsentlige fremstår klarere.

Beskrivelse af net

En beskrivelse af netværk og enkle maskiner udfra simple abstraktioner, kunne indføres med henblik på lettere at kunne beskrive trafikken mellem disse. Følgende tre abstraktionstyper foreslås som udgangspunkt for beskrivelse af de maskiner, der skal indgå i filtreringen.

Vært. Er en enkelt maskine (eks. 10.8.12.1).

Netværk. Er en mængde af maskiner på samme subnet (eks. 10.8.12.*).

Adresser. Er en mængde af ikke nødvendigvis afhængige adresser (eks. konjunktionen af 10.*.12.* og 192.168.1.*).

Der kunne indføres mekanismer, således de tre typer af adresser kunne grupperes, samt at der kunne udføres simple boolske operationer på disse mængder. Foreningsmængden af to grupper eller negationen af en gruppe, kunne således danne udgangspunkt for en filtrering.

Beskrivelse af services

Efter der er defineret abstraktioner, som beskriver de enkelte net, ønskes abstraktioner til en beskrivelse af de services, der kunne udveksles mellem de enkelte net.

Eksempelvis kunne følgende services implementeres.

IP. Denne service vil beskrive IP trafik fra et givent source net til et givent destination net. Desuden kunne defineres en række kriterier ved IP headeren, som skulle være opfyldt. Alle felter i IP headeren kunne medtages i denne kontrol, hvilket blandt andet vil sige oplysninger om routning, protokoller og levetid.

TCP. Ville beskrive TCP trafik mellem de givne net. Herunder kunne specificeres hvilke portintervaller, der var gyldige, samt hvilke TCP headerfelter, der skulle undersøges. Dette kunne eksempelvis være SYN og ACK med henblik på identifikation af et gyldigt initierende handshake.

UDP. Ville som TCP beskrive trafik udfra en række portintervaller. På grund af den noget mindre header, ville noget færre headerchecks være muligt.

ICMP. Kunne benyttes til beskrivelse af ICMP trafik udfra dens type og kode.

Der kunne, som ved net abstraktionerne, indføres en række mekanismer med henblik på gruppering af services. Eksempelvis kunne defineres en service STAFF med begrænset ICMP adgang samt TCP adgang på port 22 og 80. Denne service kunne da danne udgangspunkt for yderligere specifikation. Ligeledes kunne introduceres boolske operationer, således service grupper kunne samles, negeres og lignende.

Beskrivelse af tidsintervaller

Sammen med serviceabstraktioner, kunne indføres en tidsabstraktion med henblik på kun at tillade en given service i et givent tidsrum. Tidsabstraktioner skulle ligeledes kunne grupperes og udsættes for boolsk aritmetik.

Beskrivelse af regler og filtre

Udfra abstraktionselementerne, som er blevet beskrevet i det foregående (net, services og tidsintervaller), kan opbygges regler. Det vil eksempelvis sige, at der mellem to beskrevne net, tillades (eller forbydes) en given service i et bestemt tidsrum. Disse regler skal endvidere kombineres til opbyggelse af egentlige filtre.

3.1.2 Kombination af abstraktioner

Indførsel af ovennævnte abstraktioner i specifikationssproget, vil være en nødvendighed for at kunne bringe sproget tættere på sikkerhedspolitikkens termer. Desuden ville abstraktionerne kunne øge specifikationernes genbrugelighed, idet grupper af ofte anvendte services kunne samles med henblik på brug i nye specifikationer.

3.1.3 Sproglige konstruktioner

Udover de indførte abstraktioner, kunne der ligeledes indføres en række konstruktioner i sproget, med henblik på at gøre implementationen af filtre lettere.

Selektion

Indførsel af en kontrolstruktur til selektion, således at forskellige regler eller filtre benyttes, afhængig af pakkens karateristika. En sådan selektion kunne, beskrevet udfra eksemplet i figur 2.3 på side 13, f.eks. være, at forskellige filtre blev benyttet alt afhængig af pakkens source IP.

Operatorer

De nævnte typer af abstraktioner (net, services, tid, regler og filtre), skal eksempelvis kunne lægges sammen og trækkes fra hinanden. Således indføres disse operatorer, som en mulighed for at arbejde på to mængder af ens abstraktioner.

Hvis to net f.eks. blev lagt sammen er det eksempelvis muligt at bruge en samlet regel på disse to net på en gang.

3.2 Target model

Dette afsnit omhandler en analyse af de mulige target modeller, samt repræsentationsformer for disse modeller (target sprog).

I forbindelse med target modellen, betragtes to muligheder. Dels den eksisterende model, som benyttes af Cisco og IP Filter (beskrevet i afsnit 2.2.2 på side 12). Denne model beskriver en

række regler udfra en ordnet liste, hvor rækkefølgen af regler har en betydning. I det følgende benævnes denne model Ordnet Regel Liste (ORL).

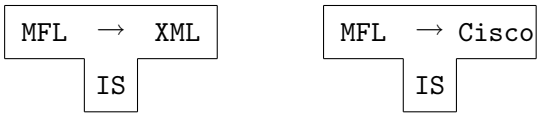
Endvidere betragtes modellen Multi Terminal Interval Decision Diagrams (MTIDD). MTIDDer opbygges af Interval Decision Diagrams (IDD). En MTIDD er en orienteret acyklisk graf, hvor hver knude i grafen består af et navn og et interval. For et filter vil navnet på noden være navnet på en variabel, og intervallet beskriver det mulige interval for den pågældende variabel.

Både MTIDDer og IDDer udtrykker prædikatlogik. I prædikatlogik beskrives udsagn der kan evalueres til sand eller falsk. Hvert udsagn består af en variabel og et prædikat, udsagnet givet en værdi til variablen evaluerer til en sandhedsværdi. Et udsagn kunne for eksempel være $dport = 1024$, hvor udsagnets variabel er *dport* og dens prædikat er $= 1024$. *dport* svarer til destinationport i en TCP header. Hvis en TCP pakke i destinationport feltet har værdien 1024 evaluerer udsagnet altså til sand. Det nævnte udsagn kan sammensættes med andre udsagn ved at bruge boolske operatorer. Udsagnet repræsentere en IDD knude, og da IDDer udtrykker prædikat logik, kan IDDer der repræsentere hver deres terminal, opbygges med boolske operatorer. En IDD har to sandhedsværdier kaldet terminaler, men da filtre har flere handlinger end de to der kan udtrykkes ved sandhedsværdierne, må MTIDDer der tillader flere terminaler anvendes til den samlede beskrivelse af et firewall filter.

For en nærmere beskrivelse af IDDer, MTIDDer og prædikat logik henvises til appendiks B på side 80.

Både ORL og MTIDD modellen kan beskrives ved en række repræsentationsformer. Den regelbaserede ordnede liste, kan naturligt beskrives ved enten Cisco eller IP Filter sproget. Desuden ville der kunne opstilles en XML[1] repræsentation af filtrene beskrevet ved denne model. For MTIDDer findes en veldefineret DTD[2] hvorfor XML vil være en oplagt repræsentationsform.

På figur 3.1 ses et T-diagram for hver af de to mulige target sprog. Et T-diagram for kompilere, beskriver sammenhængen mellem source sprog, target sprog og implementeringssprog (forkortes IS), det sidste er det sprog en kompiler implementeres i. Source sprog er i øverste venstre del, target sprog i øverste højre del og implementeringssprog i nederste del af diagrammet. I begge diagrammer er MFL source sprog, derudover er implementeringssproget ikke blevet fastlagt endnu.



Figur 3.1: T-diagram for kompilere til to mulige target sprog.

De to target sprog, XML efter en MTIDD model og Cisco/IP Filter, vil herunder blive beskrevet ud fra en række kriterier.

3.2.1 Konvertering af filtre

I det følgende beskrives, hvorledes firewall filtre beskrevet i MFL kan omdannes til de betragtede target modeller.

I ORL modellen skal reglerne defineres i en bestemt rækkefølge, da det eks. vil være formålsløst

[1] eXtended Markup Language.
[2] Document Type Definition. Beskriver hvorledes et XML dokument opbygges.

først at tilføje en regel der forbyder al WWW trafik, og derefter tilføje en regel der tillader WWW trafik til en given IP. Kompileren skal således ved benyttelse af ORL modellen, tage højde for den rækkefølge, når der tilføjes nye regler.

For MTIDD er det klart defineret, hvorledes der tilføjes regler (ved disjunktion), hvilket gør det nemt at omdanne en liste af regler skrevet i MFL til en MTIDD model.

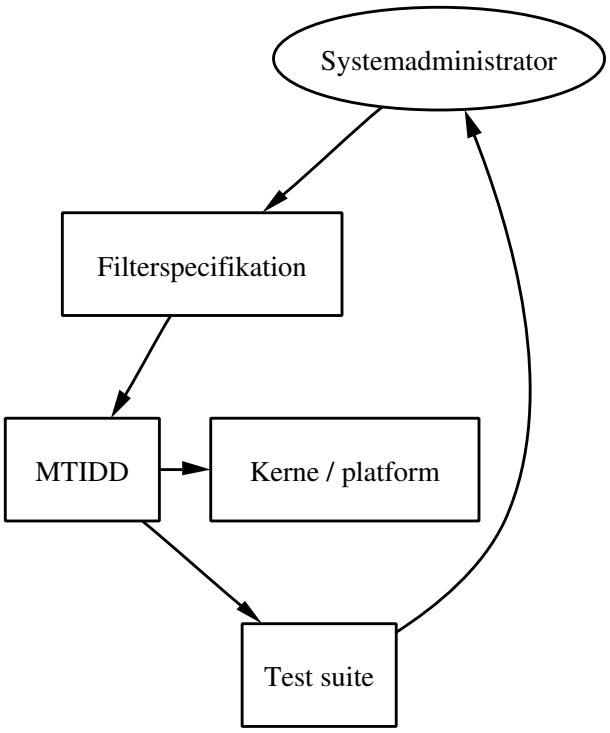
3.2.2 Test af korrekthed

Dette afsnit omhandler, hvorledes korrektheden af et filter beskrevet ved de to modeller, kan testes. Med korrekthed menes, at et filter opfører sig som den angivne sikkerhedspolitik foreskriver.

Den umiddelbare måde at teste et firewall filter på, er at afvikle programmet på den valgte platform og teste runtime om det virker hensigtsmæssigt. Både ORL og MTIDD modellerne vil kunne checkes på denne måde.

En MTIDD kan repræsenteres grafisk, hvorved det er muligt at danne sig et overblik, over hvilke handlinger der udføres ved forskellige pakker.

Det er ikke altid optimalt at teste et filter i runtime, eller ved grafisk at få et overblik over korrektheden. Derfor er det praktisk at anvende en verifier til at teste designet offline, inden det implementeres og afvikles. Dette er illustreret på figur 3.2.



Figur 3.2: Test i forbindelse med udvikling.

I [CF02] foreslås opbygningen af et værktøj (Network Access Verifier) til analyse og test af MTIDDer, med henblik på bestemmelse af hvilken netværkstrafik der tillades. Dette giver således mulighed for, at lave en udtømmende test af egenskaber for netværket. Denne test kan

foretages offline og giver derigennem mulighed for, at afgøre hvorledes pakker håndteres uden at foretage en runtime test.

3.2.3 Algoritmisk kompleksitet

I det følgende betragtes den algoritmiske kompleksitet for de filtreringsalgoritmer, der anvendes i de to filtermodeller.

I afsnit 2.3 på side 17, blev det for ORL modellen vist, at kompleksiteten i pakkefiltreringen er $O(n)$, hvor n er antallet af regler i den ordnede liste.

Kompleksiteten for filtreringsalgoritmen i MTIDD modellen er ikke på samme måde afhængig af antal regler. I det følgende betragtes antallet af operationer, der kræves for at finde en enkelt rute igennem MTIDDen for en enkelt pakke. For at finde den rute som headeren følger, skal der i hvert lag af MTIDDen foretages en binær søgning, med $log(w)$ operationer, hvor w er antallet af forgreninger ud fra knuden. Hvis der er flere felter i en pakkes header, der skal undersøges, vil der være flere lag i MTIDDen, der skal gennemløbes, før pakken er helt undersøgt. Med dette udgangspunkt kendes mængden af operationer i en gennemløbning af en MTIDD til $m \cdot log(w)$, hvor m er antallet af informationer fra pakkens header der skal undersøges. Da både m og w er begrænsede, kan de i kompleksitetssammenhæng betragtes som konstanter, hvilket medfører at worst case kompleksiteten bliver $O(1)$.

3.3 Implementeringssprog

I dette afsnit vil overvejelser i forbindelse med valg af implementeringssproget til en kompiler kort blive analyseret.

Ved implementeringen af en kompiler er det vigtigt at tage hensyn til både source- og target sprog for at kunne bestemme implementeringssproget. Sproget der skal oversættes fra (source) er i nogle tilfælde givet, men i dette tilfælde skal det designes i forbindelse med kompileren. At dette sprog skal designes samtidigt med kompileren, har ikke direkte indflydelse på valget af implementeringssproget. Derimod har implementeringssproget for allerede tilgængelige værktøjer indflydelse på valget. Idet der allerede findes et bibliotek til generering af IDDer og MTIDDer i det funktionelle programmeringssproget OCaml, vil det være fordelagtigt at vælge netop dette sprog til implementation af kompileren. Sprog i klassen af funktionelle sprog er specielt beregnede på, at løse problemer indenfor de matematiske og datalogiske paradigmer, deriblandt konstruktion af kompilere.

I forbindelse med valget af implementeringssproget bør det også overvejes, hvordan kompileren skal afvikles. Derved bør undersøges om der allerede findes en kompiler til at oversætte den nye kompiler til kode der kan afvikles på den ønskede maskine, eller man også skal til at lave denne.

3.4 Valg af løsning

Der er valgt at udvikle en kompiler, der kompilerer fra sproget MFL, et abstraktionspræget sprog til beskrivelse af pakkefiltre, til target modellen MTIDD beskrevet ved repræsentationsformen XML.

MFL skal indeholde en række abstraktioner over strukturer i problemområdet f.eks. vært, net og adresse, samt en række sproglige konstruktioner herunder operationer og selektionsmekanismer

der kan bruges i kombination med de beskrevne abstraktioner.

At kompileren skal kompilere til MTIDDer begrundes med, at MTIDDer giver en række muligheder for test af den sikkerhed det endelige pakkefilter vil yde samt mulighed for optimering f.eks. ved at fjerne overlappendene regler. Derudover ligger MTIDDer naturligt op til en række af de sproglige kombinationer der er ønskelige i et sprog til firewalls, for eksempel boolske operationer og selektionsmekanismer.

OCaml vælges som implementationssprog, da der i forvejen eksisterer et implementeret MTIDD bibliotek til dette sprog.

KAPITEL 4

Kravspecifikation

I dette kapitel specificeres de krav som systemet skal opfylde. Opstillingen af kravspecifikationen samt overvejelser omkring det ønskede system, er inspireret af SPU metoden [BSHKM98, p. 69-96].

4.1 Formål

Formålet med projektet er at udvikle et sprog og en kompiler som del af et firewall system, hvor der forsøges at øge effektiviteten og overskueligheden af pakkefiltre ved anvendelse af MTIDD modellen.

4.2 Generel beskrivelse

Her følger en generel beskrivelse af systemet og hvilke overordnede krav der findes.

4.2.1 Systembeskrivelse

En systemadministrator skal have sikret et netværk med et pakkefilter. Dette filter beskrives af systemadministratoren i et højniveausprog, som så kompileres til et sprog der er mere effektivt for operativsystemet at fortolke. Sproget til systemet skal være effektivt, da det skal udføres hver gang der kommer en pakke ind fra netværket.

Det udviklede programmel skal oversætte højniveaukoden til lavniveaukoden. Programmellet modtager en tekstfil fra systemadministratoren, og skal danne en fil som kan læses af den del af operativsystemet der skal filtrere pakker.

4.2.2 Programmellets funktion

Udviklingen af programmellet består af to dele. Der skal udvikles et sprog som kan beskrive et pakkefilter, og der skal udvikles en kompiler, der kan oversætte dette sprog til MTIDDer.

Kompileren skal oversætte et firewall filter til et sprog (XML), som kan læses af et kernemodul, der så kan agere den egentlige firewall.

4.2.3 Programmellets begrænsninger

For at gøre sproget mere universelt, er der mulighed, for at beskrive stateful firewalls i sproget. Denne funktionalitet vil dog ikke blive videreført i kompileren, da dette går udenfor målet, nemlig at lave et pakkefilter. Derudover vil der hverken i sprog eller kompiler være mulighed for anvendelse af NAT.

Indlæsningen i et kernemodul, samt deraf følgende mulighed for runtime tests vil ikke blive foretaget.

Hovedinteressen i dette projekt ligger på sprog og kompilere, og derfor arbejdes der ikke videre, med at lave en testsuite til sproget.

4.2.4 Programmellets fremtid

Kompileren og sproget er prototyper, der udvikles som led i et forsøg på, at effektivisere den måde hvorpå pakkefiltre skrives og benyttes. Effektiviseringen sker ved anvendelse af mere effektive klassificeringsalgoritmer, end der anvendes i nuværende firewalls, hvilket er hvad brugen af MTIDDer medfører. Det er derfor meningen, at sproget og kompileren skal bruges, som værktøjer i et alternativ til nuværende firewalls.

Da sproget skal understøtte beskrivelsen af en stateful firewall, er en udvidelse af kompileren, at den bør kunne forstå dette og sende det til det kernemodul, der skal implementere firewallen.

4.2.5 Brugerprofil

Brugerne, af både sprog og kompiler, vil være systemadministratorer, som er bekendt med firewalls og deres funktion.

4.2.6 Krav til udviklingsforløbet

Kompileren skrives i OCaml . For at harmonisere koden, så det bliver lettere at overskue koden for andre end vedkommende der har skrevet den, benyttes udvalgte koderetningslinier opstillet i [Wei02]. Dette vil bla. sige retningslinier omkring navngivning af funktioner, samt at kommentarer skrives således de forståes af OCamldoc[1].

4.3 De specifikke krav

Dette afsnit indeholder de specifikke krav til opbygningen og funktionaliteten af kompileren i projektet.

4.3.1 Funktionelle krav

Kompileren skal opbygges som en multipass kompiler, og den skal indeholde de tre grundlæggende faser: syntaktisk analyse, semantisk analyse og kodegenerering. Den leksikalske analyse anses som værende en del af den syntaktiske analyse. [WB00]

De tre faser er efterfølgende beskrevet med hensyn til de funktionelle krav.

Syntaktisk analyse

I syntaktiske analyse bestemmes kildekodens sproglige struktur. Denne proces kaldes også for parsing. Kildekoden skal checkes for overensstemmelse med source sprogets syntaks og der skal dannes et AST, der er en intern repræsentation af koden, beskrevet i en træstruktur.

[1] Værktøj, der genererer dokumentation fra specielle indlejrede kommentarer.

Semantisk analyse

I den semantiske analyse skal ASTet analyseres yderligere, for at bestemme om det overholder source sprogets kontekstuelle begrænsninger. Dette indbefatter check af scope- og typeregler. Dermed skal der opnås et dekoreret AST.

Kodegenerering

Kodegeneratoren skal generere kode ud fra det syntaktiske og kontekstuelle analyserede kildekode. Dette er den sidste oversættelse af filtret. Her genereres en model af det, i source sproget, beskrevne filter med targetmodellen.

4.4 Eksterne grænsefladekrav

I dette afsnit beskrives de relevante krav til de eksterne grænseflader. Dette vedrører bruger- og software grænsefladen, som er beskrevet i det efterfølgende.

4.4.1 Brugergrænseflade

Kompileren skal kunne afvikles som et kommandolinieprogram. Der skal være mulighed for at få adgang til hjælp, enten via en liste af kommandoer eller en fil med beskrivelse af funktionaliteterne.

4.4.2 Software grænseflade

For at kunne generere kompilerprogrammet udfra den skrevne OCamlkode, kræves der en platform med en OCamlkompiler.

Den generede kompiler skal endvidere have adgang til biblioteket libIDD, der bruges til generering af MTIDDer. Denne grænseflade består i, at der bliver refereret til libIDD biblioteket i programkoden for kompileren. Dermed er det et krav at biblioteket er tilgængeligt når kompileren afvikles.

4.5 Krav til programmellets ydelse

Der stilles ikke krav til programmellets ydelse i form af tidsbegrænsninger og størrelsen af programmet.

4.6 Kvalitetsfaktorer

Følgende afsnit er en gennemgang af de kvalitetsfaktorer der er for henholdsvis MFL og kompileren hertil. Først beskrives kvalitetsfaktorer for kompileren, derefter for MFL.

4.6.1 Kompiler

Tabel 4.1 viser vægtning af kvalitetsfaktorer for kompilerprogrammet. Faktorerne stammer fra [BSHKM98, p. 93].

Kriterium	**Meget Vigtigt**	**Vigtigt**	**Mindre Vigtigt**	**Irrelevant**
Pålidelighed		√		
Vedligeholdelsesvenlighed			√	
Udvidelsesvenlighed		√		
Brugervenlighed		√		
Genbrugbarhed	√			
Integritet			√	
Effektivitet			√	
Testbarhed		√		

Tabel 4.1: Vægtning af kvalitetsfaktorer for kompileren.

Pålidelighed *betyder i hvor stor grad det er garanteret, at programmet udfører den specificerede funktion.* Da programmets output skal benyttes i en firewall, som netop er et sikkerhedssystem, er det vigtigt at det endelige output stemmer overens med specifikationen uden fejl introduceret af kompileren.

Vedligeholdelsesvenlighed *er udtryk for hvor nemt det er at vedligeholde programmet efter det er færdigt.* Det er prioriteret mindre vigtigt, idet programmet udvikles som en prototype.

Udvidelsesvenlighed *er udtryk for hvor nemt det vil være at udvide funktionaliteten af programmet.* Der er nævnt muligheder for udvidelse af programmellet (i afsnit 4.2.4), og det er vigtigt at det vil være muligt at implementere dette i fremtiden.

Brugervenlighed *beskriver, hvor meget vægt lægges der på, at det er nemt at benytte programmellet.* Dette prioriteres vigtigt fordi selvom programmet er et kommandolinie værktøj og den gennemsnitlige bruger af programmet, må forventes at have grundigt kendskab til computerværktøjer i almindelig, ønskes stadig at kompileren kan give meningsfyldte fejlmeddelelser.

Genbrugbarhed *betegner, hvor godt programmet understøtter genbrug af koden.* Da der udvikles en prototype, som muligvis skal overføres til videre produkt, er det nødvendigt at kunne overføre dele af programmellet til det færdige produkt.

Integritet *beskriver, hvor vigtigt det er, at programmet beskytter sine data fra ekstern påvirkning.* Programmellet skal kun køres under kompilering, og dermed er der meget få muligheder for påvirkninger udefra, så dette tages der ikke højde for.

Effektivitet *udtrykker, hvor vigtigt det er at programmet er hurtigt og effektiv i udførelse.* Siden at programmellet udføres en enkelt gang når beskrivelsen skal oversættes er det ikke vigtigst at programmet er meget effektivt. Det er dog vigtigt, at programmet kan oversætte de fleste beskrivelser indenfor en rimelig tid, da de fleste brugere ikke vil vente i længere tid.

Testbarhed *betegner, muligheden for at teste den implementerede applikation.* I kraft af at det er vigtigt, at programmet er pålideligt, så er det også vigtigt, at det er muligt at teste programmet.

4.6.2 Sprog

I tabel 4.2 er der listet en række egenskaber et godt programmeringssprog skal have. I dette afsnit er disse egenskaber prioriteret, ud fra hvilke egenskaber et sprog til beskrivelse af pakkefiltre skal have. I kursiv er definitionen af egenskaberne kort skrevet op. Kriterierne og deres overordnede definition er inspireret af [PZ01]

Kriterium	**Meget Vigtigt**	**Vigtigt**	**Mindre Vigtigt**	**Irrelevant**
Naturlighed		√		
Læsbarhed	√			
Ortogonalitet			√	
Klarhed og enkelhed		√		
Testbarhed		√		
Understøttelse af abstraktion	√			
Skalérbarhed	√			

Tabel 4.2: Vægtning af kvalitetsfaktorer for sprog.

Naturlighed *beskriver, hvor godt sproget understøtter applikationens naturlige struktur.* Hvis det skal være nemt, for en programmør at skrive et filter, skal den naturlige struktur naturligvis understøttes, sålænge det ikke sker på bekostning af "Læsbarhed" og "Understøttelse af abstraktion". Derfor prioriteres dette "Vigtigt".

Læsbarhed *betegner, hvor let forståelig syntaksen er. Både under skrivning og ved læsning.* En af motivationerne for at lave et nyt filterspecifikationssprog, var netop at gøre det nemmere at skrive filtre. Derfor prioriteres dette kriterium til "Meget Vigtigt", idet nogle af de eksisterende sprog manglede læsevenlighed.

Ortogonalitet *beskriver, i hvilken grad kombinationer af sprogets attributter også har mening i sproget.* "Mindre Vigtigt", da sproget har en meget begrænset anvendelse.

Klarhed og enkelhed *beskriver, i hvilken grad, der er et veldefineret sæt af koncepter, som hjælper program,øren i designfasen.* Dette kriterier prioriteres "Vigtigt", af grunde der er analoge til begrundelsen i "Naturlighed".

Testbarhed *betegner, hvor let det er at teste, om programmet kører korrekt i henhold til specifikationen.* Dette er også noget de eksisterende sprog mangler. Derfor vil dette blive forsøgt forbedret, derfor "Vigtigt".

Understøttelse af abstraktion *beskriver, hvorledes sproget understøtter abstraktioner.* Prioriteres "Meget Vigtigt", med en begrundelse analog til kriteriet "Læsbarhed".

Skalérbarhed *beskriver, sprogets skalérbarhed.* Prioriteres "Meget Vigtigt", idet dette er en af problemerne ved de eksisterende specifikationssprog, som forsøges løst i dette projekt.

KAPITEL 5

Sprogdesign

Formålet med dette kapitel er at designe target og source sprogene til kompileren. På baggrund af den foregående analyse, vælges et paradigme, hvorefter der opstilles en syntaks for det ønskede source sprog. Endelig betragtes det designede sprogs kategori.

5.1 Paradigme

For at source sproget kan fastlægges, skal der tages stilling til hvilket paradigme sproget skal opbygges efter. Udfra de mulige paradigmer, beskrevet i [PZ01], foretages et valg. Der ønskes et sprog til beskrivelse af pakkefiltre, hvor målet er et skalérbart sprog med et højt abstraktionsniveau.

Pakkefiltre består af en række regler for forskellige headere. Disse regler har hver tilknyttet en handling, såsom deny, permit og log. Det er disse regler og handlinger, der skal kunne beskrives af sproget. Naturligvis kan sproget designes, så det ligner nuværende sprog til filterspecifikation, det vil sige sprog der følger ORL modellen, såsom Cisco. For et sådant sprog, bestående af regler, er det logiske paradigme et oplagt udgangspunkt. Ved kombination med kontrolstrukturer til selektion og abstraktioner over samlinger af regler, gøres det muligt at reducere mængden af redundant kode, hvormed sproget bliver mere skalérbart.

5.2 Designovervejelser

Sproget baseres på det logiske paradigme, dette paradigme kaldes også regelbaseret, fordi det beskriver betingelser for udførelse af handlinger, f.eks.:

$$\begin{aligned} \textit{enabling condition}_1 &\rightarrow \textit{action}_1 \\ \textit{enabling condition}_2 &\rightarrow \textit{action}_2 \\ &\dots \\ \textit{enabling condition}_n &\rightarrow \textit{action}_n \end{aligned}$$

I forbindelse med MFL vil enabling condition bestå af de betingelser, der kan opskrives for pakkefiltreringen. Action vil bestå af tilladelsen, som enten kan være permit eller deny i kombination med log og state.

permit. Tillader pakker, som opfylder reglen at passere.

deny. Tillader ikke pakker, som opfylder reglen at passere.

log. Siger ikke noget om pakken kan passer eller ej, men logger den hvis den opfylder reglen.

state. Måden hvorpå stateful inspektion anvendes i MFL. Det vil sige at anvendes state som handling, vil en pakke og alle relaterede pakker blive behandlet efter samme action.

Enabling conditions kan opbygges sekventielt med boolske operatorer mellem de enkelte betingelser, f.eks. kan `sip 1.1.1.1` og `dip 2.2.2.2` kombineres således `sip 1.1.1.1 && dip 2.2.2.2` eller `sip 1.1.1.1 || dip 2.2.2.2`. `sip` og `dip` hentyder til specifikke felter i pakkens header - her source og destination IP.

Hvis man ikke skriver en boolsk operator mellem to betingelser antages `&&`. En fuldstændig liste for mulige betingelser og deres anvendelse er beskrevet i afsnit 5.3.

Det er endvidere blevet valgt, at MFL skal indeholde abstraktioner over samlinger af regler. Disse bliver designet som filtre, der kan kaldes, med parametre, fra andre filtre i samme MFL fil. Da der nu kan være mere end et filter i en firewall, skal der altid angives et main filter, hvor opbygningen af det samlede pakkefilter starter. Opbygningen af et pakkefilter i MFL kan derfor bestå af en række filtre, som kalder hinanden, samt et obligatorisk main filter. I den forbindelse skal det nævnes, at der kontrolleres, at filtrene ikke kalder hinanden cyklisk, da dette ville resultere i, at der ville blive genereret en uendelig liste af regler.

Idet sproget bliver brugt til at beskrive pakkefiltre, som er statiske, bliver det et erklæringssprog. Med erklæringssprog menes blot, at alt i sproget er statisk efter kompilering. Dette kommer til at betyde, at MFL er strongly typed, hvilket vil sige at variabler ikke ændrer sig, og derfor i realiteten er konstanter.

Main filter indeholder en standardhandling for filtret, som f.eks. kan være deny. Denne standardhandling er den pakkefiltret benytter, hvis en pakke ikke dækkes af nogen af de andre regler i filtret. Standardhandling vil være tydeligt markeret i main filtret, idet denne skal skrives i firkantede parenteser.

Udenfor main og filter scopes kan man blandt andet erklære IP-adresser og porte, som her knyttes til et konstantnavn. Det er desuden muligt at inkludere filer med filter- og konstantspecifikationer. Opbygningen af main og filtrene er nærmere beskrevet i afsnit 5.3.

5.3 Sprogspecifikation

I det følgende vil de forskellige grupper i EBNFen[1] for MFL blive gennemgået. Dette sker via, at for hver kategori i EBNFen, først at opskrive syntaksen i EBNF form, derefter gennemgås dennes semantik, for til sidst at komme med nogle små eksempler på anvendelsen. [WB00, p. 77-79]

I eksemplerne ses kommentarer til MFL koden. Disse er starter med symbolerne //, der i MFL markerer, at resten af linien er kommentarer. MFL tillader også kommentarer over flere linier, disse startes med /* og afsluttes med */. Under kompilering filtreres disse fra i syntaktisk analyse.

Den semantiske mening af reglerne i EBNFen er, at hver regel beskriver en mængde af netværkspakker der overholder denne regels krav. Hver kommando er en samling af en eller flere regler og kommandoer, og beskriver dermed også en mængde af pakker.

Med evaluering af en regel menes, at den evalueres til førnævnte mængde af pakker. Med afviklingen af en kommmando menes måden hvorpå regler og kommandoer, i kommandoen, evalueres til en mængde af pakker.

[1] Extented BNF: BNF udvidet med regulære udtryk.

5.3.1 Grundlæggende dataelementer

Først beskrives nogle grundlæggende dataelementer, som er nødvendige for at forstå det efterfølgende.

Syntaks

⟨*int*⟩ ::= [‘0’...‘9’] [‘0’...‘9’]∗

⟨*range*⟩ ::= (⟨*int*⟩ (’-’ ⟨*int*⟩)?) | ’*’

⟨*identifier*⟩ ::= [‘a’...‘z’‘A’...‘Z’] ([‘a’...‘z’‘A’...‘Z’‘0’...‘9’])∗

Semantik

int. Beskriver et heltal. De tilladte værdier afgøres udfra typen i den semantiske analyse.

range. Beskriver et interval af heltal, hvor den laveste værdi skal være repræsenteret først. Angivelse af “alle værdier” i en range kan foretages med “*”. Hvad der forstås med “alle værdier” afgøres af den semantiske analyse.

identifier. Beskriver at identifier skal starte med et bogstav og derefter et antal bogstaver eller cifre.

Eksempler

```
1 //En range
2 23-50
3
4 //En identifier
5 LaTeX2e
```

5.3.2 Dataproduktioner

Syntaks

⟨*type*⟩ ::= ‘ip’ | ‘port’ | ‘iface’ | ‘time’ | ‘day’ | ‘proto’

⟨*IP*⟩ ::= ⟨*range*⟩ ‘.’ ⟨*range*⟩ ‘.’ ⟨*range*⟩ ‘.’ ⟨*range*⟩
| ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘/’ ⟨*int*⟩

⟨*port*⟩ ::= ⟨*int*⟩

⟨*portrange*⟩ ::= ⟨*range*⟩

⟨*time*⟩ ::= ⟨*int*⟩ ‘:’ ⟨*int*⟩ ‘-’ ⟨*int*⟩ ‘:’ ⟨*int*⟩

⟨*action*⟩ ::= (‘permit’ ‘state’? | ‘deny’) ‘log’ ?
| ‘log’

⟨*weekday*⟩ ::= ‘mon’ ‘day’?
| ‘tue’ ‘day’?
| ‘wed’ ‘nesday’?
| ‘thu’ ‘rsday’?
| ‘fri’ ‘day’?

| ‘sat’ ‘urday’?
| ‘sun’ ‘day’?

⟨*interface*⟩ ::= ‘’’ ⟨*identifier*⟩ ‘’’

⟨*v-name*⟩ ::= ⟨*identifier*⟩

⟨*proto*⟩ ::= ‘TCP’ (‘[’ ⟨*tcpflag*⟩ (‘,’ ⟨*tcpflag*⟩) ‘]’)?
| ‘UDP’
| ‘ICMP’ (‘[’ ⟨*int*⟩ (‘,’ ⟨*int*⟩) ‘]’)?

⟨*tcpflag*⟩ ::= ‘SYN’ | ‘ACK’ | ‘URG’ | ‘FIN’ | ‘PSH’ | ‘RST’

⟨*any*⟩ ::= ‘any’

Semantik

type. Beskriver de typer, der er mulige i sproget.

IP. Beskriver en mængde med én eller flere IP adresser. Dette betyder at der kan ikke være nul adresser i en mængde af IPer. De semantiske krav til IP er, at fire intervaller eller de fire første heltal ikke må overskride grænseværdierne på 0 og 255. Hvis IPen består af fem heltal, skal det sidste ligge mellem 0 og 32.

port. Beskriver et tal. I semantisk analyse sikres at dette tal er i intervallet 0-65535.

time. Beskriver et tidsinterval på en 24-timers dag, med et minuts opløsning. Intervallet må ligge mellem 00:00 og 23:59.

action. Beskriver en handling. De lovlige handlingr er permit, permitlog, permitstate, permitstatelog, deny, denylog eller log. Permit og deny angiver om pakker skal filtreres fra eller ej. Log og state angiver om pakker skal logges, og om der skal gemmes state for pakkerne.

weekday. Beskriver en enkelt ugedag. Dagene kan angives med deres fulde engelske navn eller blot de tre første karakterer.

interface. Angiver et navn på en interfaceenhed[2], så denne kan tilgås. For at kunne skelne disse fra identifieres indkaples de i apostroffer.

v-name. Angiver navnet på en variabel, denne kan i en konstanterklæring bindes til et udtryk, eller et andet navn. Under semantisk analyse kontrolleres det, at der ikke er v-names, der er anvendes cirkulært.

proto. Beskriver de mulige protokoltyper. For TCP protokollen kan angives hvilke flag der skal være sat i headeren, er der ikke angivet nogle flag er alle flag lovlige. ICMP protokollen giver mulighed for angivelse af typen, i form af en liste af heltal mellem 0 og 255. Er der ikke angivet nogle heltal er alle lovlige.

tcpflag. Beskriver de flag som kan være sat i en TCP header.

any. Benyttes til at beskrive alle værdier i den sammenhæng udtrykket anvendes i.

[2]Interfaceenhed: En fysisk enhed i PCen, for eksempel netkort: eth0

Eksempler

```
1  //IP
2  192.168.120.10
3  192.168.10-255.*
4  192.168.0.0/16
5
6  //Weekday
7  mon
8  sunday
9
10 //Interface
11 'eth0'
```

5.3.3 Operatorer

Eksempel på disse operatorer kommer under 5.3.5 på næste side.

Syntaks

⟨*binaryoperator*⟩ ::= '`&&`' | '`||`' | '`&!`' | '`^^`'

⟨*unaryoperator*⟩ ::= '`!`'

⟨*portoperator*⟩ ::= '`<`' | '`<=`' | '`>`' | '`>=`'

Semantik

binaryoperator. Beskriver boolske operatorer, som tager to regler som argumenter. Da regler beskriver mængder, er den semantiske betydning af disse boolske operatorer, at der foretages en tilsvarende mængdeoperation.

- `&&` Og (Fællesmængden)
- `||` Eller (Foreningsmængden)
- `&!` Og ikke (Differensmængden)
- `^^` Eksklusiv eller (Den symmetriske differensmængde)

unaryoperator. Beskriver boolsk operator, som tager en regel som argument. Den semantiske betydning af denne operator, er en mængdeoperation på argumentet.

- `!` Ikke (Komplementærmængden)

portoperator. Beskriver operator, der kun kan bruges i forbindelse med angivelse af porte. Disse portoperatorer tager enten et heltal eller et interval som argument. Den semantiske betydning af en portoperator, er mængden der beskrives af operatoren i forhold til argumentet.

5.3.4 Udtryk

⟨*expression*⟩ ::= ⟨*v-name*⟩
| ⟨*IP*⟩

| ⟨*portrange*⟩
| ⟨*interface*⟩
| ⟨*weekday*⟩
| ⟨*time*⟩
| ⟨*proto*⟩
| ⟨*any*⟩

Semantik

expression. En abstraktion, som anvendes i andre nonterminaler. De semantiske regler for expression, er de samme som for de dataproduktioner den kan bestå af.

Eksempler

```
1  //Ip expression
2  12.12.12.12
3
4  //Time expression
5  12:00 - 20:00
6
7  //Protokol expression
8  TCP [SYN,ACK]
9
10 //Any expression
11 any
```

5.3.5 Kombinatoriske produktioner

Syntaks

⟨*pardecl*⟩ ::= ⟨*singledecl*⟩ (‘`,`’ ⟨*singledecl*⟩)∗

⟨*parseq*⟩ ::= ⟨*expression*⟩ (‘`,`’ ⟨*expression*⟩)∗

⟨*singledecl*⟩ ::= ⟨*type*⟩ ⟨*v-name*⟩

⟨*command*⟩ ::= ⟨*single-command*⟩ (‘`;`’ ⟨*single-command*⟩)∗

⟨*single-command*⟩ ::= ‘{’ ⟨*command*⟩ ‘}’ **1**
| ⟨*identifier*⟩ ‘`(`’ ⟨*parseq*⟩? ‘`)`’ **2**
| ‘`if`’ ⟨*rule*⟩ ’`then`’ ⟨*single-command*⟩ ’`else`’ ⟨*single-command*⟩ **3**
| ‘`with`’ ⟨*rule*⟩ ‘`do`’ ⟨*single-command*⟩ **4**
| ⟨*rule*⟩ ‘`->`’ ⟨*action*⟩ **5**

⟨*rule*⟩ ::= ⟨*single-rule*⟩ (⟨*binaryoperator*⟩? ⟨*single-rule*⟩)∗

⟨*single-rule*⟩ ::= ‘`sip`’ (⟨*IP*⟩ | ⟨*v-name*⟩) **1**
| ‘`dip`’ (⟨*IP*⟩ | ⟨*v-name*⟩) **2**
| ‘`sport`’ (⟨*portrange*⟩ | ⟨*v-name*⟩ | ⟨*portoperator*⟩ (⟨*port*⟩ | ⟨*v-name*⟩)) **3**
| ‘`dport`’ (⟨*portrange*⟩ | ⟨*v-name*⟩ | ⟨*portoperator*⟩ (⟨*port*⟩ | ⟨*v-name*⟩)) **4**
| ‘`input`’ (⟨*interface*⟩ | ⟨*v-name*⟩) **5**
| ‘`output`’ (⟨*interface*⟩ | ⟨*v-name*⟩) **6**
| ‘`proto`’ (⟨*proto*⟩ | ⟨*v-name*⟩) (‘`,`’ (⟨*proto*⟩ | ⟨*v-name*⟩))∗ **7**

	‘`time`’ (⟨*time*⟩ | ⟨*v-name*⟩) ’`-`’ (⟨*time*⟩ | ⟨*v-name*⟩)	**8**
	‘`day`’ (⟨*weekday*⟩ | ⟨*v-name*⟩)	**9**
	‘`(`’ ⟨*rule*⟩ ‘`)`’	**10**
	⟨*unaryoperator*⟩ ⟨*single-rule*⟩	**11**

Semantik

pardecl. Definerer typer og navne på de lokale konstanter, der skal indgå i den følgende kommandoblok.

parseq. Angiver en ordnet liste over en eller flere expression.

singledecl. Knytter en ny konstant til den angivne type.

command. Angiver en sekvens af single-commands. command afvikles ved at afvikle sekvensen af single-commands.

single-command (1). Afvikler kommandoen i kommandoblokken.

single-command (2). Kommandoerne i filtret, som er bundet til identifieren afvikles, med værdierne af udtrykkene i parseq som parametre. I semantisk analyse kontrolleres for det første, at det kaldte filter eksisterer, men også at typerne i den sekvens af udtryk der findes i parseq, matcher de typer der er knyttet til de lokale konstanter i det kaldte filter. Ved oprettelse af et filter, kontrollerer den semantiske analyse, at der ikke opstår cirkulære filterkald.

single-command (3). Afvikles som følger: De to single-commands afvikles. Den semantiske betydning af denne kommando er, at reglen betragtes som værende opfyldt i den første single-command og ikke i den anden. Dette får en semantisk betydning, for mængderne beskrevet af reglerne fundet i begge disse single-commands. Mængderne i første single-command konjugeres med mængden beskrevet af reglen i betingelsen. Hvorimod mængderne i anden single-command konjugeres med komplementærmængden af betingelsen.

single-commmad (4). Afvikles som følger: Single-command afvikles, dog er der en semantisk betydning for mængderne beskrevet af reglerne i denne single-command. Mængderne i single-command konjugeres med mængden beskrevet af reglen der her virker som betingelse.

single-command (5). Afvikles som følger: Mængden, der beskrives af reglen, tilføjes til en global mængde, for alle mængder knyttet til samme slags handling. For hver af permit, deny, log og state bestanddelene af action, findes en sådan mængde hvortil der kan knyttes mængder.

rule. Evalueres som følger: Parvist kombineres de mængder der beskrives af de single-rules som denne regel består af, indtil alle er kombinerede til én mængde. Mængderne kombineres af de binære operatorer, der står mellem dem. Er der ingen operator mellem to single-rules, konjugeres mængderne de beskriver.

single-rule (1-7). Evalueres til en mængde, knyttet til et bestemt segment i en header. Det reserverede ord i starten af denne regel, svarer til et bestemt segment i headeren på en pakke. Mængden beskrevet af denne regel, er de værdier der er beskrevet på højresiden, sammenlignet med indholdet af det angivne segment af i en header.

single-rule (8). Evalueres til en mængde, knyttet til systemets ur. Det reservere ord i starten af denne regel fortæller, at værdien på højresiden skal sammenlignes med systemets ur. Mængden beskrevet af denne regel, er det tidsinterval der er beskrevet på højresiden.

single-rule (9). Evalueres til en mængde, knyttet til systemets ur. Højresidens dagsangivelse, evalueres til en mængde bestående af en enkelt ugedag.

single-rule (10). Evalueres til den mængde, der er beskrevet af den rule, der er i parentesen.

single-rule (11). Afvikles som følger: Reglen evalueres i henhold til den unaryoperator, der er placeret foran. Den semantiske betydning er, at mængden beskrevet af reglen er argument til den mængdeoperation der er knyttet til den operator der står foran den.

Eksempler

```
1  //Single-decleration
2  ip Home := 127.0.0.1
3
4  //CallCommand
5  Mail(home,9696)
6
7  //with-do strukturen
8  with dip home do {
9    ( day sat || dport <= 9696 ) &! sip 205.30.*.* -> deny;
10   sip 205.30.*.* -> permit
11  }
```

5.3.6 Hovedstruktur

Syntaks

⟨*include*⟩ ::= '`include`' '`"`' ⟨*String*⟩ '`"`' '`;`'

⟨*constdecl*⟩ ::= ⟨*singledecl*⟩ '`:=`' ⟨*expression*⟩ '`;`'

⟨*filter_def*⟩ ::= '`filter`' ⟨*identifier*⟩ '`(`' ⟨*pardecl*⟩? '`)`' '{' ⟨*command*⟩ '}'

⟨*mainfilter*⟩ ::= '`main`' '`[`' ('`permit`' | '`deny`') '`]`' '{' ⟨*command*⟩ '}'

Semantik

include. Inkluderer det af String specificerede filnavn.

constdecl. Tilskriver højresidens værdi til konstanten, der erklæres i singledecl.

filter_def. Laver et filter. Filtret får defineret de parametre, den skal have i pardecl. Kommandoen i command bindes til filtret, så det kan afvikles når filtret kaldes.

mainfilter. Danner mainfiltret for pakkefiltret, hvor action definerer pakkefiltrets standardhandling. Denne standardhandling må kun være permit eller deny.

Eksempler

```
1 //Et main filter
2 main [deny] {
3   myfilter{};
4 }
```

5.3.7 Startproduktion

Syntaks

$\langle MFL \rangle ::= \langle include \rangle* \ (\ \langle constdecl \rangle \mid \langle filter\_def \rangle\ )* \ \langle mainfilter \rangle$

Semantik

MFL. Includes, constdecl, og filter_def danner de omgivelser som filtrene og main filtret skal evalueres i. Altså fastlægges hvilke filtre og konstanter, der kan kaldes og henvises til i de forskellige regler og kommandoer i filen.

Eksempler

```
1  //Et lille MFL eksempel
2  #include "staff.mfl"
3  ip printserver = 192.168.10.23;
4
5  filter myfilter (ip klient) {
6    sip klient -> permit
7  }
8
9  main [deny] {
10    myfilter(staff); //staff inkluderet fra staff.mfl
11 }
12
```

5.4 Håndtering af overlap

Da en regel beskriver en mængde af pakker, er der mulighed for at lave regler, der overlapper hinanden. Der opstår ingen konflikt hvis to overlappende regler har samme handling, som i figur 5.1(a) på den følgende side, hvor der er overlap fordi reglerne ikke kigger på samme segment i headeren. Da reglerne begge vil tillade en pakke fra IPen 192.168.1.1 med port 80 som destination port, opstår der ingen konflikt.

Hvis reglerne derimod har samme handling, skal der prioriteres. Da filtre kan kalde hinanden, ville det gøre programmering af et pakkefilter besværlig, hvis overlap var lovlig og rækkefølgen af regler bestemte hvilke regler der var stærkest. Uden overlap gøres programmering af pakkefiltre også sværere, da alle regler da skulle være meget specifikke. Hvis man for eksempelvis ikke under nogle omstændigheder vil lade kommunikation med en bestemt mængde af porte finde sted, skulle dette inkluderes i hver enkelt regel filtret igennem, se kildekode 5.1 for et eksempel.

Kildekode 5.1 Upraktisk metode til at undgå trafik på port 80.

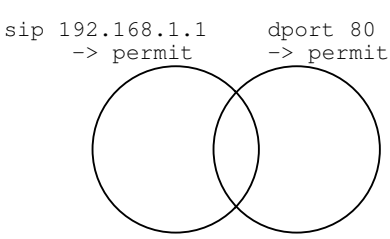
```
1 main[deny]{
2   ...
3   !dport 80 sip 192.168.1.1 -> permit;
4   !dport 80 dip 192.168.1.250 day mon -> permit;
5   !dport 80 sip 192.168.1.2.dip 192.168.1.3-> permit;
6   ...
7 }
```

Istedet er MFL designet således at man opnår samme effekt ved at skrive som det ses i kildekode 5.2 :

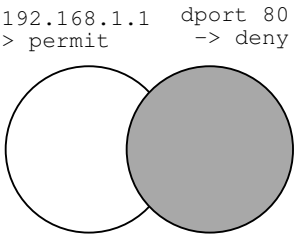
Kildekode 5.2 Bedre metode til at undgå trafik på port 80.

```
1 main[deny]{
2   ...
3   sip 192.168.1.1 -> permit;
4   dip 192.168.1.250 day mon -> permit;
5   sip 192.168.1.2.dip 192.168.1.3-> permit;
6   dport 80 -> deny;
7   ...
8 }
```

Overlap i regler i MFL, prioriteres ikke efter position, men efter om de har samme handling som standardhandlingen. Overlapper to regler, og har de modstridende handlinger, er den regel der har samme handling som main filtrets standardhandling stærkest, se figur 5.1(b). Overlapper to regler, og den ene indeholder log i dens handling, logges også fællesmængden.



(a) Eksempel på overlap mellem regler med samme handling.



(b) Overlap hvor deny reglen er stærkest fordi dens handling er den samme som standardhandling.

Figur 5.1: Overlap mellem regler.

5.5 Sprogkategori

Grammatikken for MFL er blevet designet til at opfylde LL(k)-sprogkategorien, for dermed at kunne benytte recursive-descent (top-down) parsing. LL betyder "Left to right scan Leftmost derivation" og (k) betyder det antal tokens parseren skal kigge fremad (lookahead), for at

kunne bestemme en sætnings kategori under parsing. Normalt tilstræber man at opnå LL(1). Kriterierne for LL(1) er følgende:

- Hvis grammatikken indeholder X|Y, skal *starters*[[X]] og *starters*[[Y]] være forskellige.
- Hvis grammatikken indeholder X*, skal *starters*[[X]] være forskellige fra det sæt af tokens der kan komme efter X* i den givne kontekst.

Med starters menes de start-tokens, der er mulige med det givne lookahead. Hvis grammatikken ikke umiddelbart lever op til LL(1) kan man bl.a. benytte "left factorization" og "left recursion elimination" [WB00, p. 80-83] for at tilpasse denne.

I forbindelse med MFL har det ikke ikke været muligt at løse alle problemerne med de førnævnte regler, idet IP ranges samt IP adresser med og uden mask skulle kunne skelnes fra hinanden. Derfor er der behov for et lokalt lookahead på 8.

For at kontrollere at grammatikken er entydig, samt at den overholder det generelle lookahead på 1 og det lokale lookahead på 8, er parsergenereringsværktøjet JavaCC blevet anvendt. Dette værktøj foretager nemlig en kontrol af grammatikken samtidig med at den danner en lexer og en parser. Ved implementeringen i JavaCC var der problemer med at få det lokale lookahead til at virke efter hensigten. Derfor blev det implementeret uden det lokale lookahead på 8. Resultatet blev at JavaCC kom med advarsler, som alle var relateret til det punkt hvor det lokale lookahead skulle være foretaget. Dette lookahead er blevet kontrolleret og godkendt manuelt og vil blive håndteret i designet af parseren. Dermed er MFL blevet kontrolleret til at være LL(1) med et lokalt lookahead på 8. Dette resulterer i en LL(8)-kategori for MFL.

Implementeringen i JavaCC er vedlagt på cden under: (◎ *JavaCC* `code/MFL.jj`).

Hermed er sprogets syntaks og semantik fastlagt. Således kan design af kompileren til MFL begynde, hvor syntaksen danner grundlag for lexer og parser og semantikken danner grundlag for semantisk analyse.

KAPITEL 6

Kompilerdesign

Idet target og source sprogene er blevet fastlagt, kan de forskellige faser i kompileren designes. Efter en kort beskrivelse af funktionaliteten af de enkelte kompilerfaser, fastlægges grænsefladerne mellem disse. Endeligt designes de enkelte faser i kompileren, til oversættelse fra MFL til MTIDD.

6.1 Opbygning af kompileren

Kompileren er opbygget således, at følgende opgaver skal løses under kompilering.

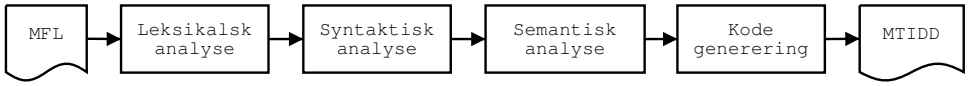
Leksikalsk analyse. Indlæsning af include filer, frasortering af kommentarer samt egentlig leksikalsk analyse.

Syntaktisk analyse. Parsing af kildekoden med henblik på at afgøre om denne kan dannes udfra den opstillede grammatik.

Semantisk analyse. Identifikation af attributter, typecheck af sætninger samt sikring mod cirkulære kald.

Kodegenerering. Generering af MTIDD.

Den beskrevne kompiler har, som vist i figur 6.1, fire opgaver. Disse opgaver løses i tre faser, idet leksikalsk analyse betragtes som en del af den syntaktiske analyse.



Figur 6.1: Opgaverne, der skal udføres under kompilering.

I det følgende gives en kort gennemgang af de enkelte faser, herunder hvilke opgaver de konkret skal løse.

6.1.1 Syntaktisk analyse af MFL

Formålet med den syntaktiske analyse er, at afgøre om den givne kildekode kan genereres udfra den opstillede grammatik og derved er et gyldigt MFL program.

Den leksikalske analyse udføres af en lexer, som indlæser kildekoden karakter for karakter. Sammenhængende karakterer grupperes i ord kaldet lexems. Hvert lexem kategoriseres udfra en række tokens, som baserer sig på sprogets syntaksdiagram (se appendiks C). [ASU86] Preprocessordirektivet include og kommentarer karakteriseres dog ikke. Læser lexeren en kommentarer frasorteres den blot, læses et include, inkluderes den efterfølgende fil, som en del af kildekoden.

Den primære del af den syntaktiske analyse, foretages af en parser, som danner et AST udfra de modtagne lexems. Dette sker ved at matche kombinationer af ord mod syntaksdiagrammet. I MFL er en regel (rule), et eksempel på kombinationer af ord, der kan parses mod syntaksdiagrammet. På denne måde skabes de interne knuder i ASTet, hvor hver knude indeholder en beskrivelse af betydningen af en række lexems. Samlet set betyder dette, at parseren laver et gennemløb af kildekoden, for at indlæse og kategorisere alle lexems i et MFL dokument og producere et AST.

6.1.2 Semantisk analyse af MFL

I semantisk analyse undersøges om det indlæste dokument, lever op til de kontekstuelle krav, der stilles til MFL. De kontekstuelle krav er beskrevet i en uformel semantik i afsnit 5.3 på side 34.

Semantisk analyse deles op i tre delaktiviteter. Identifikation af konstanter og filtre herunder main filtret, typecheck af udtryk samt sikring mod cirkulære kald.

Identifikation

Identifikation skal identificere konstanterklæringer og filterdefinitioner, samtidig med at sikre, at der i kildekoden kun anvendes erklærede konstanter og definerede filtre. Derudover skal det også sikres, at der ikke er cirkulære filterkald eller konstanterklæringer i kildekoden.

MFL anvender to typer konstanter, der er henholdsvis globalt og lokalt erklærede. Dette kaldes at sproget har en flad blokstruktur [WB00].

For at kunne lave en sammenhæng mellem kald af filtre og erklæring af konstanter, opbygges to tabeller - en identifikationstabel og en filtertabel. Identifikationstabellen bruges til registrering af globalt erklærede konstanter og indeholder hver konstants navn, type og værdi. I filtertabellen registreres hvert filter, det vil sige navn, parametre og krop.

Identifikations- og filtertabel opbygges ved gennemgang af alle knuder i ASTet, hvor konstanter og filtre tilføjes deres respektive tabeller, når disse erklæres. Under opbygningen skal det sikres, at der ikke eksisterer filtre, globalt erklærede konstanter eller formelle parametre med samme navn.

Typecheckning

MFL er, som beskrevet i afsnit 5.2 på side 33, strongly typed, derfor kan typecheck for alle konstanter gøres ved kompileringstidspunktet. Typecheck af MFL er en generel sikring, af at udtryk og typer giver mening, i den sammenhæng de optræder.

Fra sprogspecifikationen for MFL, afsnit 5.3 på side 34, kan udledes, at udtryk (expression) som anvendes ved erklæring af konstanter og i filterkald. For konstanterklæringer og regler er der en type tilknyttet for hvert udtryk.

For konstanterklæringer betyder dette, at der skal sikres, at et udtryk evaluerer til noget meningsfyldt for den tilknyttede type. Når et filter kaldes, skal det sikres, at de udtryk, der indgår som aktuelle parametre også passer til de formelle parametre. Tilsvarende skal typer i udtryk passe sammen i regler.

6.1.3 Kodegenerering

For at kunne generere en MTIDD skal det dekorerede AST tages i anvendelse.

Indledende evalueres alle filtre begyndende i main filtret. Et filter evalueres ved at evaluere alle filterkald, regler og kontrolstrukturer i dette. Når et filter evalueres, erstattes de formelle parametre med de aktuelle.

Når en regel evalueres returneres en IDD. Regler med samme handling disjungeres, så en stor IDD for hver handling (permit, deny og log) skabes. Således opbygges stadig større IDDer for hver af de tre handlinger.

Til slut genereres en MTIDD, ved kombination af de tre IDDer. Herunder tages hensyn til standardhandling defineret i main filtret, det vil sige, at hvis intet andet er beskrevet, indsættes standardhandlingen. Hvis der er overlap mellem to regler, med forskellig handling, har standardhandlingen præcedens.

6.2 Grænseflader og datastrukturer

I dette afsnit beskrives grænseflader mellem de enkelte faser i kompileringen, det vil sige de datastrukturer, der er imellem faserne.

Indledningsvis beskrives en intern grænseflade i den syntaktiske analyse, altså mellem lexeren og parseren. Efterfølgende beskrives ASTet, som danner grænsefladen mellem den syntaktiske og semantiske analyse.

Dernæst beskrives det dekorerede AST , der sendes videre til kodegenereringen. Desuden er kildekoden en grænseflade til syntaktisk analyse, og MTIDD en grænseflade til kodegenerering, disse eksterne grænseflader vil ikke blive beskrevet.

6.2.1 Efter leksikalsk analyse

Lexeren gennemløber kildekoden og identificerer lexems, hvorefter disse inddeles i en række tokens. Disse tokens er vist i tabel 6.1.

Token	Beskrivelse	Lexem eksempler
Keyword	Reserverede ord	filter ACK
Symbol	Binære operatorer og andre symboler.	:= , }
Ident	Identifier såsom navnet på et filter eller en ip.	myFilter serverIp
Int	Heltal.	127 255
Unrec	Karakterer som ikke genkendes.	æ %

Tabel 6.1: De benyttede token typer, som lexeren inddeler lexems i.

Udfra de valgte tokens opbygges en type token_t, vist ved tabel 6.2 på næste side. Tkeyword oprettes med et argument af typen keyword_t, der er de forskellige mulige keywords i MFL. Tilsvarende oprettes Tsymbol, med et argument af typen symbol_t, som beskriver alle symboler i sproget. Både keyword_t og symbol_t er beskrevet nærmere i appendiks F på side 100.

Udfra typen token_t opbygges en type lexem_t. Denne er en record, som indeholder dels typen (hvilket token) samt lokation for de enkelte lexems. Lokation hentyder i den sammenhæng

Navn	`token_t`

Metatype	`Tkeyword of keyword_t eller`
	`Tsymbol of symbol_t eller`
	`Tident of string eller`
	`Tint of int eller`
	`Tunrec of string`

Tabel 6.2: Typen `token_t` som benyttes af lexeren.

til linie nummer (line_l), start og slut kolonne nummer (s_col_l og e_col_l) samt filnavnet (filename). Typen `lexem_t` er vist ved tabel 6.3.

Navn	`lexem_t`

Metatype	`(token_l : token;`
	`line_l : int;`
	`s_col_l : int;`
	`e_col_l : int;`
	`filename_l : string) : Record`

Tabel 6.3: Typen lexem_t som benyttes af lexeren.

Lexeren returnerer en liste af `lexem_t`, som parseren efterfølgende skal analysere, denne kaldes lexemliste.

6.2.2 Efter syntaktisk analyse

Grænsefladen mellem syntaktisk og semantisk analyse består af et AST , der beskriver kildekoden på en struktureret og entydig måde.

I dette afsnit beskrives typerne, der benyttes til at opbygge et AST. Typerne til ASTet er, med små ændringer, en konvertering af EBNFen. Typerne er nærmere beskrevet i appendiksafsnittet E.2 på side 93.

For at muliggøre fejlmeldinger i semantisk analyse, der oplyser hvor den fundne semantiske fejl er, indeholder typerne i ASTet et eller to heltal. Hvert heltal henviser til et token på en specifik plads i lexemlisten (svarende til nummeret). Typer, der består af enkelte tokens, indeholder et enkelt heltal, mens typer der består af flere tokens, indeholder et tal for start tokenet og et for slut tokenet.

Her følger to eksempler, henholdsvis eksempel 3 der beskriver en specifikation af en AST type, derefter eksempel 4 på den følgende side, der er et eksempel på et AST for et eksempelfilter.

Eksempel 3 Typen `range_t` anvendt i AST. Følgende tabel viser specifikationen af typen range_t der anvendes i ASTet. Typen `range_t` stammer ikke direkte fra EBNFen, men er istedet en hjælpetype til typen `mrange_t`.

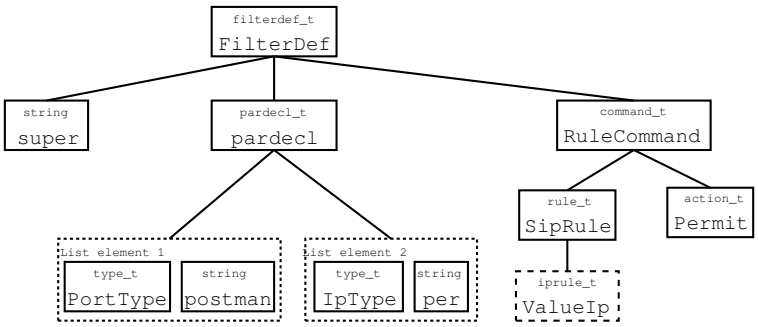
I "Metatype" ses det at `range_t` er en record bestående af to heltal, henholdsvis low og high, beskrivende øvre og nedre grænse af et interval.

Navn	`range_t`
Metatype	`(low: int; high:int): Record`
Beskrivelse	Kan indeholde en range. Bruges i `mrange_t`

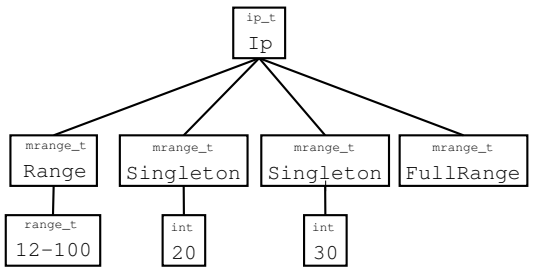
■

Eksempel 4 AST for en filterdefinition. Figur 6.2 viser et AST for følgende filter:
`filter super(port postman, ip per){dip 12-100.20.30.* -> permit;}`
`ip`, markeret med stiplet linie, er for overskuelighed lavet i sin egen AST i figur 6.3.

Hver kasse markerer en instans af en type, den øverste tekst er typens navn, nederste tekst er metatypen. Eksemplet er en definition af et filter med de formelle parameter `postman` og `per`, som er af typen `port` henholdsvis `ip`. I filtrets krop, mellem `{}`, ses en regel der tillader alle pakker med destination IP i rangen: `12-100.20.30.*`. Det bemærkes at ingen af parameterne benyttes til filtret bruges til noget, men eksemplet er valgt til demonstrationsformål.



Figur 6.2: Eksempel på AST filterdefinition.



Figur 6.3: AST for en IP.

■

6.2.3 Efter semantisk analyse

Resultatet af semantisk analyse, er en dekoreret version af ASTet indeholdende en identifikationstabel og en filtertabel, samt en beskrivelse af main filtret.

I det dekorerede AST er typerne, med fire undtagelser, fuldstændig analoge til de typer der bliver brugt til det ikke dekorerede AST, der er beskrevet i appendiksafsnittet E.2 på side 93. Disse ændringer er beskrevet i appendiksafsnittet E.3 på side 98

6.3 Design af lexer

Lexererens opgave er at scanne alle karakterer i kildekoden med henblik på at identificere en række lexems samt kategorisere disse udfra en række tokens. Derudover skal lexeren behandle en række prækompiler direktiver i kildekoden, såsom inkludering af filer.

Da de inkluderede filer tillige kan indeholde include direktiver, er der mulighed for cirkulære referencer, hvorved lexeren kan komme i en uendelig løkke af includes. Dette undgås ved brug af en tabel, indeholdende hver enkelt fil, der inkluderes. En fil inkluderes således kun, hvis dens navn ikke allerede findes i tabellen.

Inkluderede filer, må ikke indeholde et main filter, da der i den samlede kildekode kun må være et main filter.

Endelig er det også lexerens opgave at holde styr på linie og karakternumre for hver enkelt lexem, samt at fjerne eventuelle kommentarer fra kildekoden.

Til hver sourcefil der benyttes tilknyttes en type string_to_lex_t vist ved tabel 6.4.

Navn	`string_to_lex_t`
BNF	Ingen - Hjælpetype

Metatype	`(string : string;`
	`current : int;`
	`size : int;`
	`line : int;`
	`col : int;`
	`filename : string)`

Tabel 6.4: Typen string_to_lex_t som intern repræsentation af en kildekodefil.

I typen string_to_lex_t sættes *string* (selve kildekoden), *size* (længden af strengen, som indeholder kildekoden) samt *filename* (navnet på filnavnet, som indeholder kildekoden) når en instans af typen oprettes, hvorefter disse forbliver konstante.

De resterende elementer i string_to_lex_t ændres løbende igennem forløbet af lexeren og benyttes derved som interne tællere for, hvor langt lexeren er nået i den pågældende kildekode. *current*, *line* og *col* refererer til pågældende position, linie og kolonnenummer i kildekoden.

Hver gang lexeren læser en ny karakter i kildekoden, inkrementeres *col* og *current*. Ved newline karakteren nulstilles *col*, mens *line* og *current* begge inkrementeres.

Den primære lexerfunktion, tager som argument en variabel af typen string_to_lex_t, udfra hvilken den processerer kildekoden. Karakter for karakter læses kildekoden og udfra den læste karakter afgøres, hvilket mulig type lexem der er påbegyndt (eksempelvis Identifier eller Symbol)

Lexeren gennemløber hele kildekoden samt evt. inkluderede filer og returnerer en liste med lexems og tilhørende lokation i kildekoden. Denne liste danner udgangspunkt for parserens behandling af kildekoden.

For en nærmere gennemgang af funktionerne i lexeren, henvises til appendix F på side 100.

6.4 Design af parser

Under den syntaktiske analyse skal der opbygges et AST, bestående af typerne beskrevet i appendiks E.2 på side 93. Efter analysen sendes dette AST videre til den semantiske analyse.

Parseren kan enten anvende en top-down eller en bottom-up strategi til at parse koden. En bottom-up parser starter med at kombinere de enkelte tokens til lovlige produktioner i MFL syntaks. En top-down strategi begynder med den overordnede produktion i MFL syntaksen,

og bruger denne som udgangspunkt til at se om det er muligt at lave en lovlig sammensætning der passer til listen af tokens. MFL parseren anvender en top-down strategi, da denne strategi giver mulighed for at anvende recursive-descent parser algoritmen.[WB00, p. 83-109]

Parseren kan ved at læse enkelte tokens bestemme, hvilket mønster det efterfølgende kode må have. Dette svarer til den lovlige syntaks, for non-terminaler defineret for MFL. Ved f.eks. at læse tokenet "Filter" er det sikkert at der følger en filterdefinition, såfremt syntaksen er korrekt.

Når et token som "Filter" læses, opnås vished om, at en filterdefinition følger, men det er for tidligt at oprette variablen af denne type i AST. Dette skyldes at variablen, skal bestå af andre typer og der er behov for at læse flere tokens, før disse typer kan oprettes med de rigtige værdier. Derfor kaldes endnu en funktion, som fortsætter den syntaktiske analyse fra "Filter" tokenet og fremefter. Den videre analyse sikrer at syntaksen er korrekt for en filterdefinition. Denne funktion sammensætter samtidig den variabel, der kommer til at repræsentere filterdefinitionen i AST. Funktionen kan kalde andre funktioner, der kan sammensætte de variable, som filterdefinitionens variabel skal bestå af.

Som nævnt i afsnit 5.5 på side 42, er det ikke muligt at parse samtlige dele af MFL ved hjælp af en LL(1) parser, og dermed kan recursive-descent algoritmen ikke bruges i disse dele af parsefasen. Ved parsing af ⟨*expression*⟩ kræves således at der kigges længere frem end til næste token. Først kigges frem til næste token igen for at bestemme om, produktionen er et interval eller en tid. Hvis produktionen var et interval kigges derefter videre igen for at bestemme om, der skal parses en port eller en IP adresse. Denne udvidelse medfører blot, at der arbejdes med et større lookahead men ellers vil princippet være det samme.

Et andet problem for at kunne lave en LL(1) parser er IP adresseintervaller, og netværksadresser. Det ville ikke give mening at kunne skrive et interval i en netværksadresse, så derfor skal det kun være muligt at skrive enkeltstående tal i dette tilfælde. I stedet for at lave et lookahead på otte, som der ville kræves i dette tilfælde, kan der i stedet med fordel laves en funktion der undersøger om en netværksadresse er korrekt, hvis der findes en skråstreg. Derved er det også muligt at ændre netværksadresser til de tilsvarende IP adresseintervaller, hvilket fomindsker redundans i de datatyper der skal bruges til parsetræet.

I kildekode 6.1 ses et eksempel på hvor lookahead skal være længere end 1. I eksemplet ses tre forskellige filterkald, det første kalder filter1 med en port som aktuelt parameter, det andet kald er til filter2 med en IP adresse som parameter. Den tredie er kaldet til filter3 og her en IP adresse med maske. Alle disse skal parses til at passe til AST typerne. I filter1 kræves et lookahead på to for at afgøre at tallet er en port og ikke en IP, tid eller portrange. Kigges på filter2 og 3 ses det, at for at afgøre om parametren i filter3 er en IP med eller uden maske, bliver parseren nødt til at kigge otte tokens frem.

Kildekode 6.1 Eksempel hvor der er behov for lookahead større end en.

```
1 //Filterkald
2 filter1(53);
3 filter2(192.168.30.2);
4 filter3(192.168.255.0/24);
```

Ved at lave en funktion for alle lovlige mønstre af non-terminaler i MFL, designes en parser, der genererer et AST, som sendes videre til semantisk analyse. Disse funktioner kan ses i appendix F på side 100.

Funktionen skal ved at læse enkelte tokens, afgøre hvilke funktioner der skal parse det aktuelle mønster af non-terminaler. Det vil sige hvilken funktion der, for et givent mønster af non-terminaler, opretter variable til ASTet og afgør om syntaksregler for mønstret overholdes.

6.5 Design af semantisk analyse

Hovedformålet med semantisk analyse er at opbygge et dekoreret AST, som er fri for semantiske fejl. Som input til den semantiske analyse bruges ASTet som er opbygget af parseren, dette er et `ast_mfl_t` element. Resultatet, der anvendes i kodegenerering, bliver to tabeller, filtertabel, som indeholder dekorerede filtre og identifikationstabel, som indeholder dekorerede konstanter, samt et dekoreret main filter, som samlet udgør et dekoreret AST repræsenteret ved en deco_mfl_t.

6.5.1 Resultat af semantisk analyse

Nærmere beskrivelse af de tre overordnede dele af det dekorerede AST.

- Identifikationstabellen består af en liste af tupler med tre variabler, type, navn og udtryk.
- Main filtret består af en 3-tupel af en standardhandling for main filtret samt en kommando.
- Filtertabellen består af en liste af tupler, hvor hver tupel repræsenterer et filter, bestående af navn, en række parametererklæringer samt en kommando.

De to tabeller bliver hashtabeller, hvor der bliver hashet på henholdsvis konstantnavnet og filternavnet. Grunden til valget af hashtabeller er, at det er en hurtig måde at finde et element ud fra en værdi.

6.5.2 Semantiske check

Herunder listes det, der skal undersøges i den semantiske analyse.

- Ved konstanterklæringer undersøges om den værdi, givet ved et udtryk, der tildeles til konstanten passer til konstantens typen, samtidig skal sikres, at der ikke opstår cirkulære referencer mellem konstanter.
- Ved IP-, port- og timeudtryk undersøges om den medfølgende værdi er inden for pågældende types gyldige interval. Det gyldige interval for hver enkelt IP byte er 0 til 255. Hvis IP udtrykket indeholder en `FullRange` evalueres denne til en `deco_range_t` indeholdende intervallet 0 til 255. For port er det lovlige interval 0 til 65535. Indeholder port udtrykket en `FullRange` skal den laves om til en `deco_range_t` indeholdende intervallet 0 til 65535. Et tidsudtryk består af to tidspunkter, hver indeholdende en angivelse af time og minut, hvor time skal være i intervallet 0 til 23, for minut er intervallet 0 til 59. Derudover skal det første tidspunkt være tidligere end det andet.
- Når et filter kaldes, skal det sikres, at de aktuelle parametre passer med de formelle parametre i parametererklæringen for det kaldte filter. Hvis filtret indeholder kald til andre filtre skal det desuden undersøges om der opstår cirkulære kald.
- Når en regel erklæres, skal det undesøges om de typer den indeholder passer med det udtryk der knyttes til hver type. Alle regler kan indeholde en streng, svarende til en konstant i identifikationstabellen, eller navnet på en formel parameter i det filter som reglen indgår i.

- Hvis et udtryk er et `ast_any_t` udtryk skal det, hvis det er af typen IP, ikke laves om til et `deco_any_t` men derimod til et IP udtryk hvor hver del af ip adressen er en range fra 0 til 255. Det samme gør sig gældende ved port, her skal den laves om til et port udtryk med den maksimale range.

- Der skal checkes, at det samme navn ikke tildeles flere konstanter i det samlede firewall filter, flere filtre i det samlede firewall filter eller flere formelle parametre i det samme filter.

6.5.3 Overordnet forløb

Den semantiske analyse udføres, efter at parseren har opbygget et AST. Den starter når funktionen `semantik_analyzer` kaldes med det AST, som parseren opbyggede, som parameter. Derefter kaldes en række funktioner, som opbygger et dekoreret AST og checker for semantiske fejl. Tilsidst returnerer `semantik_analyszer` en deco_mfl_t.

En nærmere beskrivelse af hvordan de forskellige funktioner er designet kan ses i appendiks G på side 115.

6.6 Design af kodegenerering

Den sidste del af kompileren er kodegenereringsfasen. Denne fases opgave er, at generere den endelige targetmodel for kompileren, hvilket i dette tilfælde vil sige en MTIDD, der repræsenterer det opbyggede filter beskrevet i kildekoden.

Input til kodegenereringsfasen er det dekorerede AST, som returneres fra semantisk analyse.

Endelig er det kodegeneringsfasens opgave, at frasortere de elementer i det dekorerede AST, som ikke er valgt implementeret. Dette er eksempelvis anmodninger om stateful validering. Kodegenereringsfasen udskriver en advarsel, når disse elementer ignoreres.

6.6.1 Opbygning af IDDer

Efter den semantiske analyse er gennemløbet, returneres en instans af typen deco_mfl_t. Dette er en 3-tupel bestående af en global filtertabel (deco_filtertable_t), en global identifikationstabel (deco_identtable_t) samt en type, der beskriver main filtret. Sidstnævnte type (deco_mainfilter_t) er en totuple bestående af en standardhandling for main filtret samt selve kommandoen for main filtret (hhv deco_action_t og deco_command_t). Instansen af sidstnævnte type, danner udgangspunkt for kodegenereringen. Der henvises til afsnit 6.2.3 på side 48 for nærmere beskrivelse af typerne i det dekorerede AST.

Grundidéen i kodegenereringsforløbet er, som udgangspunkt, at definere tre IDDer - en for hver handling, der kan optræde i filtret.

- IDD_P - Permit IDD, som er en FALSE terminal.

- IDD_D - Deny IDD, som er en FALSE terminal.

- IDD_L - Log IDD, som er en FALSE terminal.

Efterfølgende gennemløbes det dekorerede AST, hvorunder enkelte regler (atomer[1]) kombineres til IDDer, som igen kombineres til større IDDer. Når en handling, for eksempel permit identificeres for en række regler, disjungeres IDD_P med den fundne IDD - på tilsvarende vis opbygges IDD_D og IDD_L. Efter gennemløbet af det dekorerede AST, kombineres IDD_P og IDD_D under hensyntagen til standardhandlingen, hvorefter IDD_L tilføjes, således en MTIDD opnås.

I det følgende beskrives med større detaljeringsgrad hvorledes de enkelte sproglige elementer i MFL giver anledning til forskellige IDDer, samt hvorledes den endelige konstruktion af MTIDDen foretages.

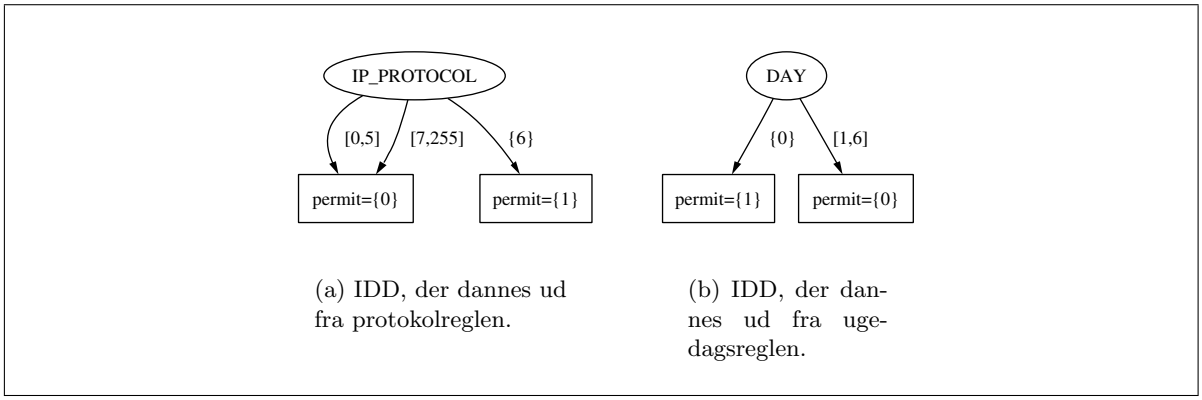
Kodegenerering udfra én regel

Indledningsvis betragtes et simpelt main filter med en enkelt regel. Dette er vist ved kildekode 6.2.

Kildekode 6.2 Et simpelt main filter med standardhandlingen deny samt en enkelt regel.

```
1 main [deny] {
2   proto TCP && day sun -> permit
3 }
```

Idéen i opbygningen af den samlede IDD for reglen er, først at nedbryde reglen i en række delregler (atomer). I dette tilfælde vil der være tale om deco_protorule_t og deco_dayrule_t. Ud fra disse opbygges to IDDer, som efterfølgende kombineres efter den ønskede logiske operation (i dette tilfælde konjunktion). De to delregler er vist ved figur 6.4, mens den resulterende permit IDD er vist ved figur 6.5 på næste side. Det bemærkes at protokol 6 er TCP, mens ugedage regnes således 0 er søndag og 6 er lørdag.

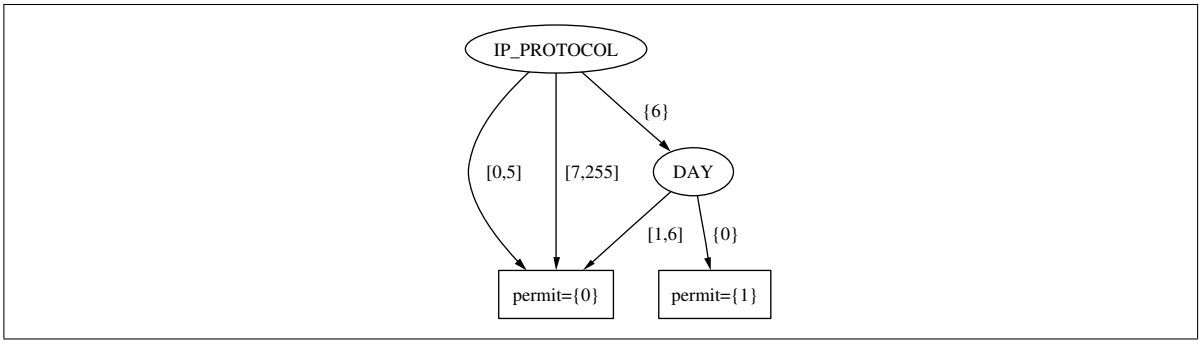


(a) IDD, der dannes ud fra protokolreglen.

(b) IDD, der dannes ud fra ugedagsreglen.

Figur 6.4: De to del IDDer, som opnås fra kildekode 6.2.

[1] Atomare regler er for eksempel sip og dport



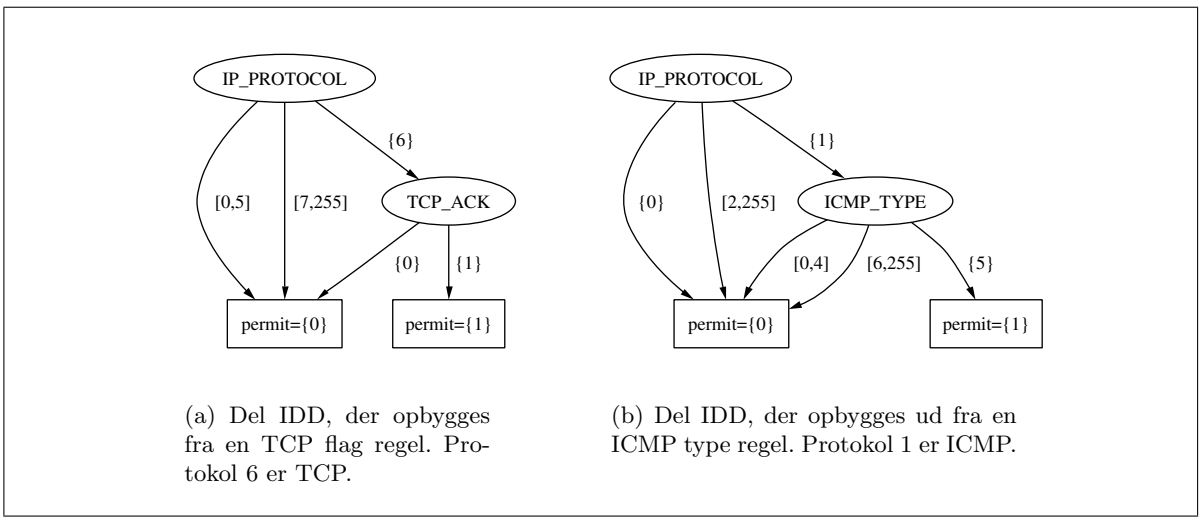
Figur 6.5: Den resulterende permit IDD, der opnås ved konjunktion af protokol IDDen og ugedags IDDen vist ved figur 6.4 på foregående side.

I kildekode 6.2 på forrige side er benyttet regler af typen *proto* og *day*. Disse regler er simple i den forstand, at de kun siger noget om protokollen henholdsvis ugedagen, og ikke har yderligere implicit information i sig. Der findes imidlertid en række regler i MFL, som er protokol specifikke. Det er eksempelvis source og destination port (*dport* og *sport*), der implicit kræver, at protokollen skal være TCP eller UDP. Tilsvarende udelukker TCP flag såsom SYN, ACK og FIN alle andre protokoller end TCP, mens specifikation af ICMP typen vil kræve at protokollen er ICMP. Ved kildekode 6.3 er vist tre eksempler på protokolspecifikke regler.

Kildekode 6.3 Et main filter med protokolspecifikke regler.

```
1 main [deny] {
2   proto ICMP [5] -> permit;
3   proto TCP [ACK] -> permit;
4   dport 80 -> permit
5 }
```

For at tage højde for denne implicitte information, skal protokol specifikke regler tilføjes en ekstra knude, under opbygningen af del IDDerne. Dette er vist for TCP flag og ICMP typer ved figur 6.6.

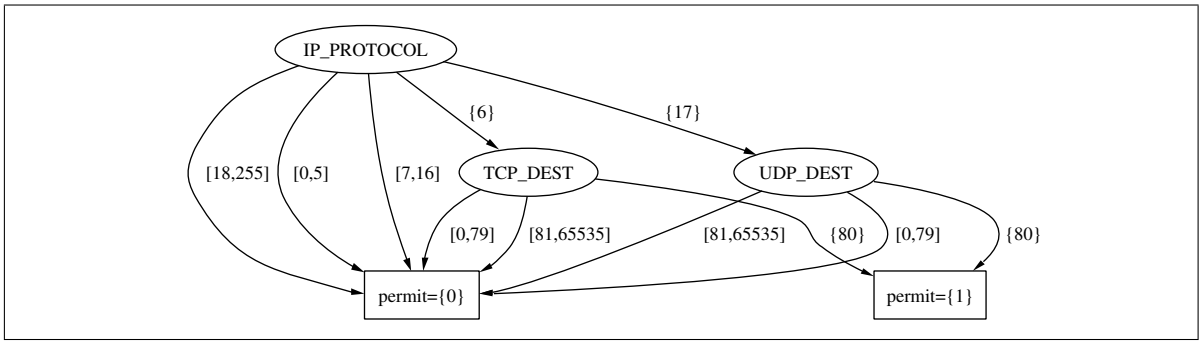


(a) Del IDD, der opbygges fra en TCP flag regel. Protokol 6 er TCP.

(b) Del IDD, der opbygges ud fra en ICMP type regel. Protokol 1 er ICMP.

Figur 6.6: Eksempler på del IDDer, der opbygges for protokolspecifikke regler.

For source og destination port (*dport* og *sport*), kan der både være tale om TCP og UDP medmindre andet er specificeret. Der opbygges derfor en del IDD, som tillader både UDP og TCP protokollen, dette er illustreret i figur 6.6 på forrige side



Figur 6.7: Eksempel på del IDD, der tillader destinationporten 80 for enten TCP eller UDP protokollen. Protokol 6 er TCP og 17 er UDP, mens knudenavnet TCP_DEST hentyder til TCP destinationport. Tilsvarende for UDP_DEST.

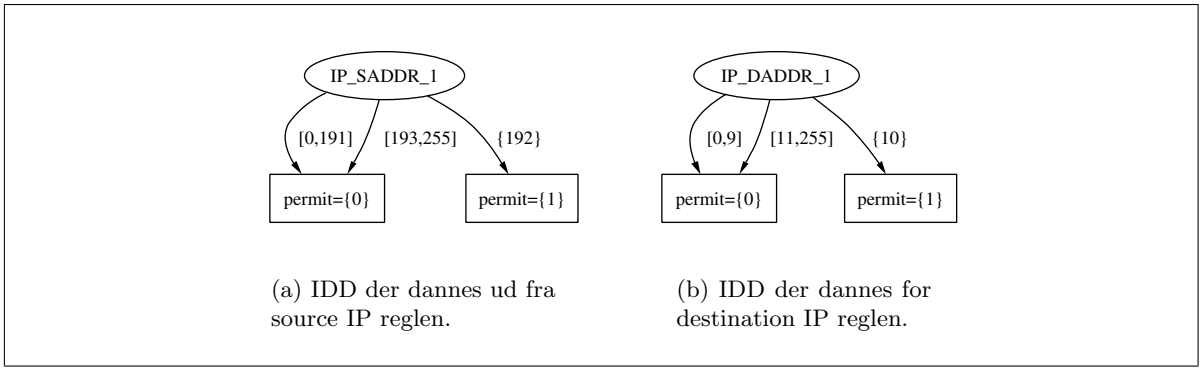
Kodegenerering udfra flere regler.

Et main filter bestående af to regler med ens handlinger betragtes. Dette er vist i kildekode 6.4.

Kildekode 6.4 Et main filter med standardhandlingen deny samt to regler.

```
1 main [deny] {
2   sip 192.168.1.0/8 -> permit;
3   dip 10.8.12.0/8 -> permit
4 }
```

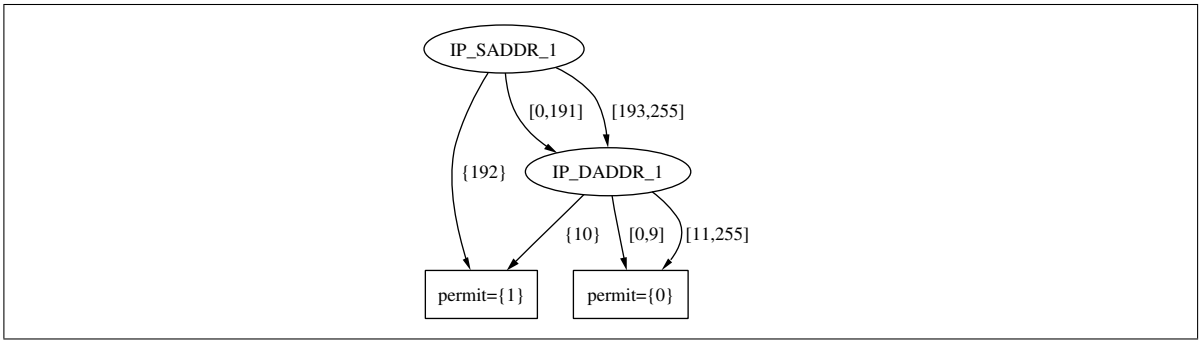
Som for det simple filter med én regel, opbygges tilsvarende en række del IDDer, som efterfølgende danner udgangspunkt for den endelig IDD. I det viste eksempel, genereres en IDD ud fra hver enkelt regel og den tilhørende handling. For eksempel 6.4 opbygges de to under IDDer, som ses i figur 6.8. Idet reglerne har samme handling, foretages en disjunktion af disse med henblik på at opnå den endelige permit IDD. Dette skal forståes således, at er blot en af reglerne opfyldt, vil pakken blive tilladt. Den resulterende permit IDD er vist ved figur 6.9 på næste side.



(a) IDD der dannes ud fra source IP reglen.

(b) IDD der dannes for destination IP reglen.

Figur 6.8: De to del IDDer, der disjungeret giver den resulterende permit IDD.



Figur 6.9: Den resulterende permit IDD, der opnås ved disjunktion af source IP IDDen og destination IP IDDen vist ved figur 6.8 på foregående side.

Kodegenerering udfra regler med kontrolstrukturer.

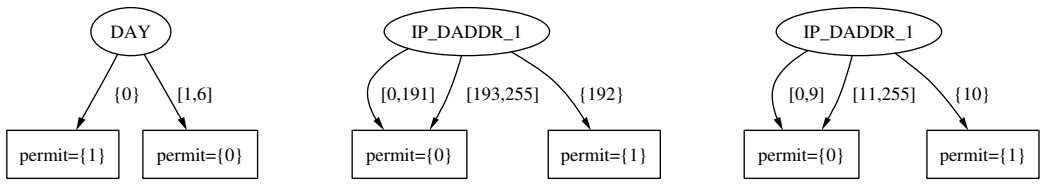
Kildekode 6.5 Et main filter med standardhandlingen deny samt en kontrol struktur.

```
1 main [deny] {
2   if day sun then
3     dip 192.168.1.0/8 -> permit
4   else
5     dip 10.8.12.0/8 -> permit;
6 }
```

I forbindelse med kontrolstrukturer, såsom if-then-else og with-do, opbygges en IDD for hver blok i strukturen, samt en IDD for betingelsen der testes på. For eksemplet vist ved kildekode 6.5, opbygges således en IDD for betingelsen (IDD_C, som i dette tilfælde er day rule), samt to IDDer for hver blok (IDD_T henholdsvis IDD_F), et eksempel på dette kan ses i figur 6.10. Disse kombineres efterfølgende ved hjælp af boolske operationer, vist ved ligning 6.1.

$$IDD_{ITE} = (IDD_T \wedge IDD_C) \vee (IDD_F \wedge \neg IDD_C) \tag{6.1}$$

For det aktuelle tilfælde bliver den kombinerede if-then-else IDD, som vist ved figur 6.11 på næste side.

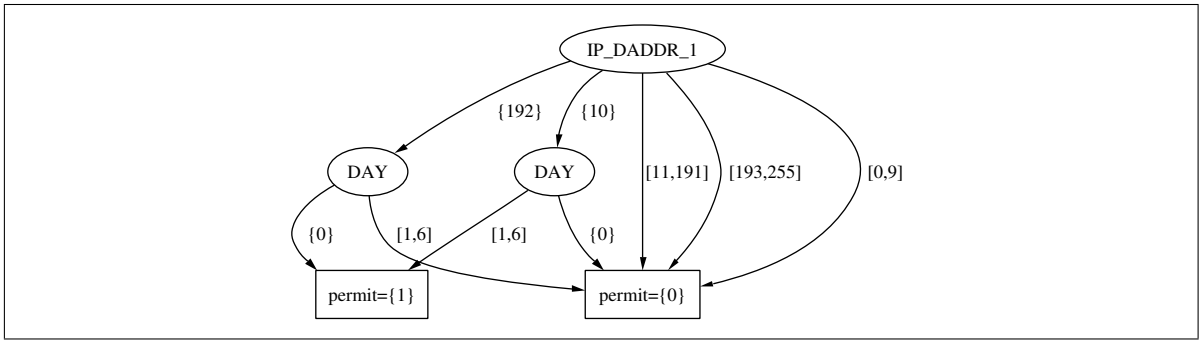


(a) IDD, der dannes ud fra betingelsen (day reglen).

(b) IDD, der dannes for TRUE blokken (første destination IP regel).

(c) IDD, der dannes for FALSE blokken (anden destination IP regel).

Figur 6.10: De tre del IDDer, der kombineret ved hjælp af ligning 6.1 giver den resulterende permit IDD.



Figur 6.11: Den resulterende permit IDD, der opnåes ud fra de tre del IDDer vist ved figur 6.10 på forrige side.

Kodegenerering udfra regler med overlap.

I det foregående blev udelukkende betragtet filtre, som beskrev opbygningen af permit IDDen. I det følgende beskrives kodegenereringens opførsel, når der specificeres både permit og deny regler, samt hvor disse overlapper.

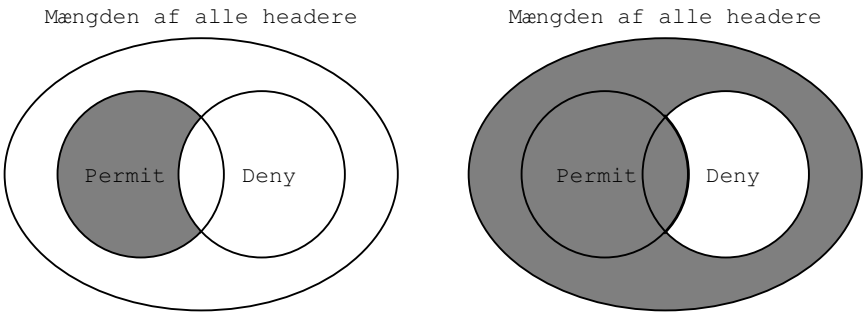
Kildekode 6.6 Et main filter med standardhandlingen deny samt overlappende regler.

```
1 main [deny] {
2   dip 192.168.1.0/8 -> permit;
3   dip 192.168.1.0/16 -> deny
4 }
```

Kildekode 6.6 viser et eksempel på et main filter, hvor to regler overlapper, idet der specificeres forskellige handlinger, for en fælles mængde af IP adresser. Som nævnt i afsnit 5.2 på side 33, vælges den gældende regel, afhængigt af main filtrets standardhandling.

Ud fra ligning 6.2 bestemmes den resulterende permit IDD. De benyttede ligninger til overlapshåndtering, er bestemt ved betragtning af mængderne vist i figur 6.12 på næste side.

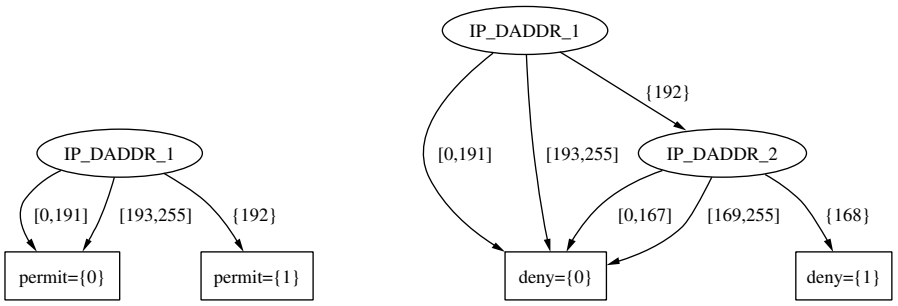
$$IDD_P = \begin{cases} IDD_P \wedge \neg IDD_D & \text{hvis standardhandling} = \text{deny}, \\ \neg (IDD_D \wedge \neg IDD_P) & \text{hvis standardhandling} = \text{permit}. \end{cases} \tag{6.2}$$



(a) Deny som standardhandling

(b) Permit som standardhandling.

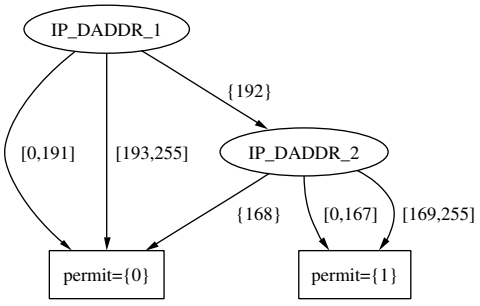
Figur 6.12: Håndtering af overlap ved kombination af deny og permit IDDerne, for givne standardhandlinger. Headere i de skraverede felter tillades.



(a) IDD, der dannes ud fra regler med handlingen permit.

(b) IDD, der dannes fra regler med handlingen deny.

Figur 6.13: De to del IDDer, der tilsammen beskriver den resulterende permit IDD.



Figur 6.14: Den resulterende permit IDD, der opnåes ved kombination af permit og deny IDDen fra figur 6.13, ved brug af ligning 6.2 på foregående side.

6.6.2 Håndtering af aktuelle parametre i filterkald.

I MFL kan filtre anvendes med parametre. Dette har betydning for hvilke konstanter, der er tilgængelige for et kaldt filter, når dette evalueres. De globalt erklærede konstanter, findes i identifikationstabellen, der opbygges under semantisk analyse, dette er beskrevet i afsnit 6.5 på side 51, men i denne identifikationstabel findes ingen information om filterparametre.

Når et filter evalueres, skal de formelle parametre derfor erstattes med de aktuelle parametre, dette gøres i MFLkompileren ved at opbygge endnu en identifikationstabel, men denne tabel indeholder kun parametre i filtret. En sådan lokal identifikationstabel gælder for filterkroppen i et filter, det vil sige for alle deco_command_t i filtret. Når kompileren finder et nyt filterkald, opbygges på ny en identifikationstabel, som bruges for alle deco_command_t i dette filter og så videre.

Dette design indebærer at funktionen, der evaluerer elementer af typen deco_command_t, i det dekorerede AST, altid skal have to identifikationstabeller som parametre: den globale og lokale identifikationstabel. Dette er også tilfældet for funktionen der anvendes til at finde værdier i identifikationstabellerne. Sidstnævnte funktion skal først søge i den lokale identifikationstabel og derpå i den globale.

Kildekode 6.7 er et eksempel på et filterkald fra main filter til filtret "SomeFilter". Ved evaluering af main filtret, starter kompileren med evaluering af filterkaldet.

Kildekode 6.7 Et main filter med filterkald.

```
1 ip SomeIp := 127.0.0.1;
2 filter SomeFilter(ip src){
3      sip src -> deny
4 }
5 main[permit]{
6      SomeFilter(SomeIp);
7 }
```

Når kompileren evaluerer filterkaldet, kombineres de formelle parametre, "ip src", samt de aktuelle parametre, "SomeIp", ind i en ny lokal identifikationstabel. Derfor undersøges hvad værdien af "SomeIp" er ved opslag i den globale identfikationstabel.

6.7 Sammenkobling af kompilerfaserne

For at integrere de enkelte faser til en komplet kompiler til MFL, skal der designes en overordnet funktion, herefter kaldet main, som kan sørge for at køre de overordnede funktioner for de enkelt faser. Desuden skal der sørges for at overføre datastrukturerne mellem de enkelte faser. For at hjælpe med at ensarte fejlhåndtering styres dette også centraliseret fra hovedfunktionen. De enkelte delopgaver bliver følgende beskrevet.

6.7.1 Udførsel af overordnede funktioner

Udførsel af de enkelte faser sker på følgende måde:

1. Indhent filnavn for den overordnede MFL fil fra kommandolinien.
2. Kald af lexer med et filnavn som parameter. Lexeren returnerer listen af lexems lavet på baggrund af MFL filen.

3. Parseren kaldes med listen af lexems, og returnerer det AST i form af en instans af mfl_t.
4. Den semantiske analyse kaldes med det AST, og returnerer en dekoreret version af ASTet.
5. Nu udføres kodegenerering sammen med det dekorerede AST. Kodegenerering returnerer med en MTIDD repræsentation af filtret fra MFL filen.
6. Til sidst konverteres MTIDD repræsentationen til XML, og skrives til en brugervalgt fil eller stdout[2].

Alle dataene mellem de enkelte faser gemmes i lokale variabler. Bortset fra lexerlisten bruges data kun af fasen efter den fase, hvor den er oprettet. Lexerlisten bruges udover i parseren også til at sammensætte fejlbeskeder i forbindelse med den semantiske analyse.

6.7.2 Fejlhåndtering

Der kan komme følgende kategorier af brugerforskyldte fejl under kompilering af en MFL fil: Lexerfejl, syntaktiske fejl, semantiske fejl, og kodegenereringsfejl. Fejl, der falder udenfor disse kategorier, er algoritmiske fejl i kompileren som ikke er opdaget endnu. De enkelte kategorier af fejl dækker over følgende:

Lexerfejl. Lexeren kommer med fejl i det tilfælde, at en kommentar over flere linier aldrig afsluttes. Sammen med denne fejl fortælles hvor i koden kommentaren startede. En anden fejl der kan opstå i lexeren, er hvis den fil, der skal læses fra ikke findes.

Syntaktiske fejl. Syntaktiske fejl sker som følge af, at kildekoden, som forsøges parset, indeholder tekst der ikke kan parses op mod MFL syntaks. Sammen med fejlen fortælles ved hvilket lexem fejlen opstod og derfor også placeringen i kildekoden.

Semantiske fejl. Kategorien af semantiske fejl er fejl, som følge af, at kildekoden ikke overholder de semantiske regler. De semantiske fejl følges altid af et (eller to) tal for, hvor i listen af lexems at fejlen opstod. Dette tal kan i main funktionen omsættes til det lexem som det repræsenterer, så det er muligt at udskrive hvor fejlen opstod sammen med fejlen. Hvis der er to tal, er det andet tal nummeret på det lexem, hvor den syntaktiske produktion sluttede.

Kodegenereringsfejl. De eneste fejl, der som følge af kildekoden, kan opstå under kodegenerering er ting der er platformsspecifikke, f.eks. genkendelse af netværksinterface. Under kodegenerering er det ikke muligt at determinere hvor henne i den originale kildekode fejlen opstod, så der gives kun en fejlbeskrivelse.

[2] stdout er den enhed brugeren bruger til at vise tekst, som standard vises det på skærmen.

KAPITEL 7

Implementation

Nærværende kapitel omhandler implementering af MFL kompileren, herunder om begrænsninger i det implementerede og hvorfor disse er nødvendige.

I det følgende laves en kort gennemgang af det implementerede. Kapitlet er opdelt i tre afsnit. Det første afsnit fortæller om parallel udvikling i projektet og om hvordan OCaml har influeret denne proces. I det andet afsnit gives et eksempel på et lille filter i MFL, derpå beskrives kort hvordan dette bearbejdes igennem kompilerens enkelte faser. Til slut, i det tredie afsnit, fortælles om de begrænsninger, der er på kompileren i forhold til det designede.

Forinden har der været opstillet nogle få enkelte regler for formatering af koden, altså en kodestandard. For det første er al kode, og kommentarer hertil, på engelsk, dernæst skal alle funktionsnavne skrives med småt, eventuelle opdelinger laves med understreg (_). Derudover er kommentarene skrevet i OCamldoc syntaks. OCamldoc bruges til at generere dokumentation af koden i f.eks. HTML eller LaTeX format. (◎ *Dokumentation af koden genereret af OCamldoc i HTML format* `/doc/ocamldoc/index.html`)

7.1 Parallel udvikling

Anvendelsen af OCaml, til udvikling af MFL kompileren, har gjort udviklingsarbejdet nemmere, forstået således, at typesystemet i OCaml, har gjort det enkelt, at definere grænseflader mellem de enkelte delmoduler i kompileren.

Dette arbejde forenkles, da OCaml tillader datatyper, der kan indeholde forskellige datatyper, afhængigt af hvilken konstruktor der anvendes til oprettelse af instans, dette kaldes på engelsk "tagged union". Sidenhen er det i OCaml muligt at lave mønstergenkendelse på den enkelte types konstruktorer, for at afgøre med hvilken konstruktor en type er lavet. Et eksempel på "tagged union" er vist i kildekode 7.1. Eksemplet viser typen `deco_command_t`, der har fem forskellige konstruktorer (eller tags): `CallCommand`, `IfCommand`, `WithCommand`, `RuleCommand` og `SequentialCommand`. Hver konstruktor laver en instans af `deco_command_t` med forskelligt indhold.

Kildekode 7.1 Anvendelse af såkaldt "tagged union" i OCaml.

```
1 type deco_command_t =
2     CallCommand of string * deco_expression_t list
3   | IfCommand of deco_rule_t * deco_command_t   * deco_command_t
4   | WithCommand of deco_rule_t * deco_command_t
5   | RuleCommand of deco_rule_t * deco_action_t
6   | SequentialCommand of deco_command_t * deco_command_t
```

Allerede tidligt i designfasen har det på den måde, været muligt at definere typer i det abstrakte og dekorerede AST. Dette, vel og mærke, uden forinden at have gjort mange overvejelser, om design af de enkelte kompilerfaser.

Denne tidlige definition af grænseflader, har gjort det muligt, at lave design og implementering, af hver enkelt kompilerfase, parallelt. Denne løsning kunne have været lavet i mange andre

programmeringssprog, fordelen ved brugen af OCaml er sprogets statiske typesystem, herunder brugen af en såkaldt "garbage collector".

At OCamls typesystem er statisk betyder, at en instans ikke kan ændre type. Dette sikres blandt andet af OCamls garbage collector, der sørger for at allokere og deallokere hukommelse som nødvendigt. Dette betyder at programmøren ikke er i direkte berøring med computerens hukommelse, herved kan mange fejl undgås. Dette betyder, sammen med typecheck af funktioner og udtryk, at der i færdigkompileret OCaml kode alene kan være logiske- eller algoritmiske fejl.

7.2 Gennemgang af faser

I dette afsnit gennemgås et lille eksempel MFL filter, igennem alle faser i kompileren.

Til eksemplet anvendes et mindre filter for et netværk, hvor udvalgte computere har tilladelse til at bruge alle porte (*), mens de resterende kun kan bruge port 80. Filtret kan ses i kildekode 7.2.

Kildekode 7.2 Eksempel MFL filter for computernetværk.

```
1 port WWW := 80;
2 ip network := 130.225.194.0/24;
3
4 main[deny]{
5   if sip network then dport * -> permit
6   else dport WWW -> permit }
```

I eksemplet har computere på network tilladelse til at bruge alle porte, hvor alle andre computere kun har tilladelse til at forbinde sig til port 80. Al anden trafik bliver som standard filtreret bort på grund af at standardhandling er deny.

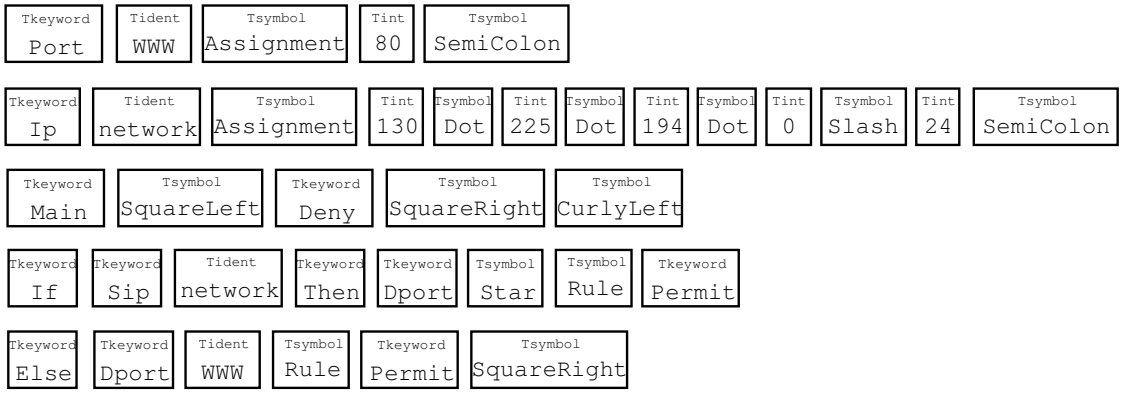
7.2.1 Syntaktisk analyse

I syntaktisk analyse indlæses eksempel filtret som lexems, i parseren, ved hjælp af lexer-funktionen. Lexems er delt op i fire overordnede typer: keyword, symbol, ident og int, disse er beskrevet i designafsnittet 6.2.1 på side 46, hvor hver types konstruktorer i OCaml tillige beskrives.

På figur 7.1 på næste side ses de lexems der genereres for MFL koden i eksempel 7.2. Hver kasse på figuren svarer til et lexem i eksemplet. For hvert lexem vises hvilken af token typens konstruktorer der er brugt til at oprette pågældende instans, dette er den øverste tekst. Den nederste tekst beskriver indholdet af hvert lexem, der er en tekststreng for ident, et heltal for int, eller en tom konstruktor for symbol og keyword. En tom konstruktor betyder blot at instansen ikke indeholder anden information end hvilken konstruktor den er oprettet med.

Hver funktion i parseren har en mængde af lexems de accepterer når de kaldes. Denne mængde af lexems blev i afsnit 5.5 på side 42 defineret som starters. Når en parser-funktion kaldes og det læste lexem er i dennes mængde starters, da bruges funktionen accept. Koden for accept er vist i eksempel 7.3 på næste side.

Funktionen virker grundlæggende ved at tage to lister, som parametre. Den ene er listen af lexems fra lexeren, denne kaldes lexerlist, den anden er listen af de lexems der skal accepteres fra lexerlisten, kaldet tokenlist. Funktionen er rekursiv og for hvert kald returneres lexerlist fratrukket det første element i tokenlist.



Figur 7.1: Liste af lexems svarende til eksempel 7.2.

I det følgende henviser tal i parentes, (x), til linienumre i kildekoden. Først laves et match på tokenlist(2) sådan at hvis tokenlist er tom, returneres lexerlist (3). Hvis tokenlist ikke er tom(4) afgøres om det første element i henholdsvis tokenlist og lexerlist er ens (5). Hvis de er ens kaldes accept igen med de resterende elementer i lexerlist og tokenlist (6). Hvis de derimod ikke er ens, er der syntaksfejl(9).

Kildekode 7.3 Funktionen accept der anvendes i parseren til at acceptere lexems.

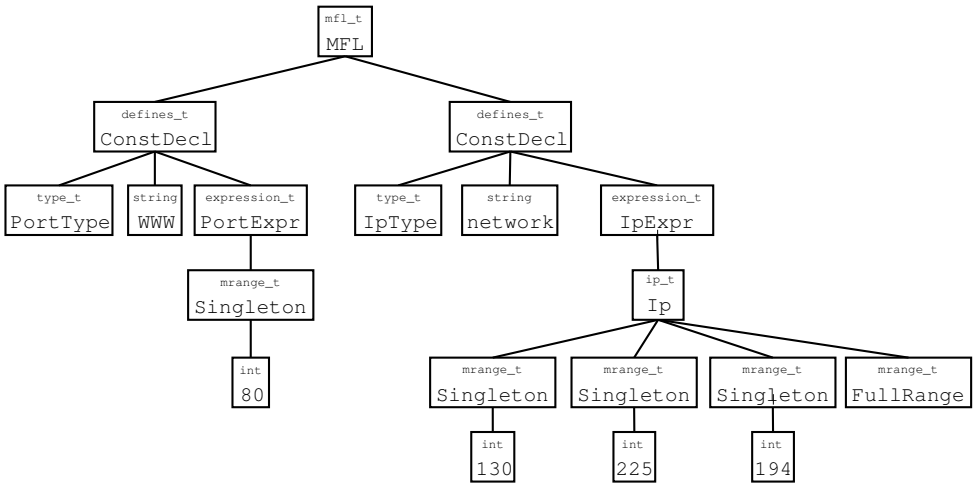
```
1  let rec accept lexerlist tokenlist =
2    match tokenlist with
3        [] -> lexerlist
4      | _ ->
5          if (List.hd tokenlist) = ((List.hd lexerlist).Lexer.token_l) then
6            let lexerlist = List.tl lexerlist in
7              accept lexerlist (List.tl tokenlist)
8          else
9            raise (ERR_SYNTAX
10             ((Lexer.string_of_token (List.hd tokenlist)),(List.hd lexerlist)))
11 ;;
```

Efterhånden som parseren accepterer de indkommende lexems opbygges et AST, samtidig sikres at kildekoden er syntaktisk korrekt. Beskrivelse og opbygning af AST findes i afsnit 6.2.2 på side 47. I figur 7.2 på næste side ses ASTet for konstanterklæringer, der opbygges ud fra lexems i figur 7.1 (to første linier). Hver kasse i figuren repræsenterer en instans af AST type. For hver instans er beskrevet hvilken type den er, dette er den øverste tekst. Den nederste tekst beskriver hvilken konstruktor instansen er oprettet af, undtagen for string og int, for disse to typer er det istedet henholdsvis indholdet af strengen eller tallet, den pågældende type repræsenterer.

Sammenlignes listen af lexems i figur 7.1 med ASTet i figur 7.2 på den følgende side, ses at alle symboler er væk. Symboler i MFL sikrer, at parseren ved kun at kigge på et lexem, kan afgøre hvilke lexems der skal følge. Samtidig gør disse symboler også koden mere læsbar.

7.2.2 Semantisk analyse

Det vigtigste mål med semantiske analyse er, at sikre at kildekoden lever op til de semantiske krav der gælder for MFL. Populært sagt betyder dette at der undersøges om de enkelte sætninger i kildekoden giver mening. For MFL betyder dette primært at sikre at såkaldte “expressions”



Figur 7.2: AST for konstanterklæringer i eksempel 7.2.

i sproget faktisk giver mening, og giver mening i den sammenhæng de anvendes. I syntaktisk analyse kan en port for eksempel godt have værdien 100000, men dette giver ikke semantisk mening. Det samme gælder hvis en port anvendes som aktuel parameter i et filterkald til et filter, der har en IP som formel parameter.

Sidstnævnte undgås ved brug af den semantiske funktion find_deco_command. Der ses et uddrag af koden for netop denne funktion i kildekode 7.4 på næste side

Funktionen kaldes med en ast_command_t som parameter, i eksemplet kaldes denne command (1). Derpå matches command med konstruktorerne for ast_deco_command (2), hvis command ikke matches som en CallCommand, returneres blot en deco_command_t svarende til command. Hvis command er en CallCommand (4), anvendes funktionen check_formal_param til at sikre at formelle og aktuelle parametre har samme type (5), alternativ er der semantisk fejl (11). Afsluttende sikres at filtret ikke bevirker cirkulære filterkald, til dette anvendes funktionen check_filter_cirk (6). Hvis der er cirkulære filterkald gives en semantisk fejl (8), ellers returneres en command som en deco_command_t.

Funktioner svarende til find_deco_command findes for alle elementer i AST, disse er beskrevet i appendiks G på side 115. Ved anvendelsen af disse funktioner er ASTet fra syntaktisk analyse omdannet til et dekoreret AST. En beskrivelse af dekoreret AST forefindes i afsnit 6.2.3 på side 48.

I figur 7.3 på side 66 ses en del af det dekorerede AST efter semantisk analyse af MFL eksemplet i kildekode 7.2 på side 62. I semantisk analyse bliver stjerne symbolet for dport i kildekoden erstattet af en range, svarende til et interval, fra 0 til 65535. Dette svarer til alle porte. Derudover er det nu sikkert at alle "sætninger" i kildekoden giver mening.

I semantisk analyse opbygges en identifikationstabel for MFL dokumentet. Identifikationstabellen indeholder alle globalt erklærede konstanter og deres værdi. Disse konstanter kan findes ved opslag på konstanternes navn.

Kildekode 7.4 Uddrag af semantikfunktionen find_deco_command.

```
1  let rec find_deco_command command =
2    match command with
3        Ast.CallCommand(name,exprlist,ftok,ltok) ->
4          let (ok, exprDecoList) = check_formal_param
5                                   name
6                                   exprlist
7                                   ftok
8                                   ltok in
9            if ok then
10             if (check_filter_cirk name ftok ltok) then
11               Deco.CallCommand(name,exprDecoList)
12             else
13                 raise (SEM_EXPN
14                         ("Circular filter call:",ftok, ltok))
15           else
16               raise (SEM_EXPN
17                       ("Formal parameters don't fit with actual
18                         parameters:", ftok, ltok))
```

7.2.3 Kodegenerering

I kodegenerering omdannes det dekorerede AST først til en intern MTIDD repræsentation, derefter til en XML repræsentation af MTIDDer.

I kodegenerering anvendes OCaml biblioteket libIDD, der indeholder typer til repræsentation af IDDer og MTIDDer, samt funktioner svarende til de boolske operationer der tillades på disse. Derudover er der en funktion til at optimere IDDer og MTIDDer, og en til at kombinere IDDer til MTIDDer.

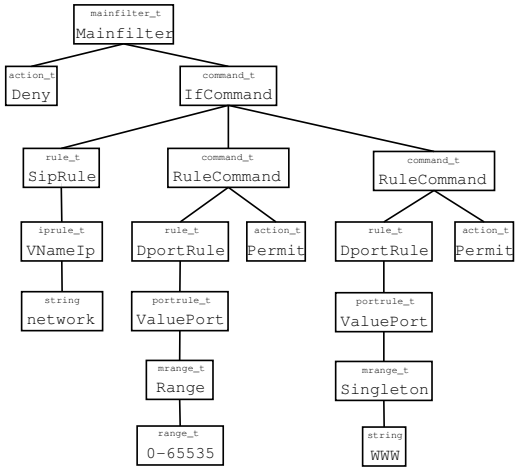
Hvordan IDDer og MTIDDer bygges op i kodegenerering, med anvendelsen af boolske operationer er beskrevet i design af kodegenerering, afsnit 6.6 på side 52. Istedet fokuseres der her på hvorledes hver enkelt regel i MFL evalueres til en IDD.

En IDD er, som beskrevet i appendiks B på side 80, en graf, hvor hvert punkt svarer til et "felt" i en netværksheader og en kant indeholder et heltalsinterval. For hvert headerfelt sammenlignes værdien fra feltet med intervallerne på kanterne fra punktet svarende til headerfeltet. Hvert interval beskriver altså en handling, hvis værdien af headerfeltet er i intervallet. For IDDer er handlingen, hvilken node eller terminal der navigeres til.

Kodegenerering genererer disse intervaller med funktionen determine_interval. I kildekode 7.5 på side 67 ses et uddrag af OCaml koden for denne funktion. Funktionen tager en deco_mrange_t, to heltal beskrivende nedre og øvre grænse af et interval, samt to IDD noder eller terminaler som parametre (1). Den ene node eller terminal er endepunktet for de kanter hvis intervaller fører til sand, modsat gælder for den anden node eller terminal.

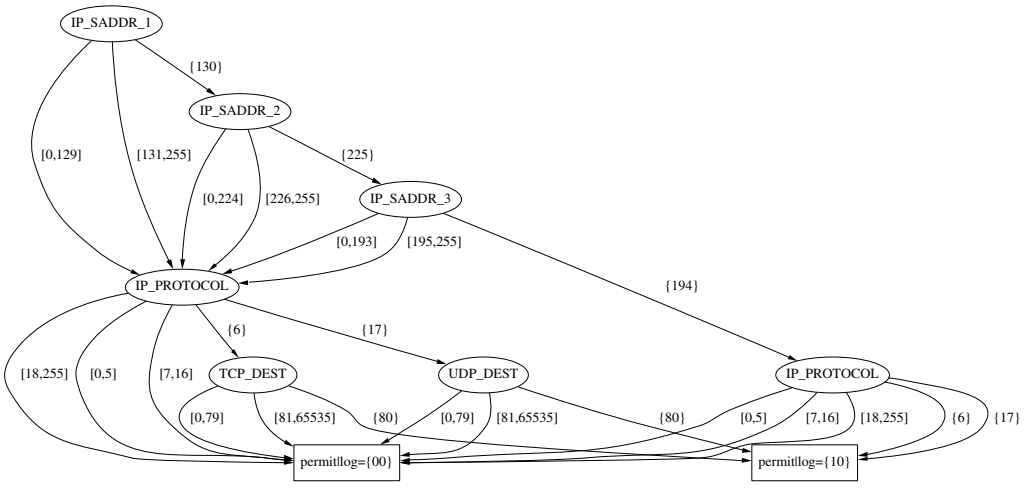
Funktionen undersøger om mrangen indeholder et interval eller en singleton, dvs en enkelt værdi. Dette gøres ved at match mod konstruktorer for deco_mrange_t (2). Hvis mrange matches som en range (3) da skabes værdierne lowerRange og higherRange (5). Disse anvendes i et match (5). Hvis både lowerRange og higherRange er 0 betyder dette, at mrange dækker hele intervallet beskrevet i parametrene min og max (6). Da returneres et IDD interval dækkende hele dette interval, gående mod den sande terminal.

Afsluttende indeholder libIDD også funktionalitet til at generere den færdige XML repræsentation for en MTIDD. I figur 7.4 på næste side ses en grafisk repræsentation af det beskrevne MFL



Figur 7.3: Dekoreret AST for main filter i eksempel 7.2.

eksempel fra kildekode 7.2 på side 62. Dette eksempel illustrerer resultatet af kodegenerering.



Figur 7.4: MTIDD genereret fra MFL eksempel i kildekode 7.2.

7.3 Begrænsninger

Dette afsnit introducerers de begrænsninger, der er afstedkommet af implementeringen af kompileren. Afsnittet er delt op i tre underafsnit. Første underafsnit omhandler navne på netværkinterfaces i MFL. Anden underafsnit fortæller om begrænsninger, på de handlinger der i MFL. Det tredie og sidste afsnit fortæller om håndteringen af dag og tid under implementeringen, og hvorfor disse ikke er praktisk anvendelige endnu.

Kildekode 7.5 Uddrag af funktionen determine_interval.

```
1  let determine_interval mrange min max tterm fterm =
2    match mrange with
3      | Deco.Range x ->
4          begin
5            let lowerRange = x.Deco.low - min
6            and higherRange = max - x.Deco.high in
7              match lowerRange,higherRange with
8                | 0,0 ->
9                    [Interval.Interval(x.Deco.low, x.Deco.high), tterm;]
10
11               | 1,1 ->
12                   [Interval.Singleton(min), fterm;
13                    Interval.Interval(x.Deco.low, x.Deco.high), tterm;
14                    Interval.Singleton(max), fterm]
15
16               | 0,1 ->
17                   [Interval.Interval(x.Deco.low, x.Deco.high), tterm;
18                    Interval.Singleton(max), fterm]
19
```

7.3.1 Interfaceenhed

I MFL er det muligt at specificere på, hvilke netværksinterfaces henholdsvis indkommende og udgående pakker accepteres. I MFL, samt syntaktisk og semantisk analyse heraf, tillades enhver identifier[1] som navn på et netværksinterface. I kodegenerering skal netværksinterfacet så konverteres til en heltalsværdi. Dette giver dog problemer hvis interfacet for eksempel hedder "mitnetkort". Dette er ikke umiddelbart til at konvertere til en heltalsværdi, i hvert fald ikke en værdi der giver mening. Dette er i kodegenerering løst ved kun at genkende identifiers med prefixet "eth" og en heltalsværdi. Da accepteres for denne heltalsværdi og de resterende værdier i intervallet 0 til 255 accepteres ikke. Alternativ accepteres pakker i hele intervallet 0 til 255.

7.3.2 Tid og dag

Det er i MFL muligt at specifere regler for dage og tidspunkter i et filter. Ideen var, at det skulle være muligt at specificere på hvilke dage, samt tidspunkter på dagen traffik igennem pakkefiltret var muligt.

MFL kompileren genkender og accepterer regler for både tid og dag. Kodegenerering kan lave IDDer for dage, hvor hver dag har en værdi i intervallet mellem 0 og 6, hvor søndag er 0 og lørdag er 6. Tilsvarende laver kodegenerering IDDer for tidsintervaller, indenfor intervallet 0 til 86400, svarende til antallet af sekunder på et døgn.

Disse er desværre ikke brugbare, da kernemodulet der ved pakkefiltrering anvender MTIDD, på dette tidspunkt ikke understøtter filterering på tid og dag.

[1] En identifier er en arbitrær rækkefølge af bogstaver og tal, startende med et bogstav

KAPITEL 8

Test

I dette afsnit vælges hvilke tests, der skal udføres på den implementerede kode. Der tages stilling til funktionel, strukturel og integrationstests.

8.1 Metode

Idet kompileren er implementeret i ML, er der i dette projekt ikke er noget stort behov for strukturtests. Endvidere er ML et strongly typed sprog, og derfor kan der efter kompilering kun være tale om logsike fejl i koden. Disse logiske fejl vil komme til udtryk i forkerte output, fra de funktioner de er lavede i. Da runtime fejl ikke findes i MFL kompileren, bliver udbyttet af strukturelle tests mindsket, til kun at finde forgreninger i koden der aldrig bliver fulgt. Funktionaliteten af kompileren skal derimod testes grundigt. Funktionaliteten testes med blackbox tests, i en bottom-up rækkefølge, så allerede testede funktioner kan anvendes i andre tests. Udtrykskraften af testene falder en smule, idet der ikke laves stubbe[1] som de funktioner, der testes kalder, hvorved funktionskaldene kan skrives ud. Til gengæld bliver de interne grænseflader, mellem de forskellige funktioner testet, og arbejdsopgaven ved testene reduceres.

8.2 Testteknikken

For hver funktion vælges en række input. Hvorefter funktionen afvikles med disse input, og de forventede og aktuelle output sammenlignes. Som nævnt udføres testene bottom-up, hvilket vil sige at de forskellige funktioner sættes sammen fra bunden af, altså tages udgangspunkt i funktioner der ikke kalder andre funktioner. Når en funktion således er blevet testet, anvendes den derefter i de følgende tests.

Først når alle funktioner er sat sammen, og hovedfunktion i de forskellige moduler er testet, udføres en såkaldt systemtest. Systemtesten er en test af hele kompileren, hvor et lovligt filter skrevet i MFL, gives som input. Filtret der bruges som input vil indeholde alle elementer i MFL, alle slags regler, kommandoer og typer. Bliver det aktuelle output det samme som det forventede, er testen slut, ellers må fejlen selvfølgelig findes og rettes, hvorefter testen gentages forfra.

8.3 Eksempel

Dokumentationen til alle testene fylder en del, derfor vises her kun et eksempel. (Alle testene er dokumenteret i et dokument for sig selv på cden, (◎ *Dokumentation af funktionelle tests* `/test`))

[1] Stubbe: Små funktioner, som bliver kaldt istedet for, de funktioner, som den funktion der testes, ellers skulle kalde [Nør]

init_lex

Input: string: Kildekoden fra en fil i en streng, String: Et filnavn og List: den liste filerne er skrevet op i

Fra fil: lexer.ml

Afhængig af:

Input	*Forventet output*	*Faktisk output*
En fil hvor alt er udkommenteret med `/*` og `*/`	"Intet" current: 0 size: 1 line: 0 col: 0 filename: file1.mfl	"Intet" current: 0 size: 1 line: 0 col: 0 filename: file1.mfl
En fil hvor alt er udkommenteret med `//`	"Intet" current: 0 size: 21 line: 0 col: 0 filename: file2.mfl	"Intet" current: 0 size: 21 line: 0 col: 0 filename: file2.mfl
En fil hvor tomme linier er udkommenteret og flere linier er udkommenteret med `/**/`	"Den kode der ikke var udkommenteret" current: 0 size: 305 line: 0 col: 0 filename: file3.mfl	"Al kode mellem første `/*` og sidste `*/` var væk" current: 0 size: 263 line: 0 col: 0 filename: file3.mfl

Alle testene er skrevet ind i et skema som ovenstående, hvor funktionsnavn, parameter, source fil afhængigheder er nævnt først. Derefter følger testresultaterne i tre kolonner, en til input, en til forventet output og en til faktisk output. Den testede funktion er fejlfri[2], hvis de sidste koloner er ens.

Som det ses, er der i den viste funktion en fejl, denne fejl består i at lexeren var "grådig" i den måde hvorpå den fjernede kommentarer over flere linier i kildekoden. Dvs. skrev som vist ved kildekode 8.1

Kildekode 8.1 Input til funktionen.

```
1 filter(ip ipvar){
2 /*
3   sip ipvar -> permit;
4 */
5   dip ipvar -> permit;
6 /*
7   port 80 -> deny;
8 */
9 }
```

fjernede lexeren alt mellem det første `/*` til det sidste `*/`, hvorved kildekoden så ud som i 8.2.

hvor det naturligvis skulle have set ud, som vist ved kildekode 8.3.

[2]Fejlfri, så langt som den er testet

Kildekode 8.2 Faktiske output.

```
1 filter(ip ipvar){
2 }
```

Kildekode 8.3 Forventet output.

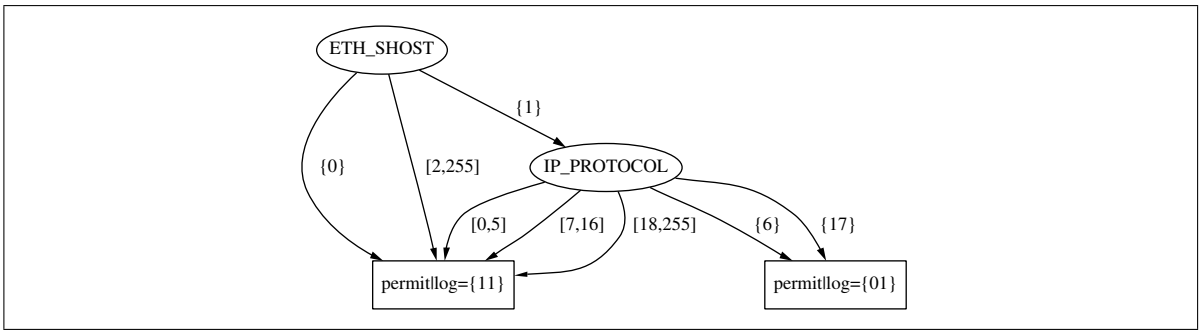
```
1 filter(ip ipvar){
2   dip ipvar -> permit;
3 }
```

Fejlen er efter følgende blevet rettet og testen kørt igen med positivt resultat.

8.4 Systemtest

Systemtesten blev foretaget med MFL kildekode 8.4. I koden er alle reserverede ord brugt mindst en gang.

Den IDD der kommer ud af en kompilering er overraskende simpel, se figur 8.1



Figur 8.1: IDD der kom ud at systemtest.

I første omgang blev der fundet en fejl, idet at parseren ikke godtog det reserverede ord: `RST`, det viste sig at være hurtig at rette. Derfor anses systemtesten for en succes.

8.5 Resultat

Ikke alle funktioner i kompileren er blevet testet, da der ikke har været tid til dette. De mest komplicerede funktioner har dog haft højst prioritet, hvorfor det hovedsageligt er små og simple funktioner der endnu ikke er blevet testede. Disse mindre funktioner er kun blevet testet indirekte, idet de er integreret i en eller flere andre funktioners tests. De testede funktioner ses i tabel 8.1 på side 72. Der er ialt udført 69 dokumenterede tests, ud af 165 implementerede funktioner.

Kildekode 8.4 Kode til brug i systemtest

```
1  day aday := mon; day bday := monday;
2  day cday := tue; day dday := tuesday;
3  day eday := wed; day fday := wednesday;
4  day gday := thu; day hday := thursday;
5  day iday := fri; day jday := friday;
6  day kday := sat; day lday := saturday;
7  day mday := sun; day nday := sunday;
8  time atime := 12:00-13:00;
9  proto aproto := UDP;
10 proto bproto := TCP [SYN, ACK, URG, FIN, PSH, RST];
11 proto cproto := ICMP [1,2];
12 ip aip := 1.2-3.4.*;
13 ip bip := 1.2.2.4/24;
14 port aport := 80;
15 port bport := 0-1024;
16 iface aiface := 'eth0';
17
18 filter afilter(ip ipvar, port portvar, day dayvar,
19                time timevar, proto protovar, iface ifacevar){
20   sip ipvar || dip ipvar -> permit log;
21   dport portvar &! sport portvar -> permit state;
22   if day dayvar then {
23     time timevar -> permit state log;
24   }
25   else
26     time atime -> deny;
27   with input ifacevar do
28     proto TCP,UDP -> deny log;
29 }
30
31 main[deny]{
32   afilter(any, aport, any, atime, aproto, 'eth1');
33   output aiface ^^ input 'eth1'-> permit;
34   sport <20 && sport >80 && (dport <=80 dport >= 30) -> permit;
35 }
```

Lexer	Parser
is_int	accept
is_alpha	peek
is_alpha_int	peek_token
is_symbol	lexernumber
is_not_newline	mrange
is_valid_keyword	port_rule
keyword_of_string og string_of_keyword	mrange_of_singleton **(Fejl rettet)**
symbol_of_string og string_of_symbol	ipmask **(Fejl rettet)**
is_valid_keyword	ip
get_src_from_file	ip_rule
init_lex **(Fejl rettet)**	interface_rule
forward	proto
forward_line	proto_rule
extractor	time
alpha_starter	time_rule
int_starter	day_rule
symbol_starter	single_rule
single_symbol	rule
hash_starter	expression
lexer	command
Semantik	**Codegen**
split_defines_in_consts_and_filters	find_expression
get_l1 lexem	find_filter
check_const_cirk	do_ip
check_global_and_local_const	do_iprule
check_time	do_port
find_deco_PortRule **(Fejl rettet)**	do_port_operator **(Fejl rettet)**
find_deco_TimeRule	do_input_output_rule
find_deco_expression **(Fejl rettet)**	do_icmp
find_deco_filter_expression	do_tcpprotocol
check_formal_param **(Fejl rettet)**	do_protorule_list
check_filter_cirk	do_time_rule
find_deco_command	do_day_rule
build_filter	match_action
semantik_analyzer	match_command
	do_mainfilter

Tabel 8.1: Testede funktioner

KAPITEL 9

Konklusion

Formålet med dette kapitel er at konkludere på projektet, hvilket vil sige sammenligne problemformuleringen og de opstillede krav med det realiserede programmel, samt diskutere hvilke fremtidige udvidelsesmuligheder systemet kunne underlgå.

Netværkssikkerhed er et område, der i den senere tid har fået større opmærksomhed. Dette skyldes hovedsageligt, at langt flere computere er blevet koblet på Internettet, hvorigennem mange flere mennesker er blevet afhængige af nettet til udførsel af opgaver i både arbejdsmæssige og privat øjemed. For virksomheder er netværkssikkerhed ligeledes et område, der må tages meget alvorligt, for at sikre interne ressourcers konfidentialitet og integritet, samt opretholde tilgængeligheden af netværksressourcer som benyttes i forbindelse med virksomhedens daglige rutiner. Firewalls er en af flere forskellige værktøjer til at opnå et givent sikkerhedsniveau på et netværk og derigennem opretteholde en sikkerhedspolitik.

Igennem denne rapport er en række problemer omkring eksisterende firewall specifikationssprog blevet belyst. Det drejer sig mere specifikt om, at sprogene skalerer dårligt i den forstand, at regelafhængigheder gør udvidelse af store filtre besværligt. Endvidere er en række problemer omkring filtrenes effektivitet og testbarhed blevet identificeret. Kompleksiteten af den eksisterende pakkefiltrering er lineær i antallet af regler, hvilket gør store filtre ineffektive. Implementerede filtre er problematiske at teste og verificere, idet eksisterende værktøjer til dette udelukkende er baseret på runtime tests af systemet og således ikke kan tilbyde offline tests.

For at imødekomme de eksisterende specifikationssprogs mangler, er der således blevet konstrueret et nyt specifikationssprog (MFL), med elementer fra de eksisterende specifikationssprog, samt en række nye tiltag. Regelbegrebet er blevet bevaret, mens der er blevet indført abstraktionsmekanimser i form af parametriserede filtre. Endvidere er overlap mellem regler håndteret ved indførsel af en standardhandling, der medfører større gennemskuelighed i forbindelse med tilføjelse af regler til store filtre. Den valgte target model MTIDD, muliggjorde endvidere, at specificerede filtre let kunne optimeres, samt en reduktion af filtreringskompleksiteten til konstant tid.

Kompileren til oversættelse fra det designede sprog MFL til MTIDD repræsentationen blev implementeret i OCaml. Kompilerens parser er implementeret via recursive desent parsings algoritmen og grænsefladene mellem faserne var klart defineret som hhv. AST og dekoreret AST. Dette samt OCamls typesystem gjorde det blandt andet muligt, præcist at definere grænseflader i kompileren fra begyndelsen, hvilket lettede udviklingsprocessen. Endvidere muliggjorde definition af rekursive typer samt mønstergenkendelse (pattern matching), at træstrukturer let kunne defineres og traverseres.

Kriterier for kompileren, som blev prioriteret meget vigtige, var: brugervenlighed og genbrugbarhed. Brugervenlighed er blevet opnået gennem sigende og præcise fejlmeddelser og genbrugbarhed er blevet opnået ved at dokumentere koden via kommentarer(OCamldoc) og en ensartet opbygning af moduler og navngivning af funktioner

Tillige var de meget vigtige kriterier for sproget: skalérbarhed, understøttelse af abstraktion og læsbarhed. Alle tre kriterier er opnået via abstraktion og kontrolstrukture.

Idet der er blevet konstrueret et specifikationssprog til firewalls, med flere abstraktionsniveauer end de hidtidige sprog, samt en kompiler der kan omsætte filtre specificeret i det designede

programmeringssprog til en repræsentationsform, som muliggør offline test og optimering, må formålet med projektet siges at være opnået.

9.1 Udvidelsesmuligheder

Idet projektet har berørt mange emner indenfor både firewall specifikationssprog og kompilerdesign, er der naturligvis en lang række områder, hvor der kunne indføres udvidelser og forbedringer.

Det udviklede programmel må betragtes som en protoype, idet mange essentielle områder af en firewall specifikation er valgt udeladt. Her tænkes specielt på tilstandsbegrebet, som findes i nuværende stateful firewalls. Endvidere er der en række protokoller, som det ikke er muligt at benytte i MFL i dets nuværende form. Derudover findes headerfelter i de implementerede protokoller, som heller ikke kan specificeres (eks. TTL for IP protokollen). I fremtidige versioner af MFL kan der således introduceres mulighed for, at nye protokoller kan defineres i sproget, uden nødvendigvis at kræve ændring af kompileren.

Effektiviteten af kompileringsprocessen er endvidere et område, som er valgt nedprioriteret i projektet. Dette må nødvendigvis tages i betragtning såfremt kompileren skal have en fremtid. Her tænkes specielt på parallelisering af kompileringsprocessen, således større filtre hurtigere kan kompilers ved brug af beregnings clustre.

I øjeblikket eksisterer der kun et Linux kernemodul, som kan benytte et filter beskrevet ved en MTIDD. En oplagt udvidelsesmulighed, ville være at implementere kernemoduler til andre operativsystemer og arkitekturer.

Endelig kunne der udvikles testværktøjer, således de generede MTIDDer kan verificeres udfra en række opstillede kriterier og derigennem testes offline, inden de tages i brug.

FiXme Note: *skal der være en nomenklatur liste?*[1]

[1] **FiXme Note**: *skal der være en nomenklatur liste?*

Litteratur

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Programming Language Processors in Java*. Addision-Wesley, 1986. 44

[BSHKM98] Stephen Biering-Sørensen, Find Overgaard Hansen, Susanne Klim, and Preben Thalund Madsen. *Håndbog i Struktureret Program Udvikling*. Teknisk Forlag, 1 edition, 1998. 27, 30

[CF02] Mikkel Christiansen and Emmanuel Fleury. Using IDDs for packet filtering. rs RS-02-43, brics, iesd, oct 2002. 25 pp. 12, 24, 80

[Cis02] Cisco Systems. *Securing Your Network with the Cisco Centri Firewall*, 1 edition, 2002. 7, 10, 11

[GB98] Barbara Guttman and Robert Bagwill. *Implementing Internet Firewall Security Policy*. Information Technology Laboratory, Computer Security Division, 1 edition, 1998. 20

[Hel00] Gilbert Held. *TCP/IP Professional Reference Guide*. Auerbach, 2000. 77

[hlf03] High level firewall language, Maj 2003. http://www.hlfl.org. 15

[Knu97] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 3 edition, 1997. 2

[KR01] Jim Kurose and Keith Ross. *Computer Networking*. Addison-Wesley Publishing, 1 edition, 2001. 77, 78, 79

[nfp03] The netfilter project, Maj 2003. http://www.netfilter.org. 14

[nma03] Nmap - stealth port scanner, Maj 2003. http://www.insecure.org/nmap/. 18

[Nør] Kurt Nørmark. Test og dokumentation. Internet, del af Forelæsningsnoter i Objekt-orienteret Programmering. http://www.cs.auc.dk/ñormark/prog1-01/html/noter/test-dokumentation-book.html. 68

[Pos80a] J. Postel. *RFC760-ICMP*. J. Postel, 1980. 78

[Pos80b] J. Postel. *RFC768-UDP*. J. Postel, 1980. 78

[Pos81] J. Postel. *RFC793-TCP*. J. Postel, 1981. 9, 77

[Pos92] J. Postel. *RFC1340-IP*. J. Postel, 1992. 78

[PZ01] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages - Design and Implementation*. Prentice Hall, 4. edition, 2001. 31, 33

[Ros03] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 2003. 84

[Sed01] Jeff Sedayao. *Cisco IOS Access Lists*. O'Reilly, 1 edition, 2001. 13

[Ste94] Richard W. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Publishing, 1 edition, 1994. 9

[WB00] David A. Watt and Deyck F. Brown. *Programming Language Processors in Java.* Prentice Hall, 2000. 28, 34, 43, 45, 50

[Wei02] Pierre Weis. Caml programming guidelines, December 2002. http://caml.inria.fr/FAQ/pgl-eng.html. 28

APPENDIKS **A**

TCP/IP protokolsuiten

Formålet med dette appendiks er kort og præcist, at beskrive nogle af de informationer, som TCP/IP protokoldatapakker indeholder. Mere specifikt er det de informationer, en firewall skal bruge til at afgøre om den ønskede trafik kan tillades.

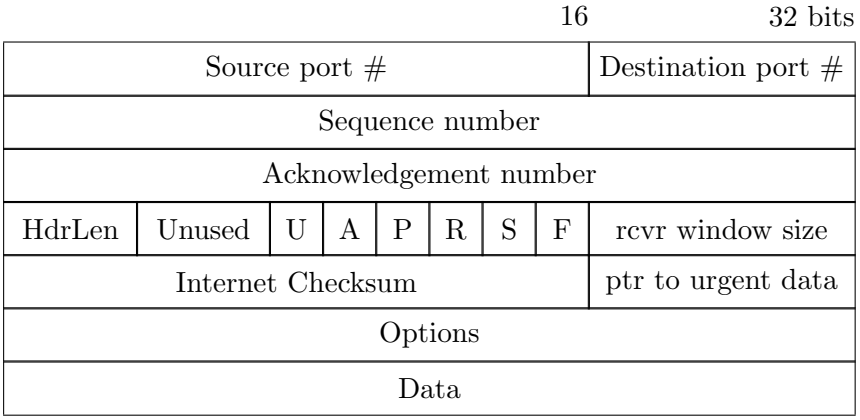
TCP/IP protokolsuiten indeholder blandt andet transportprotokollerne: Transmission Control Protocol (TCP) og Uniform Datagram Protocol (UDP) og netværksprotokollen: Internet Protocol (IP). De relevante strukturelle informationer er specificeret, for hver af disse protokoller, i det efterfølgende.

TCP/IP protokolsuiten indgår i Open Standards Interconnection (OSI) reference modellen, der er et framework til standardisering af kommunikationssystemer. OSI er specificeret af den Internationale Standardiserings Organisation (ISO). [Hel00, p. 15]

A.1 TCP protokollen

TCP protokollen er en pålidelig, forbindelsesorienteret transportprotokol, som muliggør overførsel af data mellem computerer i et netværk. Protokollen er placeret på OSI modellens fjerde lag, transportlaget.

TCP headerstrukturen er vist på figur A.1.



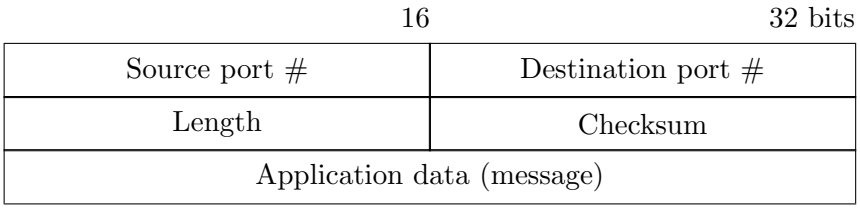
Figur A.1: TCP headerstruktur [KR01].

Yderligere detaljer er beskrevet i specifikationen for TCP-protokollen, RFC793 [Pos81].

A.2 UDP protokollen

UDP protokollen er en upålidelig, forbindelsesløs transportprotokol, som muliggør overførsel af data mellem computerer i et netværk. Protokollen er placeret på OSI modellens fjerde lag, transportlaget.

UDP headerstrukturen er vist på figur A.2.



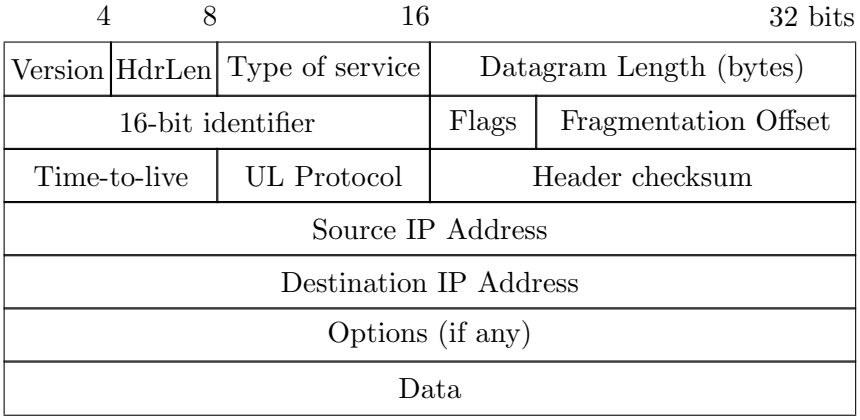
Figur A.2: UDP headerstruktur [KR01].

Yderligere detaljer er beskrevet i specifikationen for UDP-protokollen, RFC768 [Pos80b].

A.3 IP protokollen

IP protokollen er en netværksprotokol, hvis primære funktion er addressering og fragmentering af datapakker. Protokollen er placeret på OSI modellens tredie lag, netværkslaget.

IP headerstrukturen er vist på figur A.3



Figur A.3: IP headerstruktur [KR01].

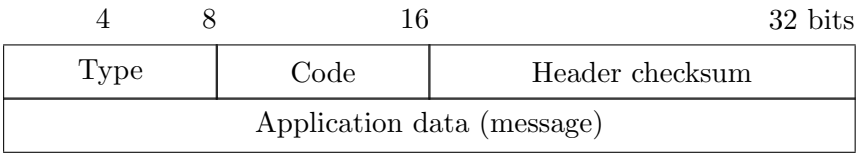
Datafeltet indeholder IP data eller en header fra et højere protokollag, eksempelvis en TCP header.

Yderligere detaljer er beskrevet i specifikationen for IP-protokollen, RFC1340 [Pos92].

A.4 ICMP protokollen

ICMP (Internet Control Message Protocol) er en IP protokol der benyttes til at kontrollere IP traffiken. Protokollen benyttes til at rapportere problemer omkring afsendelse og modtagelse af IP datagrammer.

Yderligere detaljer er beskrevet i specifikationen for ICMP protokollen, RFC760 [Pos80a].



Figur A.4: ICMP headerstruktur [KR01].

APPENDIKS **B**

Interval Decision Diagrams

Dette appendiks giver, på baggrund af [CF02], en teoretisk gennemgang af Interval Decision Diagrams(IDD) og Multi Terminal Interval Decision Diagrams(MTIDD).

B.1 IDD

En IDD er en orienteret graf uden kredse (DAC), som har to mulige terminalværdier: true og false. Hver knude i grafen indeholder et logisk udtryk, der checker på værdien af en variabel. Værdien af variablen afgør hvilken kant der fortsættes fra.

En kant k har en label I, som repræsenterer et heltalsinterval hvorom der gælder:

$$I \subseteq P_i \tag{B.1}$$

Hvor P_i er et domænesæt for en variabel x, $P_i \subseteq N$, hvor N er alle heltal.

Hvis variablen x repræsenterer de første fire cifre af en IP adresse, så vil domænesættet for x være $[0, 255]$. Der gælder for en knude p, at der for alle værdier i domænesættet, for den variabel der undersøges for, skal være en kant fra den pågældende knude, hvis label repræsenterer et interval, der dækker den fundne værdi.

Reglerne for hvordan et domænesæt deles op i intervaller, gives ved følgende to definitioner.

Definition 1 *sættet $I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ af p_i delte intervaller I_j er en intervaldækning af P_i hvis hver I_j er en delmængde af P_i. Dvs. $I_j \subseteq P_i$ og $I(P_i)$ er komplet, dvs.*

$$P_i = \bigcup_{I \in I(P_i)} I \tag{B.2}$$

Definition 2 *En intervaldækning er adskilt hvis.*

$$\forall_{j,k} 1 \leq j, k \leq p_i, j \neq k : I_j \cap I_k = \emptyset \tag{B.3}$$

Det vil sige, at intet element i P_i er inkluderet i mere end et delinterval. En adskilt intervaldækning kaldes en intervalpartition.

Intervalpartitioner bliver brugt i nedenstående definition af en IDD knude.

Definition 3 *Lad x være en heltalsvariabel defineret ved domænet $D_x \subseteq N$, hvor N er alle heltal. Lad t være et logisk udtryk på heltalsvariabler. Så er t en IDD knude hvis et af følgende udtryk er sand.*

- $t \in \{True, False\}$
- $t = (x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee \ldots (x \in I_k \wedge t_k)$

$(I_i) i \leq k$ er en intervalpartition af D_x og $(t_i) i \leq k$ er et sæt af IDD knuder. Kanterne fra t defineret ved $t = x \rightarrow (I_0, t_0)(I_1, t_1) \ldots (I_n, t_k)$, hvor I_n er intervallet, som kantens label dækker og t_k er den knude, som kanten går hen til.

For at kunne definere en IDD mangler der to begreber: rod og konsistens. Samt en funktion $var(t)$.

- En rod er en IDD knude uden forfædre
- Et sæt af IDD knuder $(t_i)i \leq n$ er konsistent hvis der kun er en rod

Navnet på heltalsvariablen kan findes ved hjælp af funktionen $var(t)$.

- $var(t) = \{x$, hvis $t = x \rightarrow (I_0, t_0)(I_1, t_1)...(I_k, t_k)\}$
- $var(t) = \{t$, hvis $t \in \{True, False\}\}$

Herefter er det muligt at definere en IDD.

Definition 4 *En IDD er et konsistent sæt af punkter og en ordning af heltalsvariablerne. $I = ((t_i)i \leq n, \succ)$ hvor $(t_i)i \leq n$ er et konsistent sæt af IDD knuder, og $\succ$ er en ordning af heltalsvariablerne. For alle $t \in (t_i)i \leq n$ med $t = x \rightarrow (I_0, t_0^{'})(I_1, t_1^{'})...(I_k, t_k^{'})$, har vi $x \succ var(t_i^{'})$ for hver $i \leq k$.*

En IDD repræsenterer sammensatte logiske udtryk på heltalsvariabler. Det er derfor muligt at bruge logiske operatorer, som f.eks. $\vee, \wedge$ og $\neg$, på IDDer. Det er også muligt at specificere en algoritme, der automatisk opdaterer IDDer, hvilket er en stor fordel når der tilføjes nye regler. Algoritmen har følgende punkter:

1. Hvis en ikke-terminal knude kun har en udgående kant, så skal den beskæres.
2. Hvis to knuder har den samme udgående kant og repræsenterer den samme variabel så skal de slås sammen.
3. Hvis to kanter, fra en knude, med efterfølgende intervaller referer til det samme barn, så skal de slås sammen.

Algoritmen fungerer ved at teste alle tre regler på alle knuder i IDDen, ud fra en dybde først søgning. Derefter sammenlignes den nye IDD med den gamle. Hvis de er ens så terminerer algoritmen, ellers kører den en gang mere. Algoritmen garanterer, at den altid finder det optimale antal knuder, samt den minimale dybde på IDDen.

B.1.1 Eksempel på IDD

Et firewall filter kan laves om til en IDD, ved at lave hver regel i filtret om til et sammensat logisk udtryk.

I kildekode B.1 ses et filtereksempel givet ved et Cisco lignende sprog, åbnes for udgående TCP pakker til alle IP adresser på port 80, for alle computere på netværket 192.168.*.*. Stjerne (*) markerer at alle værdier i intervallet 0 til 255 tillades.

Kildekode B.1 Eksempelfilter til konvertering.

```
1 Access-list 108 permit tcp 192.168.*.* Any eq WWW
```

Eksemplet i kildekode B.1 på foregående side laves om til et sammensat logisk udtryk, ved at lave hver af parametrene om til variabler. Det logiske udtryk for det nævnte eksempel kan ses i ligning B.4:

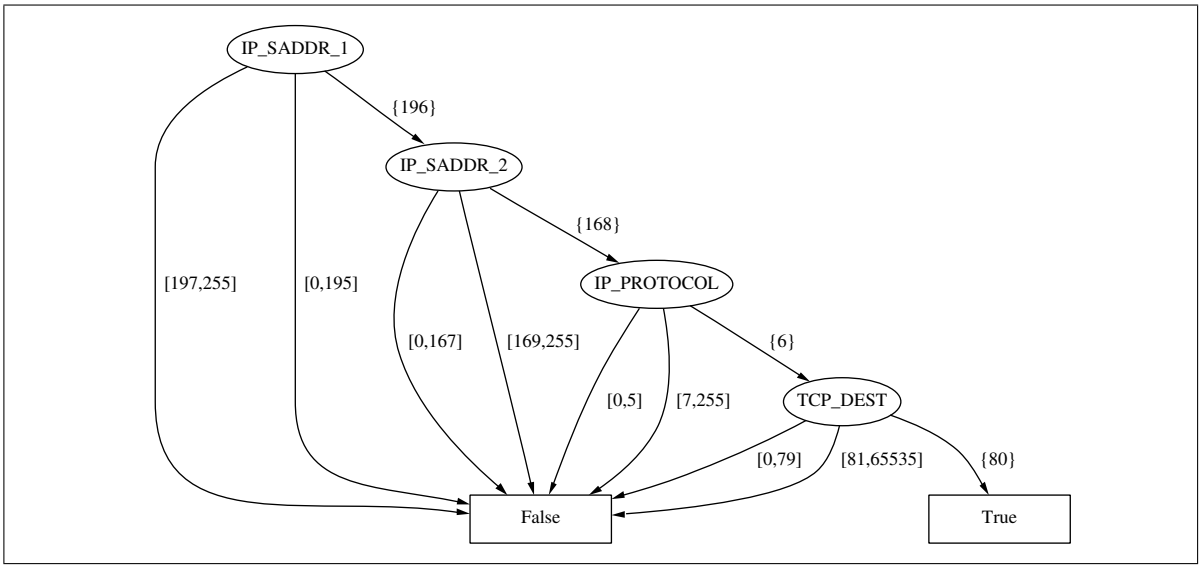
$$\begin{array}{l} (IP\_PROTOCOL = 6) \ \wedge \ (IP\_SADDR = 192.168.*.*) \ \wedge \\ (IP\_DADDR = *.*.*.*) \ \wedge \ (TCP\_DEST = 80) \\ \Downarrow \\ (IP\_PROTOCOL = 6) \ \wedge \ (IP\_SADDR\_1 = 192) \ \wedge \\ (IP\_SADDR\_2 = 168) \ \wedge \ (TCP\_DEST = 80) \end{array} \tag{B.4}$$

IP_PROTOCOl elementet, er hvilken protokol der må anvendes, hvor 6 svarer til TCP. IP_-SADDR beskriver source IPen. IP_DADDR beskriver destionation IPen og TCP_DEST beskriver den tilladte port. I linie to er det logiske udtryk reduceret. Alle fulde intervaller (*) er frasorteret da disse blot modsvarer terminalen true. Derudover er IP_SADDR opdelt i to dele, en for hver byte i den betydende del af IP adressen. Før PROTOCOL og SADDR er ses prefixet IP_, dette beskriver at disse elementer findes i IP headere, tilsvarende beskriver TCP_prefixet at disse elementer findes i TCP headere.

Den tilsvarende IDD er defineret i følgende ligninger:

$$IP\_SADDR\_1 = x \rightarrow (\{0,191\},F)(\{192\},SADDR\_2)(\{193,255\},F) \tag{B.5}$$
$$IP\_SADDR\_2 = x \rightarrow (\{0,167\},F)(\{168\},PROTOCOL)(\{169,255\},F) \tag{B.6}$$
$$IP\_PROTOCOL = x \rightarrow (\{0,5\},F)(\{6\},TCP\_DEST)(\{7,255\},F) \tag{B.7}$$
$$TCP\_PORT = x \rightarrow (\{0,79\},F)(\{80\},T)(\{81,65535\},F) \tag{B.8}$$

IDDen for udtrykket vil bestå af en knude for hver variabel, samt for true og false terminalerne, altså ialt seks knuder. False terminalen, samt kanterne til den, er tilføjet for at opnå intervaldækning. IDDen for kildekode B.1 på foregående side kan ses på figur B.1.



Figur B.1: Eksempel på IDD for filtret beskrevet i kildekode B.1 på forrige side

B.2 MTIDD

Da et filter kan have flere terminaler end true og false (eller permit og deny), for eksempel log, er det nødvendig at udvide IDD definitionen for at håndtere dette. Løsningen hedder MTIDD, hvilket står for Multi Terminal Interval Decision Diagram. En MTIDD er næsten det samme som en IDD bortset, fra at i stedet for at definere terminalerne som true eller false, er de for MTIDDer defineret med T som er en mængde af terminaler.

Forskellen mellem IDD og MTIDD kan ses ved at sammenholde definitionerne på IDD knude og funktionen til at finde heltalsvariablen for en knude med nedenstående definitioner for de tilsvarende begreber for en MTIDD.

Definition 5 *Lad x være en heltalsvariabel defineret ved domænet $D_x \subseteq N$, hvor N er alle heltal. Lad t være et logisk udtryk på heltalsvariabler. Så er t en IDD knude hvis et af følgende udtryk er sand.*

- $t \in T$
- $t = (x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee ...(x \in I_k \wedge t_k)$

$(I_i)i \leq k$ er en intervalpartition af D_x og $(t_i)i \leq k$ er et sæt af MTIDD knuder. Kanterne fra t defineret ved $t = x \rightarrow (I_0, t_0)(I_1, t_1)...(I_n, t_k)$, hvor I_n er intervallet, som kantens label dækker og t_k er den knude, som kanten går hen til.

Rod og konsistens er det samme som for IDD, men funktioner som giver navnet på integervariablen for en knude er ændret til at kunne håndtere multiple terminaler.

- $var(t) = \{x$, hvis $t = x \rightarrow (I_0, t_0)(I_1, t_1)...(I_k, t_k)\}$
- $var(t) = \{t$, hvis $t \in T$ $\}$

Selve definitionen for en MTIDD er den samme som for en IDD, og kan ses i definition 6.

Definition 6 *En MTIDD er et konsistent sæt af punkter og en ordning af heltalsvariablerne. $I = ((t_i)i \leq n, \succ)$ hvor $(t_i)i \leq n$ er et konsistent sæt af MTIDD knuder, og $\succ$ er en ordning af heltalsvariablerne. For alle $t \in (t_i)i \leq n$ med $t = x \rightarrow (I_0, t_0')(I_1, t_1')...(I_k, t_k')$, har vi $x \succ var(t_i')$ for hver $i \leq k$.*

Da en MTIDD ikke benytter boolske terminaler kan man ikke bruge de boolske operationer på en MTIDD. Det vil sige det vil være nødvendig at benytte IDDer, mens regler evalueres til IDDer og til sidst kombinere de forskellige IDDer til en MTIDD.

B.3 Udtrykskraft for IDD og MTIDD

En IDD udtrykker prædikat logik, som kan beskrives som et udsagn der beskriver egenskaber ved et objekt eller forholdet mellem objekter repræsenteret ved variabler. Påstanden $x < 3$ består af to dele, en variabel x og et prædikat < 3. Når x får tildelt en værdi kan der findes en sandhedsværdi for prædikatet. Sandhedsværdien af en IDD er sandhedsværdien af konjunktionen mellem hver sætningsfunktion $P(x)$, hvor P er prædikatet og x er en variabel. Hver $P(x)$ giver sandhedsværdien for en knude i IDDen.

Der er to mulige sandhedsværdier for en IDD: sand eller falsk. De sætningsfunktioner der indgår i en IDD er logiske udtryk indeholdende variabler. Da en IDD kun har to mulige sandhedsværdier kan der benyttes boolske operationer til at opbygge en IDD. For en MTIDD kan der være flere end to sandhedsværdier, og der kan derfor ikke bruges boolske operatorer til at opbygge en MTIDD.[Ros03]

APPENDIKS **C**

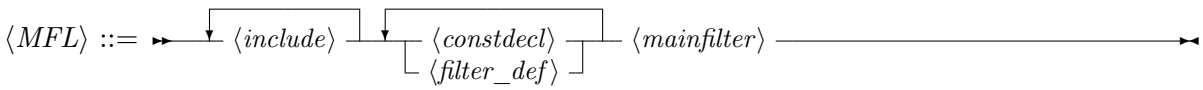
Konkret syntaks for MFL

Dette appendiks beskriver konkret syntaks for MFL , ved brug af syntaksdiagrammer.

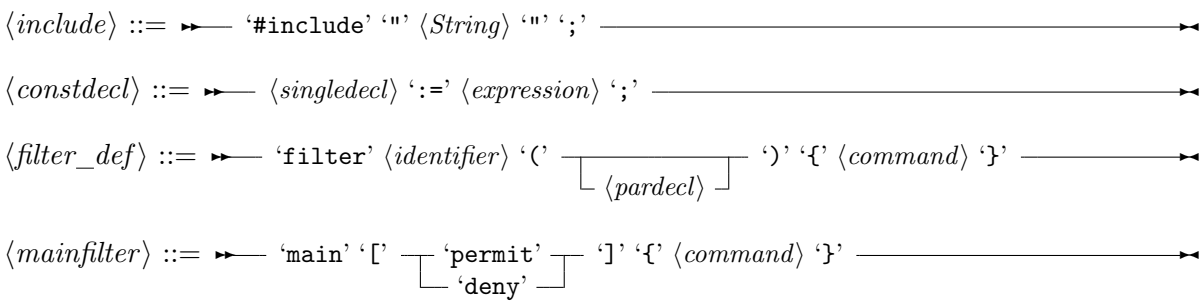
Appendikset er opdelt i flere afsnit, startende med startproduktionen, derpå præambel, det vil sige overordnede produktioner under startproduktionen. I tredie afsnit beskrives kombinatoriske produktioner og i fjerde afsnit gennemgås udtryk. I afsnit fem gennemgås operatorer og i afsnit seks vises dataproduktioner. Til slut, i syvende afsnit, gennemgås de grundlæggende dataproduktioner.

Som forklaring til syntaksdiagrammerne gives her en forklaring af startproduktionen. Alle produktioner består af mindst to elementer: på venstre side af ::= ser vi produktionens navn, på højre side hvad produktionen kan indholde. Produktionen $\langle MFL \rangle$ kan indeholde et vilkårligt antal $\langle include \rangle$ produktioner (pilen der går tilbage), derpå et vilkårligt antal $\langle constdecl \rangle$ og/eller $\langle filter\_def \rangle$ produktioner i tilfældig rækkefølge. $\langle mainfilter \rangle$ produktionen afslutter produktionen.

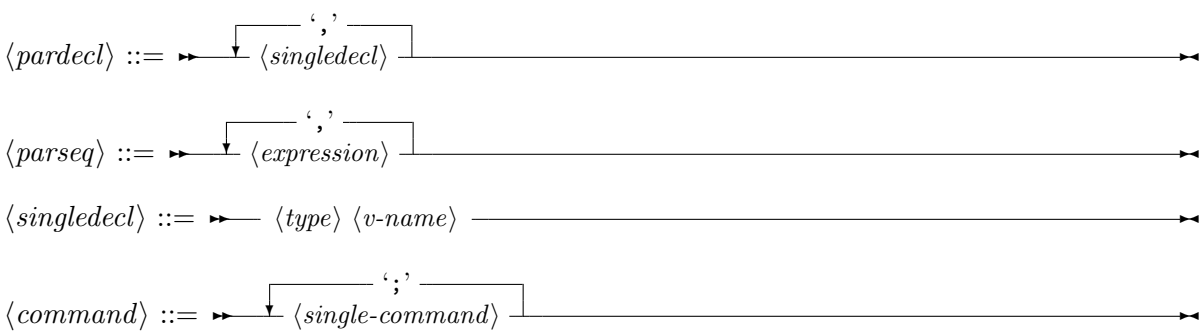
C.1 Startproduktion



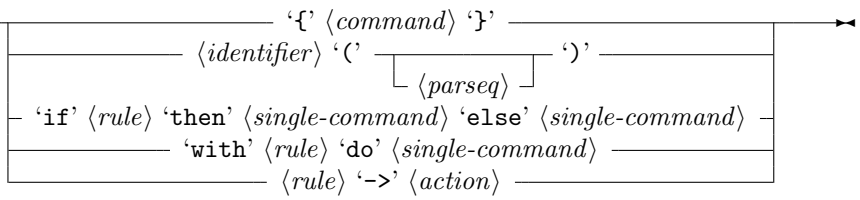
C.2 Præambel



C.3 Kombinatoriske produktioner

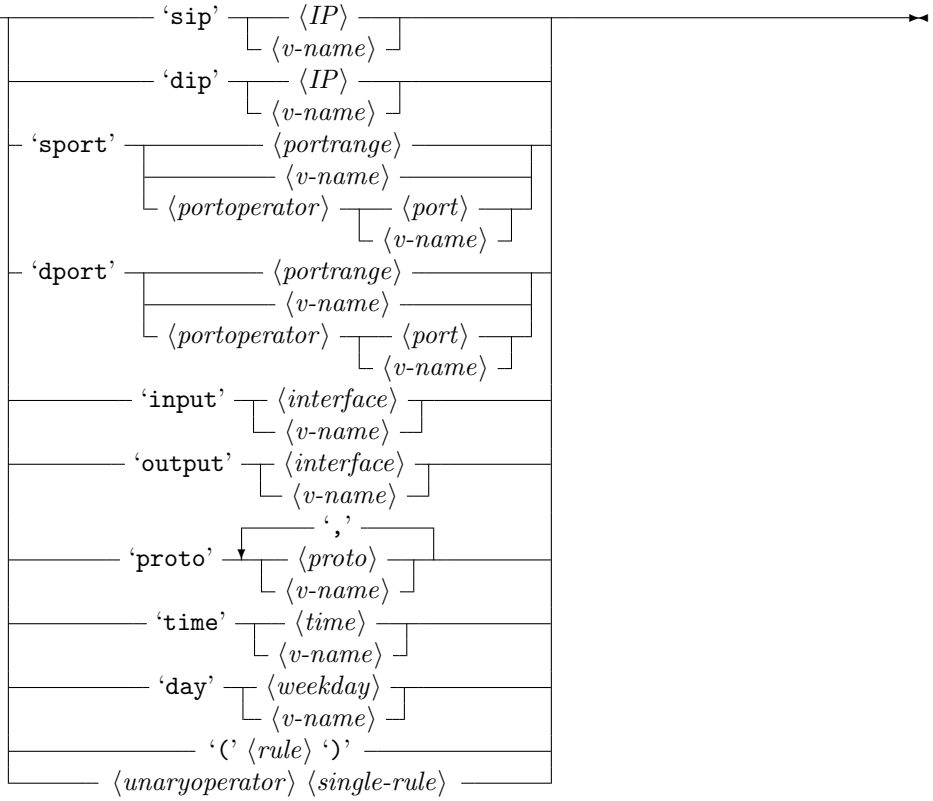


⟨*single-command*⟩ ::=



⟨*rule*⟩ ::=

⟨*single-rule*⟩ ::=



C.4 Udtryk

⟨*expression*⟩ ::=

C.5 Operatorer

⟨*binaryoperator*⟩ ::=

⟨*unaryoperator*⟩ ::= ‘!’

⟨*portoperator*⟩ ::= ‘<’ | ‘<=’ | ‘>’ | ‘>=’

C.6 Dataproduktioner

⟨*type*⟩ ::= ‘ip’ | ‘port’ | ‘iface’ | ‘time’ | ‘day’ | ‘proto’

⟨*IP*⟩ ::= ⟨*range*⟩ ‘.’ ⟨*range*⟩ ‘.’ ⟨*range*⟩ ‘.’ ⟨*range*⟩ | ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘.’ ⟨*int*⟩ ‘/’ ⟨*int*⟩

⟨*port*⟩ ::= ⟨*int*⟩

⟨*portrange*⟩ ::= ⟨*range*⟩

⟨*time*⟩ ::= ⟨*int*⟩ ‘:’ ⟨*int*⟩ ‘-’ ⟨*int*⟩ ‘:’ ⟨*int*⟩

⟨*action*⟩ ::= ‘permit’ [‘state’] [‘log’] | ‘deny’ [‘log’] | ‘log’

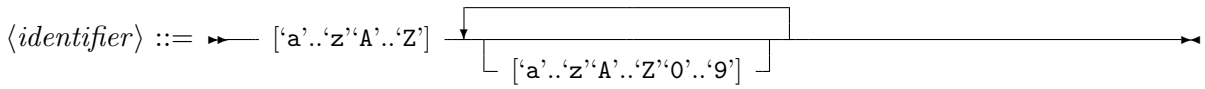
⟨*weekday*⟩ ::= ‘mon’ [‘day’] | ‘tue’ [‘sday’] | ‘wed’ [‘nesday’] | ‘thu’ [‘rsday’] | ‘fri’ [‘day’] | ‘sat’ [‘urday’] | ‘sun’ [‘day’]

⟨*interface*⟩ ::= ‘’’ ⟨*identifier*⟩ ‘’’

⟨*v-name*⟩ ::= ⟨*identifier*⟩

⟨*proto*⟩ ::= ‘TCP’ [‘[’ ⟨*tcpflag*⟩ (‘,’ ⟨*tcpflag*⟩)* ‘]’] | ‘UDP’ | ‘ICMP’ [‘[’ ⟨*int*⟩ (‘,’ ⟨*int*⟩)* ‘]’]

C.7 Grundlæggende dataelementer



APPENDIKS **D**

Abstrakt syntaks for MFL

Formålet med dette appendiks er at give et indblik i den abstrakte syntaks for MFL , ved hjælp af EBNF grammatikker.

Dette appendiks er opdelt i flere afsnit, begyndende med startproduktionen i første afsnit. Derpå andet afsnit om præambel, det vil sige overordnede produktioner under startproduktionen. I tredie afsnit beskrives kombinatoriske produktioner og i fjerde afsnit gennemgås udtryk. I afsnit fem gennemgås operatorer og i afsnit seks vises dataproduktioner. Til slut, i syvende afsnit, gennemgås de grundlæggende dataproduktioner.

I det følgende gives en kort gennemgang af startproduktionen, for at illustrere, hvorledes en EBNF grammatik er opbygget. Hver produktion består af minimum to elementer. På venstre side af ::= ses produktionens navn, på højresiden hvad produktionen kan indeholde. Startproduktionen, $\langle MFL \rangle$, kan bestå af en arbitrær mængde (*) af $\langle vardecl \rangle$ og/eller $\langle filter\_def \rangle$ efterfulgt af produktionen $\langle mainfilter \rangle$ der skal være præcist en gang for hver startproduktion.

D.1 Startproduktion

$\langle MFL \rangle$ ::= ($\langle vardecl \rangle$ | $\langle filter\_def \rangle$)* $\langle mainfilter \rangle$ **Document**

D.2 Præambel

$\langle vardecl \rangle$ ::= $\langle type \rangle$ $\langle v\text{-}name \rangle$ ‘`:=`’ $\langle expression \rangle$ ‘`;`’ **VariableDeclaration**

$\langle filter\_def \rangle$ ::= ‘`filter`’ $\langle identifier \rangle$ ‘`(`’ $\langle pardecl \rangle$? ‘`)`’ ‘{’ $\langle command \rangle$ ‘}’ **FilterDefinition**

$\langle mainfilter \rangle$ ::= ‘`main`’ ‘`[`’ (‘`permit`’ | ‘`deny`’) ‘`]`’ ‘{’ $\langle command \rangle$ ‘}’ **MainFilter**

D.3 Kombinatoriske produktioner

$\langle pardecl \rangle$ ::= $\langle type \rangle$ $\langle v\text{-}name \rangle$ (‘`,`’ $\langle type \rangle$ $\langle v\text{-}name \rangle$)* **ParameterDeclaration**

$\langle command \rangle$ ::= $\langle identifier \rangle$ ‘`(`’ ($\langle expression \rangle$ (‘`,`’ $\langle expression \rangle$)*)? ‘`)`’ **CallCommand**
| ‘`if`’ $\langle rule \rangle$ ‘`then`’ $\langle command \rangle$ ‘`else`’ $\langle command \rangle$ **IfCommand**
| $\langle rule \rangle$ ‘`->`’ $\langle command \rangle$ **RuleCommand**
| $\langle command \rangle$ ‘`;`’ $\langle command \rangle$ **SequentialCommand**

$\langle rule \rangle$ ::= ‘`sip`’ ($\langle IP \rangle$ | $\langle v\text{-}name \rangle$) **SipRule**
| ‘`dip`’ ($\langle IP \rangle$ | $\langle v\text{-}name \rangle$) **DipRule**
| ‘`sport`’ ($\langle portrange \rangle$ | $\langle v\text{-}name \rangle$ | $\langle portoperator \rangle$ ($\langle port \rangle$ | $\langle v\text{-}name \rangle$)) **SportRule**
| ‘`dport`’ ($\langle portrange \rangle$ | $\langle v\text{-}name \rangle$ | $\langle portoperator \rangle$ ($\langle port \rangle$ | $\langle v\text{-}name \rangle$)) **DportRule**
| ‘`input`’ ($\langle interface \rangle$ | $\langle v\text{-}name \rangle$) **InputRule**

	‘`output`’ (⟨*interface*⟩ | ⟨*v-name*⟩)	**OutputRule**
|	’proto’ (⟨*proto*⟩ | ⟨*v-name*⟩) (‘`,`’ (⟨*proto*⟩ | ⟨*v-name*⟩))∗	**ProtoRule**
|	‘`time`’ (⟨*time*⟩ | ⟨*v-name*⟩)	**TimeRule**
|	‘`day`’ (⟨*weekday*⟩ | ⟨*v-name*⟩)	**DayRule**
|	⟨*unaryoperator*⟩ ⟨*rule*⟩	**UnaryOpRule**
|	⟨*rule*⟩ ⟨*binaryoperator*⟩? ⟨*rule*⟩	**CombRule**

D.4 Udtryk

⟨*expression*⟩ ::=	⟨*v-name*⟩	**VarNameExpr**
|	⟨*IP*⟩	**IpExpr**
|	⟨*portrange*⟩	**PortExpr**
|	⟨*interface*⟩	**IfaceExpr**
|	⟨*weekday*⟩	**WeekdayExpr**
|	⟨*time*⟩	**TimeExpr**
|	⟨*proto*⟩	**ProtoExpr**
|	⟨*any*⟩	**AnyExpr**

D.5 Operatorer

⟨*binaryoperator*⟩ ::= ‘`&&`’ | ‘`||`’ | ‘`&!`’ | ‘`^^`’

⟨*unaryoperator*⟩ ::= ‘`!`’

⟨*portoperator*⟩ ::= ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’

D.6 Dataproduktioner

⟨*type*⟩ ::= ‘`ip`’ | ‘`port`’ | ‘`iface`’ | ‘`time`’ | ‘`day`’ | ‘`proto`’

⟨*IP*⟩ ::= ⟨*range*⟩ ‘`.`’ ⟨*range*⟩ ‘`.`’ ⟨*range*⟩ ‘`.`’ ⟨*range*⟩
| ⟨*int*⟩ ‘`.`’ ⟨*int*⟩ ‘`.`’ ⟨*int*⟩ ‘`.`’ ⟨*int*⟩ ‘`/`’ ⟨*int*⟩

⟨*portrange*⟩ ::= ⟨*range*⟩

⟨*port*⟩ ::= ⟨*int*⟩

⟨*action*⟩ ::= (‘`permit`’ ‘`state`’? | ‘`deny`’) ‘`log`’?
| ‘`log`’

⟨*time*⟩ ::= ⟨*int*⟩ ‘`:`’ ⟨*int*⟩ ‘`-`’ ⟨*int*⟩ ‘`:`’ ⟨*int*⟩

⟨*weekday*⟩ ::= ‘`mon`’ ‘`day`’?
| ‘`tue`’ ‘`day`’?
| ‘`wed`’ ‘`nesday`’?
| ‘`thu`’ ‘`rsday`’?
| ‘`fri`’ ‘`day`’?
| ‘`sat`’ ‘`urday`’?
| ‘`sun`’ ‘`day`’?

⟨*interface*⟩ ::= ‘’’ ⟨*identifier*⟩ ‘’’

⟨*v-name*⟩ ::= ⟨*identifier*⟩

⟨*proto*⟩ ::= ‘`TCP`’ (‘`[`’ ⟨*tcpflag*⟩ (‘`,`’ ⟨*tcpflag*⟩) ‘`]`’)?
| ‘`UDP`’
| ‘`ICMP`’ (‘`[`’ ⟨*int*⟩ (‘`,`’ ⟨*int*⟩) ‘`]`’)?

⟨*tcpflag*⟩ ::= ‘`SYN`’ | ‘`ACK`’ | ‘`URG`’ | ‘`FIN`’ | ‘`PSH`’ | ‘`RST`’

⟨*any*⟩ ::= ‘`any`’

D.7 Grundlæggende dataelementer

⟨*identifier*⟩ ::= [‘`a`’...‘`z`’‘`A`’...‘`Z`’] ([‘`a`’...‘`z`’‘`A`’...‘`Z`’‘`0`’...‘`9`’])∗

⟨*range*⟩ ::= (⟨*int*⟩ (’`-`’ ⟨*int*⟩)?) | ’`*`’

⟨*int*⟩ ::= [‘`0`’...‘`9`’] [‘`0`’...‘`9`’]∗

APPENDIKS **E**

Typer

Formålet med dette appendiks give et indblik i de datatyper MFL kompileren anvender i lexeren.

E.1 Lexertyper

For hver type listes dens navn, metatype, dvs. hvordan typen er opbygget af primitive datatyper, samt en kort beskrivelse.

Navn	`keyword_t`

Metatype	`Main | Filter | Any`
	`| If | Then | Else | With | Do`
	`| Sip | Dip | Sport | Dport | Input | Output`
	`| Flags | Proto | Time | Day | Ip | Port | Iface`
	`| Mon | Tue | Wed | Thu | Fri | Sat | Sun`
	`| Permit | Deny | Log | State`
	`| Syn | Ack | Urg | Fin | Psh | Rst`
	`| Tcp | Udp | Icmp`

Beskrivelse	Beskriver et keyword i MFL.

Navn	`symbol_t`

Metatype	`Left | Right | CurlyLeft | CurlyRight`
	`| SquareLeft | SquareRight | Comma`
	`| Dot | Slash | Semicolon | Star`
	`| Not | Or | Xor | Apos`
	`| Colon | Assignment | Dash | Rule`
	`| Lt | LtE | Gt | GtE | And | Nand`

Beskrivelse	Beskriver et symbol i MFL.

Navn	`token_t`

Metatype	`Tkeyword of keyword_t`
	`Tsymbol of symbol_t`
	`Tident of string`
	`Tint of int`
	`TunRec of string`

Beskrivelse	Beskriver et token i MFL.

Navn	`lexem_t`

Metatype	`(token_l : token_t;`
	` line_l : int;`
	` s_col_l : int;`
	` e_col_l : int;`
	` filename_l : string)`

Beskrivelse	Beskriver et lexem i MFL.

Navn	`string_to_lex_t`

Metatype	`(string : string;`
	` current : int;`
	` size : int;`
	` line : int;`
	` col : int;`
	` filename : string)`

Beskrivelse	Der oprettes en instans af denne type, for hver kildekodefil der inkluderes. Denne type benyttes som en intern pointer i kildekodefilen der holder styr på den nuværende position, filnavnet samt kildekodens længde.

E.2 AST typer

Her gives et indblik i de datatyper MFL kompileren anvender, til opbygning af et AST, under syntaktisk analyse. For hver type listes dens navn, en reference til hvor den kommer fra i BNFen, metatype, dvs. hvordan typen er opbygget at primitive typer, samt en kort beskrivelse. Hvis typen er til for at opfylde krav i OCaml syntaksen, eller på anden måde ikke er direkte relateret til BNFen, vil dette blive angivet under BNF punktet.

Navn	`range_t`
BNF	Ingen - Hjælpetype
Metatype	`(low: int; high:int): Record`
Beskrivelse	Kan indeholde en range. Bruges i `mrange_t`

Navn	`mrange_t`
BNF	`<range>`

Metatype	`Range: range_t eller`
	`Singleton: int eller`
	`FullRange`

Beskrivelse	Kan indeholde en `range_t`, et enkelt integer eller være en FullRange, dvs. hele intervallet. Beskriver sammen med `range_t <range>` i BNFen.

Navn	`any_t`
BNF	`<any>`
Metatype	`Any`
Beskrivelse	Any bruges alle steder i kald af filter hvor man ikke ønsker at benytte sig af en given parameter, kan sammenlignes lidt med NULL i C.

Navn	`proto_t`
BNF	`<tcpflag>`

Metatype	`Syn eller`
	`Ack eller`
	`Urg eller`
	`Fin eller`
	`Psh eller`
	`Rst eller`

Beskrivelse	Indeholder en af de 6 TCP protokolflag

Navn	`proto_t`
BNF	`<proto>`

Metatype	`Tcp: tcpflag_t list eller`
	`Udp eller`
	`Imcp: int list`

Beskrivelse	Indeholder en af de 3 protokoltyper, TCP, UDP eller IMCP.

Navn	`weekday_t`
BNF	`<weekday>`

Metatype	`Mon eller`
	`Tue eller`
	`Wed eller`
	`Thu eller`
	`Fri eller`
	`Sat eller`
	`Sun`

Beskrivelse	Direkte konvertering fra BNFens `<weekday>` hvor alle dage er forkortet til tre bogstaver.

Navn	`time_t`
BNF	`<time>`
Metatype	`(hour: int; minute:int): Record`
Beskrivelse	Direkte konvertering fra BNFens `<time>`

Navn	`action_t`
BNF	`<single-command>`

Metatype	`Log eller`
	`Permit eller`
	`PermitState eller`
	`PermitLog eller`
	`PermitStateLog eller`
	`Deny eller`
	`DenyLog`

Beskrivelse	Lille del af `<single-command>`. Hvor alle mulige kombinationer skrevet op

Navn	`ip_t`
BNF	`<IP>`

Metatype	`(ip1: mrange_t,`
	` ip2: mrange_t,`
	` ip3: mrange_t,`
	` ip4: mrange_t): Record`

Beskrivelse	I BNFen er det også muligt at skrive IPer med netmasker, disse vil blive konverteret til IPer med ranges i parseren.

Navn	`type_t`
BNF	`<type>`

Metatype	`IpType eller`
	`ProtType eller`
	`IfaceType eller`
	`TimeType eller`
	`DayType eller`
	`FlagsType eller`
	`Prototype`

Beskrivelse	Direkte konvertering af `<type>` fra BNFen.

Navn	`portoperator_t`
BNF	`<portoperator>`

Metatype	`GT eller`
	`GtE eller`
	`LT eller`
	`LtE`

Beskrivelse	GT er `>`, GtE er `>=`, LT er `<` og LtE er `<=`

Navn	`unaryoperator_t`
BNF	`<unaryoperator>`
Metatype	`Not`
Beskrivelse	Not er `!` i BNFen

Navn	`binaryoperator_t`
BNF	`<binaryoperator>`

Metatype	`And eller`
	`Or eller`
	`AndNot eller`
	`XOr`

Beskrivelse	And er `&&`, Or er `||`, AndNot er `&!` og XOr er `^^` i BNFen

Navn	`expression_t`
BNF	`<expression>`
Metatype	`VarNameExpr: string` eller `IpExpr: ip_t` eller `PortExpr: mrange_t` eller `IfaceExpr: string` eller `WeekdayExpr: weekday_t` eller `TimeExpr: time_t` eller `FlagExpr: flag_t` eller `ProtoExpr: proto_t` eller `AnyExpr: any_t`
Beskrivelse	Direkte konvertering af `<expression>` fra BNFen.

Navn	`iprule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueIp: ip_t` eller `VNameIp: string` eller
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`portrule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValuePort: mrange_t` eller `VNamePort: string` eller `OpValuePort: tuple(portoperator_t, int)` eller `OpVNamePort: tuple(portoperator_t, string)` eller
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`interfacerule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueIface: string` eller `VNameIface: string` eller
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`flagrule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueFlag: flag_t` eller `VNameFlag: string` eller
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`protorule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueProto: proto_t` eller `VNameProto: string` eller
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`timerule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueTime: time_t  eller` `VNameTime: string  eller`
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`dayrule_t`
BNF	Hjælpetype - lille del af `<single-rule>`
Metatype	`ValueDay: weekday_t  eller` `VNameDay: string  eller`
Beskrivelse	Hjælpetype til `rule_t` (`<single-rule>` fra BNFen.)

Navn	`rule_t`
BNF	`<single-rule>` og `<rule>`
Metatype	`SipRule: iprule_t` eller `DipRule: iprule_t` eller `SportRule: portrule_t` eller `DportRule: portrule_t` eller `InputRule: interfacerule_t` eller `OutputRule: interfacerule_t` eller `FlagsRule: flagrule_t list` eller `ProtoRule: protorule_t list` eller `TimeRule: timerule_t * timerule_t` eller `DayRule: dayrule_t` eller `UnaryOpRule: unaryoperator_t * rule_t` eller `CombRule: rule_t * binaryoperator_t * rule_t` eller
Beskrivelse	Direkte konvertering fra BNFens `<single-rule>` konkateneret med `<rule>`. Nogle af mulighederne er lister, indikeret at *list*. Samme syntaks forekommer ligeledes i nogle af de næste typer.

Navn	`command_t`
BNF	`<single-command>` og `<command>`
Metatype	`CallCommand: string * expression_t list` eller `IfCommand: rule_t * command_t  * command_t` eller `WithCommand: rule_t * command_t` eller `RuleCommand: rule_t * action_t` eller `SequentialCommand: command_t * command_t` eller
Beskrivelse	Direkte konvertering fra BNFens `<single-command>` konkateneret med `<command>`.

Navn	`pardecl_t`
BNF	`<pardecl>` og `<parseq>`
Metatype	`tuple(type_t, string) list`
Beskrivelse	Direkte konvertering fra BNFens `<pardecl>` konkateneret med `<parseq>`.

Navn	`mainfilter_t`
BNF	`<mainfilter>`
Metatype	`tuple(action_t, command_t)`
Beskrivelse	Direkte konvertering fra BNFens `<mainfilter>`.

Navn	`filterdef_t`
BNF	`<filterdef>`
Metatype	`tuple(string, pardecl_t, command_t)`
Beskrivelse	Direkte konvertering fra BNFens `<filterdef>`.

Navn	`vardecl_t`
BNF	`<vardecl>`
Metatype	`tuple(type_t, string, expression_t)`
Beskrivelse	Direkte konvertering fra BNFens `<vardecl>`.

Navn	`defines_t`
BNF	Hjælpetype - En del af `<MFL>`
Metatype	`VarDecl: vardecl_t` eller `FilterDef: filterdef_t` eller
Beskrivelse	Enten en variabel- eller en filtererklæring til brug i `mfl_t` (`<MFL>`).

Navn	`mfl_t`
BNF	`<MFL>`
Metatype	`tuple(defines_t list, mainfilter_t)`
Beskrivelse	Direkte konvertering fra BNFens `<MFL>`.

E.3 Dekoreret AST typer

I den dekorerede AST er typerne med fire undtagelser fuldstændig analoge til de typer der bliver brugt til den ikke dekorerede AST, se afsnit E.2 på side 93. Derfor vil der herunder kun bliver listet de ændringer der er foretaget.

1. Alle navne på typerne har fået prefixet `deco_`, se tabel E.1 på næste side.
2. I `deco_mrange_t`, se tabel E.1 på modstående side, er `FullRange` fjernet. Dette skyldes at hvor det i ASTet ikke var muligt at bestemme om en given range var en portrange eller en iprange, er det nu muligt. Derfor anvendes nu `deco_range_t` til at beskrive den range.
3. I `deco_expression_t` er `VarNameExpr` og `AnyExpr` blevet ændret til tupler, hhv.
   `tuple(string, deco_type_t)` og `tuple(deco_any_t, deco_type_t)`.
   Dette fordi at det nu er muligt at angive typen på en given konstant eller any udtryk i MFL, se tabel E.2 på næste side.

Navn	`deco_mrange_t`
BNF	`range`
Metatype	`Range: deco_range_t` eller `Singleton: int`

Tabel E.1: Dekoreret ASTtype for mrange.

Navn	`deco_expression_t`
BNF	`expression`
Metatype	`VarNameExpr: tuple(string, deco_type_t)` eller `IpExpr of deco_ip_t` eller `PortExpr: deco_mrange_t` eller `IfaceExpr: string` eller `WeekdayExpr: deco_weekday_t` eller `TimeExpr: deco_time_t` eller `FlagExpr: deco_flag_t` eller `ProtoExpr: deco_proto_t` eller `AnyExpr: tuple(deco_any_t, deco_type_t)`

Tabel E.2: Dekoreret ASTtype for expression.

APPENDIKS **F**

Syntaktisk analyse

Dette appendiks er en gennemgang af de funktioner, der anvendes i syntaktisk analyse. Formålet er at give et indblik i hvorledes syntaktisk analyse udføres for MFL, samt at vise hvorledes et parsetræ opbygges.

Syntaktisk analyse omhandler leksikalsk analyse og parsing. Den leksikalske analyse foretages af en lexer, der implementeres som en funktion, der læser fra kildekoden og returnerer en liste af tokens, som parseren efterfølgende skal behandle.

Parseren skal på baggrund af de tokens, den modtager fra lexeren, analysere kildekoden, så det sikres at den er syntaktisk korrekt. Samtidig opbygges et parsetræ, ud fra de analyserede tokens.

I de følgende afsnit beskrives først funktioner i lexeren, derefter funktioner i parseren. Under denne gennemgang er følgende beskrevet.

- Funktionens navn.
- Funktionens formål.
- Lovlige tokens, det vil sige de tokens parseren forventer at læse, for at kunne kalde pågældende funktion. (kun for parser funktioner).
- Parametre, udelades hvis der ikke er parametre.
- Returtype.
- Beskrivelse af funktionen.

I funktionsgennemgangen skrives et funktionsnavn på formen `funktions_navn`, lokale variable skrives på formen *lokal_variabel* og globale variable skrives som *global_variabel*. Tokens fra lexeren skrives som **Tokennavn** og typer skrives som `entype_t`.

F.1 Funktioner i lexer

I det følgende beskrives de funktioner, som udgør lexeren.

F.1.1 Funktionsbeskrivelser

`init_lex`
Formål: `At initiere en række variabler som lexeren bruger.`
Parametre: `s:string, filename:string, filenamelist:string list`
Returtype: `(src:string_to_lex_t,filelist:string list)`
Beskrivelse:
Denne funktionen benyttes til at initialisere en kildekode fil. Funktionen returnerer efterfølgende en record indeholdende kildekoden, der samtidig fungere som container for en række tællere, der benyttes som interne pointere til at holde styr på, hvor langt lexeren er kommet i

koden. Endelig returnerer funktionen en opdateret liste med filer, med den aktuelle fil tilføjet.

■

alpha_starter

Formål: At identificere og opbygge et lexem, hvor første tegn er et bogstav
Parametre: src: string_to_lex_t
Returtype: lexem_t
Beskrivelse:
Funktionen kaldes når den næste karakterer i kildekoden er a-z eller A-Z. Funktionen udtrækker efterfølgende alfanumeriske karakterer, indtil den møder et ikke-alfanumerisk tegn. Derefter afgøres, om det fundne lexem er af typen Tkeyword eller Tident. Efterfølgende returneres lexem_t indeholdende token typen samt linie, kolonne samt filnavn for det aktuelle lexem.

■

int_starter

Formål: At identificere og opbygge et lexem, hvor første tegn er et tal
Parametre: src: string_to_lex_t
Returtype: lexem_t
Beskrivelse:
Funktionen kaldes når den næste karakterer i kildekoden er 0-9. Funktionen udtrækker efterfølgende numeriske karakterer, indtil den møder et ikke-numerisk tegn. Efterfølgende returneres lexem_t indeholdende token typen samt linie, kolonne samt filnavn for det aktuelle lexem.

■

single_symbol

Formål: At identificere og opbygge et lexem, som indeholder en karakter
Parametre: src: string_to_lex_t
Returtype: lexem_t
Beskrivelse:
Funktionen kaldes når den næste karakter i kildekoden er et gyldigt symbol med længde 1, hvor symbolet ikke kan være starten på andre symboler. Funktionen returnerer efter at have opdateret tællerne i *string_to_lex*, typen lexem_t indeholde Tsymbol med attributten for det aktuelle symbol.

■

symbol_starter

Formål: At identificere og opbygge et lexem, hvor første tegn hverken er et tal eller et bogstav
Parametre: src: string_to_lex_t
Returtype: lexem_t
Beskrivelse:
Funktionen kaldes når den næste karakter i kildekoden er et gyldigt symbol karakter, der ikke i sig selv er et gyldigt symbol. Funktionen udtrækker efterfølgende gyldige symbol karakterer fra

teksten indtil den møder et alfanumerisk eller ugyldigt symboltegn. Endeligt afgøres om den fundne symbol streng er et gyldigt symbol. Hvis dette er tilfældet returneres lexem_t indeholde Tsymbol med attributten for det aktuelle symbol - ellers TunRec.

■

hash_starter

Formål: At identificere og håndtere præprocessing direktiver
Parametre: src: string_to_lex_t
Returtype: string
Beskrivelse:
Funktionen kaldes når næste karakter i kildekoden er et hash tegn, hvilket vil sige at resten af linien skal betragtes som et preprocessing direktiv. Efterfølgende udtrækkes resten af linien til en streng, på hvilken der søges vha. regulære udtryk. Det eneste preprocessing direktiv der findes i sproget er include. Udfra linien udtrækkes filnavnet på den fil, som ønskes inkluderet. Kan filnavnet ikke matches, returneres den tomme streng - ellers returneres filnavnet.

■

lexer

Formål: Funktinen er den primære lexer funktion
Parametre: src: string_to_lex_t,filelist:stringlist
Returtype: lexem_t list
Beskrivelse:
Denne funktion er den primære lexer funktion. Udfra *string_to_lex* som den modtager fra `init_lex`, processerer den karakter for karakterer og afgører hvilke funktioner der skal kaldes (`alpha_starter`, `int_starter`, `symbol_starter`, `hash_starter` eller `single_symbol`). Funktionen benyttes rekursivt, idet den kalder sig selv efter hvert lexem er blevet bestemt - medmindre slutningen af kildekoden nåes. Ved preprocessing direktiver omkring inkludering af filer, undersøges om filen eksisterer, samt at den ikke er blevet inkluderet før. Hvis dette er opfyldt kaldes funktionen på den inkluderede fil. Funktionen returnerer en liste med de fundne lexems.

■

get_lexem_list

Formål: At returnere lexemlisten til parseren
Parametre: filename: string
Returtype: lexem_t list
Beskrivelse:
Denne funktion benyttes af parseren - som indgangspunkt til `init_lex` og `lexer`. Funktionen forsøger at læse et filnavn fra parameteren filename, hvorefter den initieres med `init_lex`. Efterfølgende kaldes `lexer` på den returnerede *string_to_lex*, hvorved der opnåes en liste med lexems. Denne liste returneres.

■

F.2 Funktioner i parser

Parseren læser en gang fra lexeren, hvor den modtager en liste over alle de lexems kildekoden består af. Hvert enkelt lexem matches mod de tokens, parseren på et givent tidspunkt forventer at læse. Denne mængde af forventede tokens kaldes: Lovlige tokens. Det vil sige, at læser parseren i en given funktion, et lexem der ikke er forventet er der syntaksfejl.

Efterhånden som lexems læses opbygges et parsetræ. Parsetræet opbygges af en række typer, beskrevet i appendiks E.2 på side 93.

Der finders tre vigtige hjælpefunktioner, som alle de andre funktioner kalder: `accept`, `peek` og `peek_token`.

`accept` kaldes med en liste af tokens, det kontrolleres om disse tokens stemmer overens med de næste lexems i *lexerlisten*, hvis de gør returners en *lexerlist* hvor disse er fjernet. Hvis ikke, kastes en syntaksfejl.

`peek` kaldes med den nuværende *lexerlist* og et tal, hvorefter den returner det lexem som står på tallets plads. Listen starter ved 1.

`peek_token` returner tokenet fra lexemet returneret af kaldet til `peek(1)`.

Som nævnt i afsnit 6.2.2 på side 47, skal de forskellige konstruktorer der genererer typerne i ASTet indeholde 1 eller 2 heltal. Disse heltal oplyser placeringen i den originale *lexerlist*, af første og sidste token der tilhører den pågældende type. Når en type kun skal have et heltal, er det fordi den kun består af et enkelt token. I funktionsbeskrivelserne, der følger herefter, nævnes disse heltal ikke i forbindelse med konstruktorne, da det ville blive trivielt gentagne gange at nævne dem.

Parseren har fået *lexerlisten* fra `main`, som har den fra lexeren. Denne sendes rundt mellem funktionerne i parseren, hver gang `accept` kaldes, fjernes mindst ét lexem fra starten listen. Dette medfører, at alle de funktioner der nu vil blive beskrevet, modtager denne liste som parameter, og sender den retur, ofte lidt mindre. Da det ville blive trivielt at beskrive hvordan denne liste sendes rundt i hver eneste funktion, undlades herunder at nævne den.

F.2.1 Funktionsbeskrivelser

`mfl`
Formål: `Hovedfunktion for parsing`
Lovlige tokens: `Filter Main Proto Time Day Ip Port Iface`
Returtype: mfl_t
Beskrivelse:
`peek_token` kaldes og der matches på de lovlige tokens, hvis `peek_token` ikke returner et lovlig token er der syntaksfejl i kildekoden.

Hvis det returnerede er starten af en konstantdeklaration eller filterdefinition, f.eks. **Ip**, **Proto** eller **Filter**, kaldes funktionen `defines`, der returnerer en liste af typen defines_t. Den returnerede liste gemmes, som en lokal variabel. Derefter kalder `main_filter`, hvis returværdi også gemmes i en lokalvariabel.

Hvis det returnerede er af typen **Main**, kaldes funktionen `main_filter`, der returnerer en instans af typen mainfilter_t, som gemmes i en lokal variabel og en lokal varaiabel til defines_t sættes til en tom liste.

Til slut returneres listen af defines_t og den returnerede instans mainfilter_t, som instans af typen mfl_t.

■

main_filter

Formål: At opbygge et ast_mainfilter_t
Lovlige tokens: Main
Returtype: mainfilter_t
Beskrivelse:
Først kaldes accept med parameteren **Main** og **SquareLeft**.

Næste peek_token skal returnere typen keyword_t og være enten **Permit** eller **Deny**. Dette angiver standardreglen for det samlede filter.

accept kaldes med den tilsvarende action, **SquareRight** og **CurlyLeft**,

command kaldes, og returnerer en instans af command_t, der gemmes i en lokal variabel, her kaldet *mainCommand*.

Herefter kaldes accept med **CurlyRight**, og der returneres en instans af mainfilter_t indeholdende *mainCommand* og **Permit** eller **Deny**

■

defines

Formål: At parse alle konstanterklæringer og filterdefinitioner
Lovlige tokens: Ip Port Iface Time Day Proto Filter
Returtype: defines_t list
Beskrivelse:
Hvis peek_token er **Filter** kaldes funktionen filter_def. Denne funktion returnerer en instans af typen filterdef_t, her kaldet cur_fdef. Derpå kaldes defines, hvor returværdien gemmes i den lokale variabel cur_def. cur_fdef bruges som parameter til konstruktoren FilterDef, hvilket indsættes som hovedet i cur_def, der jo er en liste af definitioner på filtre og konstanter. Listen bestående af både cur_fdef og cur_def returneres herpå.

Er token returneret af peek_token istedet **Ip**, **Portrange** eller en anden type kaldes constdecl. constdecl returnerer en instans af typen constdecl_t, denne gemmes som $current\_constdecl$. Derpå kaldes defines, hvor returværdien gemmes i den cur_def. $current\_constdecl$ bruges som parameter til konstruktoren ConstDecl, hvilket som ovenfor indsættes som hovedet i cur_def. Listen bestående af både $current\_constdecl$ og cur_def returneres herpå.

Er tokenet fra peek_token intet af ovenstående, returneres blot en tom liste.

■

constdecl

Formål: At parse en enkelt konstanterklæring
Lovlige tokens: Ip Port Iface Time Day Proto
Returtype: constdecl_t

Beskrivelse:
Uanset hvilke af de lovlige tokens der returneres af peek_token, kaldes accept med denne token som parameter. Herpå læses og gemmes den følgende identifier, såfremt der faktisk er tale om en identifier, da der ellers kastes en syntaksfejl. Identifieren gemmes i $current\_ident$, accept kaldes med denne identifier og **Assignment** som parametre. Nu gemmes et funktionskald til expression i den lokale variabel cur_expr. Herefter kaldes accept med tokenet **Semicolon**.

En instans af typen type_t svarende til det i starten læste token, returneres sammen med de lokale variable *current_ident* og *cur_expr*.

■

filter_def

Formål: At parse en filterdefinition
Lovlige tokens: Filter
Returtype: filterdef_t

Beskrivelse:
Først kaldes accept med **Filter**.

Indeholder peek_token derefter en **identifier**, gemmes denne i en lokal variabel, og bruges sammen med **Left** som parameter til accept. Derpå kaldes pardecl, som returnere en liste med de parameter filtret har. Denne liste gemmes også midlertidigt i en lokal variabel. Efter kaldet til pardecl, kaldes accept med **Right** og **CurlyLeft** En sidste lokal variabel oprettes med returværdien fra funktionskald til command. Derefter kaldes accept med **CurlyRight**.

Til sidst oprettes en filterdef_t type, med de omtalte lokale variabler, som derefters returneres.

■

pardecl

Formål: At parse en parameterdeklaration
Lovlige tokens: Ip Port Iface Time Day Proto Right
Returtype: pardecl_t
Beskrivelse:
Der oprettes en variabel af typen pardecl_t, der er en liste af tupler hver repræsenterende en parametererklæring. Indholdet i peek_token matches på de lovlige tokens og accept kaldes. Var indholdet af peek_token **Right**, returneres en tom liste. Var tokenet derimod en type, oprettes en tupel hvor første parameter er en instans af type_t, se appendiks E.2 på side 93, svarende til det læste token. I den anden plads i denne tupel gemmes det følgende token, såfremt det er en **identifier**, da der ellers er en syntaksfejl. Herpå gemmes tuplen i den tidligere oprettede liste, og accept kaldes. Nu foretages et kald til pardecl, altså rekursivt, hvor den returnerede liste appenderes på den tidligere erklærede liste. Til sidst returneres listen.

■

parseq

Formål: At parse en række parametre
Lovlige tokens: identifier Int Mon Tue Wed Thu Fri Sat Sun Syn Ack Urg Fin Psh Rst Tcp Udp Icmp Apos any
Returtype: parameterSeq_t

Beskrivelse:
Indholdet af peek_token matches på de lovlige tokens og der kastes en syntaksfejl, hvis peek_token returnerer noget andet. Uanset hvilken af de lovlige tokens der læses, kaldes expression. Returværdien fra expression gemmes i den lokale variabel *current_par*. Nu kaldes peek_token igen, for at tjekke om det næste token er **Comma**. Hvis det var **Comma**, kaldes accept med **Comma** som parameter, et funktionskald til parseq gemmes i *cur_parseq* og accpet kaldes igen, med parametren **Right** denne gang. Nu returneres *current_par* konkateneret med *cur_parseq*. Hvis der ikke findes et **Comma** ved andet kald til peek_token,

kaldes `accept` med parametren **Right**, hvorefter *current_par* returneres som eneste element i en liste.

■

command

Formål: `At parse en række ast_command_t`
Lovlige tokens: `identifier If With Sip Dip Sport Dport Input Output Proto Time Day Not`
Returtype: `command_t`

Beskrivelse:
`command` kalder først `single_command`, og gemmes i *current_command*.

Efter returværdien er gemt, kaldes `peek_token`. Returnerede `peek_token` et **Semicolon**, kaldes `accept` med **Semicolon** som parameter, og `peek_token` kaldes igen. Returnerer `peek_token` nu et **CurlyRight**, returneres *current_command* blot. Returners **CurlyRight** ikke anden gang, er der tale om en sekventiel kommando, og konstruktoren `SequentialCommand` bruges med *current_command* som første og et kald til `command` som anden parameter, til at returnere. Returnerede `peek_token` ikke et **Semicolon** første gang, returneres *current_command*.

■

expression

Formål: `At parse en expression_t`
Lovlige tokens: `identifier Int Mon Tue Wed Thu Fri Sat Sun Syn Ack Urg Fin Psh Rst Tcp Udp Icmp Apos any`
Returtype: `expression_t`

Beskrivelse:
Returværdien af `peek_token` matches mod de lovlige tokens for denne funktion. Hvis der læses et af de lovlige tokens, returneres en instans af typen `expression_t`, med en, til det fundne token, svarende konstruktor. Dette er herunder beskrevet for hver af de lovlige tokens.

Var `peek_token` en identifier, kaldes `accept` med denne som parameter, og `VarNameExpr` konstruktoren genererer, med identifieren som parameter, den værdi funktionen returnerer.

Returnerede `peek_token` en token svarende til en af ugedagene, bruges denne som parameter til `accept`. Konstruktoren `WeekdayExpr` genererer, med den fundne token som parameter, returværdien.

Var `peek_token` et **Apos**, er der tale om et interface udtryk. `accept` kaldes med **Apos** som parameter, hvorefter `peek_token` kaldes, for at teste at der følger en identifier og gemme denne. `accept` kaldes med det fundne indetifier og **Apos** som parametre. Til sidst anvendes `IfaceExpr` konstruktoren med den fundne identifier som parameter, til at generere en returværdi.

Kom en af de tokens der svarer til en protokol, altså **Tcp**, **Udp** eller **Icmp**, er der tale om et protokol udtryk. `proto` kaldes, denne returnerer en instans af typen `proto_t`. Konstruktoren `ProtoExpr` bruges med returværdien fra `proto`, til at generere en returværdi.

Var `peek_token` tokenet **Any**, kaldes `accept` med **Any** som paremter. Herefter bruges konstruktoren AnyExpr til at generere en returværdi.

Returnerede `peek_token` et heltal, kan det udtryk der skal parses både være et tid, port eller IP udtryk. For at undersøge dette nærmere, kaldes `peek` med tallet 2 som parameter, hvilket medfører at der returneres et lexem to trin henne i lexemlisten. Hvis der her er tale om et

Colon, kaldes `time` hvis returværdi gemmes. Derefter kaldes `accept` med Dash og `time` kaldes endnu engang og returværdien brugens til konstruktoren `TimeExpr`, sammen med den anden returværdi.

Hvis der ikke blev returneret en **Colon** af `peek` kaldet, må der være tale om enten et port- eller IP udtryk. Da både port- og IP udtryk starter med en mrange, kaldes funktionen `mrange`, der returnerer det læste mrange. Herpå kaldes `peek_token`, for at se om det næste token er et **Dot**, i hvilket tilfælde der vil være tale om et IP udtryk. Er der tale om et port udtryk, anvendes konstruktoren `PortExpr` med det returnerede mrange som parameter, til at generere en returværdi. Fandt `peek_token` en **Dot**, betyder det at der er tale om et ip udtryk, og `accept` kaldes med **Dot** som parameter. For at finde de tre resterende mranges, som IP udtrykket består af, kaldes `mrange` tre gange mere, dog med `accept` kald med **Dot** som parameter, mellem dem. Alle mranges bliver gemt undervejs. Efter det fjerde mrange er fundet, kaldes `peek_token`, for at se om det følgende token er en **Slash**, da der da ville være tale om en maske på IPen. Er dette tilfældet, kaldes `ipmask` med de fire fundne mranges som parametre. Returværdien fra `ipmask` anvendes som parameter til konstruktoren `IpExpr`, hvilket returneres. Hvis IP udtrykket ikke havde en maske, får `IpExpr` de fire fundne mranges som parametre, og genererer derved returværdien.

■

mrange

Formål: `At parse en mrange`
Lovlige tokens: `Int Star`
Returtype: mrange_t

Beskrivelse:
Hvis `peek_token` er **Star**, kaldes `accept` med paremetren **Star** og der returneres en instans af mrange_t, med konstruktoren `FullRange`.

Hvis *token* er **Int**, kaldes `accept` med **Int** som parameter. Hvis `peek_token` ikke er et **Dash** nu, returneres det fundne heltal som en instans af mrange_t, oprettet med konstruktoren `Singleton`. Hvis `peek_token` var et **Dash** anden gang, kaldes `accept` med **Dash** som parameter. Da skal `peek_token` være **Int**, ellers er der syntaksfejl. `accept` kaldes med **Int** som parameter, og der returneres nu en instans af mrange_t indeholdende, en range_t bestående af de to fundne heltal.

■

ipmask

Formål: `At konvertere en IP af singletons til en IP af mranges via IPens mask`
Parametre: ip:ip_t
Returtype: ip_t

Beskrivelse:
For at lave en IP om til en range vha. masker, undersøges først om de enkelte tal i IPen er af typen mrange_t, alle oprettet med `Singleton`, hvis ikke er der systemfejl, da `ipmask` ikke burde være kaldt.

Masken beskriver hvor mange bits, af den samlede adresse, der ligger fast for et specifikt net. Er masken 16 ligger de to første tal i IPen for eksempel fast, mens de sidste to kan have alle værdier mellem 0 og 255, dvs. de skal oprettes med konstruktoren `FullRange` (*).

Mere besværligt bliver det når antallet af bits i masken ikke svarer til et helt antal bytes. Da skal et enkelt tal i IPen konverteres til `mrange_t`, denne oprettes med `mrange_of_singleton`, mens resten af tallene enten skal være `Singleton` eller `FullRange`.

Først kaldes `accept` med **Slash**, derefter matches på om det næste token er **int**, som gemmes i *mask* og `accept`eres, hvis ikke er der syntaksfejl.

Hvis $mask = 0$ returneres en IP, hvor alle fire tal er `FullRange`.

Hvis $0 < mask \leq 8$ returneres en IP, hvor de sidste tre tal er `FullRange` og det første tal fåes ved et kald til `mrange_of_singleton`, med første tal i IPen og masken som parametre.

Hvis $8 < mask \leq 16$ returneres en IP, hvor de sidste to tal er `FullRange` og det første tal er det samme som fra den indkommende IP. Tal nummer to fåes ved et kald til `mrange_of_singleton` med andet tal fra den indkommende IP og masken som parametre.

Hvis $16 < mask \leq 24$ returneres igen IP, hvor der er det sidste tal der erstattes med `FullRange` og de første to tal er magen til de første to tal i den indkommende IP. Det tredie tal fåes via `mrange_of_singleton` med parametrene analog til ovenfor

Hvis $24 < mask \leq 32$ da bliver den returnerede IP dannet, via de tre første tal fra den indkommede IP og det sidste tal via et kald til `mrange_of_singleton` med det sidste tal fra IPen og masken som parametre. Hvis ikke $0 \leq mask \leq 32$, er der syntaksfejl.

■

mrange_of_singleton

Formål: `At konvertere en maske og et tal mrange til en mrange`
Parametre: `mrange:mrange_t, lenmask:int`
Returtype: `mrange_t`

Beskrivelse:
Hvis maskelængden er et multiplum af 8, vil netværksadressen være hele `Singleton`. Derfor returneres blot denne.

Hvis ikke maskelængden er et multiplum af 8, regnes først den egentlige bitmaske for netværksadressen udfra den opgivne længde. Masken udregnes via nedenstående ligning: (<< er logisk shift left og & er logisk "og")

$$(2^{lenmask} - 1) << (8 - lenmask)$$

Det er her vigtigt at sørge for at længden er mellem 1 og 8. Da længden fra input ligger mellem 1 og 32, er det nødvendigt at sørge for at dette foldes om, så den ligger indenfor dette område. Dette gøres ved at bruge følgende ligning:

$$((lenmask - 1) \bmod 8) + 1$$

Efter at have fundet masken bruges denne til at regne de egentlige netværksadresse ved at tage logisk "og" af masken og heltalsværdien. Netværksadressen returneres som den nedre grænse for intervallet.

For at finde den øvre grænse for intervallet, lægges den største værdi for maskinadressen til netværksadressen. Den største værdi for maskinadressen vil være en negering af masken.

■

single_command

Formål: At parse en single_command_t
Lovlige tokens: CurlyLeft identifier If With Sip Dip Sport Dport Input Output Proto Time Day Not
Returtype: command_t

Beskrivelse:
For at parse en single_command, kaldes peek_token først, og denne matches på de lovlige tokens.

Hvis peek_token indeholder et **CurlyLeft** token, kaldes accept med den som parameter. Derpå gemmes returværdien af et funktionskald til command i *cur_command*. accept kaldes med parametren **CurlyRight**, og *cur_command* returneres.

Returnerede peek_token en identifier, kaldes accept med denne og tokenet **Left**. Da der nu er tale om et funktionskald, kaldes funktionen parseq, for at samle alle parametre i en liste. Til sidst bruges konstruktoren CallCommand, med det læste identifier og returnerede liste af parametre som parametre, til at generere returværdien.

Fandt peek_token et **If** token, kaldes accept med dette token som parameter. Herpå kaldes rule funktionen, for at returnere og gemme betingelsen i if-then-else, strukturen i den lokale variabel *current_rule*. Nu kaldes accept med parametren **Then**, hvorefter single_command kaldes for at finde den første af de to kommandoer der er i if-then-else strukturen, denne gemmes i *then_cmd*. Nu kaldes accept igen, denne gang med parametren **Else**. Endnu en gang kaldes single_command, for at finde den sidste af de to kommandoer i if-then-else strukturen, denne gemmes i *else_cmd*. Herpå returneres der ved hjælp af konstruktoren IfCommand, med *current_rule*, *then_cmd* og lokvarelse_cmd som parametre.

Hvis peek_token returnerede *With*, oprettes og returneres en instans af command_t, med konstruktoren WithCommand, der skal bruge to parametre. Første parameter findes ved et funktionskald til funktionen rule, inden dette kald skal accept dog kaldes med parametren **With**. rule kaldet returnerer en instans at typen rule_t, se appendiks E.2 på side 93, denne gemmes midlertidigt i en lokal variabel, kaldet *current_rule*. Herpå kaldes accept med parametren **Do**. Derefter findes den anden parameter ved et kald til singel_command, denne gemmes i *cur_single_command*. Nu returneres ved konstruktoren WithCommand med de to gemte parametre.

Returnerede peek_token **Sip**, **Dip**, **Sport**, **Dport**, **Input**, **Output**, **Proto**, **Time**, **Day** eller **Not**, kaldes rule og returværdien gemmes i *current_rule*. Herpå kaldes accept med parametren **Rule**, for at acceptere den pil der indgår i reglerne der laves i MFL. Herpå kaldes funktionen action og returværdien gemmes i *cur_action*. Nu returneres med konstruktoren RuleCommand der får *current_rule* og *cur_action* som parametre.

■

rule

Formål: At parse en række regler kombineret med binære operatorer
Lovlige tokens: Sip Dip Sport Dport Input Output Proto Time Day Not
Returtype: rule_t
Beskrivelse:
rule funktionen kalder først peek_token, hvis der ikke returneres en af de lovlige tokens kastes en syntaksfejl.

Returnerede peek_token et af de lovlige tokens, tilskrives værdien af et funktionskald til

`single_rule` den lokale variabel *current_rule*. Nu kaldes `peek_token` for at tjekke om der efter reglen følger nogle binære operatorer eller en af de fra starten lovlige tokens. Følger der en af de binære operatorer **Or**, **Xor**, **And** eller **Nand**, kaldes `accept` med det læste token som parameter, hvorefter der foretages endnu et kald til `single_rule`, denne gemmes i *cur_rule*. Herpå returneres med konstruktoren `CombRule`, der tager *current_rule* den læste binære operator og *cur_rule* som parametre. Læser `peek_token` ikke en binær operator, men en af de fra starten lovlige tokens, kaldes `accept` ikke, men ellers fortsættes som om der var læst et **And** token. Læser `peek_token` derimod hverken en binær operator eller en af de fra starten lovlige tokens, anden gang den bliver kaldt, returneres *current_rule* bare.

■

single_rule

Formål: `At parse en single_rule eller blok af single_rules`
Lovlige tokens: `Sip Dip Sport Dport Input Output Proto Time Day Not`
Returtype: rule_t

Beskrivelse:
`peek_token` kaldes og afhængig af returværdien anvendes en tilsvarende konstruktor, til at oprette en variabel af typen rule_t, se appendiks E.2 på side 93, og denne returneres. `accept` kaldes i hvert tilfælde, med det læste token som parameter. Konstruktorerne skal hver have én parameter (**Not** skal have to). Denne parameter fås altid ved et kald til en funktion, og de bliver midlertidigt gemt i lokale variable. De forskellige muligheder kan ses i nedenstående tabel.

Token	Konstruktor	Parameter til konstruktor
Sip	SipRule	ip_rule
Dip	DipRule	ip_rule
Sport	SportRule	port_rule
Dport	DportRule	port_rule
Input	InputRule	interface_rule
Output	OutputRule	interface_rule
Proto	ProtoRule	proto_rule
Time	TimeRule	time_rule
Day	DayRule	ip_rule
Not	UnaryOpRule	Not og single_rule

Not er lidt speciel, idet den kalder `single_rule` rekursivt og at dens rule_t konstruktor skal to parameter. En af typen unaryoperator_t og en af typen rule_t.

■

port_rule

Formål: `At parse en port_rule`
Lovlige tokens: `Int identifier Gt GtE Lt LtE Star`
Returtype: portrule_t
Beskrivelse:
På baggrund af `peek_token`, bestemmes det hvilken konstruktor der skal bruges.

Returnerede `peek_token` et **Int** eller **Star** kaldes `mrange` og returværdien bruges i konstruktoren: `ValuePort`.

Er det returnerede token ikke et **Int**, men en identifier, bruges denne identifier som parame-

ter til konstruktoren `VNamePort`. Dette producerer en variabel af typen portrule_t, der derpå returneres efter `accept` er kaldt.

Skulle det første token være **Gt**, **GtE**, **Lt** eller **LtE**, kaldes `accept`, og `peek_token` kaldes. Returneres et **Int**, `accept`eres denne og konstruktoren `OpValuePort` anvendes, med den gemte operator som første parameter og den nyligt læste **Int** som anden parameter. Dette producerer et output af typen portrule_t, der kan returneres. Returneres der derimod en identifier `accept`eres denne, og der anvendes konstruktoren `OpVNamePort`, med den gemte operator og den læste identifier som parametre, til at generere et output der kan returneres.

■

proto_rule

Formål: `At parse en proto_rule`
Lovlige tokens: `identifier Tcp Udp Icmp`
Returtype: protorule_t liste
Beskrivelse:
`peek_token` kaldes og med mindre der er syntaksfejl, vil den indeholde et token svarende til en protokol eller en **identifier**.

`peek_token` matches dog på Tkeyword, selv om det ikke udelukker at der er fundet et ulovligt token som **Main**. Blev der fundet noget af typen Tkeyword, bliver `proto` funktionen kaldt, hvori det syntaktiske tjek bliver foretaget. Returværdien fra `proto` gemmes i *cur_proto*, hvorefter `peek_token` kaldes, for at tjekke om der følger et **Comma**. Fulgte der et **Comma**, kaldes `accept` med denne som parameter, hvorefter `proto_rule` kaldes og gemmes i *cur_proto_rule*. Herpå returneres konstruktoren `ValueProto` med *cur_proto* som parameter, alene i en liste konkateneret med *cur_proto_rule*. Blev der ikke fundet et **Comma** anden gang `peek_token` blev kaldt, returneres konstruktoren `ValueProto` med *cur_proto* som parameter, i en liste som eneste element.

Returnerede det første `peek_token` kald en identifier, kaldes `accept` med denne identifier som parameter. Herefter undersøges der, med `peek_token`, om der følger et **Comma** token. Er dette tilfældet, kaldes `accept` med **Comma** som parameter. Herefter kaldes `proto_rule` og returværdien gemmes i *cur_proto_rule*. Konstruktoren `VNameProto` anvendes med den fundne identifier som parameter, til at generere en instans af typen protorule_t, der bruges som eneste element i en liste. Listen konkateneres med *cur_proto_rule* og returneres. Blev der ikke fundet et **Comma**, anvendes konstruktoren `VNameProto` med den fundne identifier som parameter, til at generere en instans af typen protorule_t, der bruges som eneste element i en liste. Listen returneres.

■

time

Formål: `At parse en time_t`
Lovlige tokens: `Int`
Returtype: time_t
Beskrivelse:
Indeholdet `peek_token` en **Int**, kaldes `accept` med parametrene **Int** og **Colon**. `peek_token` kaldes for at tjekke at det følgende token er et **Int**, denne gemmes i *cur_int*. En record bestående af to elementer, den første kaldet hour tilskrives værdien af det først læste heltal, mens det andet, kaldet minute, tilskrives værdien *cur_int*.

■

time_rule

Formål: At parse en time_rule_t
Lovlige tokens: identifier Int
Returtype: time_rule_t
Beskrivelse:
peek_token kaldes, og returværdien matches på identifier og **Int**.

Indeholdet peek_token en **Int**, kaldes time og returværdien gemmes i *cur_time*1. accept kaldes på Dash og time kaldes endnu engang, og returværdien gemmes

De to returværdier bruges som parameter til konstruktoren ValueTime, dette returneres.

Skulle det være tilfældet at peek_token ikke indeholdt tokenet **Int** fra starten af, men en **identifier**, bruges denne **identifier** som parameter til konstruktoren VNameTime, der genererer et output af typen time_rule_t. Efter et kald til accept med identifieren som parameter, kan konstruktorens output returneres.

■

interface_rule

Formål: At parse en interface_rule_t
Lovlige tokens: identifier Apos
Returtype: type_t
Beskrivelse:
Først kaldes peek_token, hvor returværdien matches på **Apos** og **identifier**. Hvis ingen af disse to findes, kastes en syntaksfejl.

Læses **Apos** kaldes accept, med **Apos** som parameter. Hvorefter peek_token kaldes igen, med henblik på at tjekke at næste token er en identifier, og gemme denne i den lokale variabel *current_str*. Herpå kaldes accept igen med parametren **Apos**. Nu returneres konstruktoren ValueIface, kaldt med parametren *current_str*.

Blev en identifier i stedet returneret af peek_token, kaldes accept, med denne identifier som parameter. Herefter returneres konstruktoren VNameIface, kaldt med den læste identifier som parameter.

■

day_rule

Formål: At parse en day_rule_t
Lovlige tokens: identifier Mon Tue Wed Thu Fri Sat Sun
Returtype: type_t
Beskrivelse:
peek_token kaldes, og matches med de lovlige tokens.

Returnerer peek_token et token svarende til en af ugedagene, kaldes accept med det læste token som parameter. Derefter returneres konstruktoren ValueDay, med det læste token som parameter.

Returnerede peek_token i stedet en identifier, kaldes accept med den læste identifier som parameter. Derefter returneres konstruktoren VNameDay, med det læste identifier som parameter.

■

action

Formål: `At parse en action_t`
Lovlige tokens: `Permit Deny Log`
Returtype: `action_t`
Beskrivelse:
Først kaldes `peek_token`, for at matche returværdien på de tre lovlige tokens.

Returnerer `peek_token` et **Permit**, kaldes `accept` med det læste **Permit** som parameter. Herefter skal der testes om tokenet **State** følger, hvilket gøres med `peek_token`, hvis dette er tilfældet kaldes `accept` derefter med **State** som parameter. Ligeledes testes derefter om **Log** følger, hvorefter `accept` kaldes med **Log** som parameter hvis den var det næste token. Afhængig af hvilke af disse tokens der blev fundet, returneres Permit, PermitState, PermitStateLog eller PermitLog.

Blev Deny returneret af `peek_token`, kaldes `accept` med det læste **Deny** som parameter. Herefter skal der testes om tokenet **Log** følger, hvilket gøres med `peek_token`, hvis dette er tilfældet kaldes `accept` derefter med **Log** som parameter. Nu returneres DenyLog. Blev **Log** ikke fundet ved andet kald til `peek_token`, returneres Deny.

Returnerede `peek_token` **Log** fra starten af, kaldes `accept`, med det læste **Log**. Herefter returneres Log.

■

proto

Formål: `At parse en proto_t`
Lovlige tokens: `Udp Icmp Tcp`
Returtype: `proto_t`
Beskrivelse:
Først kaldes `peek_token` og returværdien matches på de tre lovlige tokens.

Returneres **Udp** af `peek_token` kaldet, kaldes `accept` med **Udp** som parameter. Herefter returneres Udp.

Returnerede `peek_token` **Icmp**, kaldes `accept` med **Icmp** som parameter. Herefter skal det testes om der følger en liste af tal, derfor kaldes `peek_token` for at se om der følger et **SquareLeft**, hvis det er tilfældet kaldes `accept` med parametren **SquareLeft**. Herefter læses listen af **Int** tokens. Ved at læse en **Int** med `peek_token` gemme den i listen og kalde `accept` med det læste **Int** som parameter. Efter en **Int** er læst ind i listen, kaldes `peek_token` for at se om det næste token er et **Comma**, er dette tilfældet gentages søgningen efter **Int** tokens og listen udvides således. Når der ikke læses et **Comma** kaldes `accept` med parametren **SquareRight**, da listen jo skal afsluttes af en kantet parentes. Uanset hvor lang listen er, returneres konstruktoren `Icmp` med listen som parameter.

Returnerede `peek_token` et **Tcp** token, kaldes `accept` med **Tcp** som parameter. Herefter skal det testes om der følger en liste af flag, derfor kaldes `peek_token` for at se om der følger et **SquareLeft**, hvis det er tilfældet kaldes `accept` med parametren **SquareLeft**. For at samle listen af flag, kaldes `peek_token`, for at se hvilket flag der kommer, denne tilføjes en liste og anvendes så som parameter i et kald til `accept`. Herefter kaldes `peek_token` igen, for at se om der følger et **Comma**, er dette tilfældet anvendes den som parameter til `accept`. Blev der ikke fundet et **Comma**, kaldes `accept` med **SquareRight** som parameter. Hver gang der finde et **Comma**, gentages søgningen efter flag, og listen udvides således indtil der findes et **SquareRight**. Uanset

hvor lang listen er, returneres konstruktoren `Tcp` med listen som parameter.

■

ip_rule

Formål: At parse en ip_rule_t
Lovlige tokens: identifier Int
Returtype: type_t
Beskrivelse:
Først kaldes `peek_token` og matches på de to lovlige tokens.

Læses en identifier, bruges denne som parameter, først til `accept` og derefter til konstruktoren `VNameIp`, der returneres.

Læses et **Int** token, kaldes funktionen `ip`. Returværdien fra `ip` kaldet anvendes som parameter til konstruktoren `ValueIp`, der returneres.

■

ip

Formål: At parse en IP
Lovlige tokens: Int
Returtype: ip_t
Beskrivelse:
Først kaldes `peek_token`, for at kontrollere at det indkommende token er **Int**. Returværdien at et kald til `mrange` gemmes i en lokal variabel

Derefter kaldes `accept` med **Dot** og endnu en returværdi fra `peek_token` kontroleres til af være **Int**. Returværdien at et kald til `mrange` gemmes i en lokal variabel Dette gøres to gange mere, så man således har 4 mrange_t.

Disse 4 bruges til at lave en ip_t, hvis det næste token, fra `peek_token` er en **Slash**, kaldes `ipmask` med den dannede IP og returværdien returnes. Hvis der ikke er en **Slash** returneres den dannede IP direkte.

■

APPENDIKS G

Semantisk Analyse

Formålet med dette appendiks er at give en gennemgang af funktionerne i semantisk analyse.

Formålet med semantisk analyse er, som beskrevet i afsnit 6.5 på side 51, at sikre at kildekoden er semantisk korrekt.

I MFLkompileren omfatter semantisk analyse tre hovedområder: Identifikation af konstanter og filtre, detektion af cirkulære referencer for filterkald og konstanterklæringer, samt typecheck af udtryk.

Først identificeres konstanter, det vil sige der opbygges en identifikationstabel for globale konstanter. Derefter opbygges en tilsvarende tabel for filterdefinitioner.

Derpå opbygges først en tabel af dekorerede konstanter, hvor der undersøges for cirkulære referencer, samt laves typecheck. Derefter opbygges en tabel af dekorerede filtre, hvor der laves typecheck, samt undersøges for cirkulære filterkald.

Tilsidst opbygges et dekoreret main filter, som sammen med den dekorerede konstanttabel og den dekorerede filtertabel skal bruges som input til kodegenereringen.

I det efterfølgende afsnit beskrives de funktioner, der anvendes i semantisk analyse. Under denne gennemgang er følgende beskrevet.

- Funktionens navn.
- Funktionens formål.
- Parametre, udelades hvis der ikke er parametre.
- Returtype
- Beskrivelse af funktionen.

I funktionsgennemgangen skrives et funktionsnavn på formen `funktions_navn`, lokale variable skrives på formen *lokal_variabel*, globale variable skrives som *global_variabel* og typer skrives enten som `ast_entype_t` eller `deco_entype_t`.

G.1 Funktioner i semantisk analyse

I det følgende beskrives de vigtigste funktioner i semantisk analyse.

`semantik_analyzer`

Formål: `At koordinere opbygningen af outputtet af den semantiske analyse.`
Parametre: `mfl:ast_mfl_t`
Returtype: `deco_mfl_t`
Beskrivelse:
`semantik_analyzer` tager et AST som input, og fungerer på følgende måde. Først erklæres en variabel *defines* til at være lig med første element i ASTet som er af typen `ast_defines_t`. Dermed har man en liste af filterdefinitioner og konstantdefinitioner. Derudover erklæres en variabel

mainfilter til at være lig med det andet element i ASTet, som er af typen ast_mainfilter_t, og som indeholder main filtret. Derefter kaldes funktionen `build_all_constants` som opbygger en hashtabel af dekorerede konstanter. Derefter kaldes funktionen `build_all_filtres`, som opbygger en hashtabel af dekorerede filtre og til sidst erklæres en variabel *dmain*, som sættes lig returværdien fra et kald til `build_mainfilter`, som opbygger et dekoreret main filter. Der returneres så en tupel som består af den dekorerede konstant hashtabel, den dekorerede filter hashtabel og det dekorerede main filter.

■

build_all_filters

Formål: `At tage en liste af filtre og for hver af dem kalde build_filter.`
Parametre: `filtHash:Hashtable med filtre`
Beskrivelse:
Listen der indeholder filterdefinitioner gennemløbes, hvorefter der for hvert filter kaldes metoden `build_filter` som sørger for at opbygge et semantisk korrekt dekoreret filter og tilføje det til en hashtabel.

■

build_all_constants

Formål: `At tage en liste af konstanter og for hver af dem kalde` *build_constant*.
Parametre: `constantHash:Hashtable med konstanter`
Beskrivelse:
Listen der indeholder konstanterklæringer gennemløbes og for hver konstant kaldes metoden `built_konstant` som sørger for at opbygge en semantisk korrekt dekoreret konstant og tilføje den til en hashtabel.

■

build_constant

Formål: `At opbygge en semantisk korrekt dekoreret konstant.`
Parametre: `name:string,const:en konstant fra` *constHash*
Beskrivelse:
Der tages en ast_constdecl_t som input og ud fra de tre forskellige elementer som den indeholder opbygges tilsvarende dekorerede elementer som samtidig checkes for semantisk korrekthed. De tre elementer er, type, navn samt det udtryk konstanten er assignet til. Der oprettes en variabel *dtype*, som sættes lig den dekorerede type, der tilsvarer konstantens type. Navnet skal der ikke gøres noget ved. Der erklæres en variabel *dexpr* som sættes lig det dekorerede udtryk, som findes ved et kald til *find_deco_expression*. Når alle dele af konstanten er blevet dekorerede og checket for semantisk korrekthed så tilføjes den dekorerede konstant, indeholdende *dtype*, navn og *dexpr*, til en hashtabel.

■

build_filter

Formål: `At opbygge et semantisk korrekt dekoreret filter.`
Parametre: `name:string,filter:et filter fra` *filterHash*
Beskrivelse:

Der tages en ast_filterdef_t som input og ud fra de tre forskellige elementer som den indeholder opbygges tilsvarende dekorerede elementer som samtidig checkes for semantisk korrekthed. De tre elementer er navn, ast_pardecl_t, og ast_command_t. Navn gøres der ikke noget ved. Der erklæres en *dpardecl* som sættes lig det den dekorerede ast_pardecl_t. ast_pardecl_t bruges også til at opbygge en liste af lokale konstanterklæringer, som skal bruges i forbindelse med dekoreringen af ast_command_t. Så oprettes en *dcommand*, som sættes lig den dekorerede ast_command_t. ast_command_t checkes for semantisk korrekthed og dekoreres ved at kalde funktionen `find_deco_expression`. Når alle dele af filtret er blevet dekorerede og checket for semantisk korrekthed, så tilføjes det dekorerede filter indeholdende navn, *dpardecl* og *dcommand* til en hashtabel.

■

`find_deco_command`

Formål: `At opbygge en semantisk korrekt deco_command_t.`
Parametre: `command:ast_command_t`
Returtype: `deco_command_t`
Beskrivelse:
Der tages en ast_command_t som input, udfra hvilken der opbygges en semantisk korrekt deco_command_t. Funktionen gøres rekursiv, da en ast_command_t kan indeholde yderligere ast_command_t. For hver slags ast_command_t kaldes funktioner, der finder en dekoreret udgave af de elementer de indholder, herunder også `find_deco_command` selv. Ved en call command skal der undersøges, om de aktuelle parametre i kaldet passer med de formelle parametre i det filter der er blevet kaldt, hvilket gøres med et kald til funktionen `check_formal_param`. Derudover skal der også checkes, om der opstår cirkulære filterkald, hvilket gøres med et kald til funktionen `check_filter_cirk`. En if command opbygges ved et kald til `find_deco_rule` og to kald til `find_deco_command`. En with command opbygges ved et kald til `find_deco_rule`, samt et kald til `find_deco_command`. En rule command opbygges ved et kald til `find_deco_rule` og et kald til `find_deco_action`. Til sidst er der sequential command, der opbygges ved to kald til `find_deco_command`.

■

`check_formal_param`

Formål: `At checke om de aktuelle parametre i et filterkald passer med de formelle parametre i det filter der bliver kaldt.`
Parametre: `callName:string,exprList: ast_expression_t liste`
Returtype: `bool`
Beskrivelse:
Der tages navnet på det kaldte filter, samt listen af de aktuelle parametre i form af en liste af ast_expression_t som input. Så findes det kaldte filter og dets parameterdeklaration sammenholdes med listen af ast_expression_t. Der undersøges først, om længden af parameterdeklarationer er lig længden af listen med udtryk. Hvis det er tilfældet så gennemløbes listen af udtryk, og for hver ast_expression_t undersøges der om det er et variabelnavn udtryk. Hvis det er tilfældet så undersøges der, om der findes enten en global eller lokal konstant med det pågældende navn, og om en sådan konstant har samme type, som det parameter der er på samme plads i paramterdeklaratioen, som variabelnavn udtrykket har i listen af udtryk. Hvis det ikke er et variabelnavn udtryk så undersøges der, om det parameter der står på den tilsvarende plads i parameterdeklarationen har en type, der tilsvarer det pågældende udtryk. Derudover tilføjes hvert udtryk også til en liste af udtryk som tilsidst omdannes til en liste

af dekorerede udtryk, ved for hvert udtryk at kalde `find_deco_filt_expression`. Tilsidst returneres sandt eller falsk, alt efter om det forløb som det skulle, samt listen af dekorerede udtryk.

■

check_filter_cirk

Formål: `At checke at der ikke forekommer cirkulære filterkald`
Parametre: `name:string`
Returtype: `bool`
Beskrivelse:
Funktionen tager navnet på det kaldte filter som input, og den er rekursiv, da det kaldte filter kan indeholde yderligere filterkald som også kalder funktionen. Først tilføjes navnet på det kaldte filter til en hashtabel *cirkFilterHash*, dog først efter at der er blevet undersøgt om det kaldte filter i forvejen er i den. Dernæst findes det kaldte filter, og derefter undersøges det fundne filters `ast_command_t` for callCommands. Hver callCommand tilføjes til en liste, og for hver callCommand i den liste kaldes `check_filter_cirk`. Til sidst fjernes det kaldte filter fra *cirkFilterHash*, da evt. cirkularitet der inkludere det kaldte filter vil have forårsaget en fejl på det tidspunkt, og det derfor skal være muligt at kalde filtret igen, uden at der meldes fejl. Der returneres true, hvis der ikke blev fundet nogen fejl.

■

find_deco_expression

Formål: `At opbygge en semantisk korrekt deco_expression_t.`
Parametre: `expr:ast_expression_t`
Returtype: `deco_expression_t`
Beskrivelse:
Funktionen tager en `ast_expression_t` som input, og ud fra den opbygges en semantisk korrekt `deco_expression_t`. I forbindelse med at udtrykket laves om til et dekoreret udtryk, skal det checkes for semantisk korrekthed. Det gøres ved at undersøge, hvilken slags udtryk der er tale om. Hvis det er et variabelnavn udtryk, så skal der checkes, at der findes en konstant med det navn som udtrykket indeholder, samt at der ikke opstår cirkulære konstant erklæringer. Ved port og IP udtryk skal der checkes, om de er inden for det tilladte interval. Any udtryk skal laves om, hvis konstanten har en port eller IP type, skal det laves om til henholdsvis et IP udtryk med maksimal interval eller et port udtryk med maksimal interval. Derudover skal et dekoreret variabelnavn og any udtryk også indeholde den dekorerede type af konstanten, da det er nødvendig for kodegenereringen. Funktionen `find_deco_filt_expression` er magen til pånær, at den ikke checker for om et variabelnavn udtryk forårsager en cirkulær konstant erklæring, da dette allerede er undersøgt for alle konstanter.

■

find_deco_rule

Formål: `At opbygge en semantisk korrekt deco_rule_t.`
Parametre: `rule:ast_rule_t`
Returtype: `deco_expression_t`
Beskrivelse:
Funktionen tager en `ast_rule_t` som input, og ud fra den opbygger den en semantisk korrekt `deco_rule_t`. Den er rekursiv da en regel kan bestå af andre regler. Der er en række forskellige

regler, og der findes ud af hvilken regel der gives som input, hvorefter der kaldes funktioner der finder den dekorerede udgave af de elementer reglen består af. Alle slags regler kan bestå af et variabelnavn, og der skal så undersøges om der findes en global eller lokal konstant med sammen navn som variabelnavnet, samt om den konstants type passer til den pågældende regel. Ved en timeRule checkes om det er en gyldig tid, og ved IP og port checkes om det er et gyldig interval.

■

check_const_cirk

Formål: At checke for cirkulære konstanterklæringer.
Parametre: name:string,originType:deco_type_t
Returtype: bool
Beskrivelse:
Funktionen tager navnet på den konstant, som skal undersøges for om den forårsager en cirkulær fejl, samt typen af den konstant som startede kæden af konstant erklæringer. Først findes den konstant med samme navn som input navnet, hvorefter der undersøges om den konstant indeholder et variabelnavn udtryk, hvis den gør det så kalder funktionen sig selv og tilføjer den fundne konstant til en hashtabel $cirkConstHash$. Hvis den fundne konstant ikke indeholder et variabelnavn så undersøges om det udtryk den indeholder er af samme type som typen af den konstant der startede kæden af konstanterklæringer. Til sidst returneres true eller false.

■

find_deco_mrange_ip

Formål: At checke en ast_mrange_t, og lave den om til en deco_mrange_t.
Parametre: ast_mrange_t
Returtype: deco_mrange_t
Beskrivelse:
Der undersøges om det er en Range, Singleton eller FullRange, og checkes om værdien er inden for den gyldige interval for en IP. Der er en tilsvarende funktion for port, som checker om værdien er inden for det gyldige interval for port.

■

build_mainfilter

Formål: At opbygge en deco_mainfilter_t.
Parametre: main:ast_mainfilter_t
Returtype: deco_mainfilter_t
Beskrivelse:
Der erklæres en variabel *daction*, der sættes lig det dekorerede action som er blevet fundet ud fra det action som main filtret indeholder. Derudover erklæres en variabel *dcommand*, som sættes lig et kald til funktionen `find_deco_command`. Til sidst returneres en tupel bestående af *daction* og *dcommand*.

■

APPENDIKS **H**

Kodegenerering

I det følgende appendiks beskrives kodegenerering i MFL kompileren. Herunder beskrives hvilke funktioner der anvendes, samt hvorledes en MTIDD opbygges fra et dekoreret AST.

Målet med kodegenerering er at omdanne kildekode til target kode. I den konkrete kompiler vil det sige at omdanne fra kompilerens interne repræsentation af MFL koden til en MTIDD.

Efter syntaktisk og semantisk analyse vides med sikkerhed at kildekoden er velformet MFL. I semantisk analyse er der, som beskrevet i appendiks G på side 115, opbygget et dekoreret AST. Dette syntakstræ er opbygget af typer beskrevet i design af grænseflader, afsnit 6.2.3 på side 48 og bruges som parameter i kodegenereringsfunktionerne.

I det følgende afsnit beskrives disse funktioner, startende med den funktion, der starter kodegenerering.

H.1 Funktioner i kodegenerering

Det følgende er beskrivelser af funktioner i kodegenerering. I denne funktionsgennemgang skrives funktionsnavne på formen `funktions_navn`. Lokale variable skrives som *en_variabel* og typer fra OCaml, samt det dekorerede AST skrives som entype_t.

codegen

Parametre: `decotree:mfl_t`
Returtype: `MTIDD`
Beskrivelse:
Funktionen laver et match med *decotree* for at få indholdet af denne. Indeholdt i *decotree* er en 3-tupel, bestående af en identtable_t, en filtertable_t og en mainfilter_t. Disse tre bruges som parametre til funktionen `do_mainfilter` der kaldes. Fra `do_mainfilter` returneres en MTIDD der igen returneres.

■

do_mainfilter

Parametre: `mainfilter: mainfilter_t, identtable: identtable_t, filtertable: filtertable_t`
Returtype: `MTIDD`
Beskrivelse:
Først oprettes endnu en identtable_t der gemmes i *localident_table*. *localident_table* indeholder lokale konstanterklæringer, til forskel fra *identtable* der indeholder globale konstanterklæringer.

Derefter laves et match på *mainfilter* for at få adgang til dens interne action_t og command_t. Den interne action_t gemmes i *action*, den interne command_t gemmes i *command*.

Derpå kaldes funktionen `match_command` med parametrene *command*, *identtable*, *localident_table*, *filtertable*, samt en 3-tupel af IDDer, hvor hver enkelt IDD, er IDD terminalværdien falsk. Den returnerede 3-tupel af IDDer gemmes i *iddTriple*.

Inden generering af MTIDD fra *iddTriple*, skal standardhandling og overlap håndteres. Dette er beskrevet i afsnit 6.6 på side 52 med ligning 6.2 på side 57. Ligningen beskriver de IDD operationer der skal anvendes for at undgå overlap, under hensyntagen til standardhandlingen.

Der laves et match på *action* samt *iddTriple*. Sidstnævnte for at kunne lave operationer på de tre IDDer, herefter kaldet *permit_idd*, *deny_idd* og *log_idd*.

Hvis *action* matcher konstruktoren `Permit` skal overlap undgås ved følgende IDD operationer. På *permit_idd* bruges NOT operationen, den resulterende IDD, konjugeres med *deny_idd*. På IDDen fra denne operation bruges NOT operationen, dette er den nye *permit_idd*.

Hvis *action* i stedet matcher konstruktoren `Deny`, undgås overlap ved følgende IDD operationer. For *deny_idd* bruges NOT operationen. Den resulterende IDD konjugeres *permit_idd*. Resultatet heraf er den nye *permit_idd*.

Derefter kombineres den nye *permit_idd*, samt *deny_idd* og *log_idd* til en MTIDD. Denne MTIDD returneres.

■

`match_command`

Parametre: `command: command_t, it; local_it: identtable_t, filtertable: filtertable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:

Parametret *command* matches mod konstruktorer for `command_t`.

Hvis *command* matches som `SequentialCommand`, da kaldes `do_sequential_command` med *command*s interne `command_t` som parametre. Derudover er *it*, lokvarlocal_it, lokvarfiltertable og tretuplen af IDDer parametre til funktionen. Den returnerede 3-tupel af IDDer returneres.

Matches *command* som en `RuleCommand`, kaldes `do_rule_command` med de i *command* indeholdte `rule_t` og `action_t` som parametre. De resterende parametre er *it*, lokvarlocal_it, lokvarfiltertable og tretuplen af IDDer parametre til funktionen. Den returnerede 3-tupel af IDDer returneres.

Hvis *command* er en `CallCommand` kaldes funktionen `do_callcommand` med *command*s interne `string`, `expression_t list` og `command_t` som parametre. Herudover er *it*, *local_it*, *filtertable* og tretuplen af IDDer som parametre. Den returnerede 3-tupel af IDDer returneres.

Matches *command* som konstruktoren `WithCommand` kaldes `do_with_command` hvor *command*s interne mltypecommand_t er parameter. Derudover er følgende parametre til funktionen: *it*, lokvarlocal_it, lokvarfiltertable og tretuplen af IDDer. Den returnerede 3-tupel returneres.

Hvis *command* matches som `IfCommand` kaldes funktionen `do_if_command`. Parametrene til funktionen er den interne `rule_t`, samt de to interne `command_t` indeholdt i *command*. Derudover er parametrene de samme som beskrevet for tidligere matches. Den returnerede 3-tupel returneres.

■

`do_callcommand`

Parametre: `filtername: String, paramexp: expression_t list, it; local_it: identtable_t, filtertable: filtertable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:
Funktionen kalder `find_filter` med parametrene *filtername* og *filtertable*. Fra `find_filter` returneres en filterdef_t.

Denne filterdef_t er en 3-tupel af filtrets navn, parametre, som en pardecl_t, samt en command_t der repræsenterer filtrets krop. Filtrets parametre gemmes i en *paramdecl*, filtrets krop gemmes i *filtercommand*.

Herpå skal en ny lokal identifikationstabel opbygges, fra aktuelle og formelle parametre, i filterkaldet. Dette gøres ved kald af funktionen `make_identtable`, med parametrene *paramdecl*, *paramexp*, *it* og *local_it*. Her repræsenterer *paramdecl* de formelle parametre og *paramexp* de aktuelle parametre. Den returnerede identtable_t gemmes i den lokale variabel *new_local_it*.

Afsluttende kaldes `match_command` for at evaluere filterkroppen til en IDD. Parametrene til funktionen er *filtercommand*, *it*, *new_local_t*, *filtertable* samt tretuplen af IDDer. Den returnerede 3-tupel af IDDer returneres.

■

do_sequential_command

Parametre: `command_1; command_2: command_t, it; local_it: identtable_t, filtertable: filtertable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:
Funktionen kombinerer to tretupler af IDDer, skabt fra to command_t, indeholdt i en command_t, `SequentialCommand`.

Først kaldes `match_command` med parametrene *command_1*, *it*, *local_t*, *filtertable*, samt tretuplen af IDDer. Funktionen returnerer en tilsvarende 3-tupel af IDDer der gemmes i *idd_1*.

Tilsvarende gøres for *command_2*, den resulterende 3-tupel gemmes i *idd_2*.

Derpå kaldes funktionen `combine_idd_tuples` med *idd_1* og *idd_2* som parametre. Den returnerede 3-tupel returneres.

■

do_if_command

Parametre: `rule: rule_t, command_1; command_2: command_t, it; local_it: identtable_t, filtertable: filtertable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:
Funktionen kalder `do_rule` med parametrene *rule*, *it* og *local_t*, hvorfra der returneres en IDD. Denne IDD udgør betingelsen i if sætningen. Desuden kaldes funktionen `match_command` for *command_1* og *command_2* med de resterende fælles parametre *it*, *local_t*, *filtertable* samt tretuplen med IDDer. Funktionen `match_command` returnerer en 3-tupel med IDDer for hver af de to kommandoer, hvilke svarer til then og else operationerne. Til sidst kaldes funktionen `combine_ITE_idd` med IDDen for betingelsen i if sætningen samt de to tretupler med IDDer for then og else operationerne, hvorfra der returneres en kombineret if then else IDD 3-tupel.

■

do_with_command

Parametre: `rule: rule_t, command: command_t, it; local_it: Identtable_t, filtertable: filtertable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:

Funktionen kalder `do_rule` med parametrene *rule*, *it* og *local_it*, hvorfra der returneres en IDD. Denne IDD udgør betingelsessætningen. Desuden kaldes funktionen `match_command` for *command* med parametrene *it*, *local_it*, *filtertable* samt tretuplen med IDDer. Funktionen `match_command` returnerer en 3-tupel med IDDer, som svarer til do operationen. Til sidst kaldes funktionen `combine_ITE_idd` med IDDen for betingelsen samt tretuplen med IDDer for do operationen og en 3-tupel med IDDer der har terminaler der går til falsk. Ved at anvende denne 3-tupel med "dummy IDDer" som det sidste led i funktionen `combine_ITE_idd`, opnås en do-with funktion, hvorfra der returneres en kombineret do-with IDD 3-tupel.

■

do_rule_command

Parametre: `rule: rule_t, action: action_t, it; local_it: identtable_t, (permit_idd, deny_idd, log_idd): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:

Funktionen kalder den rekursive funktion `do_rule`. Som parameter til funktionen bruges *rule*, *it* og *local_it*. Funktionen `do_rule` evaluerer *rule* til en IDD, der gemmes i den lokale variabel *rule_idd*.

Derpå kaldes funktionen `match_action`, med *rule_idd*, *action* og tretuplen af IDDer. Funktion *match_action* returnerer tillige en 3-tupel af IDDer, som igen returneres.

■

combine_ITE_idd

Parametre: `rule_idd; (permit_idd_1, deny_idd_1, log_idd_1); (permit_idd_2, deny_idd_2, log_idd_2): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:

Funktionen kombinerer to tretupler af IDDer, med en IDD brugt som kontrolstruktur. De operationer der skal anvendes for at kombinere alle disse IDDer er beskrevet i design af kodegenerering, 6.6. Ligningen 6.1 beskriver hvorledes f.eks. *permit_idd_1* og *permit_idd_2* kombineres med *rule_idd*.

permit_idd_1 konjugeres *rule_idd*. For *permit_idd_2* og *rule_idd* bruges AND NOT operationen. De resulterende IDDer lægges sammen med OR operationen.

Således gøres for hver IDD i tretuplerne, hvorefter de nye IDDer returneres i en ny 3-tupel.

■

combine_idd_tuples

Parametre: `(permit_idd_1, deny_idd_1, log_idd_1); (permit_idd_2, deny_idd_2, log_idd_2): IDD`

Returtype: `IDD: (permit_idd, deny_idd, log_idd)`

Beskrivelse:
Funktionen kombinerer to tretupler af IDDer ved at bruge IDD operationen or på IDDerne der er indeholdt i tuplerne. Således bruges OR operationen på den første IDD i hver af tretuplerne. Dette gøres for alle IDDer i tuplerne, derpå laves en ny 3-tupel af de sammenlagte IDDer. Til sidst returneres den nye 3-tupel.

■

make_identtable

Parametre: `paramdecl_l: pardecl_t, paramexp_l: expression_t list, it; old_local_it: identtable_t`
Returtype: `identtable_t`

Beskrivelse:
Funktionen genenerer en ny identifikationstabel fra en liste af pardecl_t og en liste af expression_t.

Der oprettes en hashtabel, kaldet *new_it*.

Derpå kaldes en intern rekursiv funktion med de samme parametre som `make_identtable` tilføjet *new_it*, den nye identifikationstabel. I denne interne funktion laves et match for *paramdecl_l*, hvor startelementet i listen kaldes *param* og resten af listen kaldes *restparam*. I dette match laves et indlejret match på *paramexp_l* hvor startelementet i listen kaldes *expression* og resten af listen kaldes *restexp*. Hvis listen *paramexp_l* er er tom returneres *new_it*

Hvis *paramexp_l* ikke er tom oprettes en ny constdecl_t indeholdende *param* samt indholdet af *expression*. Denne nye constdecl_t indsættes i *new_it* sammen med navnet på parametren, fundet i *param*. Tilsidst kaldes den interne funktion rekursivt.

■

match_action

Parametre: `rule_idd; (permit_idd, deny_idd, log_idd): IDD, action: action_t`
Returtype: `IDD: (permit_idd, deny_idd, log_idd)`
Beskrivelse:
Parameteren *action* matches mod konstruktorer for action_t

Afhængigt af hvilken *action* der matches på, foretages der en opdatering af en eller flere af IDDerne i 3-tuplen med permit, deny og log. Opdateringen foretages ved at bruge IDD operationen iddOr med *rule_idd* og den tilhørende IDD i 3-tuplen.

I de tilfælde hvor der forekommer en action indeholdende state, vil state blive ignoreret, da dette ikke er implementeret.

Der returneres en opdateret IDD 3-tupel fra denne funktion.

■

do_rule

Parametre: `rule: rule_t, it; local_it: identtable_t`
Returtype: `IDD`
Beskrivelse:
Parameteren *rule* matches mod konstruktorer for rule_t.

Hvis *rule* matches som `SipRule`, `DipRule`, `SportRule`, `DportRule`, `InputRule`, `OutputRule`,

`ProtoRule`, `TimeRule` eller `DayRule` kaldes en underliggende funktion med reglen som parameter samt parametrene *it*, *local_it* og *nodenavn*. De underliggende funktioner returnerer hver en IDD med den pågældende regel.

Hvis *rule* matches som `UnaryOpRule` eller `CombRule` kaldes de interne rekursive funktioner henholdsvis `do_unaryop_rule` og `do_comb_rule` med parametrene *it* og *local_it* samt de specifikke parametre for hver at funktionerne. Funktionen `do_unaryop_rule` tager yderligere parametrene *unaryoperator* samt reglen *rule*. Ligeledes tager funktionen `do_comb_rule` parametrene *binaryoperator* samt reglerne *rule_1* og *rule_2*. Reglerne i parametrene er i begge funktioner repræsenteret ved IDDer og der returneres en IDD fra disse.

■

do_unaryop_rule

Parametre: `unaryoperator: unaryoperator_t, rule: rule_t, it; local_it: identtable_t`
Returtype: `IDD`
Beskrivelse:
Der laves et match på operatoren *unaryoperator_t*, hvortil der kun findes et match på nuværende tidspunkt, nemlig `Not`. Den interne rekursive funktion `do_rule` kaldes for den pågældende regel *rule_t* som returnerer en IDD for reglen. Denne IDD bruges som parameter til IDD operationen `Not`, som returnerer en inverteret IDD for reglen.

■

do_comb_rule

Parametre: `rule_1; rule_2: rule_t, binaryoperator: binaryoperator_t, it; local_it: identtable_t`
Returtype: `IDD`
Beskrivelse:
Funktionen kalder den interne rekursive funktion `do_rule` for hver af de to regler, hvorfra der returneres en IDD. Derefter laves et match på *binaryoperator_t* hvorfra den tilsvarende IDD operation kaldes med den definerede IDD order samt de to rule-IDDer. Der returneres en samlet IDD, indeholdende de to givne regler.

■

do_day_rule

Parametre: `dayrule: dayrule_t, it; local_it: identtable_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen konverterer en `dayrule_t` til en tilsvarende IDD.

Først laves et match på *dayrule* mod konstruktorer for `dayrule_t`. Hvis *dayrule* matches med konstruktoren `ValueDay`, indeholder *dayrule* en dag beskrevet ved typen `weekday_t`, denne gemmes i den lokale variabel *weekday*. Derpå kaldes den interne funktion `make_weekday_idd` med *weekday* og *nodename* som parameter. `make_weekday_idd` returnerer en IDD, denne returneres.

Matches *dayrule* i stedet med konstruktoren `VNameDay` er *dayrule* en "reference" til en foruddefineret konstant. Værdien af denne konstant findes ved kald af funktionen `find_expression`

med navnet på konstanten, samt *it* og *local_it* som parameter. Den returnerede expression_t matches derpå mod konstruktorer for expression_t. Hvis der matches med `AnyExpr` returneres terminalen sand. Matches der i stedet med konstruktoren `WeekdayExpr` indeholder den returnerede expression en weekday_t. Denne weekday_t bruges sammen med *nodename* som parameter ved kald af den interne funktion `make_weekday_idd`. Den returnerede IDD returneres.

Funktionen `make_weekday_idd` kalder først en anden intern funktion, `daytoint` med *weekday* som parameter. `daytoint` returnerer en heltalsværdi i intervallet 0 til 6 svarende til dagen. Hvis *weekday* matches med konstruktoren `Mon` returneres 1 osv. Da genererer `make_weekday_idd` en ny IDD hvor den returnerede værdi skal gå til terminalen sand, og alle andre værdier skal gå til falsk. Denne nye IDD returneres.

■

do_time_rule

Parametre: `timerule: timerule_t, it; local_it: identtable_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Til start matches *timerule* mod konstruktorerne for timerule_t. Matches *s*om en `ValueTime` kaldes den interne funktion `gettime` med de, i *timerule* indeholdte time_t, som parameter. Disse parametre udgør det tidsinterval denne regel beskriver. Der returnerede IDD, returneres.

Hvis *dayrule* matches som en `VNameTime` kaldes i stedet funktionen `find_expression` med parametrene *it*, *local_it* og navnet på den konstant som *timerule* refererer til. `find_expression` returnerer en expression_t, kaldet *expression*, der ligeledes skal matches mod sine konstruktorer.

Hvis *expression* er en `TimeExpr` indeholder *expression* to time_t der bruges som parameter ved kald af funktionen `gettime`. Funktionen returnerer en IDD, der igen returneres.

Matches *expression* som `AnyExpr` returneres IDD terminalen indeholdende værdien sand.

`do_day_rule` har to interne funktioner. Den ene, `gettime` anvendes som tidligere beskrevet. Funktionen genererer en IDD, ved at evaluere to time_t til intervaller, der indsættes i IDDen. Til at evaluere de to time_t til heltalsværdier, anvendes den anden interne funktion, `timetoint`, der konverterer en time_t til en heltalsværdi svarende til det antal sekunder fra 00:00 time_t repræsenterer.

■

do_protorule_list

Parametre: `protorulelist: rule_t list, it; local_it: identtable_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen evaluerer en liste af rule_t til en IDD. Hver rule_t svarer til en protokol, f.eks. TCP.

Først oprettes en IDD, hvor alle værdier for protokol, dvs. 0 til 255, går til terminalen falsk. Denne IDD kaldes $fullproto$.

Derpå kaldes den interne, rekursive funktion `do_proto_idd` med *protorulelist*, *it*, *local_t*,

nodename og *fullproto* som parametre. Funktionen returnerer en IDD, der returneres videre.

Den interne funktion `do_proto_idd` laver et match på *rulelist*, det første element i listen gemmes i *protocol*, den resterende liste gemmes i *restlist*. Så kaldes `do_protorule` med *protocol* som parameter. Den returnerede IDD lægges sammen med fullproto ved brug af and-operationen. Denne nye IDD bruges som parameter når `do_proto_idd` kaldes rekusivt.

Når listen er tom returneres *fullproto*.

■

do_protorule

Parametre: `protorule: protorule_t, it; local_it: identtable_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen laver et match på *protorule*. Hvis den matches med konstruktoren `ValueProto` da kaldes `do_protocol` med *protorule*s interne `proto_t`, samt *nodename* som parameter. Den returnerede IDD returneres.

Hvis *protorule* i stedet matches med konstruktoren `VNameProto` kaldes `find_expression` med tekststrengen indeholdt i *protorule*, samt *it* og *local_it* som parametre. Funktionen returnerer en `expression_t`, der gemmes i den lokale variabel *expression*. Protokollen indeholdt i *expression* bruges sammen med *nodename* som parameter til funktionen `do_protocol`, som beskrevet tidligere.

■

do_protocol

Parametre: `protocol: proto_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen matcher *protocol* med konstruktorer for `proto_t`. Matches *protocol* som en `Tcp` da kaldes funktionen `do_tcpprotocol` med *protocol*s liste af `tcpflag_t`, samt *nodename* som parameter. Den returnerede IDD returneres.

Hvis *protocol* i stedet er af typen `Udp`, returneres en IDD, hvor værdien 17 (svarende til UDP) tillades og alle andre værdier går til IDD terminalen falsk.

Hvis *protocol* matches som en `Icmp` da kaldes funktionen `do_icmp` med *protocol* liste af ICMP flag og*nodename* som parameter. Den returnerede IDD returneres.

■

do_tcpprotocol

Parametre: `flaglist: tcpflag_t list, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen evaluerer en række TCP flag, til en IDD for disse samt for protokol.

Først oprettes en IDD for protokol, hvor værdien 6 tillades og alle andre værdier går til terminalen falsk. Denne IDD gemmes i den lokale variabel *proto_idd*.

Herefter kaldes den interne, rekursive funktion `tcp_idd` med *proto_idd* og *flaglist* som parametre.

Funktionen indeholder to interne funktioner. Den første, `tcp_idd`, matcher *flaglist*. Hvis *flaglist* er tom returneres *proto_idd*. Hvis listen ikke er tom kaldes den interne funktion `tcp_nodename` med det første element fra *flaglist*. Den returnerede streng bruges som navn i en ny IDD, hvor værdien 1 går til terminalen sand, værdien 0 går til falsk. Derpå lægges *proto_idd* og den nye IDD sammen ved brug af and-operationen. Denne nye IDD gemmes i den lokale variabel *new_proto_idd*. Til sidst kaldes `tcp_idd`, med *new_proto_idd* og den resterende liste som parametre.

Den anden interne funktion, `tcp_nodename` matcher en `tcpflag_t` med konstruktorer for denne og returnerer et IDD nodenavn svarende til denne konstruktor. Matches dette `tcpflag_t` f.eks. med konstruktoren `Syn` da returneres strengen "TCP_SYN".

■

do_icmp

Parametre: `flaglist: int list, nodename: string`
Returtype: `idd`
Beskrivelse:
Funktionen evaluerer en række ICMP flags, til en IDD for disse samt for protokol.

Funktionen afgør først om der er sat nogle ICMP flag, ved at undersøge antallet af elementer i *flaglist*. Hvis der elementer i listen oprettes først en fuld IDD for ICMP, hvor alle værdier går til den falske IDD terminal. Derpå kaldes den interne funktion `create_icmp_idd` med denne IDD og *flaglist* som parameter. Den returnerede IDD, sættes sammen med en IDD for protokol, hvor værdien 1 går til den returnerede IDD, alle andre værdier får til den IDD terminalen falsk.

Er der ikke sat nogle flag i listen returneres en IDD tilsvarende den ovenstående, men hvor alle ICMP værdier tillades, dvs. går til den IDD terminalen sand.

Funktionen har en intern rekursiv funktion, kaldet `create_icmp_idd` der tager en liste af heltal og en IDD som parametre. Funktionen laver et match på listen af heltal. Der er nu to muligheder, den første: listen er tom, så returneres IDD fra parameteren. Alternativt tages det første element i listen, dette kaldes *flag*, den resterende liste kaldes *restlist*. Derpå oprettes en ny IDD for *flag*, denne nye IDD bruges som parameter i IDD operationen or, der tager endnu en IDD som parameter, dette er *full_idd*. Den returnerede IDD og *restlist*, bruges som parameter i det rekursive kald til `create_icmp_idd`.

■

do_input_output_rule

Parametre: `inf_rule: interfacerule_t, it; local_it: identtable_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Parameteren *inf_rule* matches mod konstruktorer for `interfacerule_t`.

Hvis *inf_rule* matches som `ValueIface`, kaldes funktionen `match_ethname` med parametrene *iface* og *nodename*. Funktionen `match_ethname` returnerer en IDD der accepterer det givne interface.

Hvis *inf_rule* matches som `VNameIface`, bruges funktionen `find_expression` til at finde udtrykket, som der efterfølgende matches. Hvis udtrykket matches på *IfaceExpr*, kaldes funktionen `match_ethname` med interface navnet som parameter. Dette vil returnere en IDD som

accepterer det pågældende interface. Hvis udtrykket matches på *AnyExpr*, returneres en IDD som accepterer alle interfaces.

■

match_ethname

Parametre: `ethname; nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen anvender OCamls indbyggede string operationer, hvor der kan lave et regulært udtryk for en streng og sammenligne om syntaksen for en given streng er korrekt. Dette anvendes på parameteren *nodename*. Det antages at syntaksen er ethx, hvor x betegner et tal i intervallet 0-255. Hvis *nodename* overholder syntaksen, dannes en IDD node der accepterer det givne tal x, ellers dannes en IDD der ikke accepterer noget.

■

do_portrule

Parametre: `portrule: portrule_t, it; local_it: identtable_t, udp_nodename; tcp_nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen danner en IDD udfra en portregel.

Parameteren *portrule* matches mod konstruktorer for `portrule_t`. Afhængigt af hvad der matches på kaldes henholdsvis funktionen `do_port` og `do_port_operator`. I de tilfælde hvor der er tale om variabelnavne er det nødvendigt at anvende funktionen `find_expression` for at finde de regler der er refereret til. Derefter laves et match på det udtryk der returneres fra `find_expression` og den pågældende IDD kan dannes udfra de underliggende funktioner.

Der returneres en IDD med portreglen.

■

do_port_operator

Parametre: `operator: portoperator_t, port: mrange_t, udp_nodename; tcp_nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen danner en IDD der accepterer en eller flere porte i henhold til den givne portoperator.

Først findes grænseværdierne for *port* med de interne funktioner `get_port_higher_bound` og `get_port_lower_bound`. Derefter matches parameteren *operator* mod konstruktorer for `portoperator_t`. For hvert match kaldes funktionen `do_port` med de fundne grænseværdier for portene og der returneres en IDD fra `do_port` med den givne portregel.

■

do_port

Parametre: `port: mrange_t, udp_nodename; tcp_nodename: string`
Returtype: `IDD`
Beskrivelse:

Funktionen danner en IDD der accepterer TCP og UDP protokollerne med portnumrene defineret udfra parameteren *port*.

Først kaldes funktionen `determine_interval` som returnerer en intervalliste udfra parameteren *port*. Denne intarvalliste bruges til at danne to referencer til en IDD, som accepterer dette interval af porte. Nodenavnene på disse er henholdsvis *udp_nodename* og *tcp_nodename*.

Efterfølgende dannes en UDP IDD og en TCP IDD, som ved accept af de respektive protokoller anvender de førnævnte IDD referencer, for derved at kunne afgøre om portnumrene også kan accepteres.

Til sidst anvendes IDD operationen OR til at danne en samlet IDD udfra UDP og TCP IDDen, som er den IDD der returneres fra denne funktion.

■

do_iprule

Parametre: `iprule: iprule_t, it; local_it: identtable_t, nodeprefix: string`
Returtype: `IDD`
Beskrivelse:
Funktionen evaluerer en iprule_t til en tilsvarende IDD.

Funktionen matcher først *iprule* mod konstruktorer for iprule_t. Hvis *iprule* matches som `ValueIp`, da kaldes funktionen `do_ip` med $nodeprefix$ og den i iprule_t indeholdte ip_t som parametre. Den returnerede IDD returneres.

Matches *iprule* i stedet som `VNameIp`, skal funktionen `find_expression` med *it*, *local_it* og strengen indeholdt *iprule* som parametre. Den returnerede expression_t gemmes i *expression*. Derpå matches *expression* med konstruktorer for expression_t. Hvis *expression* er en `IpExpr` kaldes funktionen `do_ip` med indholdet af *expression*, samt $nodeprefix$ som parameter. Matches *expression* som en `AnyExpr` returneres terminalen sand.

■

do_ip

Parametre: `ip: ip_t, nodename: string`
Returtype: `IDD`
Beskrivelse:
Funktionen returnerer en IDD svarende til *ip*.

Denne funktion opretter en IDD for hver byte i *ip*. Hver byte er beskrevet ved en mrange_t, denne mrange_t bruges som parameter i kald af funktionen `determine_interval`. Funktionen `determine_interval` skal desuden have en reference til den næste IDD, hvis et interval accepterer eller ikke gør. For den sidste byte skal denne reference være til hhv. terminalen sand og falsk. Men ved den næste skal reference for sand være til IDD for den sidste byte, sådan gøres for alle fire bytes. Derpå returneres IDDen.

■

determine_interval

Parametre: `mrange: mrange_t, min; max: int, tterm; fterm: IDD`
Returtype: `Intervalliste`
Beskrivelse:

Funktionen danner et sammenhængende interval mellem de angivne grænseværdier og indsætter et tal eller et interval indenfor dette. Værdierne af intervallerne defineres også som parametre.

Der foretages et match på parameteren *mrange* mod konstruktorer for mrange_t. Resultatet kan enten være et interval eller et tal. I begge tilfælde kontrolleres hvor *mrange* ligger i det overordnede interval, for dermed at kunne opbygge et korrekt kontinuerligt interval. For det overordnede interval og for det interne interval eller tal, indsættes en reference til en IDD node, som eksempelvis kan være en terminal med sand eller falsk.

Funktionen returnerer dermed en intervalliste med referencer til IDDer.

■

find_filter

Parametre: `filtertable: filtertable_t:, filtername: string`
Returtype: `filterdef_t`
Beskrivelse:
Bruger funktionen `Hashtbl.find` med parametrene *filtertable* og *filtername*. Dette virker da *filtertable* blot dækker over en hashtabel af filternavne og filterdefinitioner. Findes *filtername* i *filtertable* returneres en filterdef_t der igen returneres. Findes filtret ikke rapporteres en fejl, men dette bør ikke kunne forekomme efter semantisk analyse.

■

find_expression

Parametre: `it; local_it: identtable_t, constname: string`
Returtype: `expression_t`
Beskrivelse:
Funktionen foretager en søgning i den lokale eller globale identifikationstabel og returnerer det udtryk der matcher på parameteren *constname*.

Der foretages først en søgning efter *constname* i den globale identifikationstabel og såfremt det ikke giver resultat, søges derpå i den lokale identifikationstabel. Hvis der findes et udtryk for det givne *constname* foretages et match på denne. Såfremt det igen findes en reference, kaldes funktionen `find_expression` rekursivt indtil udtrykket findes, og udtrykket returneres efterfølgende fra denne funktion.

■