Sprogkonstruktion
Madsen, Per Printz

Publication date:
2001

Link to publication from Aalborg University

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

? Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
? You may not further distribute the material or use it for any profit-making activity or commercial gain
? You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.
Sprogkonstruktion

Per Printz Madsen
Aalborg Universitet
Forord

Denne rapport omhandler sprogkonstruktion inden for proceskontrol. Med sprogkonstruktion menes opbygning af strukturen (syntaksen) for et dedikeret sprog. Rapporten er opbygget som en "Case Study". Der tages udgangspunkt i dedikerede sprog anvendt til proceskontrol.

Per Printz Madsen AUC
1. februar 2001
# Indhold

1 **Hvorfor dedikerede sprog** 1
   1.1 Overordnet indfaldsvinkel 1
   1.1.1 Hvorfor og hvorfor ikke udvikle et dedikeret sprog 2

2 **Hvilke opgaver skal kunne løses af sproget** 4
   2.1 Indledning 4
   2.2 Aflåsninger 5
   2.3 Sekvensstyring 6
   2.4 Regulering 8
   2.5 Konklusion 12

3 **Udvikling af sprogets struktur og syntaks** 14
   3.1 Indledning 14
   3.2 Aflåsninger 14
   3.3 Sekvensstyring 17
   3.4 Regulering 22
   3.5 Konklusion 25

4 **Udenomskrymmel** 26
   4.1 Proces-io 26
   4.2 Netværks-import/export 27
   4.3 Parametre 27
Kapitel 1

Hvorfor dedikerede sprog

1.1 Overordnet indfaldsvinkel

Med begrebet dedikerede sprog menes sprog, som er målrettede mod en passende snæver anvendelse, som fx. dedikerede sprog anvendt til proceskontrol.

Den overordnede opgave, ved udvikling af dedikerede sprog, er derfor at udvikle et sprog, der er så simpelt som muligt dog således, at det indeholder de nødvendige værktøjer for at kunne anvendes inden for det målrettede område.

Et overordnet krav til sproget kan fx. være, at sprogets struktur og de programmeringstekniske værktøjer, som sproget indeholder, skal have en sådan natur, at det er let for personer uden kendskab til generelle programmeringsprog, men med kendskab til de opgaver der skal løses, at formuler sig i sproget.

Sprogets opbygning skal derfor tage udgangspunkt i de opgaver, man agter at løse med sproget (computersystemet). *Der skal ikke tages udgangspunkt i et generelt programmeringsprog som fx. C.*

Vil man lave et dedikeret sprog til proceskontrol, skal man derfor begynde med at afklare, hvilke type af opgaver den pågældende computer skal kunne løse, samt hvordan opgaverne løses mest hensigtsmæssigt mht. styring, regulering og overvågning det pågældende processtekniske anlæg.

Nå man er blevet færdig med at analyserer den gruppe af processtekniske anlæg som computeren skal kunne håndtere (styre, regulere og overvåge) skal man til at være *kreativ*. Man skal omsætte disse styrings-, regulerings- og overvågningsopgaver til en programmeringsteknisk udtryksform. (*Et programmeringssprog*).

Kreativ betyder her, at der ikke er én rigtig løsning. Der er mange gode og brugbare løsninger. Disse gode og brugbare løsninger er de løsninger, som dygtige og erfarne ingeniører kommer til. Der udover er der de geniale og banebrydende løsninger, som ingen finder på.
1.1.1 Hvorfor og hvorfor ikke udvikle et dedikeret sprog

Når man er stillet overfor opgaven at skulle skrive kildeteksten\(^1\) til et program, der skal kunne løse et givet problem, kan man göre det efter to modeller:

1. Anvende et generelt programmeringsprog til applikationsskrivningen.
2. Anvende et dedikeret sprog til applikationsskrivningen.

![Diagram](image)

Figur 1.1: **Udviklingstiden for de to modeller.**

Model 1. er at foretrække hvis, der fra gang til gang, er stor forskel på de opgaver, der skal løses af computerystemet. Såd med andre ord: hvis man starter forfra hver gang, først der kun er en lille del, der kan genbruges, så vil det bedste programmeringsværktøj være et generelt sprog med et tilhørende bibliotek af generelt anvendelige standard procedurer. Model 1. kræver derfor en person med både kendskab til programmering i et generelt sprog fx. C og kendskab til den opgave der skal løses. (En computereksperter med kedeldragt på\(^2\)... Den slags mennesker er meget sjældne.)

Model 2. er at foretrække, hvis der er stor lighed fra gang til gang. Som fx. ved anvendelse af computere til styring, regulering og overvågning i procesetekniske anlæg. I et sådant tilfælde skal man kun bruge denne "computereksperter med kedeldragt på" én gang. Han skal udvikle et dedikert sprog, der skal kunne anvendes af folk, som kun har "kedeldragt på", dvs ikke computereksperter.

Figur 1.1 er et forsøg på at illustere forskellen i udviklingstid ved de to grund software udviklingsmodeller. Den del, der her kaldes for **system**, er den del af

\(^1\)Kildeteksten, til løsning af en bestemt opgave kaldes ofte for **applikationen**. I det følgende, når udtæktet "**applikationen**" anvendes, menes derfor dette program.

\(^2\)Med vendedingen: "Computereksperter med kedeldragt på" menes her en person med kendskab til dels computer software/hardware og dels til de opgaver, der skal løses af computeren. Han har således både set og rørret ved en computer, en servo motor, en ventil osv.
1.1. OVERORDNET INDFALDSVINKEL


Antages det, at der er en stor del, der kan genbruges fra gang til gang, hvad får man så ud af at udvikle et dedikeret sprog til "indpakning" af denne genbrugs-software??.

Hvis det dedikerede sprog er godt, hvad jeg vil antage det for at være, har det følgende fordele:

1. Den tid applikationsskrivningen tager bliver minimeret.
2. Antallet af fejl bliver minimeret.
4. Det er lettere at vedligeholde kildeteksten.
5. Afhængig af sproget og den person der har skrevet applikationen, kan kildeteksten være selvdokumenterende - også over for ikke computerekspert.

Dette er de vigtigste argumenter for at udvikle et dedikeret sprog. Men som I ved er der ingen udvikling, der er rigtigt eller forkert³, ingen udvikling er enten sort eller hvidt. Der er således altid mange synsvinkler på en konkret sag. Så derfor er her nogle af argumenterne imod dedikerede sprog.

1. Programmet bliver bundet af sproget, idet det ikke er generelt anvendeligt.
2. Det er besværligt at tilføje nye faciliteter.
3. Flere har kendskab til et generelt sprog (fx. C) end til et et dedikeret sprog.
5. Kan kun løse dedikerede opgaver.
6. Det er næsten umuligt at gette på fremtidige anvendelser.

Nok om det. Fra nu af antages det, at man har valgt at udvikle et dedikeret sprog til styring, regulering og overvågning af procesetekniske anlæg.

³Der er mange mennesker, der tror de kender sandheden og tror, de ved, hvad der er rigtigt og forkert, hvem der har ret og hvem der har uret. Ofte udspringer denne ensidige tro (Janatisme) af uvidenhed og/eller hjernevæske... Citat: PPM.
Kapitel 2

Hvilke opgaver skal kunne løses af sproget

2.1 Indledning

Som før nævnt skal man tage udgangspunkt i de opgaver, der ønskes løst af computersystemet. I det følgende vil der derfor blive givet eksempler på, hvilke opgaver der er og hvordan de ønskes løst, når anvendelsen er proceskontrol.

Alle processteknikke opgaver (jeg kender) kan puttes ind under tre begreber:


Disse tre begreber er de overordnede styrings-, regulerings- og overvågnings-tekniske begreber og kræver en række andre underopgaver løst for at kunne
2.2. AFLÅSNINGER


Lad os i en stund glemme alt om, hvordan vi får stoppet de tre begreber (aflåsninger, sekvensstyring og regulering) ned i en computer og i stedet dæle lidt ved hver enkelt af disse overordnede begreber.

2.2 Aflåsninger

Aflåsninger anvendes, når aktioner skal udføres som følge af bestemte hændelser eller kombinationer af hændelser.

F.x. en motoren skal starte, når der trykkes på startknappen og alt er ok. Dette lille eksempel kræver, at man har signalet alt er ok til rådighed. alt er ok signalet er ofte genereret udfra en mængde andre signaler: alt er ok hvis kølevand ok og ikke temp. for høj og ......

I det følgende vil jeg beskrive et mere konkret eksempel.

![Diagram](image_url)

Figur 2.1: Start-stop af blæser med alarm.

Figur 2.1 et luftanlæg, der fx. kan anvendes til at føde tre brænderere med luft.

En simpel funktionsbeskrivelse kan fx. være følgende: Blæseren M1 skal kunne startes og stopses ved tryk på luv. start- og stopknappen på kontrolpanelet. Hvis der opstår en alarm, skal motoren stoppe og lampe Alarmlampe på kontrolpanelet skal blinke. Der er alarm, hvis trykket i beholderen ikke er over et
passende tryk efter en given tid fra motoren er startet.

Knapperne, Startknap og Stopknap, på kontrolpanelet er ringetryk og motoren M1 er en on/off motor. P er en trykstyring kontakt, som slutter når trykket i bebojseren er nået over et passende tryk.

Alarmlampen skal lyse, hvis trykket ikke er vokset op, givet ved P, efter en given tid, givet ved alarmaf fra at motoren M1 har fået et start signal.

Pseudokoden kan se således ud:

M1 er lig ( M1 eller startknap) og ikke stopknap og ikke alarm;

alarm er lig alarm eller ikke P efter alarmaf fra M1;

Alarmlampe er lig alarm og blink;

Størrelsen blink er her tænkt som en logisk variabel som blinker (skifter mellem sand og falsk). Bemærk at pseudokoden ovenfor kun fungerer, hvis M1 og alarm initialiseres til falsk.

Konstruktionen (M1 er lig ( M1 eller ...) er et såkaldt selvhold på M1. Synes man, at dette er en uoverskuelig konstruktion, kan man indbygge en Filp-Flop i sproget, således at den vil kunne anvendes i stedet.

Konklusionen på ovenstående er, at der er brug for at kunne lave logiske udtryk sammensat af følgende:

- De logiske operatorer: OG, ELLER og IKKE.
- Tidsforskydning af hændelser: Bliver sand efter given tid fra hændelse.
- Mulighed for anvendelse af parantezer.
- En systemvariabel blink som blinker. Denne variabel kan realiseres vha. tidsforskydning af hændelser, men det vil virke kloset.
- Evt. Flip-flops.

2.3 Sekvensstyring

Hvis blæsermotoren fra figur 2.1 har en vis størrelse, skal den startes vha. en en såkaldt +Δ starter. En +Δ starter er en sekvens af fire tilstande, som kan beskrives som nedenfor:

1. tilstand Denne tilstand er den såkaldte stoptilstand og er den tilstand motoren befinder sig i, når den er stoppet.

2. tilstand Denne tilstand er den såkaldte stjæmetilstand. I denne tilstand skal motoren forsynes med det såkaldte stjæme-signal samsidig med at hovedstrømmen er sluttet. Denne tilstand bibeholdes i en given tid fx. 5 sek.
3. tilstand Denne tilstand er en overgangstilstand, hvor man skifter fra stjerne til trekant. Det er nødvendigt med denne overgangstilstand, da hovedstrømmen skal afbrydes i skiftet mellem stjerne og trekant. Tilstanden kan fx. vare 0.2 sek. hvor hovedstrømmen er afbrudt og stjernesignalet forsvarer samtidigt med at trekantsignalet bliver sat.

4. tilstand Denne tilstand er den endelige driftstilstand, hvor trekantsignalet er sat samtidigt med at hovedstrømmen er sluttet. Denne tilstand bliver mån i, til der indtræder en hændelse, der gør at der skiftes til 1. tilstand.

Dette lille eksempel er et eksempel på en simpel sekvensstyring. Sekvensstyringer er karakteriserede ved, at styringenfinder sig i en af mange tilstande ad gangen og at der kan skiftes fra en tilstand til en anden, når der indtræffer forskellige hændelser. En sådan hændelse kan også være kombinationer af andre hændelser, beregnet ud fra logiske udtryk, som ved aflæsninger. Ofte er det, indenfor den samme sekvens, tilladt at skifte til bestemte tilstande fra en vilkårlig anden tilstand og til andre tilstande hun fra bestemte tilstande. Tilstand 1. (Stop tilstanden) fra ovenstående må man skifte til fra en vilkårlig anden tilstand, hvorimod man kun må skifte til tilstand 2 fra tilstand 1. til tilstand 3 fra tilstand 2 og til tilstand 4 fra tilstand 3. En sekvensstyring med tilhørende skiftetilstande kan beskrives via et såkaldt tilstandsdiagram. På figur 2.2 er *Δ starteren udtrykt på tilstandsdiagramform.

![Tilstandsdiagram for *Δ starter.](image)

Denne *Δ startersekvens vil typisk være en undersekvens til en anden sekvens, som fx. hvis motoren skulle trække en omrører i en blandingsbeholder.

Figur 2.3 viser en sådan blandingsbeholder, hvor der blandes N. forskellige komponenter. Nedenstående sekvens et eksempel på en sådan blandingssekvens.
Figur 2.3: Blandingsbeholder til illustration at blendingsekvens.

1. tilstand Første komponent fyldes i indtil V1 % fuld.

2. tilstand Anden komponent fyldes i indtil V2 % fuld.

:\

N. tilstand N’te komponent fyldes i indtil 100 % fuld.

N+1. tilstand Omrøren startes. (Hvis aktiveres underskærslen + Δ starter.) Den kører fx. indtil en bestemt tid er gået.

N+2. tilstand Udtømning indtil beholderen er tom.

Ofte ser man, at der er flere apparater der skal synkroniseres på en ganske bestemt måde. Har man fx. to beholder og ønskede at imens der er omrøring i den ene forberedes omrøring i den anden. Man har måske kun et piskeri men to skåle. I dette tilfælde skal den ene være i tilstand N+1, mens den anden gennemløber tilstandene N+2, 1, 2, ..., N og derefter skiftes. Dette kræver, at de to sekvensstyringer kan synkroniser sig indbyrdes.

Konklusionen på sekvensstyring er, at det skal være muligt, i sproget, at formulere sekvensstyringer på flere niveauer. Sekvensen skal være opdel i tilstande, således at programmet befinder sig i en bestemt tilstand på hvert niveau i sekvensstyringshierarkiet. Desuden skal det være muligt at synkronisere sekvensstyringer, som befinder sig på samme niveau.

2.4 Regulering

Regulering adskiller sig meget fra de to forrøgende begreber:
2.4. **REGULERING**

- Regulering skal afvikles cyklik med konstante periodetider. Samlingstiden.

- Regulering er overvejende beregning på reelle tal, hvorimod de to foregående begreber overvejende har været beregninger på boolske størrelser

I det følgende vil jeg vise en række eksempler på ofte anvendte regulatoropkoblinger og derigennem bestemme, hvilke krav der bør stilles til regulatorernes funktion.

Eksempel 1. er en simpel regulatoropkobling, som skal regulere vandstanden i en beholder, som vist på figur 2.4

![Regulatoropkobling](image)

Figur 2.4: *Vandstandsejering.*

Regulatoropkoblingen, vist på figur 2.4, er den mest simple, man ser ude i industrien. Den består af en enkelt PI-regulator med auto/manuel omkobling og en op- og en ned-knap.

En sådan regulator har to tilstande, en auto-tilstand og en manuel-tilstand.

I auto-tilstanden er det regulatorfunktionen, der bestemmer outputtet fra regulatoren, dvs. at outputtet er bestemt ud fra nedenstående udtryk:

\[ \text{Output}(t) = P \cdot (\text{Ref}(t) - \text{Mv}(t)) + I \cdot \int_{0}^{t} (\text{Ref}(T) - \text{Mv}(T)) dT \]

Jeg antager, at regulatorfunktionen er af PI-typen, da det er den regulatortype der praktisktalt altid anvendes ude i industrien. Dette skyldes, at den er nøjagtig og rimelig let at indstille i hånden.
For at denne funktion skal kunne fungere i praksis, er det nødvendigt at sikre sig mod integrale windup. Dvs. man skal sikre, at opdateringen af integraleledet i regulatoren standses, når den aktuator som styrer af outputsignalet (motor ventilen M1 på figur 2.4) er i møtning. Dette kan let realiseres, hvis regulatoren kender møtningsgrænserne for den pågældende aktuator.

I manuel-tilstanden er det op- og ned-knapperne, der bestemmer outputtet fra regulatoren. Når op-knappen aktiveres, skal outputtet øges efter en passende støj rampe, og tilsvarende skal outputtet mindskes når ned-knappen aktiveres.

I skiftet mellem auto og manuel skal outputtet holdes konstant (hopsri omkobling). Dette sikres ved, at lade regulatoren holde det til enhver tid værende output ved omkoblingen til manuel tilstand. I manuel-tilstand skal integraleledet justeres således, at reguleringsfunktionen giver det rigtige output, når der igen kobles til auto-tilstand.

Eksempel 2. er en meget anvendt regulatoropkobling. Det er en såkaldte kaskadekobling. Denne opkobling er vist på figur 2.5

Regulatoren benævnt med PI-Reg1 har samme funktion, som regulatoren på figur 2.4. Regulatoren benævnt med PI-Reg2 er indført for at udkompensere ulineariteter i reguleringsventilen. Det er PI-Reg2, som styrer ventilen. Det er derfor også denne regulator (PI-Reg2), som anvendes til auto/manuel omkoblingen.
2.4. REGULERING

Det nye her er, at PI-Reg1 har to ekstra indgange, kaldt *Preset* og *Preset værdi*. Disse to indgange er nødvendige for at kunne sikre hopfrø omkobling, fra manuel til auto, i denne omkobling.

Hvis *Preset* signalet er sat er outputtet fra regulatoren lig *Preset værdi*. Husk at justere integralkoeficienten i reguleringsfunktionen således, at der er hopfrø omkobling, når *Preset* signalet forsvinder.

Problem opstår, når PI-Reg2 kobles over i manuel tilstand. I dette tilfælde vil PI-Reg1 miste kontakten med processen og derfor pga. integralkoeficienten drive mod $\pm \infty$. Når der igen kobles tilbage til auto-tilstand vil kravet til flowet gennem reguleringsventilen være $\pm \infty$ - det er meget. For at sikre mod dette presettes PI-Reg1 til en værdi lig med måleværdier for PI-Reg2, her værdien fra flowmåleren dp2, så længe PI-Reg2 er i manuel-tilstand.

Eksempel 3. er et eksempel på feedforward- kombineret med feedback-regulatoropkobling. Dette er ligeledes en ofte anvendt regulatoropkobling og er i visse tilfælde en nødvendig opkobling for at kunne regulere processen tilstrækkelig hurtigt. Ideen her er, at man flyder lige så meget vand ind i beholderen, som man tager ud. Dette gøres ved at måle flowet ud af beholderen og så styre outputtet fra PI-Reg1 således at det svarer til udløbsflowet. Et eksempel på en sådan feedforward-regulering kombineret med feedback-regulering er vist på figur 2.6.

![Diagram](image)

**Figur 2.6:** Feedforward-kombineret med feedback-regulering.

Figur 2.7 viser en ofte anvendt måde at afbilde en sådan feedforward-regulering på, som noget der er adskilt fra selve feedback-regulatoren.


De to sidste ting er meget praktiske at have med i kufferten da det ofte viser sig, at mør for brug for dem i praksis.

### 2.5 Konklusion

Fra afsnittet om aflæsering haves, at det skal være muligt at håndtere følgende:

- De logiske operatorer: OG, ELLER og IKKE.
- Tidsforskydning af hændelser: Bliver sand efter given tid fra hændelse.
- Mulighed for anvendelse af parenteser.
2.5. KONKLUSION

- Evt. Flip-flop’s.

Fra afsnittet om sekvensstyring hæves, at det skal være muligt at håndtere følgende:

- Sekvensstyringer på flere niveauer.
- Det skal være muligt at synkronisere sekvensstyringer på samme niveau.
- Sekvensen skal være opdelt i tilstande således, at programmets tilstand på hvert niveau i sekvensstyringshierarkiet.

Fra afsnittet om regulering hæves, at det skal være muligt at håndtere følgende:

- PI-regulatorer.
- Omkobling mellem auto drift og manuel drift.
- Det skal være muligt at koble flere regulatorer sammen i kaskader.
- Det skal være muligt at presette outputt fra regulatoren.
- Det skal være muligt at lave simpel feedforward-regulering.
- Det skal evt. være muligt at summere og multiplicere reelle tal.
- Det skal evt. være mulig at anvende ulineære funktioner til at linearisere signaler.
Kapitel 3

Udvikling af sprogets struktur og syntaks

3.1 Indledning


Min baggrund er tre år ude i industrien, hvor jeg arbejdede med dels et proceskontrollsystem kaldt, ABCL og dels med Søren T. Lyngsø's proceskontrolsystem, kaldt Stella. Jeg er derfor præget af de måder disse systemer blev programmeret på. Det jeg vil gøre mest anvendelse af her er påvirkning jeg har fra ABCL, da det var et proceskontrolsystem mere i min smag end Stella.

Jeg vil tage udgangspunkt i de tre hovedbegreber: Aflåsninger, sekvensstyring og regulering.

3.2 Aflåsninger

Kunsten er nu at bestemme en passende struktur og syntaks, som vil kun tilfredsstille den type logiske udtryk, som er beskrevet i afsnit 2.2 side 5. Tages der udgangspunkt i pseudokoden herfra er den givet ved følgende:

\( M1 \) er lig ( \( M1 \) eller startknap) og ikke stopknap og ikke alarm;

alarm er lig alarm eller ikke \( P \) efter alarmtid fra \( M1 \);

Alarmlampe er lig alarm og blink;
3.2. AFLÅSNINGER

Det vil være naturligt at udtrykke dette med følgende lille programeksempel:

\[ M1 = (M1 \text{ eller STARTKNAP}) \text{ og ikke STØPKNAP og ikke ALARM}; \]

\[ \text{ALARM} = \text{ALARM} \text{ eller delay}(M1, \text{ALARMTID}) \text{ og ikke } P; \]

\[ \text{ALARMLAMPE} = \text{ALARM} \text{ og blink}; \]

De ord der her er skrevet med lille er de, af compileren, predefinerede ord.

Denne udtryksform svarer rimelig godt til den måde, hvorpå man ville formulere sig mundtligt hvis man skulle forklare programmet. Det forudsætter dog at de størrelser, som indgår i programmet, navngives, på en formfind måde. Programmeringsformen kan derfor siges at være rimelig selvdukkumenterende.

Det nye i denne sammenhæng er funktionskaldet \text{delay}(M1, \text{ALARMTID}). Dette kald er en tidsforsinkelse af \text{M1} med tiden \text{ALARMTID}, som vist på figur 3.1.

\[ \text{Figur 3.1: Tidsforsinkelse af en hændelse.} \]

Oftå får man brug for at kunne anvende reelle størrelser i sine logiske udtryk. Fx. hvis \text{P} er en reel angivelse af trykket fx. i [bar], så har man brug for konstruktionen:

\[ \text{ALARM} = \text{ALARM} \text{ eller delay}(M1, \text{ALARMTID}) \text{ og ikke } (P \geq \text{ALARMTRYK}); \]

eller

\[ \text{ALARM} = \text{ALARM} \text{ eller delay}(M1, \text{ALARMTID}) \text{ og } (P < \text{ALARMTRYK}); \]

Der kan derfor udvides med relationsoperatorerne: \(< \leq >\geq >\geq >\). 

Afviklingen af disse logiske udtryk skal udføres således, at brugeren opfatter det som omt. Alle udtryk bliver evalueret enten hele tiden eller også hver gang, der sker en ændring. En bruger vil ikke kunne se forskel på disse to afviklingsmetoder.

Udtrykkene evalueres hver gang, der sker ændringer Denne afviklingsmetode bygger på, at afviklingssystemet holder øje med, om der sker ændringer på input til computeren. Hvis et input ændrer sig evalueres alle de udtryk, som er relevante for det pågældende input. Evt. evalueres alle logiske udtryk af administrative årsager. Denne metode ser umiddelbart ud til at økomisere med CPU'en, men der kan dog nemt blive en del administration for afviklingssystemet, hvorved fordelen forsvinder.

En anden måde, at beskrive disse logiske udtryk på, er en grafisk form. Programmet består her af en række logiske symboler (AND -, OR - og NOT gate osv.) samt forbindelser mellem disse. Man tegner sig således frem til sit applikationsprogram.

Figur 3.2: Grafiskprogrammering.
3.3. SEKVENSSTYRING

Figuur 3.2 er det samme program, som ovenfor her bare udtrykt på grafisk form. Denne programmeringsform bliver mere og mere udbredt i industrien. Dette skyldes antageligt, at den er let overskuelig for folk uden den store programmeringsbaggrund.

Min erfaring (den er meget sparsom) siger dog, at denne programmeringsform kan virke uoverskuelig og er langsom at programmere, specielt hvis der er tale om store applikationer.

Ofte ser man tekstbaserede programmeringsprog, der ligeledes er bygget op omkring logiske gates. Disse udtryksformer har samme svagheder, som den grafiske type og vil derfor ikke blive omtalt i denne note.

3.3 Sekvensstyring

I dette afsnit vil der blive givet eksempler på, hvorledes sekvensstyringer kan omformes til en programmeringsteknisk udtryksform. Der vil således blive forsøgt at bestemme en passende struktur og syntaks, som vil kunne tilfredsstille de krav til sekvensstyringer, som er beskrevet i afsnit 2.3 side 6.

Det skal være muligt, på en naturlig måde, at kunne programmere sekvensstyringer på flere niveauer og at kunne synkronisere sekvensstyringer på samme niveau. Herunder at opdele sekvenserne i tilstande således, at programmet befinder sig i én bestemt tilstand på hvert niveau i sekvensstyringshierarkiet.

I afsnitet omhandlende aflåsninger blev der udviklet en simpel programstruktur. Denne bestod bare af en række statements. For at kunne udtrykke sekvenser på en naturlig måde, vil det være oplagt at udvikle en programstruktur, der kan udtrykke det nødvendige.

En sådan sekvensprogramstruktur kunne se således ud:

```plaintext
sekvæs <NÅVN>
 reset = <håndelse>;
 hold = <håndelse>;

tilstand init
   "Her er vi når programmet starter eller når sekvensen aktiveres" 
   "Her kan være logiske udtryk, som skal evalueres, når 
       programmet er i denne tilstand"

   gå til <NÅVN> når <håndelse>;
   slut tilstand

tilstand <NÅVN>
   "Her kan være logiske udtryk, som skal evalueres, når 
       programmet er i denne tilstand"
```
gå til <NAVN> når <hændelse>; 
slut tilstand
.
.
.

slut sekvens

Sekvensprogramstrukturen starter med sekvens <NAVN>. Dette betyder, at her starter sekvensen med navnet <NAVN>. Herefter er der to statements, som er overordnede kontrolstatements for sekvensen. reset = <hændelse>; betyder at hvis hændelser sker, går sekvensen til init-tilstanden, uanset hvilken tilstand den måtte være i. Denne indgang har højeste prioritet. hold = <hændelse>; betyder at hvis hændelser sker, standser det sekventielle forløb. Sekvensen bliver i den tilstand, den befinder sig i. Denne indgang har næst højeste prioritet.

Herefter starter en beskrivelse af init-tilstanden tilstand init. Init-tilstanden er den tilstand sekvensen vil befinde sig i ved opstart af programmet. De statements som vil blive afviklet i denne tilstand er de, der findes mellem tilstand init og slut tilstand. Imellem disse to linier er der et eller flere statements af typen gå til <NAVN> når <hændelse>; Indtræffer denne hændelse vil sekvensen skifte til tilstanden med navnet <NAVN>.

Alle øvrige tilstande i sekvensen har samme struktur som init-tilstanden. Har kaldes tilstanden dog et entydigt navn istedet for init.

I hver tilstand kan der være en eller flere undersekvenser.

Disse undersekvenser har samme struktur som den overordnede sekvens. Når den overordnede sekvens forlader tilstanden vil undersekvenser automatisk blive resat og derved gå til init-tilstanden.

Synkronisering af sekvenser på samme niveau kan gøes på flere forskellige måder. En måder er at anvende hold signalen til at standse en sekvens. En sekvens' holdsignal skal kunne aktiveres fra et vilkårligt sted i programmet dvs. også fra andre sekvensers tilstande.

En anden måde, og nok den der i de fleste tilfælde kan anbefales, er at indføre en eller flere ventetilstande, hvori sekvensen venter til, der sker en bestemt hændelse. En ventetilstand er en alm. tilstand, hvori der ikke udføres nogen handling udover, at der checkes på hændelser via gå til <NAVN> når <hændelse>. Der hoppes derved til den specificerede tilstand, når hændelsen indtræffer.

Man kan i visse tilfælde også indbygge synkroniseringen i gå til <NAVN> når <hændelse> for de enkelte tilstande, men det vil nok gøre programmet mindre overskueligt.

Nedenstående eksempel viser programmet for blandingsbeholderen fra ansnit 2.3 side 6. Det er valgt kun at have to komponenter, der skal blandes 50% - 50%.
I programeksemplet er der anvendt følgende proces-io:

**START_BLANDER**: Trykknap, der anvendes til at starte blanderen.
**KOMPONENT1_VENTIL, KOMPONENT2_VENTIL og UDTØMNING_VENTIL**: Outputsignal fra computeren, der åbner den pågældende ventil.
**TØM**: Inputsignal til computeren, som angiver, at beholderen er tom.
**HALV_FULD**: Inputsignal til computeren, som angiver, at beholderen er halv fuld.
**FULD**: Inputsignal til computeren, som angiver, at beholderen er fuld.
**STØPSIGNAL**: Trykknap, der anvendes til, at stoppe omrøringen.
**HØVEDSTRØM**: Outputsignal fra computeren, der slutter hovedstrømen til motoren.
**STJERNESIGNAL**: Outputsignal fra computeren, der sætter motoren i stjernekobling.
**TREKANTSIGNAL**: Outputsignal fra computeren, der sætter motoren i trekantkobling.

**sekvens BLÅNDER**

reset = falsk;
hold = falsk;

tilstand init
    KOMPONENT1_VENTIL = falsk;
    KOMPONENT2_VENTIL = falsk;
    UDTØMNING_VENTIL = falsk;
    gå til KOMP1_IFYLDNING når START_BLANDER;
slut tilstand

tilstand KOMP1_IFYLDNING
    KOMPONENT1_VENTIL = sand;
    gå til KOMP2_IFYLDNING når HALV_FULD;
slut tilstand

tilstand KOMP2_IFYLDNING
    KOMPONENT1_VENTIL = falsk;
    KOMPONENT2_VENTIL = sand;
    gå til OMRØRING når FULD;
slut tilstand

tilstand OMRØRING
    KOMPONENT2_VENTIL = falsk;
    aktiver(OMRØRERMOTOR); "Her aktiveres undersekvensen"
    gå til UDTØMNING_S når delay(OMRØRERMOTOR.DRIFT,OMRØRINGSTID)
    eller STØPSIGNAL;
slut tilstand

tilstand UDTØMNING
    UDTØMNING_VENTIL = sand;
    gå til init når TØM;
slut tilstand
20 KAPITEL 3. UDVIKLING AF SPROGETS STRUKTUR OG SYNTAKS

slut sekvens

sekvens OMRØRER_MOTOR

hold= falsk;

tilstand init
   HØVEDSTRØM = falsk;
   STJERNESIGNAL= falsk;
   TREKANTNSIGNAL= falsk;
   gå til STJERNEN når sand;
   slut tilstand

tilstand STJERNEN
   HØVEDSTRØM = sand;
   STJERNESIGNAL= sand;
   gå til init når STØPSIGNAL
   gå til OVERgang når delay(OMRØRER_MOTOR.STJERNEN,5.0);
   slut tilstand

tilstand OVERgang
   HØVEDSTRØM = falsk;
   STJERNESIGNAL= falsk;
   TREKANTNSIGNAL= sand;
   gå til init når STØPSIGNAL
   gå til DRIFT når delay(OMRØRER_MOTOR.OVERgang,0.2);
   slut tilstand

tilstand DRIFT
   HØVEDSTRØM = sand;
   TREKANTNSIGNAL= sand;
   gå til init når STØPSIGNAL;
   slut tilstand

slut sekvens

Størrelsen <sekvens navn>,<tilstands navn> er en boolsk størrelse, som angiver hvorvidt den pågældende sekvens er i den pågældende tilstand.

Størrelserne sand og falsk er, af compileren, predefinerede størrelser. Man bør måske i stedet anvende termerne on og off for disse størrelser, da det ofte vil give et mere læsevenligt program.

Hvis alle sekvensers reset-signal default initialiseres til sand kan kaldet aktiver(OMRØRER_MOTOR) let realizeres. Det skal da bare sætte reset-signalet for OMRØRER_MOTOR til falsk.

Antages det nu at det skal være muligt at starte en lille motor vha. udtrykkene
3.3. SEKVENSTYRING

i afsnit 2.2, når og kun når ØMRØRER_MOTOR er i DRIFT tilstand, skal man bare udvide denne tilstand, som vist nedenfor.

**tilstand DRIFT**

HØVEDSTRØM = sand;
TREKANTSIGNAL= sand;

M1 = (M1 eller STARTKnap) og ikke STØPNAP og ikke ALARM;
ALARM = ALARM eller delay(M1, ALARMMTID) og ikke P;
ALARMLAMPE = ALARM og blink;

gå til init når STØPSIGNAL;
slut tilstand

Hvis man ønsker altid, at kunne starte den lille motor, kan disse tre statements anbringes helt uden for sekvenserne. Man kan således let håndtere situationer, hvor noget skal ske i en enkelt tilstand eller hvor det altid skal ske. Hvis man ønsker at kunne starte motoren i en række tilstande, er det hensigtsmæssigt at anvende et procedure aktiveringskald.

Selve proceduren skal være placeret uden for sekvenserne og skal have strukturen:

```plaintext
procedure LILLE_MOTOR
    M1 = (M1 eller STARTKnap) og ikke STØPNAP og ikke ALARM;
    ALARM = ALARM eller delay(M1, ALARMMTID) og ikke P;
    ALARMLAMPE = ALARM og blink;
slut procedure
```

Man kan herefter skrive som nedenstående i de tilstande, hvor motoren skal kunne startes:

```plaintext
tilstand DRIFT
    HØVEDSTRØM = sand;
    TREKANTSIGNAL= sand;
    aktiver(LILLE_MOTOR);
    gå til init når STØPSIGNAL;
slut tilstand
```

Det bemærkes, at alle sekvenser og procedurer som er aktiveret i en tilstand skal pauslceres automatisk, når programmet forlader tilstanden.

Vi har nu en programstruktur, der ser således ud:
program "Her starter programmet"

"Statements der afvikles cyklisk med en frekvens større end 5 Hz"

procedure <NAVN>
"Statements der afvikles cyklisk med en frekvens større end 5 Hz
når proceduren er aktiveret"
slut procedure

sekvens <NAVN>
"Statements der afvikles cyklisk med en frekvens større end 5 Hz"

tilstand init
"Statements der afvikles cyklisk med en frekvens større end 5 Hz
når sekvensen enten: startes op første gang, resettes, hvis
sekvensen ikke er aktiv eller hvis der skiftes til init-tilstanden vha.
gå til statments"
slut tilstand
tilstand <NAVN>
"Statements der afvikles cyklisk med en frekvens større end 5 Hz
når der skiftes til <NAVN> tilstanden vha. gå til statments"
slut tilstand
slut sekvens

slut program "Her slutter programmet"

Det her beskrivne er mit bud på, hvorledes en sekvensstyring skal bygges op.
Oplysning af programmet er et skud fra høften, så der er sikkert mange andre
og måske bedre måder at bygge sådan noget op på.

3.4 Regulering

I dette afsnit vil der blive givet eksemplet på, hvorledse krav til regulering,
som er beskrevet i afsnit 2.4 side 8, kan omformes til nogle byggeblokke i det
dedikerede sprog.

Disse byggeblokke skal kunne tilfredsstille følgende krav:

Det skal være muligt at håndtere PI-regulatører.
Det skal være muligt at omkoble mellem auto drift og manuel drift.
Det skal være muligt at sammenkoble flere regulatører i kaskader og det skal
derfor være muligt at presette outputtet fra regulatoren.
Det skal være muligt at lave simpel feedforward-regulering.
Det skal evt. være muligt at summere og multiplikere reelle tal.
Det skal evt. være muligt at anvende ulinære funktioner til at linearisere sig-
naler.
3.4. REGULERING

En regulator er en selvstændig enhed, som indeholder en lille sekvensstyring og en del beregninger. Den indbyggede sekvensstyring er ens for alle regulatorer (auto/manuel omkobling). De beregninger, der skal foretages i autotilstanden er forskellige fra regulator type til regulator type. Det er derfor naturligt at simplificere brugerprogrammet ved at brugeren bare skal vælge mellem en række predefinerede regulatortyper og derved skjule sekvensstyringen og beregningerne for brugeren. Her vil dog kun blive beskrevet regulatorer af typen PID og en variant, der ofte kaldes en 3-PC. Dette valg er gjort, da det vil kunne dække behovet i langt de fleste tilfælde.

Jeg foreslår, at der opbygges følgende konstruktioner i prøvemæt:

```
pid_regulator <NAVN>
  ref = <NAVN på reel størrelse>;
  mv = <NAVN på reel størrelse>;
  p_led = <NAVN på reel størrelse>;
  i_led = <NAVN på reel størrelse>;
  d_led = <NAVN på reel størrelse>;
  samplingstid=<NAVN på reel størrelse>;

  max_output= <NAVN på reel størrelse>;
  min_output= <NAVN på reel størrelse>;

  manuel = <håndelse>;
  op = <håndelse>;
  ned = <håndelse>;

  preset= <håndelse>;
  preset_værdi=<NAVN på reel størrelse>;

  ff_værdi= <NAVN på reel størrelse>;
  ff_gain= <NAVN på reel størrelse>;
slut pid_regulator
```

```
3pc_regulator <NAVN>
  ref = <NAVN på reel størrelse>;
  mv = <NAVN på reel størrelse>;
  p_led = <NAVN på reel størrelse>;
  i_led = <NAVN på reel størrelse>;
  d_led = <NAVN på reel størrelse>;
  samplingstid=<NAVN på reel størrelse>;

  i_max= <håndelse>;
  i_min= <håndelse>;

  manuel = <håndelse>;
  op = <håndelse>;
  ned = <håndelse>;
slut 3pc_regulator
```


Nedenstående programeksempel er vandstandsreguleringen fra figur 2.6 afsnit 2.4 side 11.

I programeksemplet er der anvendt følgende proces-i0 og parametre:

HØJDE: Måleværdien for vandstanden. dp1.
IND_FLOW: Måleværdien for flowet ind i beholderen. dp2.
UD_FLOW: Måleværdien for flowet ud af beholderen. dp3.
MANUEL: Ringetryk, der kobler PI-reg2 over i manuel tilstand.
AUTO: Ringetryk, der kobler PI-reg2 over i auto tilstand.
OP: Ringetryk, der kører motoren M op i manuel tilstand.
NED: Ringetryk, der kører motoren M ned i manuel tilstand.
REFERENCE: Referenceværdi for vandstanden.
VENTIL_OP: Output signal fra computeren, der kører motoren M op.
VENTIL_NED: Output signal fra computeren, der kører motoren M ned.
P, I, P1, I1, SAMPLETID, MAX_FLOW, MIN_FLOW; Parametre, der angiver hlv: p og i leddet for PI-reg1, p og i leddet for PI-reg2, samplingstiden, det maksimale flow ind i beholderen og det minimale flow ind i beholderen.

I_MANUEL = (I_MANUEL eller MANUEL) og ikke AUTO;

pid_regulator PI-REG1
  ref = REFERENCE;
  mv = HØJDE;
  pLed = P;
  iLed = I;
  dLed = 0.0;
  samplingstid = SAMPLETID;
  max_output = MAX_FLOW;
  min_output = MIN_FLOW;
  preset = I_MANUEL;
  preset værdi = IND_FLOW;
  ff værdi = UD_FLOW;
  ff_gain = 1.0;
  slut pid_regulator

3pc_regulator PI-REG2
3.5. **KONKLUSION**

```plaintext
ref = PI-REG1.out;
mv = IND_FLOW;
p_led = P1;
i_led = I1;
d_led = 0.0;
samplingstid = SAMPELTID;
manuel = I_Manuel;
op = OP;
ned = NED;
slut 3pc_regulator

VENTIL_OP=PI-REG2.op;
VENTIL_NED=PI-REG2.ned;
```

3.5 **Konklusion**

En passende konklusion på dette kapitel vil være at vise syntaksgrafen for den beskrevne programstruktur, da det vil forklare strukturen i detaljer. Men jeg synes ikke, jeg vil være så konkret da, den beskrevne struktur er et skud fra hoften og derfor ikke skal gøres til genstand for detaljeret beskrivelse. Det er nu op til læseren, om han/hun vil godtage den udviklede programstruktur, lave om i den eller digte sin egen.
Kapitel 4

Udenomskrymmel

I det foregående afsnit er forudsat at man kan sammenknytte symboliske navne og deres fysiske repræsentation (proces-io). Denne sammenknytning kan passende foregå i en instantierings-, initialisering- eller erklæringsdel om man vil, (i det efterfølgende kaldt erklæringsdel). Denne erklæringsdel er placeret før det endelige program.

4.1 Proces-io

Denne del af erklæringsdelen skal sammenknytte de logiske navne med de fysiske io-kanaler dvs. hvilke kort og kanaler der er tale om samt fortælle ind-/udlægningsproceduren hvorledes der skal skelres mellem ADC/DAC's værdi og programmets værdi. (Vi vil ikke arbejde med tal fra 0 til 1024 i programmet.)

Jeg foreslår, at erklæringen, af en analog ind- og udgang, ser således ud:

<NAVN>:  <KÖRT_ID>,<KÖRT_TYPE>,<KANAL_ID>,<MIN_VÆRDI>,<NÅX_VÆRDI>;

Der skal muligvis mere information til for at kunne sammenknytte fysiske io med logiske navne, så tag dette som et udgangspunkt.

Tilsvarende foreslår jeg, at erklæringen, af en digital ind- og udgang, ser således ud:

<NAVN>:  <KÖRT_ID>,<KÖRT_TYPE>,<KANAL_ID>,<POLARITET>;

Med termen <POLARITET> menes, hvis <POLARITET> er sat til +, så betyder det, at der er signal, når størrelsen <NAVN> er sand og hvis <POLARITET> er sat til -, så betyder det, at der ikke er signal, når størrelsen <NAVN> er sand.
4.2  Netværks-import/export

Denne del skal sammenknytte de logiske navne med størrelser, der hhv. importeres eller exporteres over et givet netværk mellem computeren selv og andre proceskontrolcomputer.

En måde kan være at anvende formuleringen i erklæringsdelen:

<NAVN>: import,<COMPUTER ID>,<VARIABEL NAVN>;
<NAVN>: export;

Hvor <VARIABEL NAVN> angiver det <NAVN>, som blev exporteret i computeren <COMPUTER ID>.

Dette bevirker, at man kan importere størrelser fra andre computere under forudsætning af, at den pågældende computer exporterer disse størrelser. De to computere, der snakker sammen, skal således være enige om, hvilke størrelser der overføres. Dette dobbelte check er gjort for at beskytte de enkelte programmer fra omverdenen, således at de enkelte computere ikke kan lave ulykker på hinanden. Der er set forsøgende eksempler på netværkshåndtering. Fx. kan en hver computer, af et kendt dansk fabrikat, kopiere en variabel over i en hvilken som helst variabel i en hvilken som helst anden computer af dette kendte danske fabrikat. Dvs. at det er umuligt at spore, hvor fejlen er, hvis en variabel pludselig, utilgittert skifter værdi. Eller indnu være hvis en computer går "amok" hvad kan der så ikke ske?

4.3  Parametre

Denne erklæring kan anvendes til alle de størrelser, som er interne for computeren, dvs. ikke er proces-iø eller netværks-import/export.

Parametre kan formuleres således:

<NAVN> = 17.3;
<NAVN> = falak;
program → erklæring start handlingsdel
erklæring → sektion | erklæring sektion
sektion → aisektion
    | aosektion
    | disektion
    | dosektion
    | aimpsektion
    | aexpsektion
    | dimpsektion
    | dexpsektion
    | parssektion
    | relaysektion
aisektion → analog_input akanalspes
aosektion → analog_output akanalspes
akanalspes → akanal | akanalspes akanal
akanal → variabel : intval, variabel, intval, realval - realval;
disektion → digital_input akanalspes
dosektion → digital_output akanalspes
dkanalspes → dkanal | dkanalspes dkanal
dkanal → variabel : intval, variabel, intval, polaritet;
polaritet → - | +
aimpsektion → analog_import impkanalspes
aimpspektion → analog_export impkanalspes
aimpspektion → analog_export expkanalspes
impkanalspes → impkanal | impkanalspes impkanal
impkanal → variabel : comp_id, variabel;
comp_id → intval
expkanalspes → expkanal | expkanalspes expkanal
expkanal → variabel;
parssektion → parametre parspec
parspec → parm | parspec parm
parm → dparm | aparm
dparm → variabel = dinit
dinit → on ; | off ;
aparm → variabel = realparm
realparm → realval ; | realval ( realval, realval ) ;
4.3. PARAMETRE

relaysektion → hjælpe_relay variabler

variabler → variabellist ;

variabellist → variabel | variabellist , variabel

handlingsdel → tilskrivning | handlingsdel tilskrivning

blok → pidreg

  | tre_pcreg
  | procedure
  | sekvens

pidreg → pid_regulator : variabel pidsek slut variabel

tre_pcreg → 3pc_regulator : variabel pc3sek slut variabel

pidsek → pidpårm | pidsek pidpårm

pidpårm → ε | ref = reel

  | mv = reel
  | pLed = reel
  | iLed = reel
  | dLed = reel
  | samplingstid = reel
  | manuel = bølsk
  | op = bølsk
  | ned = bølsk
  | max_output = reel
  | min_output = reel
  | preset = bølsk
  | preset_vaerd = reel
  | ff_gain = reel
  | ff_vaerd = reel

pc3sek → pc3pårm | pc3sek pc3pårm

pc3pårm → ε | ref = reel

  | mv = reel
  | iMin = bølsk
  | iMax = bølsk
  | run_time = reel

procedure → procedure : variabel handlingsdel slut variabel
sekvens → sekvens ; variabel sekvensctrl inittilstand tilstandslist slut variabel
sekvensctrl → \( \varepsilon \mid \text{reset} = \text{bolk} \)
   | \( \text{reset} = \text{bolk hold} = \text{bolk} \)
   | \( \text{hold} = \text{bolk} \)
tilstandslist → tilstand \mid tilstandslist tilstand
inittilstand → tilstand \mid init handlingsdel gaa til list slut init
tilstand → tilstand : variabel handlingsdel gaa til list slut variabel
gaa til list → gaa til variabel naar udsagn ; | gaa til init naar udsagn ;
tilskrivning → variabel = udsagn ; | blok | aktiver ( variabel ) ;
udsagn → delay ( udsagn , realval )
   | delay ( udsagn , variabel )
   | udsagn og udsagn
   | udsagn eller udsagn
   | ( udsagn )
   | ikke udsagn
   | variabel
   | on
   | off
   | blink
reel → variabel ; | realval ;
bolk → variabel ; | on ; | off ;
variabel → letter tegnlist
tegnlist → \( \varepsilon \mid \text{tegn} \mid \text{tegnlist tegn} \)
tegn → letter \mid digit
letter → [a-zA-Z][_\-].
digit → [0123456789]
realval → \(((0-9)*\.[0-9]+)([eE][-+]?[0-9]+)?\) \"Se LEX\"
intval → [0-9]+ "Se LEX"
LEX programmet:

```c
%
#include "y.tab.h"
#include <string.h>
#include <math.h>
#include "pp.h"
extern int lineno;
%

Ws [ \t]+
comment #[^\n]*[^\n]
QString "[^\n]*[^\n]
intval [0-9]+
realval (([0-9]\.\[0-9]+)\([eE][-+]?[0-9]+\))
variable [a-zA-Z][a-zA-Z0-9_]*
ln \n
%

{ws} ;
{comment} ;

analog_input {return ANALOGIN;}
analog_output {return ANALOGOUT;}
digital_input {return DIGITALIN;}
digital_output {return DIGITALOUT;}
analog_import {return ANALOGIMPORT;}
analog_export {return ANALOGEXPORT;}
digital_import {return DIGITALIMPORT;}
digital_export {return DIGITALEXPORT;}
parametre {return PARAMETRE;}
hjælpereelay {return RELAY;}
start {return START;}
oeg {return OG;}
eeller {return ELLER;}
ikeks {return IKKE;}
delay {return DELAY;}
sekvens {return SEKVENS;}
reset {return RESET;}
hold {return HOLD;}
tilstand {return TILSTAND;}
init {return INIT;}
gaa {return GAA;}
til {return TIL;}
naar {return NAAR;}
slut {return SLUT;}
on {return ON;}
```
off {return OFF;}
procedure {return PROCEDURE;}
aktiver {return AKTIVER;}
blink {return BLINK;}
pid_regulator {return PIDREG;}
ref {return REF;}
mv {return MV;}
p_led {return PLED;}
i_led {return ILED;}
d_led {return DLED;}
max_output {return MAXOUT;}
min_output {return MINOUT;}
manuel {return MANUEL;}
op {return OP;}
ned {return NED;}
preset {return PRESET;}
preset_værdi {return PRESETV;}
ff_værdi {return FFV;}
ff_gain {return FFG;}
3pc_regulator {return PC3;}
max {return IMAX;}
min {return IMIN;}
run_time {return RUNTIME;}

{intval} {yyival.ival = atoi(yytext);
return INTVAL;}

{realval} {yyival.fval = atof(yytext);
return REALVAL;}

{variabel} { yyival.symb = insert_sym(yytext, VARIABEL);
if (yyival.symb == NULL) {
    yyival.symb = lookup_sym(yytext, VARIABEL);
    yyival.symb->nacc++;
} return VARIABEL;
}

{nl} {lineno++;}

{return *yytext;}

%%

printerror(int n)
{
    yyerror(errortxt(n));
4.3. PARAMETRE

}

yyerror(const char *msg)
{
    printf("Line% d: %s at < %s > \n", lineno, msg, yytext);
}

YACC programmet:

%
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "pps.h"

extern FILE *yyin, *yyout;

struct symbol *symptr;
struct symtab *tmpptr[500];
struct symtab *dummyptr;
char string[80];
int tmpindex = 0;
%
union {
    char *string;
    double fval;
    int ival;
    struct symtab *symb;
}
%
token <#symb> VARIABEL
%token ANALOGIN ANALOGOUT DIGITALIN DIGITALOUT
%token ANALOGIMPORT ANALOGETXPORT DIGITALIMPORT DIGITALEXPORT
%token PARAMETRE RELAY
%token OG ELLER IKKE DELAY ON OFF BLINK
%token SEKVENS RESET HOLD TILSTAND INIT GAA TIL NAAR START SLUT
%token PROCEDURE AKTIVER
%token PIDREG
%token REF MV PLED ILED ILED Sampeltid Maxout Minout
%token MANUEL OP NED PRESET PRESE1 PV PV FFG FG3 IMAX IMIN RUNTIME
%token <fval> REALVAL
%token <ival> INTVAL
%
%left OG
%left ELLER
%right IKKE

%start program

%%

program:   erklæring START handlingedel;

erklæring: sektion | erklæring sektion;

sektion:  aisektion
             | aosektion
             | disektion
             | dosektion
             | aimpsektion
             | aexpsektion
             | dimpektion
             | dexpsektion
             | parmsektion
             | relaysektion;

aisektion: ANALOGIN akanalspes
{for (; tmpindex > 0; tmpindex--) {(tmpptr[tmpindex - 1])->type = ANALOGIN;}};

aosektion: ANALOGOUT akanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=ANALOGOUT;}};

akanalspes: akanal | akanalspes akanal;

akanal:   VARIABEL ":" INTVAL "," VARIABEL "," INTVAL "," REALVAL "," REALVAL ";"
{if ($1->nacc > 0) printerror(1);tmpptr[tmpindex++]=$1;};

disektion: DIGITALIN dkanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=DIGITALIN;}};

dosektion: DIGITALOUT dkanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=DIGITALOUT;}};

dkanalspes: dkanal
             | dkanalspes dkanal;

dkanal:   VARIABEL ":" INTVAL "," VARIABEL "," INTVAL "," polaritet ";"
{if ($1->nacc > 0) printerror(1);tmpptr[tmpindex++]=$1;};
polaritet: "-" | "+"

aiimpsektion: ANALOGIMPORT impkanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=ANALOGIMPORT;}};
dimsektion: DIGITALIMPORT expkanal
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=DIGITALIMPORT;}};

aexpsektion: ANALOGEXPORT expkanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=ANALOGEXPORT;}};
dexpsektion: DIGITALEXPORT expkanalspes
{ for (;tmpindex > 0;tmpindex--) { tmpptr[tmpindex-1]->type=DIGITALEXPORT;}};

impkanalspes: impkanal
| impkanalspes impkanal;

impkanal: VARIABEL '"' comp_id '"' VARIABEL '"''
{if ($1->nacc > 0) printerror(1);tmpptr[tmpindex++]=$1;};

comp_id: INTVAL;

expkanalspes: expkanal
| expkanalspes expkanal;

expkanal: VARIABEL '"''
{if ($1->nacc > 0) printerror(1);tmpptr[tmpindex++]=$1;};

parmsektion: PARAMETRE parmspec;

parmspec: parm
| parmspec parm;

parm: dparm
| aparm;

dparm: VARIABEL '=' dinit
{if ($1->nacc > 0) printerror(1); $1->type=DIGITALPARM;};

dinit: ON '"''
| OFF '"'';

aparm: VARIABEL '=' realparm
{if ($1->nacc > 0) printerror(1); $1->type=ANALOGPARM;};

realparm: REALVAL '"''
| REALVAL '(' REALVAL ',' REALVAL ')' '"'';

relaysektion: RELAY variabler;

variabler: variabellist '"'';

variabellist: VARIABEL
{if ($1->nacc > 0) printerror(1); $1->type=RELAY;}
| variabel\text{list \ '}, \text{VARIABEL}
{\text{if ($3->nacc > 0) printerror(1); $3->t\text{ype}= RELAY;}};

\text{handlings\text{del: til\text{skrivning}}}
| \text{handlings\text{del til\text{skrivning}};}

\text{blok: pid\text{reg}}
| \text{tre_pcreg}
| \text{procedure}
| \text{sekvens};

\text{pid\text{reg: PIDREG ":" VARIABEL pid\text{sek SLUT VARIABEL}}
{\text{if ($3!=6) printerror(3);}}
\text{strcpy(string,$3->name);}
\text{strcat(string,"\text{,out}"};
\text{if ($3->nacc != 1) printerror(4);}
\text{if (((\text{dummyp\text{tr= lookup_sym(strdup(string),ANAL\text{OGIN}}})!= NULL}}
\text{printerror(11);}
\text{else}
\text{dummyp\text{tr=insert_sym(strdup(string),ANAL\text{OGIN}); }}

\text{tre_pcreg: PC3 ":" VARIABEL pc3\text{sek SLUT VARIABEL}}
{\text{if ($3!=6) printerror(5);}}
\text{if ($3->nacc != 1) printerror(6);}}
\text{strcpy(string,$3->name);}
\text{strcat(string,"\text{,op}"};
\text{if (((\text{dummyp\text{tr= lookup_sym(strdup(string),DIGITALIN}}})!= NULL}}
\text{printerror(11);}
\text{else}
\text{dummyp\text{tr=insert_sym(strdup(string),DIGITALOUT});}
\text{strcpy(string,$3->name);}
\text{strcat(string,"\text{,ned}"};
\text{if (((\text{dummyp\text{tr= lookup_sym(strdup(string),DIGITALIN}}})!= NULL}}
\text{printerror(11);}
\text{else}
\text{dummyp\text{tr=insert_sym(strdup(string),DIGITALOUT}); }$

\text{pid\text{sek: pidparm | pid\text{sek pidparm};}}

\text{pidparm : | REF"=" reel}
| \text{MY "n" reel}
| \text{PLED "m" reel}
| \text{ILED "n" reel}
| \text{DLED "n" reel}
| \text{SAMPELTID "n" reel}
| \text{MANUEL "m" bolsk}
| \text{OP "m" bolsk}
4.3. PARAMETRE

```
| NED  "=" bolsk |
| MAXOUT  "=" reel |
| MINOUT  "=" reel |
| PRESET  "=" bolsk |
| PRESETV  "=" reel |
| FFQ  "=" reel |
| FFV  "=" reel; |
```

c3sek: c3parm | c3sek c3parm;

c3parm: | REF="=" reel |
| MY  "=" reel |
| IMIN  "=" bolsk |
| IMAX  "=" bolsk |
| RUNTIME  '=' reel; |

```
procedure: PROCEDURE ':' VARIABEL handlingsdel SLUT VARIABEL
{ if ($3!=86) printerror(12); $3->type= PROCEDURE;
  if ($3->nacc != 1) printerror(13);}
```

```
sekvens: SEKVENS ':' VARIABEL sekvensctrl initstand tilstandslst SLUT VARIABEL
{ if ($3!=88) printerror(14); $3->type= SEKVENS;
  if ($3->nacc != 1) printerror(15);}
```

```
sekvensctrl: | reset  '=' bolsk |
| reset  '=' bolsk hold  '=' bolsk |
| hold  '=' bolsk ; |
```

```
tilstandslst: tilstand | tilstandslst tilstand;
```

```
initstand: TILSTAND ':' INIT handlingsdel gaatilist SLUT INIT ;
```

```
tilstand: TILSTAND ':' VARIABEL handlingsdel gaatilist SLUT VARIABEL
{ if ($3!=87) printerror(16);
  if (($3->nacc >= 1) & ( $3->type != TILSTAND)) printerror(17);
  if ($3->nacc == 0) $3->type = TILSTAND;}
```

```
gaatilist: gaatil | gaatilist gaatil ;
```

```
gaatil: GAÅ TIL VARIABEL NÅR udsagn ';
{if (($3->nacc >= 1) & ( $3->type != TILSTAND)) printerror(17);
  if ($3->nacc == 0) $3->type = TILSTAND;}
  GAÅ TIL INIT NÅR udsagn ';
```

```
tilskrivning: VARIABEL "=" udsagn ";
```
{if ($1->nacc == 0) printerror(2);}

| blok
| AKTIVER '(' VARIABEL ')';

udsagn: DELAY "(" udsagn "," REALVAL")"
| DELAY "(" udsagn "," VARIABEL")"
{if ($5->nacc == 0) printerror(2);}

| udsagn OG udsagn
| udsagn ELLER udsagn
| "(" udsagn ")"
| IKKE udsagn
| VARIABEL
{if ($1->nacc == 0) printerror(2); checkdigital($1);}

| ON
| OFF
| BLINK ;

reel: VARIABEL ';
{if ($1->nacc == 0) printerror(7); checkanalog($1);}

| REALVAL ';

bolsk: VARIABEL ';
{if ($1->nacc == 0) printerror(8); checkdigital($1);}

| ON ';
| OFF ';

%%

char *progname = "fuzzy";
int lineno = 1;
#define DEFAULT_OUTFILE "fuzzy.out"
char *usage = "%s: usage [infile] [outfile]\n";

main(int argc, char **argv)
{
char *outfile;
char *infile;
progname = argv[0];
if (argc > 3) {
    fprintf(stderr, usage, progname);
    exit(1);
}
if (argc > 1) {
    infile = argv[1];
    yyin = fopen(infile, "r");
4.3. PARAMETRE

if (yyin == NULL) {
  fprintf(stderr, "%s: cannot open %s\n", proname, infile);
  exit(1);
}
}

if (argc > 2) {
  outfile = argv[2];
} else {
  outfile = DEFAULT_OUTFILE;
}

yyout = fopen(outfile, "w");
if (yyout == NULL) {
  fprintf(stderr, "%s: cannot open %s\n", proname, outfile);
  exit(1);
}

yyparse();
}

warning(char *s, char *t)
{
  fprintf(stderr, "%s: %s", proname, s);
  if (t)
    fprintf(stderr, "%s", t);
  fprintf(stderr, " line %d\n", lineno);
}

/******
*       struct symtab *insert_sym(char *str, int type)
*     Indsætter et symbol i symboltabellen og returnere
*     en pointer til den struct som indeholder symbolet.
*     *******/

struct symtab *insert_sym(char *str, int type)
{
  struct symtab *sp;
  int cnt;
  for (cnt=0; cnt < Nsymbol; cnt++) {
    sp = &symtab[cnt];
    if (sp->name && !strcmp(sp->name, str))
      return NULL;
    if (!sp->name) {
      sp->name = strdup(str);
      sp->type= type;
      sp->nacc= 0;
      return sp;
    }
    if (sp->name) {
      strcpy(sp->name, str);
      sp->type= type;
      sp->nacc= 0;
      return sp;
    }
  }
  return NULL;
}
} } 
warning("symboltable overflow", str);
return NULL; 
} 

/****************************************************
* struct symtab *lookup_sym(char *str, int type)
* 
* Underbøger om et symbol er i symboltabellen og hvis 
* symbolet eksisterer returnere en pointer til den struct 
* som indeholder symbolet ellers returnere NULL pointeren. 
* 
****************************************************************************/
struct symtab *
lookup_sym(char *str, int type)
{
struct symtab *sp;
int cnt;
for (cnt = 0; cnt < Nsymbol; cnt++) {
sp = &symtab[cnt];
if (sp->name && !strcmp(sp->name, str))
return sp;
}
if (!sp->name) {
return NULL;
}
return NULL;
} 

/****************************************************
* Semantisk analyse.
****************************************************************************/
checkdigital(struct symtab * p)
{
if (p->type != DIGITALIN && 
p->type != DIGITALOUT && 
p->type != DIGITALIMPORT && 
p->type != DIGITALEXPORT && 
p->type != DIGITALPARAM && 
p->type != RELAY ) printerror(9);
}

/****************************************************
* Semantisk analyse.
****************************************************************************/
checkanalog(struct symtab * p)
{
    if (p->type != ANALOGIN &&
        p->type != ANALOGOUT &&
        p->type != ANALOGIMPORT &&
        p->type != ANALOGEXPORT &&
        p->type != ANALOGFARM ) printerror(10);
}

Litteratur


