CODEN:LUTFD2/(TFRT-5317)/1-122/(1984)

GRAFISK PRESENTATION OCH EDITERING AV MATEMATISKA UTTRYCK OCH RELÄSCHEMA

MAGNUS TAUBE

DEPARTMENT OF AUTOMATIC CONTROL
LUND INSTITUTE OF TECHNOLOGY

DECEMBER 1984

används som analoga inenheter för att styra zoomning, scrollning och panorering av bilden.

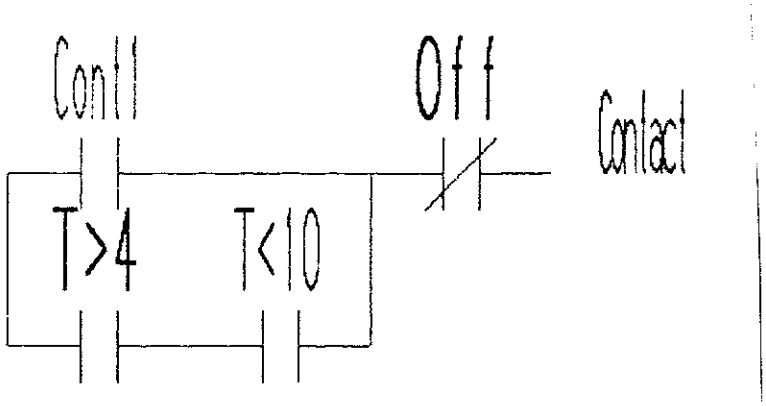
Examensarbetet har bestått av tre olika huvuddelar, där de två första delvis använder samma programvara, medan den tredje är fristående. I den första delen har en implementation för interaktiv inmatning och presentation av matematiska uttryck gjorts, i den andra presenteras logiska uttryck som reläschema och den tredje delen omfattar en implementation för bildtolkning av reläscheman. All programvara som har skrivits i examensarbetet har skrivits i pascal.

Till den färgskärm som har använts har det inom LICS-projektet skapats en mängd grafiska primitiver som har använts, dessutom härstammar en del program från Dymola, det gäller scanner, parser, uttrycks-rutiner etc. En del av dessa rutiner har dock modifierats och utökats för att även hantera den grafiska information som behövs.

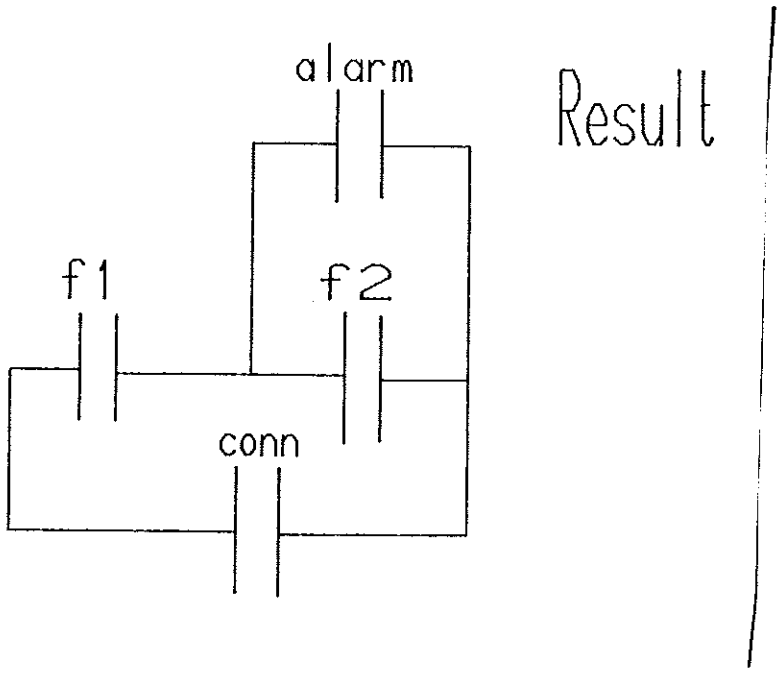
Exempel:
Den första delen av examensarbetet ger möjlighet att skriva ut och editera formeln
(1+x)**0.5 = 1 + x/2 - x**2/8 + 5*x**3/16 - 35*x**4/128 som

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{5*x^3}{16} - \frac{35*x^4}{128}$$

Den andra delen ger möjlighet att rita ut följande reläschema till uttrycket Contact=(Cont1 or (T>4 and T<10)) and not Off



Bildtolkningen i den tredje delen tolkar följande reläschema som uttrycket Result=conn or f1 and (f2 or alarm) .



2 Använt grafiskt system

I detta kapitel tas de grafiska primitiver som har använts i examensarbetet upp, och en kort beskrivning av det bakomliggande grafiska systemet.

Grafikrutinerna är samtliga utvecklade inom LICS-projektet på Institutionen för reglerteknik, LTH. Grafikrutinerna är skrivna i pascal, med vissa små assembler-delar för skärmuppdatering och dylikt.

Som bas finns ett universellt koordinatsystem, och skärmen kan ses som ett kikhål ner i detta. Genom att ha olika skala kan man också åstadkomma zoomning, dvs visa bilden i större eller mindre skala. Eftersom denna zoomning sker i reell tid får man samma intryck som om man befunnit sig i en helikopter som stiger eller sjunker över bilden.

I detta koordinatsystem kan man rita linjer, skriva text etc. För att hantera var linjer och text skall placeras har man tillgång till två olika cursors. Den ena gäller för linjedragning, medan den andra används för teckenutmatning. I det följande kallas dessa för bildcursor respektive textcursor. Det bör observaras att ingen av dessa cursors syns på skärmen, de är endast logiska koordinatpekare.

En kort beskrivning av de grafiska primitiver som har använts följer nedan:

```
MoveTo(x, y : real);
```
MoveTo flyttar bildcursorn till punkten x,y.

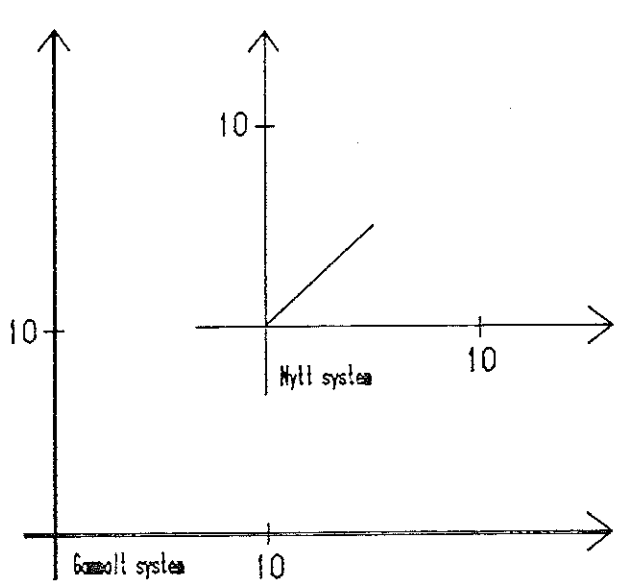
```
LineTo(x, y : real);
```
LineTo drar en rak linje från bildcursorns position till punkten x,y.

```
Translate(x, y : real)
```
Translate flyttar det aktuella koordinatsystemets origo till punkten x,y. Alla koordinater i fortsättningen gäller i det nya koordinatsystemet.

Exempel:

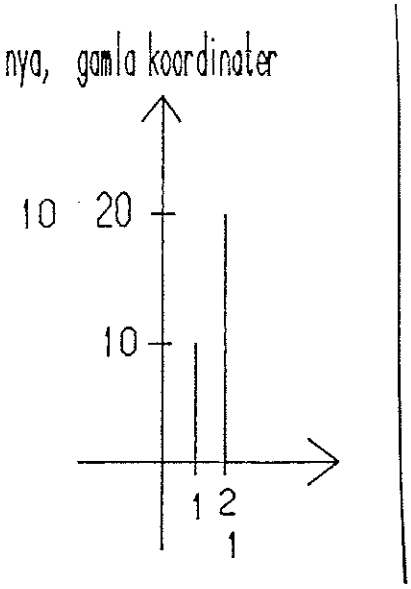
```
Translate(10,10);
MoveTo(0,0);
LineTo(3,4);
```

kan illustreras på följande sätt: (Dubbla linjer är det ursprungliga koordinatsystemet)




```
Scale(xscale, yscale : real);
```
Scale ändrar skalan för koordinatsystemet.

Exempel:
```
MoveTo(1,0);
LineTo(1,10);
Scale(2,2);
MoveTo(1,0);
LineTo(1,10);
```
ger följande bild:




```
SaveTransform(var T : TransformDescriptor);
```
Savetransform sparar undan den aktuella koordinatsystemet och skalan i variabeln T.

```
SetTransform(T : TransformDescriptor);
```
Settransform sätter tillbaka koordinatsystemet till det som gällde vid den tidpunkt man anropade SaveTransform med samma variabel.

```
ClearScreen;
```
Suddar hela bilden.

SetColor(color : integer);
Sätter den färg som all utmatning fram tills nästa anrop av SetColor skall ha.

SetCursor(x, y : real);
Sätter text-cursorn till angiven punkt. Detta anrop har ingen effekt på bild-cursorn.

CharSize(xsize, ysize : real);
Anger med vilken storlek i det viruella koordinatsystemet som tecken skall genereras. Denna storlek bestämmer alltså inte tecknens fysiska storlek i pixel, utan den bestämms av den aktuella skalfaktorn.

DrawChar(ch : char);
Skriv ut tecket ch på text-cursorns koordinater. Dessutom uppdateras textcursorn i x-led, så att två DrawChar-anrop efter varandra ger två tecken i bredd på skärmen.

2.1 Pan, Scroll och Zoom

Eftersom skärmen normalt inte kan visa hela koordinatsystemet finns det rutiner för att förflytta skärmen över bilden.

Pan(dist : real);	Flyttar skärmen över koordinatsystemet i x-led.
Scroll(dist : real);	D:o fast i y-led.
Zoom(fact : real);	Skalfaktorn för bilden ändras. ökas skalan ser man mer av bilden, fast i större skala, och vice versa.

2.2 Interaktion

Det finns tre inenheter som används i det grafiska systemet, tangentbordet, två stycken styrspakar och en mus med tre knappar. Styrspakarna används för att styra panorering, scrollning och zoomning, och musen för att styra bildcursorn. De tre knapparna på musen kan användas till olika saker beroende på vilket program man kör. I examensarbetet har ett befintligt paket för interaktion (av Hilding Elmqvist) modifierats något. Paketets utseende utåt listas nedan:

```
type InteractType = (InteractChar, InteractCursor, InteractOther,
                     InteractButton1Up, InteractButton1Down,
                     InteractButton2Up, InteractButton2Down,
                     InteractButton3Up, InteractButton3Down);
var  InteractCX,
     InteractCY : real;    { Flyttning av musen }
function Interact(var ch : char) : InteractType;
   { Returnerar InteractCursor om musen har rörts.                  }
   { Returnerar InteractChar om något tecken har matats in på       }
   {            tangentbordet. ch innehåller då tecknet.            }
   { Returnerar InteractButton(n) Up/Down om någon av knapparna     }
   {            på musen rörts.                                     }
   { Returnerar InteractOther om någon av spakarna rörts.           }
```

När någon av spakarna rörts (Interact returnerar InteractOther) har skärmen flyttats över bilden, eller skalan har ändrats. Detta syns dock inte innan det anropande programmet har genererat om bilden. När musen har rörts returneras InteractCursor och de globala variablerna InteractCX och InteractCY innehåller värdet av den flyttning som gjorts.

2.3 Speciella kommandon

Slutligen skall nämnas några specialrutiner som också används. De har mer med den aktuella hårdvaran att göra och är inte så generella som de ovanstående rutinerna.

Den hårdvara som f n används kan lagra två stycken bilder på samma gång, men bara en bild visas på skärmen. Detta kan man använda när man skapar en ny bild på så sätt att man genererar den nya bilden i det dolda bildplanet, när man är klar byter man bild. Fördelen är att ögat inte hinner uppfatta bildbytet, därigenom ser man aldrig när en bild genereras.

Följande rutiner används för att styra detta:

TwoPlanes;	Initiering av faciliteten.
DrawBackground;	Samtliga kommandon i fortsättningen skall ha effekt på det dolda bildplanet.
SwapPlane;	Byt bildplan. Dvs det tidigare dolda bildplanet blir synligt och v v.

Dessutom finns det rutiner för att kunna få ut bilden på en skrivare. Detta styrs med följande rutiner:

OKIScreen;	All utmatning på den grafiska skärmen konverteras till OKI-skrivarens format.
PrintVirtualScreen;	Den bild som genererats sedan anropet av OKIScreen matas ut på skrivaren.
CMIScreen;	Återställer den grafiska utmatningen.

3 Matematiska uttryck

3.1 Editering och presentation av matematiska uttryck

Huvuddelen av examensarbetet har bestått i att möjliggöra interaktiv inmatning och editering av matematiska och logiska uttryck. Detta skall ses som en specialfunktion i en texteditor, där man alltså skall kunna mata in uttryck och få dem presenterade med konventionell matematisk notation.

Matematiska uttryck som skall behandlas i datorer, såväl i program som i texteditorer skrivs i dag som regel på en rad, t ex skrivs upphöjt till som ** och division som /.

Uttryckseditorn skriver ut ett uttryck med konventionell matematisk (och mera lättläst) representation.
Exempel:
a = b / (2 + x) * fact ** 0.5 skrivs ut som

$$a=\frac{b}{2+x}*\sqrt{fact}$$

3.1.1 Inmatning

Vid inmatning av matematiska uttryck kan man tänka sig två olika huvudmetoder, linjär och grafisk inmatning. Linjär inmatning innebär att uttrycket matas in på en rad där parenteser anger det strukturella utseendet, medan grafisk inmatning innebär att man på något sätt, t ex med piltangenter eller mouse, flyttar cursorn till den plats på skärmen där nästa tecken skall skrivas.

Med grafisk inmatning skulle uttrycket $\frac{a}{b + c}$ matas in med följande sekvens:

Placera cursorn på platsen för täljaren.
Skriv täljaren "a".
Placera cursorn på platsen för /-linjen.
Skriv /, eller dra linjen.
Placera cursorn på platsen för nämnaren.
Skriv nämnaren "b + c".

Fördelen med grafisk inmatning är att den i mycket liknar den inmatning man har till en grafisk editor, vilket skulle göra den naturlig i ett sådant system. En nackdel med grafisk inmatning kan vara att bilden av ett uttryck ofta innehåller en stor mängd deluttryck (operatorer och operander), varför det kan vara svårt att editera upp en korrekt bild. Särskilt gäller detta vid editering av uttrycket då kanske många deluttryck måste flyttas.

T ex innehåller uttrycket

$$result = \frac{a}{k+g^2}$$

9 stycken enheter. Skall det ändras till

$$result = \frac{a}{kvot+g^2}$$

så måste ledet fr o m + -tecknet flyttas, divisionstecknet förlängas och dessutom a:et omcentreras över divisionslinjen. Hur komplicerad denna editering är beror naturligtvis på den editor man använder och problemen skall inte överskattas. Det är dessutom svårare att analysera uttryck som är grafiskt inmatat, särskilt när uttrycket innehåller operatorer och deluttryck på olika nivåer i höjdled, och med olika skala.

Med linjär inmatning måste i stället den strukturella informationen, dvs information om vilka deluttryck operatorerna skall arbeta på överföras med hjälp av parenteser (och med prioritetsregler). Detta ger att uttrycket

$$\frac{a}{b+c}$$

vid linjär inmatning matas in som a/(b+c), där parenteserna anger vilka deluttryck operatorn (här /) skall verka på. Med linjär inmatning blir alla uttryck som behandlas utskrivna efter samma, enhetliga, principer.

I den implementerade uttryckseditorn används linjär inmatning. Skälen för det är flera, man kan då använda en konventionell parser, linjär inmatning anknyter till den inmatning som är den normala i en texteditor, implementeringen förenklas, och slutligen slipper man alla problem med centrering etc, eftersom detta helt hanteras av editorn.

3.2 Interaktiv uppbyggnad av uttryck

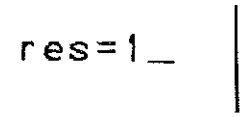
Uttryckseditorn kan arbeta dels interaktivt och dels icke interaktivt. För att ge en bild av hur resultatet blir skall i detta avsnitt visas några inledande exempel på hur det ser ut när uttryck byggs upp interaktivt.

I interaktiv mode hanteras en cursor på färgskärmen, denna visas normalt som ett streck under aktuellt tecken, men om det aktulla tecknet är ett divisionsstreck eller en * som ingår i en upphöjt till operator ritas i stället en pil på operatorn.

3.2.1 Exempel på interaktiv uppbyggnad av uttryck

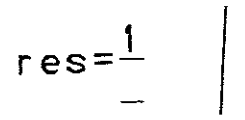
Exempel 1:
Det första exemplet är ett kedjebråk. Till vänster visas vad som har matats in, och till höger vad som syns på färgskärmen.

```
res=1_
```



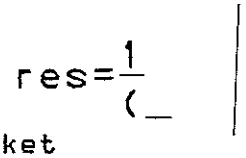
Linjen efter ettan är cursorn. Nu matas divisionstrecket in

```
res=1/_
```



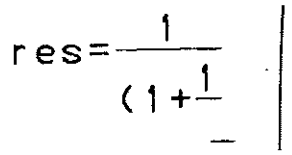
Sedan en vänsterparentes

```
res=1/(_
```



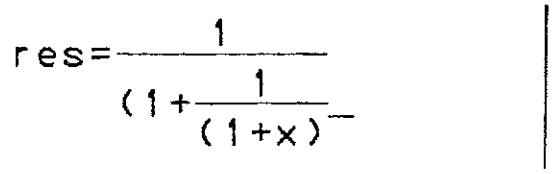
Efter ytterligare tecken är uttrycket

```
res=1/(1+1/_
```



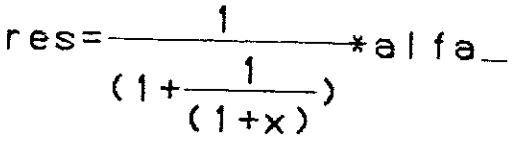
Ett nytt divisionstreck. Observera att det övre divisionstrecket blir längre när nämnaren växer, liksom att täljaren hela tiden är centrerad. När den nya nämnaren är klar har uttrycket vuxit till

```
res=1/(1+1/(1+x)_
```



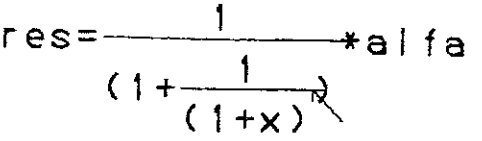
Det färdiga uttrycket ser ut så här

```
res=1/(1+1/(1+x))*alfa_
```



Om man använder piltangenterna kan man flytta cursorn genom uttrycket. Är det aktuella tecknet ett vanligt tecken ritas ett streck under det, men om det är ett divisionsstreck ritas i stället en pil.

```
res=1/(1+1/(1+x))*alfa
```



Exempel 2:
Även detta exempel visar centrering kring divisionstreck, i den vänstra bilden har /-tecknet precis matats in.

$$x=\frac{alfa*(k1+k3)}{\_} \qquad x=\frac{alfa*(k1+k3)}{(1+k2)}\_$$

Exempel 3:
I nästa exempel har tangens definierats, men n-et i sin(x) har inte kommit med.

$$tan(x)=\frac{si(x)}{cos(x)}\_$$

Genom att med piltangenterna förflytta cursorn till den plats där n-et skall stå och därefter mata in det är felet korrigerat.

$$tan(x)=\frac{si(x)}{cos(x)} \qquad tan(x)=\frac{sin(x)}{cos(x)}$$

Exempel 4:
Upphöjt till symbolen (**) är lite speciell, eftersom den som delmängd har multiplikationsoperatorn (*). Detta framgår av följande exempel. I den första bilden är en * inmatad,

```
sqrt=x*_
```

$$sqrt(x)=x*\_$$

när den andra *-an matats in flyttas cursorn upp ett halvsteg och *-orna syns inte.

```
sqrt=x**_
```

$$sqrt(x)=x^{\_}$$

Exponenten är inmatad och skrivs ut med mindre teckenstorlek.

```
sqrt=x**(1/2)_
```

$$sqrt(x)=x^{(\frac{1}{2})}\_$$

Exempel 5:
När man inte editerar uttrycket skrivs det ut utan parenteser, eller med kvadratrotstecken ifall exponenten är 0.5.

```
sqrt=x**(1/2)
```

$$sqrt(x)=x^{\frac{1}{2}}$$

```
sqrt=x**0.5
```

$$sqrt(x)=\sqrt{x}$$

Exempel 6:
Vid editering av upphöjt till operatorer syns inte *-orna, utan cursorpositionen markeras med pilar i stället. När cursorn befinner sig under den vänstra *-an ritas en horisontell pil ut, och under den högra ritas en diagonal pil. Exemplet nedan illustrerar alltså en cursorförflyttning åt vänster.

```
x**(1/2)
```

$$sqrt(x)=x^{(\frac{1}{2})}$$

```
x**(1/2)
```

$$sqrt(x)=x^{(\frac{1}{2})}$$

```
x**(1/2)
```

$$sqrt(x)=x^{(\frac{1}{2})}$$

Exempel 7:
Exponentiering kan också ske i flera led, som det sista exemplet visar, i bilden nederst är parenteserna borttagna.

$$Rett^{(tva^{(tre^{(fyra)})})}$$

$$Rett^{tva^{tre^{fyra}}}$$

3.3 Principer och överväganden vid implementeringen

3.3.1 Regler för uttryck

De uttryck som kan representeras och behandlas i uttryckseditorn följer i stort den notation och de prioritetsregler som gäller i traditionella programspråk, t ex Fortran och Simula. Dvs följande operatorer, notation och prioritetsordning används: (högst prioritet överst)

Operation	operator
funktion	functionsnamn(arg[,arg ..])
upphöjt till	**
negation och unärt plus	- resp +
division och multiplikation	/ resp *
subtraktion och addition	+ resp -
större (mindre) än	> resp <
tilldelas	=
logisk invers	not
och	and
eller	or
villkor	if .. then .. else

- Som grundenheter finns identifierare, numeriska konstanter och operatorer.

- Uttryck och identifierare kan vara logiska eller numeriska, dock sker ingen typkontroll av uttryck, det upptäcks t ex inte att följande uttryck är felaktigt: a=b + c>d.

- Två operatorer får inte följa efter varandra, utan måste i så fall omges med parenteser. Med parenteser kan man också ändra beräkningsordningen.

- Identifierare inleds med en bokstav (a..z,A..Z) och följs av 0 till 14 bokstäver eller siffror.

- Numeriska konstanter kan anges med eller utan decimalpunkt och med eller utan 10-potens. Värdet 1000 kan alltså bl a uttryckas som 1000, 1000.0, 1E3 eller 1.0E3.

Exempel på korrekta uttryck:

```
res=if larm then 0 else niva
tan(x)=sin(x)/cos(x)
a=(b+(-c*d))**2
a=10E4
Open=Temp>10 and not Alarm
```

Exempel på felaktiga uttryck:

7.4*-term	Två operatorer i rad
res=56.56.56	Felaktig numerisk konstant
res=56 fact	Utelämnad operator

3.3.2 Förenkling av uttryck

Ett avgörande man måste göra vid presentation av uttryck är hurvida uttrycken skall förenklas och/eller överflödig information skall tas bort. Den lösning som valts innebär att editorn aldrig förenklar ett uttryck, det inmatade uttrycket -(-x/1) förenklas alltså inte till bara x. Orsaken är att många uttryck av tradition skrivs på ett visst sätt, och att det kan vara svårt att känna igen sina uttryck när de blivit matematiskt bearbetade.

Ett uttryck kan dessutom innehålla redundant information, t ex extra parenteser och extra blanktecken, t ex är parenteserna i uttrycket a+(b+c) redundanta, eftersom uttrycket är ekvivalent med a+b+c. Det normala är dock att parenteser överför strukturell information, d v s de anger på vilka deluttryck som en operator skall arbeta, t ex i uttrycket a/(b+c) skall divisionsoperatorn arbeta med a som täljare och med b+c som nämnare.

Uttryckseditorn kan ta bort redundanta blanktecken och parenteser och även strukturella parenteser som blivit överflödiga när uttrycket behandlats.

Exempel: (a+ b)/c kan skrivas ut som $\frac{(a+\ b)}{c}$ eller som $\frac{a+b}{c}$

där i det andra fallet redundanta strukturella parenteser och blanktecken har tagits bort.

Det bör dock observeras att alla strukturella parenteser inte kan elimineras, exempelvis är parenteserna i uttrycket a*(b+c) strukturella, men inte överflödiga.

Om man tar bort redundant information innebär det att inte alla inmatade tecken syns på skärmen, detta upplevdes som störande vid provkörningar, särskilt vid korrigering och editering i uttryck. För att lösa problemet kan editorn arbeta i två moder, en interaktiv där samtliga tecken visas på skärmen, och en icke interaktiv där de överflödiga tecknen undertrycks. Eftersom den interaktiva moden främst är avsedd för editering ritas dessutom en textcursor ut, som anger aktuell teckenposition.

En ytterligare skillnad mellan de båda moderna är att kvadaratrotsuttryck skrivs ut med rottecken i icke interaktiv mode, medan de i interaktiv mode skrivs ut med potens, detta för att exempelvis uttrycket x**0.51 vid inmatning annars skulle ge upphov till sekvensen:

$x^{0.}$ $\sqrt{x}$ $x^{0.51}$

Exempel på uttryck : (cursorn är _-strecket)

a = (2 + x)/(3 + x) och res = (k) - x**0.5

skrivs ut som

$$a=\frac{2+x}{3+x} \qquad res=k-\sqrt{x}$$

i icke interaktiv mode och som

$$a = \frac{(2 + x)}{(3 + x)} \qquad res = (k) - x^{0.5}$$

i interaktiv mode.

3.3.3 Felaktiga och ofullständiga uttryck

Ett uttryck kan vara felaktigt på i grunden två olika sätt. Dels kan det bryta mot de regler som gäller, exempelvis innehålla två operatorer i rad, exempelvis a+-k, dels kan det vara ofullständigt, dvs uttrycket är inte komplett, exempelvis a=k* är inget korrekt uttryck, men om det kompletteras till a=k*5 så är det riktigt.

Utryckseditorn hanterar felaktiga uttryck genom att alla tecken från och med felet skrivs ut med röd färg. Någon behandling av uttrycket fr o m felet sker inte, utan den inmatade texten skrivs bara rakt ut. Ofullständiga uttryck i interaktiv mode betraktas inte som felaktiga, interaktiv mode används ju vid inmatning och editering, medan i icke interaktiv mode ritas en röd fyrkant ut efter ofullständiga uttryck.

Exempel: Det felaktiga uttrycket a=d/(k*-5)-a**0.5 skrivs ut som

$$a=\frac{d}{(k*-5)-a**0.5}$$

(Understrukna tecken skrivs med rött)

Exempel: Det ofullständiga uttrycket a=taljare/n * skrivs ut som

$$a=\frac{taljare}{n}*\_$$

i interaktiv mode (_ är textcursorn) och som

$$a=\frac{taljare}{n}*\square$$

i icke interaktiv mode (observera fyrkanten).

3.3.4 Utryck med distribuerade operatorer

Vid uppbyggnad av den grafiska informationen och vid presentation av uttrycket utgör distribuerade operatorer ett särskilt problem. Distribuerade operatorer är sådana som utgörs av mer än ett tecken. De operatorer som berörs i uttryckseditorn är

villkor (if ... then ... else)
funktioner
upphöjt till (**)

Dessa hanteras i uttryckseditorn på följande sätt:

- Upphöjt till operatorn har som delmängd multiplikationsoperatorn, varför en delvis inmatad ** tolkas som multiplikation.

- För if .. then .. else upptäcks att det är en sådan konstruktion i och med att if är inmatat. Vid presentation av uttrycket vore det dock mycket olämpligt att visa mer än vad som är inmatat.

- För funktioner, som förutom funktionsnamnet innehåller höger- och vänsterparentes och eventuellt ett eller flera kommatecken är förhållandet likartat, d v s ännu ej inmatade tecken (här högerparentesen) bör inte skrivas ut innan de är inmatade.

Exempel: Om man skulle visa hela operatorn skulle det delvis inmatade uttrycket if a>5 th matas ut som

```
if a>5 then else
```

i stället för som

```
if a>5 th
```

Lösningen är att uttryckseditorn särbehandlar dessa distribuerade operatorer, på så sätt att inte bara operator-typen lagras, utan även hur långt inmatningen har kommit. Emellertid får inte raden fortsätta efter den ofullständigt inmatade operatorn.

Exempel: (| betecknar radslut)

if a>|	är korrekt
5*(if a>10 then 20 els|	är korrekt
5*(if a>10 then 20 els) +|	är inte korrekt, eftersom raden fortsätter efter "els".
f(a,|	är korrekt
G(x, *fact|	är inte korrekt.

3.4 Implementering

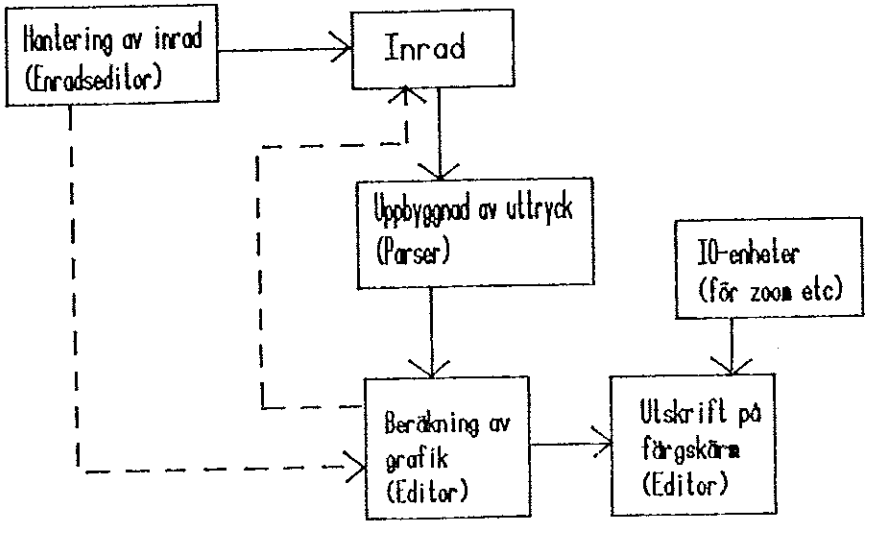
3.4.1 Översikt av olika programenheter

De olika steg som behöver genomföras för att omvandla ett uttryck skrivet som en textrad till en bild är följande:

1: Generering av en intern representation av uttrycket. Detta sker genom att textraden delas upp i sina enheter (identifierare, operatorer etc), och sedan att en trädrepresentation byggs upp. Detta sker med en scanner och en parser.
2: Generering av den nödvändiga grafiska informationen.
3: Grafisk utmatning av uttrycket.

Varje gång som inraden ändras sker samtliga steg, och eftersom editorn även hanterar en textcursor betraktas även en förflyttning till höger eller vänster på inraden som en förändring i uttrycket. Inraden, ändringar i denna och cursorförflyttningar hanteras av en enkel enradseditor. För accepterade kommadon se Appendix Enradseditor.

Ett undantag utgör dock piltangenterna för förflyttning upp och ner, eftersom dessa har mening endast om cursorn befinner sig under respektive över ett divisionsstreck, då de innebär en cursorförflyttning till positionen rakt över (under) aktuell cursorposition. En sådan förflyttning kan inte tolkas av enradseditorn, utan måste hanteras av uttryckseditorn själv, eftersom endast den känner till den grafiska informationen. Utryckseditorn räknar då ut den nya positionen för cursorn, och motsvarande position i inraden. Därefter sker generering av det nya uttrycket.



Kommentar:
De heldragna linjerna svarar mot det normala flödet mellan programenheterna. Inraden uppdateras av enradseditorn, sedan

skapas ett nytt uttryck. För detta nya uttryck räknas den grafiska informationen fram och slutligen skrivs uttrycket ut på färgskärmen. De streckade linjerna svarar mot flödet vid förflyttningar i strukturen, dvs piltangenter upp och ner. Då räknar uttryckseditorn ut den nya positionen i textraden, varefter ett nytt uttryck skapas etc.

Exempel:	inrad	editerat uttryck
	a/(b±c)	$\frac{a}{(b \pm c)}$
Pil höger ger Denna förflyttning kan hanteras av enradseditorn själv.	a/(b+c)	$\frac{a}{(b+c)}$
Pil upp ger Här kan inte enradseditorn själv avgöra den nya positionen utan flödet följer de streckade pilarna i figuren ovan.	a/(b+c)	$\frac{a}{(b+c)}$

Vid förflyttningar upp och ner blir den nya cursorpositionen den som ligger närmast över (under), om inget deluttryck står över (under) så ignoreras flyttningen.

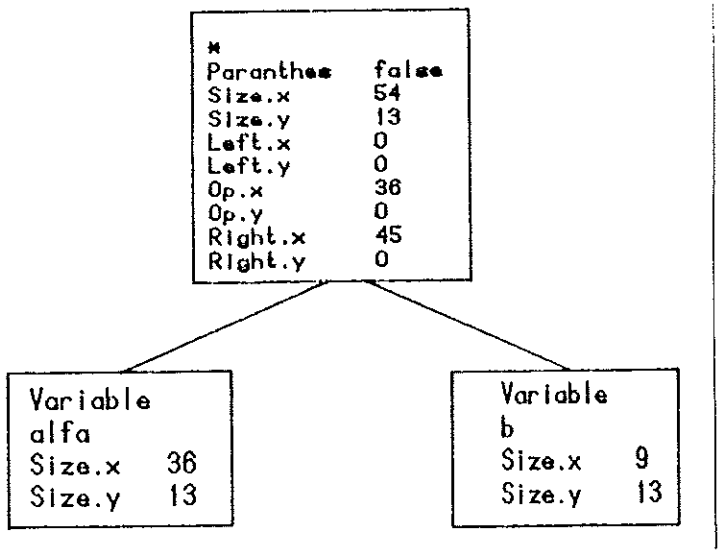
3.4.2 Grafisk information knuten till uttrycket

Detta avsnitt kommer att ta upp hur den grafiska informationen hanteras i icke interaktiv mode. I interaktiv mode lagras information även för inmatade parenteser, varför editorn hanterar dessa på ett annat sätt, se nedan.

Varje enhet (identifierare, konstant eller operator) i uttrycket upptar en rektangulär yta vars storlek lagras för alla enheter. För identifierare och konstanter lagras ingen ytterligare grafisk information, däremot finns naturligtvis de tecken som ingår i identifieraren respektive konstanten lagrade. För operatorer, som ju verkar på en eller flera operander, lagras koordinaterna till såväl operatorn som origo för operandernas koordinatsystem (i operatorns koordinatsystem). Operatorns totala platsbehov blir alltså en rektangel som omsluter operatorn och operandernas rektanglar.

Exempel: alfa*b kan åskådliggöras med följande bild

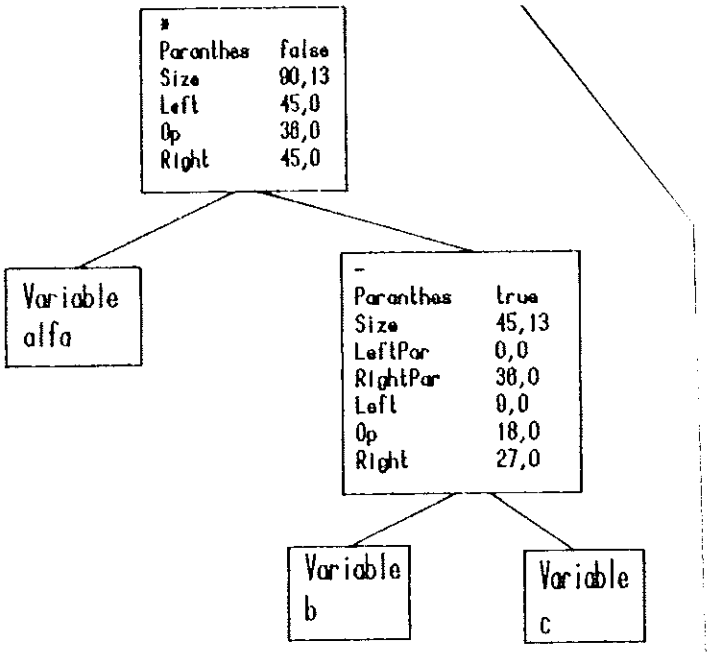
Exempel (forts):
Den lagrade informationen kan också åskådliggöras med ett träd, där operatorn är roten och operanderna löv. (Ett tecken antas vara 9 enheter brett och 13 högt i det virtuella koordinatsystemet, detta åstadkommes genom ett anrop till rutinen CharSize, se kapitlet Använda grafiska primitiver)



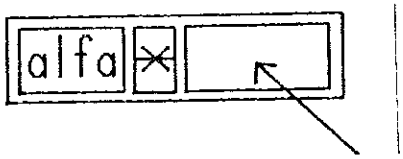
Kommentar: Storleken för rotens (*-operatorns) rektangel är alltså i x-led summan av de ingående operandernas rektanglar plus platsen som åtgår för *-tecknet. I y-led blir storleken lika med den högsta av de ingående rektanglarna (som här alla är lika höga). Left.x respektive left.y anger var den vänstra rektangeln har sitt origo (angivet i operatorns koordinatsystem). Op och right anger på motsvarande sätt var rektanglarna för operatorn och höger deluttryck skall placeras.

När editorn arbetar i icke interaktiv mode och det uttryck som behandlas innehåller parenteser som inte kan tas bort måste plats skapas för dessa. Det gör att den grafiska informationen måste utökas med koordinater och storlek för parenteserna.

Exempel: alfa*(b-c) ger följande träd (9 x 13 tecken)



Här syns det tydligt att platsen för enheten till höger om *-tecknet är given i subtraktionsnodens koordinatsystem.
Den rektangel som innehåller multiplikationsoperatorn ser ut så här:

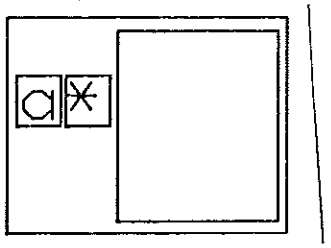


Den med pil utpekade rektangeln ovan (som innehåller subtraktionen) ser i närbild ut så här:

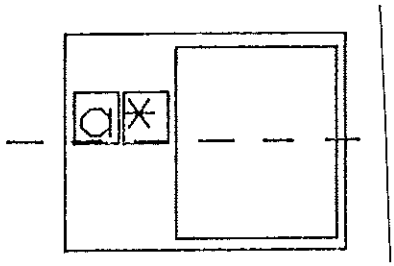
De uttryck som hittills använts har båda varit 'raka', dvs de skrivs på en rad även på färgskärmen. När man behandlar uttryck som innehåller division och/eller upphöjt till skall olika delar av uttrycket skrivas på olika höjd. Dessutom skall uttrycken centreras över divisionsstrecket och även i höjdled. Centreringen i höjdled kan ställa till problem, eftersom det inte är säkert att centreringen skall ske till mitten på rektangeln.

Exempel: a*(1/(1+(1/(1+x)))) skall skrivas ut som $a * \frac{1}{1+\frac{1}{1+x}}$

Det innebär att den multiplikationsteckent är omgivet av två olika höga rektanglar, som skall centreras, men inte kring mitten av den större. Detta ger följande figur:

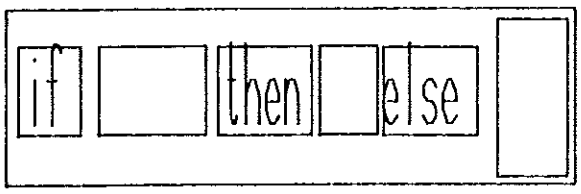


För att hantera detta arbetar uttryckseditorn med en baslinje, som anger den linje som rektangeln skall centreras kring. Om ovanstående figur kompletteras med baslinjen (streckad) erhålls följande bild:



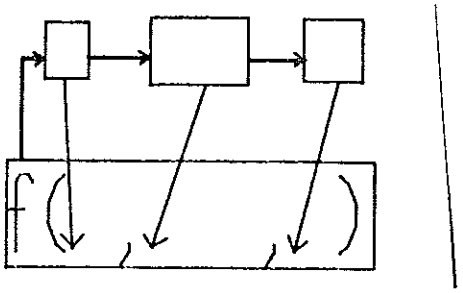
3.4.3 Grafisk information till distribuerade operatorer

De två distribuerade operatorer som accepteras är villkor och funktioner. Dessa måste särbehandlas eftersom de inte kan innesluta i en rektangel, utan måste representeras av flera.
Exempel: if a>larm then 17 else a/10 representeras av en rektangel som ser ut så här:



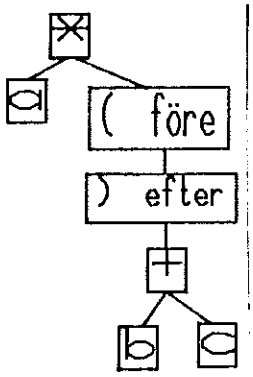
Funktioner är lite mer komplicerade eftersom de kan ha ett eller flera argument, dessutom tillkommer ett komma för varje argument förutom det första. Argumenten representeras som en länkad lista, därigenom kan en funktion ha godtyckligt många argument. Varje argument innehåller förutom den grafiska informationen även koordinaterna för argumentets rektangel i funktionens koordinatsystem (d v s argumentets placering inom hela funktionsrektangeln).

Exempel: Funktionen f(a,b+c,d) representeras med följande bild: (enkla pilar svarar mot arguments origo i funktionsrektangeln, dubbla mot den länkade listan)

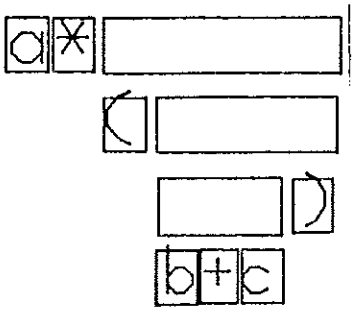


3.4.4 Ytterligare grafisk information i interaktiv mode

Det som skiljer den grafiska informationen åt vad gäller interaktiv mode är att även blanktecken och parenteser tilldelas rektanglar. För att förstå hur dessa tecken är lagrade kan det vara bra att se lite på hur parenteser i ett uttryck hanteras.
Det program som tolkar en rad och bygger upp ett uttryck kallas en parser. En parser bygger upp ett syntaxträd som representerar uttrycket, hur detta går till beskrivs kort i Appendix Parser.
Som beskrivs i appendixet lagrar den använda parsern även paranteser och blanktecken i syntaxträdet, detta för att uttrycksediterna skall kunna hantera dessa tecken. Efter parsing av uttrycket a*(b+c) erhålls syntaxträdet:



Vid genereringen av den grafiska informationen skapas även plats för parenteserna. Ovanstående uttryck ger upphov till följande rektanglar:



3.5 Beskrivning av uttrycksediterns arbetgång

Följande avsnitt behandlar icke interaktiv mode, de skiljaktigheter som gäller interaktiv mode tas upp i ett särskilt avsnitt längre fram.

Editorn arbetar i två pass, i det första passet räknas all grafisk information ut och lagras. I det andra passet sker utskrift på färgskärmen av uttrycket. Denna uppdelning i två pass är nödvändig eftersom man inte vet rektanglarnas position i det virtuella koordinatsystemet förän hela trädet är behandlat.
Dessutom blir editorn på det här viset tidseffektivare, eftersom beräkningen inte behöver ske mer än en gång, och man får en snabb omskrivning av uttrycket när man t ex zoomar.

3.5.1 Pass 1

I pass 1 gås uttrycket igenom rekursivt i suffix order, den rektangel som omsluter uttrycket och koordinater för alla enheter i uttrycket räknas ut. Eftersom eventuellt överflödiga paranteser skall tas bort sker en kontroll med hjälp av prioritetsordningen om paranteser behövs. För att kunna hantera centrering i y-led behöver varje operator känna till baslinjen för de omgivande operanderna, dvs vilken nivå operatorn skall placeras vid.
Arbetsgången vid beräkning för de binära operatorerna *, + och minus är följande:

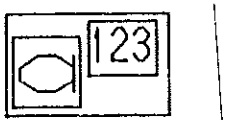
1: Beräkna grafisk information för vänster operand. Lagra dess baslinje.
2: D:o för höger delträd.
3: Beräkna den totala höjden för rektangeln. Hänsyn skall tas till de båda operandernas höjd och deras baslinjer.
4: Kontrollera om paranteser skall sättas in. Om ja, räkna ut skalan för dessa. Parenteserna skall vara lika höga som rektangeln.
   Vänster parentes placeras i punkten 0,0.
5: Placera ut vänster operand, i x-led placeras den direkt till höger om vänsterparantesen om den finns, annars på 0. I y-led tas hänsyn till baslinjen.
6: Placera ut operatorn omedelbart till höger om vänster operand. Hänsyn tas till baslinjen.
7: Placera ut vänster operator till höger om operatorn. Även här skall hänsyn tas till baslinjen.
8: Om parenteser skall sättas in så placeras höger parentes ut längst till höger.

Denna algoritm gäller med smärre modifikationer för alla typer av icke-löv noder. För divisionsnoder skall centreringen i stället ske i x-led, för unära noder (minus och not) faller vissa delar bort etc.

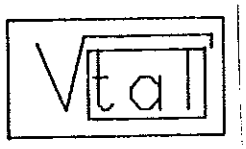
För upphöjt-till noder skrivs ingen operator ut, i stället justeras högerledets storlek med en skalfaktor < 1 för att hantera att den delen skall skrivas ut med mindre skala och

platsen för höger son ligger ett halvsteg upp.

Exempel: a**123 ger följande rektangel:



Om upphöjt-till noden svarar mot ett uttryck av type x**0.5 kommer inte 0.5 att skrivas ut, utan i stället räknas storlek och plats för kvadratrotstecknet ut. Detta gör att rektangeln inte kommer att ha mer än en delrektangel.
Exempel: tal**0.5 ger följande rektangel:



För funktioner och villkor sker en beräkning av de olika deluttrycken coh en centrering sker till baslinjen för funktionsnamnet (if-satsen). För funktioner adderas dessutom plats för parenteserna och eventuella kommatecken.

3.5.2 Pass 2

I pass två ritas uttrycket ut på färgskärmen. Eftersom varje enhet har sitt eget koordinatsystem så flyttas det aktuella koordinasytemet till aktuell enhet med hjälp av Translate (se kapitlet Använda grafiska primitiver).

Detta ger att pass två väsentligen kodats så här:

```
case expressiontype of
   binary :                    { Binära operatorer (+,-,*,/)      }
      begin                    { men ej ** som särbehandlas       }
      if Parentheses then      { Kolla ev parenteser              }
         DrawParenthes(expr);{ I så fall - rita ut dom           }
      DrawOperator(expr);      { Rita ut operatorn                }
      SaveTransform(T);        { Spara det aktuella koordinatsyst }
      Translate(left^.size); { Flytta koordinatsystemet till    }
                               { vänster operand                  }
      PrettyPass2(left);       { Rita ut vänster operand          }
      SetTransform(T);         { Sätt tillbaks det egna           }
                               { koordinatsystemet                }
      Translate(right^.size);{ Höger son                        }
      PrettyPass2(right);
      end;
   variable: .......
```

Ovanstående gäller med smärre modifieringar för de flesta noder, de viktigase undantagen är powerop, där en skalning sker och error-noder som behandlas lite speciellt, på så sätt att den text som finns i noden (dvs den text som stått efter felet på inmatnings-raden) skrivs ut med avvikande färg (röd).

3.5.3 Arbetssätt i interaktiv mode

När editorn arbetar i interaktiv mode innehåller det behandlade uttrycket förutom den vanliga informationen dessutom samtliga inmatade tecken, alltså även parenteser, redundanta parenteser och blanktecken.

Vid interaktiv mode läggs grafisk information upp även för dessa extra tecken, men inga parenteser genereras för uttryck. Dessutom skrivs kvadratrotsuttryck ut som upphöjt till 0.5.

I interaktiv mode hanterar editorn dessutom en cursor, denna läggs in i uttrycket vid omvandlingen från textraden till intern representation (av parsern). Cursorn skrivs ut som ett streck under aktuellt tecken, om inte detta är ett divisionsstreck eller en * som ingår i en upphöjt till operator, då ritas i stället en pil på operatorn. Se exempel på detta i avsnittet Interaktiv uppbyggnad av uttryck, tidigare i detta kapitel.

3.5.4 Hantering av pil upp & ner vid editering

Ett speciellt problem uppstår vid interaktiv körning när det inmatade tecknet är pil upp (ner). Detta skall ju resultera i en cursorförflyttning till motsvarande position över (under) aktuell position. En sådan vertikal förflyttning är bara möjlig när cursorn befinner sig under (över) ett divisionstreck.
Utryckseditorn hanterar detta genom att söka i det befintliga grafiska trädet för att hitta koordinaterna för aktuell cursorposition och den divisionsoperator som innehåller cursorn i det nedre (övre) deluttrycket. Därefter genomsöks divisionsoperatorns övre (undre) deluttryck för att finna det tecken som står närmast rakt över (under) cursorns position och dit flyttas cursorn. I de fall som en förflyttning inte är möjlig sker ingen ändring av cursorpositionen.

3.6 Anrops-exempel

Vid anrop av utryckseditorn skall två parametrar ges, dels det aktuella uttrycket och dels en Boolsk variabel som anger mode. True motsvarar interaktiv mode.

Följande korta programavsnitt bygger upp ett grafiskt träd till uttrycket ex och ritar ut det med de nya värden för fönster och skala som förutsättes ges av NewWindow:

```
PrePretty(ex,true);    { Pass 1. Interaktiv mode          }
repeat
   Pretty(ex,true);    { Pass 2. Rita det grafiska trädet }
   NewWindow;          { Scroll, pan eller zoom.          }
until StopShow;
```

3.7 Paket-specifikation

Vid användning av uttryckseditorn behövs förutom rutinerna Pretty och PrePretty även rutiner för uppbyggnad av ett uttryck från en textrad och eventuellt även för hantering av teckenvis inmatning. De rutiner och globala data som användaren behöver specificeras nedan.

3.7.1 Enradseditor

Enradseditorn används genom att anropa InLine med de tecken som matas in från tangentbordet. InLine returnerar värdet InLineChar om tecknet inte var pil upp eller ner, i så fall returneras InLineUp respektive InLineDown. Eftersom piltangenterna skickar tre tecken returneras InLineNothing i de lägen då enradseditorn måste ha fler tecken för att kunna avgöra vilken tangent som trycktes ner. Enradseditorn uppdaterar alltid den rad som scanner (och därmed parsern) arberar på.

```
const ImageLength = 79;
type  InLineType  = (InLineUp, InLineDown, InLineChar,
                     InLineNothing);

      image       = packed array[1..ImageLength] of char;

procedure InLineInit;
   { Den editerade raden tömms. Positionen sätts till 0  }

function InLine(ch : char) : InLineType;
   { ch läggs in i raden om det är ett tryckbart tecken. }
   { Annars kontrolleras om ch är ett kommando.          }

procedure SetInLineText(im : image; Pos : integer);
   { Editeringsraden får värdet im. Cursorpositionen     }
   { sätts till Pos. Pos skall vara i intervallet        }
   { 1..ImageLength                                      }

procedure SetInLinePos(Pos : integer);
   { Cursorpositionen sätts till Pos.                    }
```

3.7.2 Uppbyggande av uttryck

Uttryck byggs upp med en parser, se Appendix Parser. Parsern accepterar tre typer av uttryck
- tilldelningar (identifierare = uttryck)
- ekvationer (uttryck = uttryck)
- enkla uttryck, som ej får innehålla = -tecken.

Parsern förses normalt med indata från scannern via enrads-editorn. Om den inte används kan rutinen SetScannerData användas.

```
type   pexpr = ^expr;
       expr  = record
                 ...
               end;

function Assignment : pexpr;
function Equation   : pexpr;
function Expression : pexpr;
procedure SetScannerData(im : image; Pos : integer);
   { Förser scannern med im som indata. Cursorpositionen }
   { sätts till Pos.                                     }
```

3.7.3 Hantering av uttryck

Några rutiner för att hantera uttryck.

```
procedure RemoveChInfo(var ex : pexpr);
   { Tar bort alla redundanta noder, dvs de som          }
   { innehåller blanktecken och parenteser.              }

function CopyExpr(ex : pexpr) : pexpr;
   { Kopierar ett uttryck.                               }
```

3.7.4 Uttrycksediterna

Ovan har två anropbara procedurer nämnts, en för att skapa den grafiska informationen och en för att rita ut den. Ytterligare en procedur för att ta reda på vilken area ett uttryck tar finns.

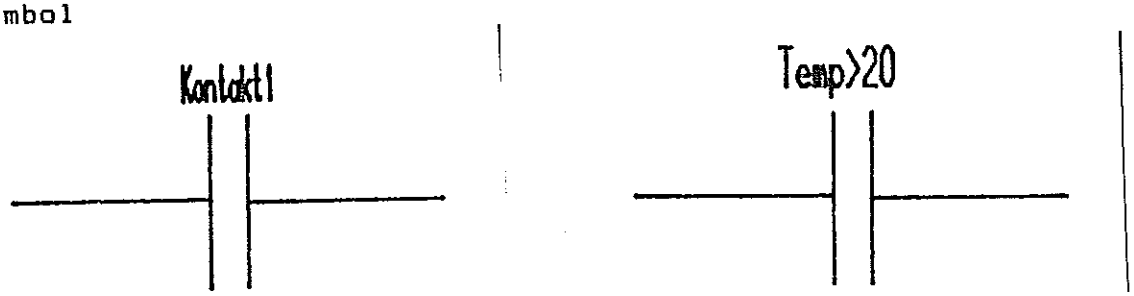
```
procedure PrePretty(ex : pexpr; Interactive : boolean);
   { Skapar grafisk information till uttrycket ex. Om    }
   { Interactive är true så arbetar rutinen i interaktiv }
   { mode.                                               }

procedure Pretty(ex : pexpr; Interactive : boolean);
   { Ritar ut den grafiska informationen till ex.        }

procedure ReturnExprSpace(ex : pexpr; var x, y : real);
   { Returnerar ändpunkten för den rektangel som spänns  }
   { upp av ex.                                          }
   { PrePretty måste vara anropad innan denna rutin      }
   { anropas.                                            }
```

4 Reläschema

Inom industrin används ofta relä-schema (ladder-diagram) för att representera logiska uttryck, även om dessa realiseras med digital teknik eller som datorprogram. Ett relä har två tillstånd, öppet och stängt (logisk 1:a respektive 0:a) på samma sätt som en digital krets eller en logisk variabel. För att demonstrera detta har en möjlighet att representera logiska uttryck som reläschema implementerats. Ett reläschema kan ritas till logiska uttryck, d v s uttryck med värdet sant eller falskt (true respektive false i Pascal), och varje villkor symboliseras då av ett relä. Detta betyder att alla logiska uttryck som kan representeras av uttryckseditorn också kan ritas som reläschema. Dock har programmet för ritning av reläschema inte realiserats för interaktivt bruk, utan uttrycket måste byggas upp med uttryckseditorn och därefter ritas ut som ett reläschema. Se även Appendix Demonstrationsprogram.

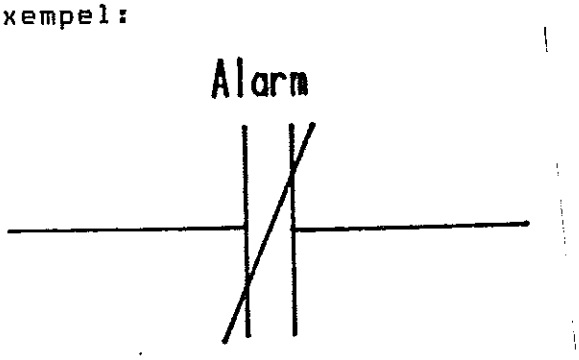
4.1 Representation

Ett relä med tillhörande villkor representeras med följande symbol



där reläet alltså är öppet (leder ström) när Kontakt1 är tillslagen respektive Temp är större än 20. Reläer kan även vara inverterade, dvs de leder ström när villkoret är falskt. Detta symboliseras av en diagonal linje över reläet.

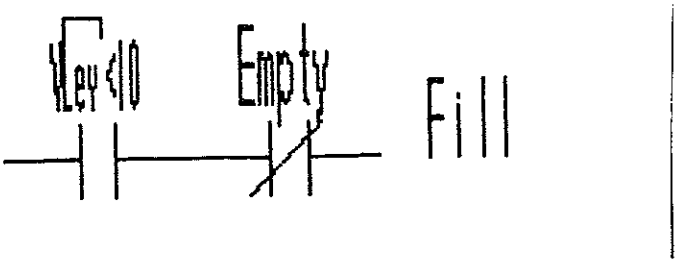
Exempel:



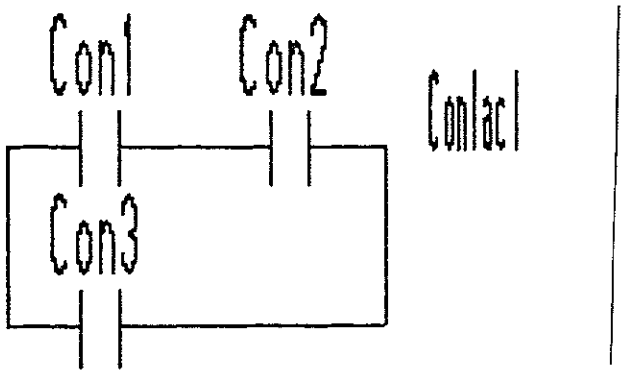
Här leder reläet ström när Alarm inte är sant. Detta motsvaras i Pascal av not Alarm.

När sammansatta logiska uttryck skall representeras med reläscheman så motsvaras den logiska operatorn and av seriekoppling och or motsvaras av parallellkoppling.

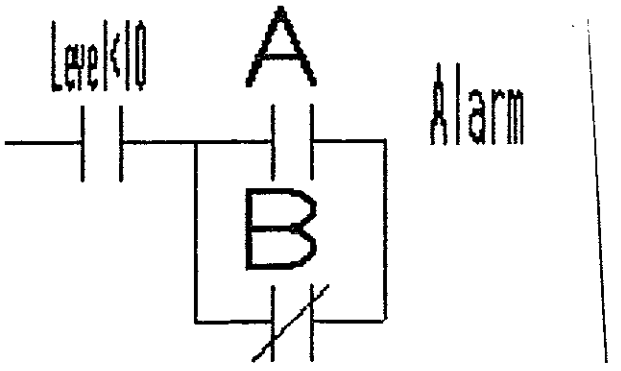
Exempel på logiska uttryck presenterade som reläschema:



Fill = Lev**0.5 < 10 and not Empty



Contact = Con1 and Con2 or Con3



Alarm = Level < 10 and (A or not B)

Resultatet av det logiska uttrycket skrivs alltså längst till höger i schemat. Om denna punkt är spänningsförande, dvs det finns en väg från vänster till höger som är framkomlig är uttrycket sant, reläet 'drar'.

4.2 de Morgans teorem

Med reläer kan man inte representera negerade sammansatta uttryck, dvs uttryck av typen not(a and b). För att kunna realisera dessa måste man omvandla dem med hjälp av de Morgans teorem:

```
not(a and b) = not a  or not b
not(a  or b) = not a and not b
```

d v s man byter and mot or och vice versa och negerar de ingående termerna.
Exempel på uttryck före och efter behandling med de Morgans teorem:

```
not(a and not b)        <==>  not a or b                    (1)
not(a and b or not c)   <==>  (not a or not b) and c        (2)
```

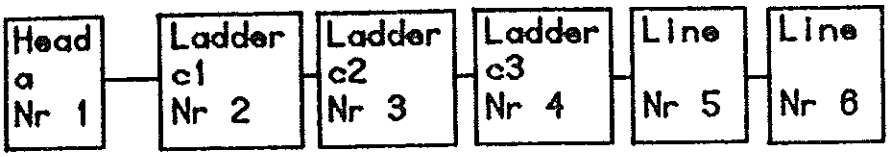
Motsvarande reläscheman blir:

4.3 Implementering

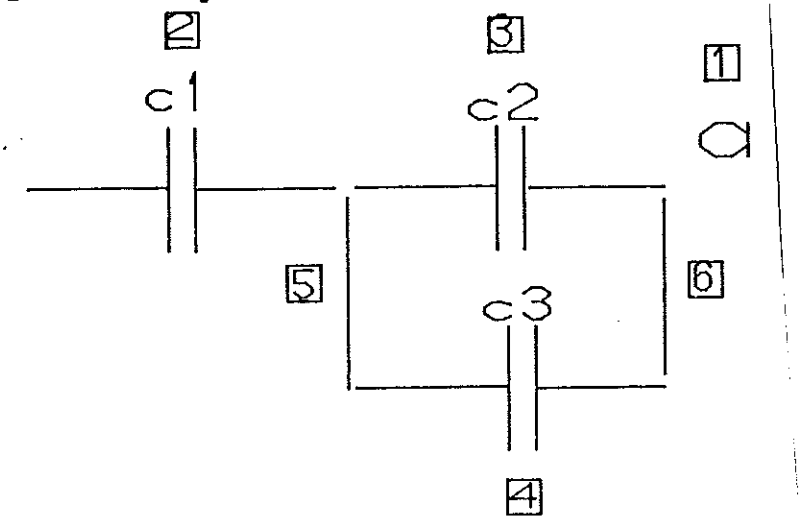
4.3.1 Datastruktur

Ett reläschema motsvaras av en länkad lista i Pascal. Varje element i listan motsvarar antingen ett relä, en kontaktledning mellan reläer eller ett listhuvud som innehåller vänstra ledet av uttrycket, d v s resultatet av reläschemat. Den viktigaste informationen som lagras för ett relä är de olika linjer som bygger upp reläet, koordinaterna för reläet och det uttryck som är kopplat till det. För en kontaktlinje behöver bara koordinaterna för linjens båda ändpunkter lagras. De uttryck som lagras i huvudet och i reläerna har först behandlats av uttrycks-editorn och innehåller därför all nödvändig grafisk information för utskrift på färgskärmen.

Exempel: Uttrycket a=c1 and c2 or c3 representeras av listan



och av följande reläschema (med numrering enligt listan):



Ett relä består i sig av en länkad lista av de linjer som ingår i reläet, detta för att kunna behandla såväl vanliga som inverterade reläer på samma sätt, ett vanlig relä består av fyra linjer, medan ett inverterat består av fem.

4.4 Arbetsgång

4.4.1 Arbetsgång i princip

Indata till reläritningsprogrammet är ett uttryck representerat av ett syntaxträd (se Appendix D, Parser). Först undersöks om trädet motsvarar en tilldelning, i så fall skapas ett reläschema för höger delträd, annars används hela trädet. För en tilldelning svarar vänstra delen mot resultatet av reläschemat. Det som skall ske sedan är att varje and- och or-nod skall motsvaras av en koppling mellan två reläer, som serie- respektive parallellkoppling.

And-noderna behandlas enligt följande algoritm:

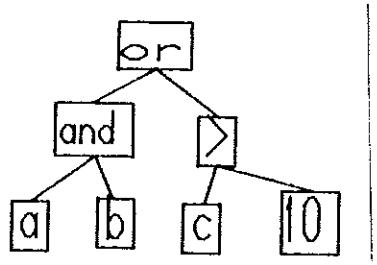
1 Skapa reläschema för vänster delträd. Då erhålls höger kontaktpunkt för detta schema.
2 Utifrån denna punkt byggs schemat ut med höger delträd.
3 Returnera kontaktpunkten för den högraste av kontaktpunkterna och platsen för det lägst belägna reläet.

För or-noderna gäller följande algoritm:

1 Beräkna reläschemat för vänster delträd. Då erhålls dels den högra kontaktpunkten och det lägst belägna reläet.
2 Räkna ut startpunkten för höger delträds schema. Denna punkt skall ligga rakt under startpunkten för vänster delträds schema. Skapa en linje mellan de båda startpunkterna.
3 Räkna ut höger delträds schema.
4 Kontrollera om de båda grenarnas reläscheman är lika långa. Om de inte är det, skapa en horisontell linje från det kortaste schemat så att längderna blir lika.
5 Skapa en vertikal linje som förenar de båda schemanas högra punkter.

För andra typer av noder skapas ett relä, och uttrycket som svarar mot noden och dess underträd behandlas med uttrycks-editorn. Sedan väljs en lämplig skala för uttrycket och det knyts till reläet.

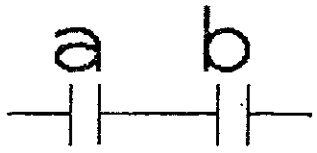
Exempel : a and b or c>10 har följande syntaxträd:



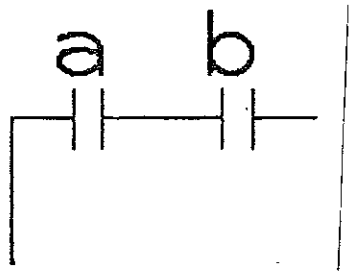
och byggs upp enligt följande (A1 anger punkt 1 i and-algoritmen etc) :

1 Först konstateras om uttrycket är en tilldelning. Detta innebär att reläschema endast skall ritas för höger delträd.
2 Or-noden träffas på. Enligt 01 beräknas reläschemat för vänster delträd.
3 And-noden träffas på. Enligt A1 beräknas även här uttrycket för vänster delträd.
4 Nu träffas noden med identifieraren a på. Ett relä skapas och uttrycket "a" knyts till det. Bild hittills:

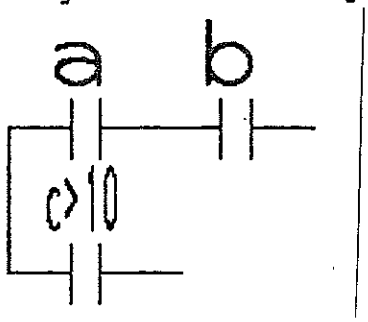
5 Nu beräknas and-nodens högra delträd (enligt A2).
6 På samma sätt som i 3 skapas ett relä till höger delträd och de två reläerna kopplas samman. Bild:



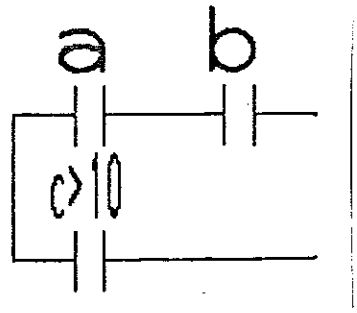
7 Nu är vänster delträd till or-noden klar. Enligt 02 skall det nu skapas en linje mellan vänster och höger delschemas startpunkter. Bild:



8 Enligt 03 räknas höger delträds schema ut. Bild:

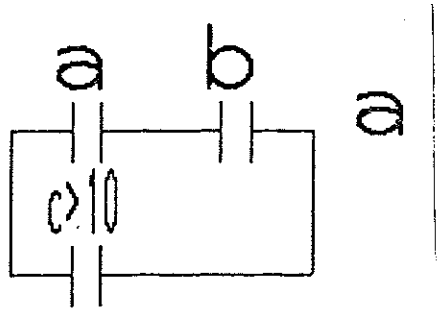


9 Eftersom de båda delschemana inte är lika långa skapas en horisontell linje från det kortare, d v s det undre (enligt 04). Bild:



10 Nu återstår kontaktlinjen mellan ändpunkterna (enligt 05).

11 Slutligen behandlas uttrycket till vänster om =-tecknet med uttryckseditorn och det placeras in till höger om reläschemat.
Slutlig bild:



4.4.2 Arbetsgång i praktiken

I den gjorda implementeringen ritas inte bilden ut när den skapas, utan i stället skapas den länkade listan och denna kan sedan ritas ut på färgskärmen. Liksom uttrycksediterna arbetar alltså reläritningsprogrammet i två pass, och skälet är att den förhållandevis tidskrävande uträkningen då bara behöver ske en gång per uttryck och att det sedan kan matas ut med olika skala och på olika delar av färgskärmen efter anrop av Scroll, Pan eller Zoom (se kapitlet Använda grafiska primitiver).

4.5 Anropsexempel

Vid anrop av pass 1 (skapandet av reläet) skall två parametrar anges, en för uttrycket som skall ritas som reläschema, och en för den lista där reläschemat skall lagras. Anropet av pass 2 skall bara innehålla listan som skapas av pass 1.
Följande korta programavsnitt bygger upp ett reläschema till uttrycket ex och ritar ut det med de nya värden för fönster och skala som förutsättes ges av NewWindow:

```
MakeLadder(ex,list);  { Pass 1. Reläet skapas.   }
repeat
   DrawLadder(list);  { Pass 2. Reläet ritas ut. }
   NewWindow;         { Scroll, pan eller zoom.  }
until StopShow;
```

4.6 Paket-specifikation

Eftersom reläritningsprogrammet använder uttrycksediterna för att rita ut de uttryck som är knutna till reläerna behövs dess rutiner. Dessutom används följande globala värden som användaren behöver känna till.

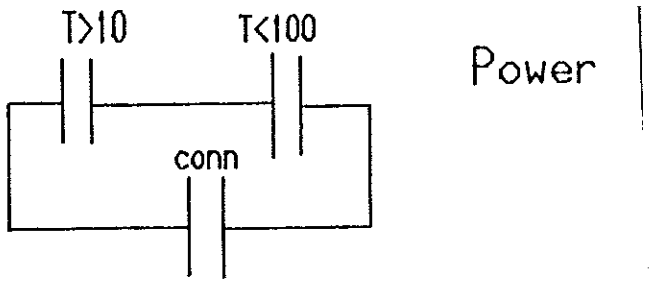
```
type   pLDItem = ^LDItem;

procedure MakeLadder(ex : pexpr; var ladder : pLDItem);
   { Skapar ett reläschema till ex.                    }

procedure DrawLadder(ladder : pLDItem);
   { Ritar ut reläschemat på skärmen.                  }

procedure DisposeLadder(var ladder : pLDItem);
   { Tar bort ett reläschema.                          }
```

5 Bildkompilering av reläschema

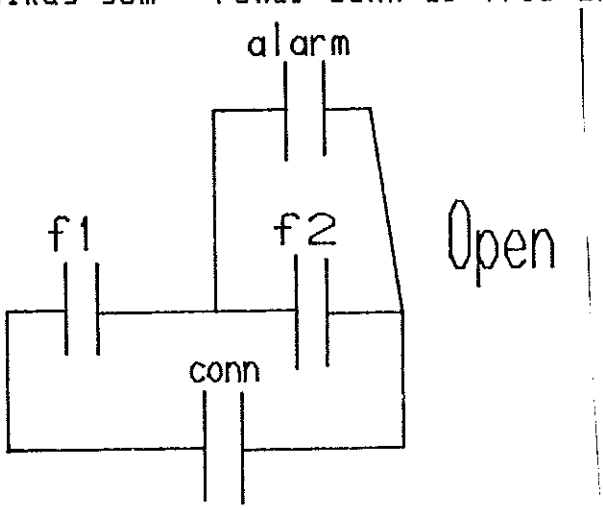
I handeln i dag finns ett antal system med vilka man ritar upp ett reläschema för den styrfunktion man önskar. Styrfunktionen realiseras sedan inte med reläer, utan med ett datorprogram eller med digitala kretsar. Som ett fristående program har en möjlighet att rita ett reläschema med LICS för att sedan analysera det och översätta det till ett syntaxträd implementerats. Detta syntaxträd skrivs sedan ut med konventionell notation på en vanlig terminal, men det skulle också kunna användas till att skapa de Pascal-programrader som realiserar styrfunktionen.

Här används alltså en grafisk inmatning som sedan tolkas, till skillnad från i de andra delarna av examensarbetet där linjär inmatning används. Det uttryck som erhålls vid tolkningen av reläschemat skulle kunna användas som inmatning till reläritningsprogrammet, man skulle då erhålla reläschema som var maskinritade, med de fördelar detta ger i uppställning på ett specificerat sätt etc. Någon sådan koppling har dock inte gjorts.

5.1 Exempel på tolkning reläscheman



tolkas som Power=conn or T>10 and T<100



tolkas som Open=conn or f1 and (f2 or alarm)

5.2 Algoritm vid bildkompileringen

Indata till bildkompileringsprogrammet är de linjer och texter som bygger upp bilden. Inledningsvis identifieras alla reläer, d v s de linjer som bygger upp ett relä knyts samman till en enhet. Ett relä byggs upp av fyra eller fem linjer och har två kontaktpunkter. Reläerna utgör ett enkelt uttryck med värde till/från (true/false). Dessa uttryck byggs sedan samman med or- respektive and-konstruktioner, där varje uttryck har två kontaktpunkter. Genom att succesivt knyta samman de olika uttrycken till större uttryck, där varje uttryck likaledes har två kontaktpunkter, byggs ett uttryck för hela reläschemat upp.

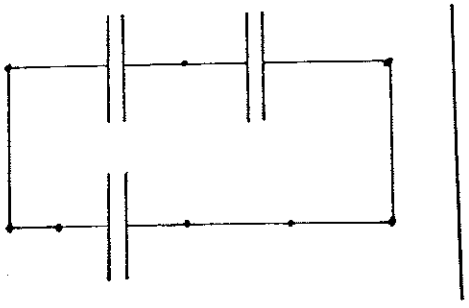
5.2.1 Uppbyggnad av reläer från linjer

Uppbyggnaden av reläer från de enskilda linjerna sker genom att först söka reda på två linjer som bildar de vertikala plattorna i reläet (figur 2 i exemplet nedan). Därefter söks efter två kontaktledningar som är anslutna till plattorna (3). Om sökningen lyckas söks efter en diagonal linje över plattorna (som markerar att reläet är inverterande) och reläet byggs upp samt de använda linjerna tas bort från listan av oanvända linjer (4). Detta upprepas tills inte fler reläer kan bildas.

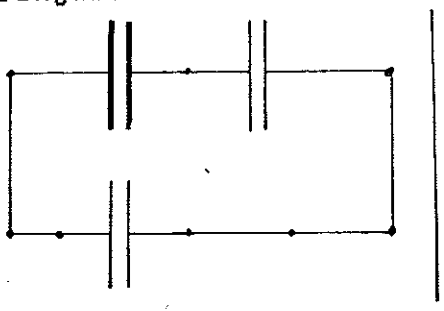
När alla reläer har byggts upp söks de kvarvarande linjerna igenom efter linjer som har kontakt (direkt eller indirekt) med något relä, dessa bildar en lista med kontaktledningar (5). I uppbyggandet av uttrycket används sedan reläerna och kontakt- ledningarna.

Avslutningsvis söks samtliga texter igenom och knyts till respek- tive relä.

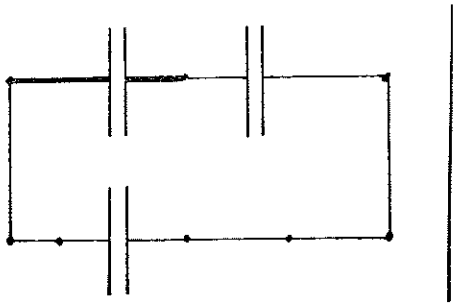
Exempel:
(1) Ursprunglig lista (Punkterna är utsatta endast för att markera linjernas ändpunkter och finns inte med på bilden).



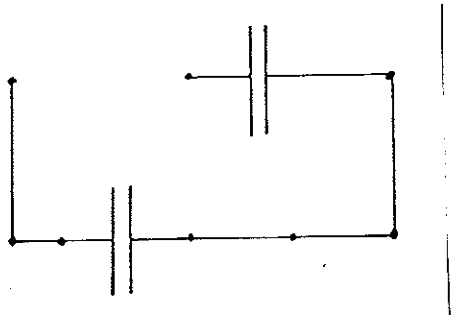
(2) Två vertikala linjer har hittats (markerade med tjockare linjer)



(3) Även två horisontella linjer har hittats.

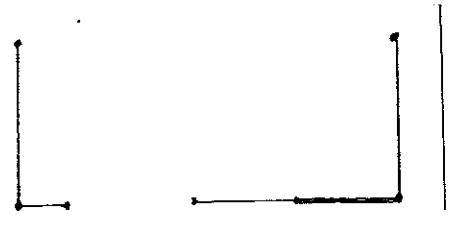


(4) De hittade linjerna bildar ett relä och tas ur listan. Kvar i listan finns:



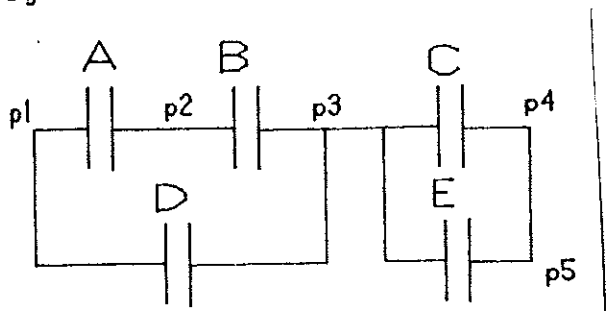
(5) Samtliga reläer är uppbyggda. Kvar finns:

(6) Kontaktledningarna hittas successivt. Även den markerade linjen, som inte har direkt kontakt med några relä, kommer att hittas och föras till listan med kontaktledningar.

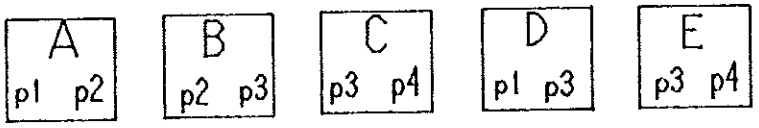


5.2.2 Uppbygnad av uttryck från reläer och kontaktledningar

Algoritmen illustreras med följande exempel:

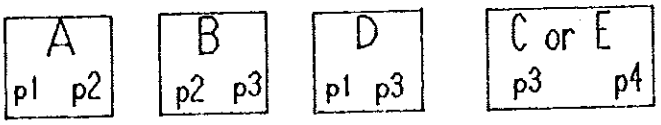


Bildar inledningsvis en lista med de fem reläerna A-E.

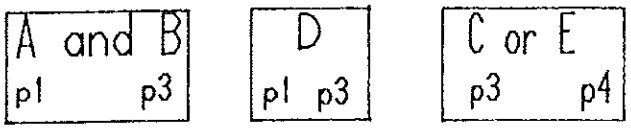


Observera att punkterna P4 och P5 är ekvivalenta, eftersom de står i direkt elektrisk kontakt med varandra. I fortsättningen av detta exempel används därför bara punkterna P1-P4.

En or-konstruktion existerar när de båda deluttryckens kontaktpunkter är ekvivalenta (dvs i elektrisk kontakt med varandra). Detta betyder att D i exemplet inte kan ingå i någon or-konstruktion, utan endast C och E kan bilda en sådan. Efter att C och E byggts samman får listan utseendet:

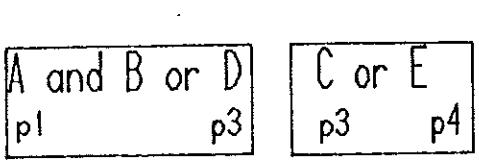


Vid byggandet av and-konstruktioner krävs att höger kontaktpunkt i ett deluttrycken är i kontakt med vänster kontaktpunkt i ett annat deluttryck. Med detta krav finns det tre möjliga and-konstruktioner i exemplet:
A and B
D and (C or E)
B and (C or E)
Av dessa tre är bara den första riktig. För att undvika att felaktiga konstruktioner byggs upp krävs därför att den gemensamma punkten mellan de två deluttrycken inte förekommer i något annat uttryck. Eftersom punkten p3 finns i tre uttryck kan and-konstruktioner kring p3 inte göras. Detta utesluter de två senare alternativen och A and B kan byggas upp och följande lista erhålls:

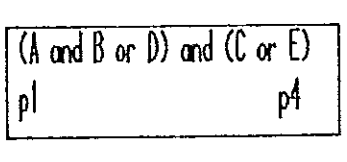


Det bör noteras att punkten p2 inte längre finns kvar som kontaktpunkt i något element, och att det nyskapade elementet har p1 och p2 som kontaktpunkter. Någon motsvarande reduktion av antalet punkter äger inte rum vid uppbyggandet av or-konstruktioner där de båda kontaktpunkterna är ekvivalenta.

Nu upprepas sökningen efter or-konstruktioner. Nu kan D ingå i en or-konstruktion, eftersom (A and B) har kontaktpunkter som är ekvivalenta med D's. Listan får då följande utseende:

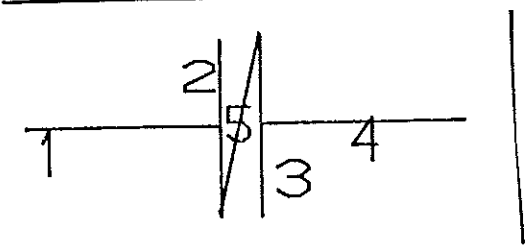


Avslutningsvis byggs and-konstruktionen upp och den slutliga listan innehåller bara det slutliga uttrycket:



I de fall där den slutliga listan innehåller mer än ett träd har den ursprungliga bilden varit felaktig, t ex med kontakttrådar som inte varit i absolut kontakt med varandra eller reläer helt utan kontakt med varandra.

5.3 Krav på bilden



När programmet skall detektera de linjer som sätter samman ett relä krävs att vissa villkor är uppfyllda.

För att de båda plattorna (linjerna markerade med 2 respektive 3) skall detekteras krävs följande:

- Att de är absolut vertikala. Detta garanteras av den grafiska editorn om "Manhattan mode" används.
- Att de båda linjernas ändpunkters y-koordinater inte avviker från varandra med mer än 20 % av deras längd.
- Att linjerna inte ligger mer än 50% av deras längd från varandra i x-led.

För att anslutningslinjerna (1 respektive 4) skall detekteras krävs följande:

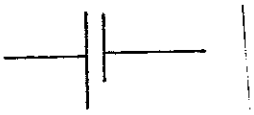
- Att linjerna har direkt kontakt med plattorna. Detta garanteras av den grafiska editorn om "Gravity mode" används.
- Att linjerna inte avviker mer än 20 % från horisontalplanet.

När reläet är inverterande krävs dessutom att den diagonala linjens (5 i figuren) ändpunkter inte avviker med mer än 10 % av plattornas längd i x-led och inte med mer än 20 % i y-led.

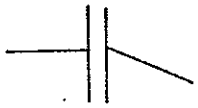
Slutligen krävs att det uttryck som är knutet till uttrycket är placerat med sin bas mindre än 25 % av textens storlek från reläets översta punkt, samt att det i x-led inte står utanför reläet.

Detta kan verka som omständliga krav, men i praktiken betyder de att reläer som ser "bra" ut också accepteras av bildkompileringsprogrammet.

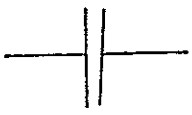
Exempel på ej detekterade reläer:



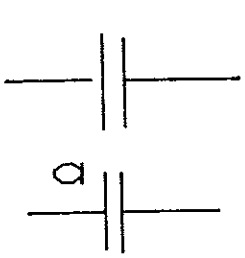
Högra plattan för kort.



Höger kontaktledning för sned.



Höger platta sned.

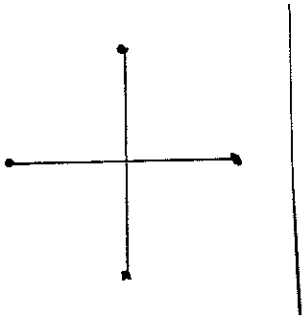


Vänster kontaktledning ej i kontakt med plattan.

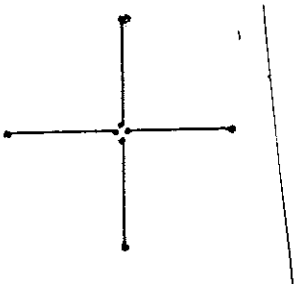
Villkoret (här a) står inte över reläet.

Även på kontaktledningarna (d v s de linjer som förbinder de olika reläerna med varandra) ställs krav. De måste vara i absolut kontakt med varandras, eller med reläernas, ändpunkter. Detta kan ses som att kontaktlinjerna består av isolerade elektriska ledare, som bara ger kontakt i ändpunkten.

Exempel: (Punkterna markerar ändpunkterna på linjerna och syns inte på bilden)



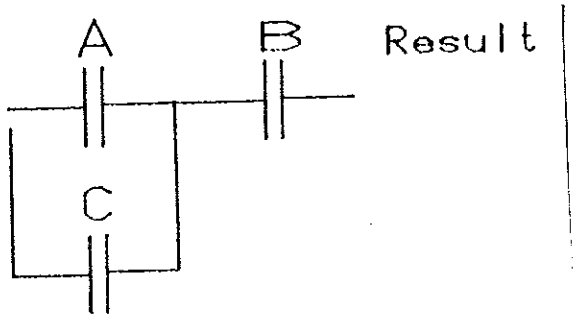
Denna linjekorsning ger ingen kontakt mellan den vertikala och den horisontella linjen. Om en sådan kontakt önskas måste de båda linjerna delas upp till fyra stycken, två vertikala och två horisontella, med gemensam ändpunkt i skärningspunkten.



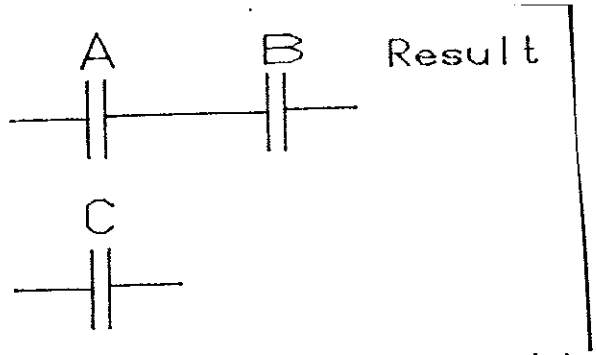
5.4 Ofullständiga och felaktiga reläscheman.

En svårighet uppstår när det inritade reläschemat inte är korrekt, det kan t ex vara ofullständigt, reläerna kan sakna logiska villkor, eller det kan vara felaktigt. En naturlig lösning hade varit att på något sätt, t ex med hjälp av cursorn, markera vart i bilden felet kan hänföras och dessutom ge en förklarande felutskrift på textterminalen. Detta skulle emellertid kräva en stark koppling till LICS skärmhantering och i stället erhålls endast felutskrifter på textterminalen.

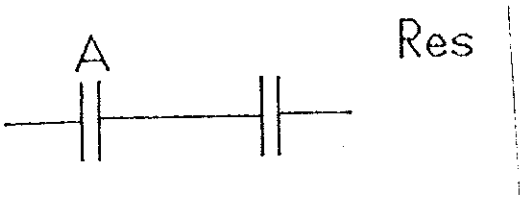
Exempel på resultat och felutskrifter vid felaktiga reläscheman:



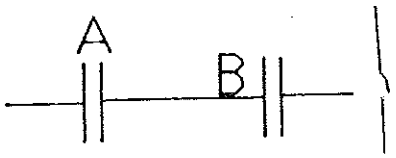
(En kontaktpunkt är ej sluten.)
Felutskrift: List is not empty after creation of ladders and connections.
3 expressions detected.
Resultat: Result = B
Kommentar: Felutskriften uppkommer under byggandet av reläerna. Då kontrolleras också att alla linjer som inte ingår i reläerna hänger fast i bägge sina ändpunkter.
Genom att C sitter fast mellan A och B kan inte uttrycket A and B byggas upp, eftersom B är ansluten till två reläer.
Vilket av de tre uttrycken (Result = A, Result = B, Result = C) som blir resultatet av bildkompileringen beror bara på i vilken ordning linjerna är genererade, och kan alltså inte förutses.



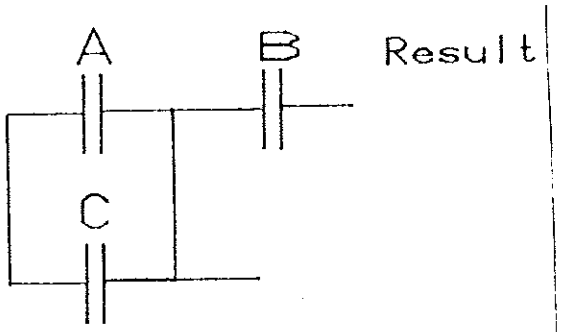
Felutskrift: 2 expressions detected.
Resultat: Result = A AND B
Kommentar: Reläet C är helt fristående. Endast ett uttryck returneras från bildkompileringsprogrammet, och liksom ovan är det inte möjligt att avgöra vilket.



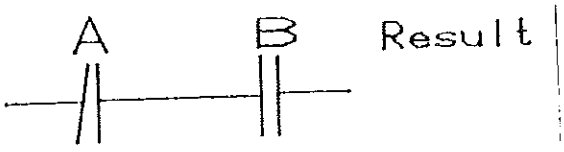
Felutskrift: Ingen.
Resultat: Res = A AND NoText01
Kommentar: Reläer utan logiska villkor erhåller en variabel med namnet NoTextnn.



Felutskrift: All texts not used.
B
Resultat: A AND NoText01
Kommentar: B är felplacerat, och hittas inte som villkor till det högra reläet. Exemplet visar att också reläscheman utan "höger-led" lämnar ett resultat, men då inte med en lika-med-relation.



Felutskrift: List is not empty after creation of ladders and connections.
Resultat: Result = (C OR A) AND B
Kommentar: Linjen till höger om C är bara knuten i ena ändpunkten.



Felutskrift: List is not empty after creation of ladders and connections.
All texts not used.
A
Resultat: Result = B
Kommentar: Den sneda vänstra kontaktplattan på A gör att reläet inte identifieras.

Appendix A
Kompilering och länkning

A.1 Källkodsfilernas utseeende

Pascal innehåller inte separatkompilering utan hanterar hela programmet som en enhet. Detta ger en rad nackdelar när programm blir stora, man saknar skydd mot användande av globala data, man kan få namnkonflikter på identifierare, programmen blir textmässigt stora och svårhanterliga, etc.

För att i någon mån kunna hantera fristående runtiner har det inom LICS-projektet skrivits ett program som sammanför olika programfiler till en kompileringsenhet. Genom att i programtexten ange speciella direktiv av typen .TYPE och .FORWARD förs programkoden till rätt del av det sammanslagna programmet. Med denna teknik slipper man inte ifrån problemen med identifierare som är synliga globalt, men textfilerna kan struktureras så att de innehåller logiska enheter.

Exempel:

```
.TYPE
Point       = record
   x, y   : real;
   end; { Point }

.VAR
LastPoint : Point;

.FORWARD
procedure SetCursor(p : Point);                          forward;
procedure DrawLine(FromPoint, ToPoint : Point);          forward;
procedure LineTo(p : Point);                             forward;

.PROCEDURE
procedure SetCursor(p : Point);
   begin
   LastPoint := p;
   end; { SetCursor }
.....
.INIT
LastPoint.x := 0,0;        { Utförs när programmet startas å
LastPoint.y := 0.0;
.END
```

Kommentar: Det enda som användaren av paketet behöver känna till är de tre forwarddeklarerade rutinerna och deklarationen av Point. Deklarationen av LastPoint måste ligga globalt, men skall bara användas av rutinerna i paketet. Någon kontroll av detta sker dock inte.

I huvudprogrammet anger man .MAIN, denna kod kommer att exekveras efter alla .INIT-rutiner.

A.2 Kompilering

Innan kompilering av systemet körs sammanslagningsprogrammet, som skapar 9 stycken filer, en för varje direktiv. Sedan kompileras ett program som bara innehåller include-direktiv till kompilatorn. Hanteringen sker lämpligtvis i en kommandofil, som kör sammanslagningsprogrammet, kompilerar och länkar.

Exempel på en kommandofil:

```
$ RUN REG:[regler.PASCON]compack        ! Sammanslagningsprogrammet
file1.pak                               ! De ingående filerna
file2.pak
file3.pak
$ on error then go after                ! För att undvika att länka
                                        ! felaktiga program
$ pascal/check/list/nostandard/nowarn use:[lics.basic]pasprog
                                        ! Komilering av ett program
                                        ! som bara består av INCLUDE
                                        ! -direktiv
$ delete/nocon *.sec;*                  ! Ta bort temporära filer
$ link pasprog                          ! Länka
$ ren pasprog.exe simple.exe            ! Döp om EXE-filen
$ after: delete/nocon pasprog.obj;*     ! Ta bort alla OBJ-filen
```

A.3 Kompilering av uttryckseditorn och reläritningsprogrammet

I examensarbetet två första delar används ett antal filer som hanterar det grafiska systemet. Dessa filer måste tas med vid varje kompilering av uttryckseditorn och reläritningsprogrammet.

Använda filer:
reg:[regler.lics.lics]licsdata.pak
reg:[regler.lics.lics]licsproc.pak
reg:[regler.lics.SCREEN]raster.old
reg:[regler.lics.SCREEN]charshell.pak
reg:[regler.lics.SCREEN]charpac.old
reg:[regler.raster]rastinit.pak
reg:[regler.raster]color.pak
use:[LICS.COLOR.GRAPH]prvscreen.pak
reg:[regler.lics.SCREEN]graphpac.old

A.3.1 Filer ingående i uttryckseditorn

Förutom de ovan nämnda filerna behövs följande filer för att kompilera uttryckseditorn:

use:[magnus.exarb.pretty]Prettybas.pak
use:[magnus.exarb.term]interact.pak
use:[magnus.exarb.term]myterm.pak
use:[magnus.exarb.term]vt100term.pak
use:[magnus.exarb.term]inline.pak
use:[magnus.exarb.expr]expr.pak

```
use:[magnus.exarb.expr]eqoper.pak
use:[magnus.exarb.expr]exprset.pak
use:[magnus.exarb.expr]scanner.PAK
use:[magnus.exarb.expr]parser.PAK
use:[magnus.exarb.pretty]Pretty.dfs
use:[magnus.exarb.pretty]Pretty.PAK
use:[magnus.exarb.pretty]mtlib.pak
use:[magnus.exarb.pretty]updown.pak
use:[magnus.exarb.pretty]Prettyini.pak
```

A.3.2 Filer ingående i reläritningsprogrammet

Eftersom reläritningsprogrammet använder uttryckseditorn för att presentera reläernas uttryck behövs samtliga ovanstående filer. Reläritningsprogrammet består av följande filer:

```
use:[magnus.exarb.expr]demorgan.pak
use:[magnus.exarb.expr]excopy.pak
use:[magnus.exarb.ladder]ladder.pak
use:[magnus.exarb.ladder]laddmake.pak
use:[magnus.exarb.ladder]laddraw.pak
```

A.3.3 Kompilering och länkning

Vid kompilering och länkning av uttryckseditorn och reläritnings-programmet används följande kommandon:

```
$ pascal/check/list/nostandard/nowarn use:[lics.basic]pasprog
$ link/nomap pasprog,proclib/lib,rasteropt/opt,paslib/lib
```

A.4 Kompilering av bildkompileringsprogrammet

Bildkompileringsprogrammet körs som en del av LICS-programmet. Det innebär att alla filer som används i LICS skall ingå vid kompilering, utom filen PICTCOMP2.PAK som ersätts med en annan fil. Nedan listas alla i filer LICS som ingår vid kompilering och de filer som innehåller bildkompilatorn.

```
reg:[regler.lics.lics]licsdata.pak
reg:[regler.lics.lics]licsproc.pak
reg:[regler.lics.dp]DPTEST.NEW
reg:[regler.lics.dp]messform.pak
reg:[regler.lics.dp]graphicsn.def
reg:[regler.lics.dp]graphicsn.bod
reg:[regler.lics.inter]interact.dfn
reg:[regler.lics.syspas]rvext.pak
reg:[regler.lics.dp]DPIO.NEW
reg:[regler.lics.dp]DPDATA.NEW
reg:[regler.lics.dp]GRAPHICS.NEW
reg:[regler.lics.dp]DPLISTS.NEW
reg:[regler.lics.dp]DPMEM.NEW
reg:[regler.lics.dp]DPSTRETCH.NEW
```

```
reg:[regler.lics.dp]DPOUT.NEW
reg:[regler.lics.dp]DPFILES.NEW
reg:[regler.lics.dp]STRING.NEW
reg:[regler.lics.dp]FILETEXT.PAK
reg:[regler.lics.dp]DPFUNCT.NEW
reg:[regler.lics.dp]DPIMMED.NEW
reg:[regler.lics.dp]DPSCALE.NEW
reg:[regler.lics.dp]DPIMAGE.PAK
reg:[regler.lics.dp]DP.NEW
reg:[regler.lics.dp]DPSEG.NEW
reg:[regler.lics.dp]dpaim.pak
reg:[regler.lics.pict]dppict.new
reg:[regler.lics.pict]pict.new
reg:[regler.lics.pict]prettypri.pak
reg:[regler.lics.pict]addconn.pak
reg:[regler.lics.pict]pdcheck.pak
reg:[regler.lics.pict]makeconn.pak
reg:[regler.lics.pict]simsys.pak
reg:[regler.lics.pict]detail.pak
reg:[regler.lics.pict]ddcsys.pak
use:[magnus.exarb.ladder]laddbasic.pak
use:[magnus.exarb.expr]expr.pak
use:[magnus.exarb.expr]exprset.pak
use:[magnus.exarb.expr]eqoper.pak
use:[magnus.exarb.expr]excopy.pak
use:[magnus.exarb.expr]scanner.pak
use:[magnus.exarb.expr]parser.pak
use:[magnus.exarb.expr]exprinf.pak
use:[magnus.exarb.ladder]ladder.pak
use:[magnus.exarb.ladder]laddanal.pak
use:[magnus.exarb.ladder]fromlics.pak
use:[magnus.exarb.pretty]mtlib.pak
use:[magnus.exarb.ladder]pictcomp2.pak
reg:[regler.lics.pict]pictcheck.pak
reg:[regler.lics.dcl]dclpak.pak
reg:[regler.lics.dcl]dclstart.pak
reg:[regler.lics.dcl]getimage.pak
reg:[regler.lics.text]text.pak
reg:[regler.lics.teve]tved.pak
reg:[regler.lics.teve]tv.pak
reg:[regler.lics.teve]tvmenu.pak
reg:[regler.lics.teve]dpmenu.pak
reg:[regler.lics.teve]termcurs.pak
```

För kompilering och länkning används samma kommandon som i uttryckseditorn.

Appendix B
Kort demonstrationsprogram för uttryckseditorn

Nedanstående program som använder enradseditorn och uttrycks-editorn visar hur dessa kan användas. Trycker man ner den vänstra knappen på musen byter uttryckseditorn mellan interaktiv och icke interaktiv mode.
Inmatning av tecken från tangentbordet gör alltid att uttrycks-editorn övergår till interaktiv mode. Inmatning av ' return' raderar inmatningsraden, och ^P skriver ut skärmen på OKI-skrivaren. övriga inamtade tecken förs vidare till enradseditorn.

```
procedure SimpleDemo;
const DummyCharHeight = 26;
      DummyCharWidth  = 18;
      CtrlP           = 16;      { ASCII code for ^P             }
      CR              = 13;      { ASCII code for Carriage Rtn   }
var AktEx          : pexpr;      { Pointer to an expression      }
    Mode           : (NoInt, Int);
    InLineTemp     : InLineType;
    InteractTemp   : InteractType;
    ch             : char;

{ Demo } procedure DrawExpression;
   begin
   DrawBackGround;              { Draw the new image in back-   }
   Draw;                        { ground                        }
   ClearScreen;                 { Erase screen.                 }
   if Mode = NoInt then
      Pretty(AktEx, false)      { NonInteractive mode           }
   else
      Pretty(AktEx, true);      { Interactive mode              }
   SwapPlane;                   { Make the drawed image visible.}
   end;

begin { SimpleDemo }
Mode     := Int;   { Start in interactiv mode }
AktEx    := nil;
while true do
   begin
   InteractTemp:=Interact(ch);
   case InteractTemp of
      InteractButton1Down:      { Left button on mouse   }
                                { Change mode            }
         begin
         if Mode = Int then
            begin
            PrePretty(AktEx, false);
            Mode := NoInt;
            end
         else
            begin
            PrePretty(AktEx, true);
            Mode := Int;
```

```
            end;
         end;

      Interactchar:                { character from key-board }
         begin
         if ch = chr(CtrlP) then
            begin
            OKIScreen;             { Print thi image          }
            DrawExpression;
            PrintVirtualScreen;
            CmiScreen;             { Back to video screen     }
            end
         else
            begin
            Mode := Int;           { Interactive mode }
            if ch = chr(CR) then
               begin
               DisposeExpr(AktEx);
               InLineInit;         { Clear inline image }
               DrawExpression;
               end
            else
               begin
               InLineTemp:=InLine(ch); { Modify inline image }
               case InLineTemp of
                  InLineChar:
                     begin
                     DisposeExpr(AktEx);
                     AktEx := equation;
                     PrePretty(AktEx, true);
                     DrawExpression;
                     end;
                  InLineUp,
                  InLineDown:
                     begin
                     MoveUpOrDown(Aktex, InLineTemp = InLineUp);
                     InLineUpDate;
                     DisposeExpr(AktEx);
                     AktEx := equation;
                     PrePretty(AktEx, true);
                     DrawExpression;
                     end;
                  InLineNothing:
                     ; { No action }
                  end; { case InLineTemp }
               end; { if }
            end;
         end; { InteractChar }

      OtherWise
         DrawExpression;
      end; { case }

   end; { while true do }
end; { SimpleDemo }
```

Appendix C
Kort demonstrationsprogram för reläritningsprogrammet

Nedanstående program är mycket likt det i appendix B, men det demonstrerar hur ett program som använder reläritningsprogrammet kan se ut. Vänstra knappen på musen används här för att byta mellan interaktiv mode i uttryckseditorn och reläritning.
Inmatning av tecken från tangentbordet gör att programmet övergår till interaktiv mode i uttryckseditorn. inmatning av 'return' raderar den editerade raden, och ^P skriver ut bilden på skrivaren.

```
procedure LadderDemo;
const CtrlP           = 16;    { ASCII code for ^P                }
      CR              = 13;    { ASCII code for Carriage Return }
var AktEx          : pexpr;    { Pointer to an expression       }
    Ladder         : pLDItem;
    Mode           : (NoInt, Int);
    InLineTemp     : InLineType;
    InteractTemp   : InteractType;
    ch             : char;

{ SimpleDemo } procedure DrawExpression;
   begin
   DrawBackGround;           { Draw the new image in background }
   Draw;
   ClearScreen;              { Erase screen.                    }
   if Mode = NoInt then
      DrawLadder(Ladder)     { NonInteractive mode              }
   else
      Pretty(AktEx, true);   { Interactive mode                 }
   SwapPlane;                { Make the drawed image visible.   }
   end;

begin { SimpleDemo }
Mode     := Int;   { Start in interactiv mode }
AktEx    := nil;
Ladder   := nil;
while true do
   begin
   InteractTemp:=Interact(ch);
   case InteractTemp of
      InteractButton1Down:      { Left button on mouse      }
         begin                  { Change mode               }
         if Mode = Int then
            begin
            DisposeLadder(Ladder);
            MakeLadder(AktEx, Ladder);
                                { Creates a ladder to AktEx }
            Mode := NoInt;
            end
         else
            Mode := Int;
         DrawExpression;
         end;
```

```
      InteractChar:                { character from key-board }
         begin
         if ch = chr(CtrlP) then
            begin
            OKIScreen;             { Draw this image on the LPT  }
            DrawExpression;
            PrintVirtualScreen;
            CmiScreen;             { Back to video screen        }
            end
         else
            begin
            Mode := Int;                    { Interactive mode   }
            if ch = chr(CR) then
               begin
               DisposeLadder(Ladder);
               DisposeExpr(AktEx);
               InLineInit;                  { Clear inline image }
               DrawExpression;
               end
            else
               begin
               InLineTemp:=InLine(ch);      { Modify inline image}
               case InLineTemp of
                  InLineChar:
                     begin
                     DisposeExpr(AktEx);
                     AktEx := equation;     { Build new expr     }
                     PrePretty(AktEx, true);
                     DrawExpression;
                     end;
                  InLineUp,         { Arrow up/down on keyboard   }
                  InLineDown:
                     begin
                     MoveUpOrDown(Aktex, InLineTemp = InLineUp);
                     InLineUpDate;
                     DisposeExpr(AktEx);
                     AktEx := equation;
                     PrePretty(AktEx, true);
                     DrawExpression;
                     end;
                  InLineNothing:
                     ; { No action }
                  end; { case InLineTemp }
               end; { if ch = chr(CR) }
            end; { case InteractTemp }
         end; { InteractChar }

      OtherWise
         DrawExpression;    { Redraw image }
      end; { case }

   end; { while true do }
end; { LadderDemo }
```

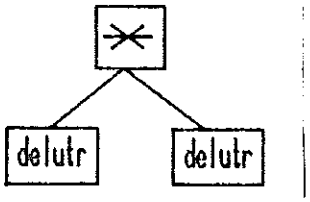
Appendix D
Parser

Avsikten med detta appendix är inte att ge en komplett beskrivning av en parsers arbetssätt, utan bara via exempel ge en kort förklaring på det och på hur ett träd för ett uttryck är uppbyggt.

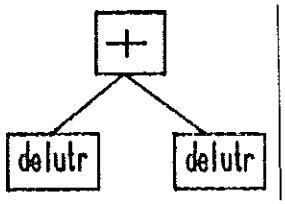
D.1 Syntaxträd

All intern representation av uttryck i examensarbetet grundar sig på träd.

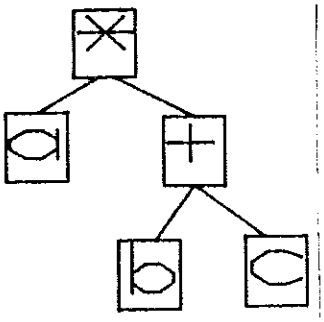
Uttrycket a*(b+c) ses från multiplikationstecknet som delutryck * deluttryck. Att det andra deluttrycket är omslutet av parenteser har ur operatorns synpunkt ingen betydelse, utan den ser bara två deluttryck. Ritat som ett träd ser multiplikationen ut så här



Deluttrycket (b+c) ser ut additionsoperatorn ut som deluttryck + deluttryck, och som träd som



Om vi sätter in identifierarna och ritar hela uttrycket a*(b+c) får vi följande bild



Observera att parenteserna inte syns i trädet, beräkningsordningen bestäms i stället av trädets utseende.

D.2 Parserns arbetssätt

Det program som översätter ett uttryck skrivet med linjär representation till ett träd kallas en parser. Arbetsgången för parsern när uttrycket ovan översätts till trädet är följande:

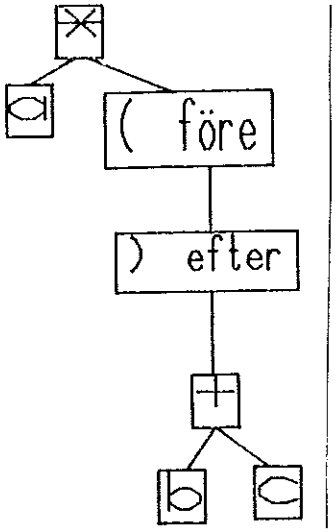
1 Första enheten, d v s variabeln a, läses och sparas.
2 Nästa enhet läses (*-tecknet). Nu vet parserna att syntaxen är deluttryck * deluttryck. Vänster deluttryck är klart och utgörs av variabeln a.
3 Nästa enhet, vänsterparantesen, läses. Nu vet parsern att alla tecken fram till högerparentesen skall betraktas som ett del-uttryck.
4 Deluttrycket b+c läses och motsvarande träd byggs upp.
5 Nu läses slutligen högerparentesen. Det innebär att multi-plikatorns bägge omgivande deluttryck är klara, och trädet med multiplikatorn som rot kan byggas.

D.3 Utökat syntaxträd

I exemplet ovan syns aldrig parenteser i trädet, men uttrycks-editorn i interaktiv mode skall skriva ut även dessa. Därför har syntaxträdet och parsern modifierats för att kunna hantera även parenteser.

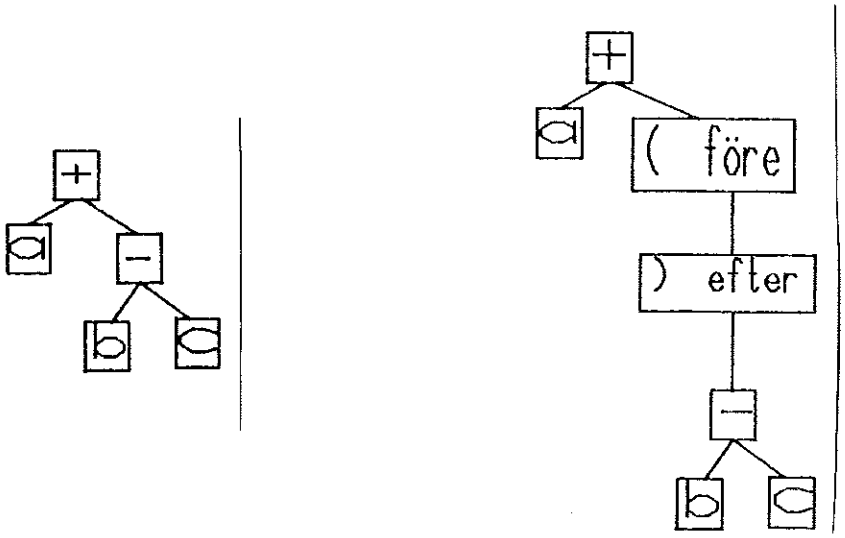
Detta har skett genom att parenteser lagras som egna noder i trädet, och varje sådan nod förses med en markering som anger om tecknet skall skrivas ut före eller efter uttrycket längre ner i trädet.

Exemplet ovan ger det utökade syntaxträdet



I detta exempel är det möjligt att utifrån trädets utseende avgöra att parenteserna måste sättas in, men i de fall parenteserna är redundanta måste de vara representerade i trädet.

Parenteserna i uttrycket a+(b-c) är redundanta. Det normala respektive det utökade syntaxträdet blir



För uttrycket a+b-c, som matematiskt är helt likvärdigt, är såväl det normala som det utökade syntaxträdet ekvivalent med det ovan, men i det utökade trädet framgår skillnaden i inmatningen.

Slutligen skall nämnas att även blanktecken lagras på samma sätt som parenteser, de har ju ingen betydelse för utrycket ur matematisk synvinkel, men väl ur estetisk.

Appendix E
Enradseditor

Interaktiv inmatning och editering av en textrad.

För interaktiv inmatning av de uttryck som skall behandlas används en enkel enrads-editor för VT100-terminaler. Textraden matas in via en vanlig terminal och raden visas också på denna terminal. De kommandon som finns i Inline är väsentligen de samma som används i reglerteknikks skärmeditor TEVE.
Eftersom enradseditorn endast arbetar en rad som innehåller 79 tecken kan längre rader än så inte behandlas.

Accepterade tecken	Effekt	Kommentar
^B	Cursorn till pos 1	
^E	Cursorn till slutet av raden	
^W	Raden skrivs om	
Delete	Tecknet till vänster om cursorn raderas	
^R	Tecknet över cursorn raderas	
^D	Om cursorn är placerad på pos 1 raderas raden	
<Return>	Cursorn till pos 1	
pil höger (vänster)	Cursorn flyttas ett steg åt höger (vänster)	
pil upp (ner)	Ingen	Strukturinformation, skickas vidare till uttryckseditorn.

Som alternativ till pil höger (vänster) kan även användas

^F	Cursorn 1 steg till höger
<BS> (^H)	Cursorn 1 steg till vänster

Appendix F
Programlistor

Följande filer, som ingår i examensarbetet, listas i detta appendix.

Fil	Kommentar
prettybas.pak	Programhuvud
interact.pak	Interaktion med mus, styrspakar och terminal
myterm.pak	Terminal-rutiner
vt100term.pak	Styrsekvenser för VT100-terminaler
inline.pak	Enradseditor
expr.pak	Data-struktur för uttryck
eqoper.pak	Enkla operationer på d:o
exprset.pak	------- " -------
excopy.pak	Kopiering av uttryck
demorgan.pak	Demorgans algoritm
scanner.PAK	Scanner
parser.PAK	Parser
Pretty.dfs	Datastruktur för uttrycksediterna
Pretty.PAK	Kod för d:o
updown.pak	Hanterar pilar upp/ner i d:o
mtlib.pak	Några biblioteksrutiner
ladder.pak	Datastruktur för reläsheman
laddmake.pak	Skapar ett reläschema
laddraw.pak	Ritar ut d:o
Prettyini.pak	Initialisering av skärm, fonter etc.

Rutiner för bildkompileringsprogrammet.

laddbasic.pak	"Dummy" deklarationer från uttryckeditorn.
exprinf.pak	Skriver ut ett uttryck på textskärmen.
laddanal.pak	Bildkompilatorn
fromlics.pak	Konverterar LICS-data till annat format.
pictcomp2.pak	Interface-rutin mot LICS

```
================================================================

                    use:[magnus.exarb.pretty]Prettybas.pak

================================================================

{ PACKAGE BASIC for Pretty-program }
Author Magnus Taube
.PROGRAM
program LICS(input,output,infile,outfile,fontfile,maskfile);

.EXTERN
procedure OutInit;extern;

.CONST
   maxIGbuttons = 4;
   maxIGchar    = 20;
   ImageLength  = 79;  { length of the text image }
.TYPE
   text3  = packed array [1.. 3] of char;
   text4  = packed array [1.. 4] of char;
   image = packed array [1..ImageLength ] of char;
   line=array[1..linelength]of char;
.VAR
   outfile:text;



.END
```

```
================================================================

                    use:[magnus.exarb.term]interact.pak

================================================================

package INTERACT is


-- Author: Hilding Elmqvist
-- Date: 1981-09-01

Modified : Magnus Taube.   Added interactype and som other modifications.
.TYPE
InteractType =
   (InteractChar, InteractOther, InteractCursor,
    InteractButton1up,   InteractButton2up,   InteractButton3up,
    InteractButton1down, InteractButton2down, InteractButton3down,
    InteractNothing);
.VAR
Interactch             : char;
Interactcx, Interactcy : real;
.FORWARD
function Interact(var ch : char): InteractType; forward;
procedure InitInteraction;                      forward;

.END

{ ============================================================== }


package body INTERACT is

.VAR
      gcteck          : array[1..21] of char;
      gcpos           : integer;
      getcharprinting : boolean;


initinteract, stopinteract                   : boolean;
panfactconst, scrollfactconst, zoomfactconst : real;

.PROCEDURE

{ -------------------------------------------------------------- }

procedure InitInteraction;
   var   idum   : integer;
         status :integer;

   function GlblMap(init: boolean; sectname,
                    filename: packed array [integer] of char;
                    size: integer): integer; extern;
   procedure GetEvCluster(b:boolean;s:packed array [integer] of char;
                          i:integer);extern;
   function TTinit: integer; extern;

   begin
   if initinteract then
      begin
```

```
      status := GlblMap(true, 'LICSSECT', 'LICSGLBLSECT', idum);
      if not odd(status) then
         writeln('GlblMap status:',status:8 HEX);
      status := TTinit;
      if not odd(status) then
         writeln('TTinit status:',status:8 HEX)
      else
         writeln('Terminal input line connected');
      GetEvCluster(false,'INTERACTSTART',100);
      end;

   initinteract := false;
   stopinteract := false;
   end { InitInteraction };


function Interact((var ch : char): InteractType);

const maxchan = 8;
      Cursorfact = -30;
type  blocktype = record
          def    : array[1..maxchan] of boolean;
          values : array[1..maxchan] of integer;
          i      : integer;
          arr    : packed array[1..25] of char;
          d      : packed array [1..7] of char;
          end;

var   block  : blocktype;
      chan   : integer;
      rv,f   : real;
      peek   : boolean;
      res    : InteractType;
      j,sl   : integer;

procedure InBlock(var block:blocktype); extern;
function MoreIo:boolean;                extern;


  begin { Interact }
  stopinteract:=false;

{ Old Getchar procedure follow (modified) }

   if gcpos ) 0 then
      begin
      ch     :=gcteck[gcpos];
      gcpos  :=gcpos-1;
      res    :=InteractChar;
      end
   else
      begin
      InBlock(block);
      res:=InteractNothing;

      if block.i ) 0 then
         begin
         ch   :=block.arr[1];
```



```
         res  :=InteractChar;
         if block.i ) 25 then { 26 to 32 is filler }
            block.i:=25;
         if block.i ) 8 then
            sl:=block.i-7
         else
            sl:=2;
         for j:=block.i downto sl do
            begin
            gcpos:=gcpos+1;
            gcteck[gcpos]:=block.arr[j];
            end;
            end
   { end of old getchar }
      else
         begin
         Interactcy:=0.0;
         Interactcx:=0.0;
         for chan := 1 to maxchan do
            if block.def[chan] then
               begin
               rv := block.values[chan]/400;
               case chan of

                  1:  begin
                      if Block.values[1] = 1 then
                         res:=InteractButton1Down
                      else
                         res:=InteractButton1Up;
                      end;

                  2:  begin
                      if Block.values[2] = 1 then
                         res:=InteractButton2Down
                      else
                         res:=InteractButton2Up;
                      end;

                  3:  begin
                      if Block.values[3] = 1 then
                         res:=InteractButton3Down
                      else
                         res:=InteractButton3Up;
                      end;

                  4:  begin                          { Scroll }
                      f := scrollfactconst*rv;
                      Scroll(f);
                      res:=InteractOther;
                      end;

                  5:  begin                          { Pan }
                      f := panfactconst*rv;
                      Pan(f);
                      res:=InteractOther;
                      end;

                  6:  begin                          { Zoom }
```

```
                    f := exp(zoomfactconst*rv);
                    Zoom(f);
                    end;

                7 : begin                          { cursor right or left }
                    res:=InteractCursor;
                    Interactcx:=rv*CursorFact;
                    end;

                8 : begin                          { cursor up or down }
                    res:=InteractCursor;
                    Interactcy:=rv*CursorFact;
                    end;

              end { case };
            end { if block };
          end;
      end;
   Interact := res;
end { Interact };

{ ------------------------------------------------------------------------ }


.INIT
initinteract     := true;
panfactconst     := 2;
scrollfactconst  := 2;
zoomfactconst    := 0.1;
InitInteraction;
gcpos            :=0;
.END
```



```
.END
{ Package for terminal handler }


.FORWARD
procedure KbDirect;forward;
procedure KbSet;extern;
procedure KbRestore;forward;
procedure WriteCh(ch: char); forward;

{ ------------------------------------------------------------------------ }

.PROCEDURE

Procedure KbDirect;
begin
  KbSet;
end;

Procedure Kbrestore;
Procedure kbreset; extern;
begin
  kbreset
end;
.PROCEDURE

procedure WriteCh (ch: char) ;
  type arr = array[1..2]of char;
  var a:arr;
  procedure Outstring(i:integer;a:arr); extern;
begin
  a[1]:=ch;
  Outstring(1,a);
end;

.INIT
gcpos:=0;
getcharprinting:=false;

.END
```

```
=============================================================

                    use:[magnus.exarb.term]vt100term.pak

=============================================================

{ Package VT100 is
{--- Routines for cursorposition on VT100-terminals
{--- Author  Magnus Taube
{
{ Calls : WriteCh
{}

.FORWARD
procedure EraseEndOfScreen;forward;
procedure EraseEndOfLine;  forward;
procedure SaveCursor;      forward;
procedure RestoreCursor;   forward;
procedure TermSetCursor(x,y:integer);forward;

{ ------------------------------------------------------------------ }



.CONST
esc = 27;
.PROCEDURE
procedure SaveCursor;
   begin
   writech(chr(esc));
   writech(chr(55));
   end;

procedure RestoreCursor;
   begin
   writech(chr(esc));
   writech(chr(56));
   end;

procedure EraseEndOfScreen;
   begin
   writech(chr(esc));
   writech(chr(91));
   writech(chr(48));
   writech(chr(74));
   end;

procedure EraseEndOfLine;
   begin
   writech(chr(esc));
   writech(chr(91));
   writech(chr(48));
   writech(chr(75));
   end;



procedure TermSetCursor(x,y:integer);
{ TermSetCursor routine, for VT100-terminals only }
```



```
   begin
   writech(chr(esc));
   writech(chr(91));                              { ESC (1B) och 5B }
   writech(chr((x div 10) + 48));
   writech(chr((x mod 10) + 48));
   writech(chr(59));                                    { 3B }
   writech(chr((y div 10) + 48));
   writech(chr((y mod 10) + 48));
   writech(chr(72));                                    { 48 }
   end { GoToxy };

.END
```

```
use:[magnus.exarb.termJinline.pak
```

```
{ package Inline is
{
{--- Short one-line editor
{ Author Magnus Taube
{}
{}
.TYPE
InlineType = (InlineUp, InlineDown, InlineChar, InlineNothing);

.VAR
InlineImage : image;
InlinePos   : integer;
InlineLast  : integer;
InlineMode  : (InlineNormal, EscMode, CSIMode);

.FORWARD
procedure InlineInit;                                   forward;
function Inline(var ch : char): InlineType;             forward;
procedure SetInline(i:integer); forward;
procedure InlineUpdate;                                 forward;
procedure SetInLineText(im : image; Pos : integer);     forward;
procedure SetInLinePos(Pos : integer);                  forward;
{ ------------------------------------------------------- }


.VAR
InlineLine : integer;

.PROCEDURE
procedure SetInLineText(im : image; Pos : integer);
   begin
   InLineImage := im;
   SetInLinePos(Pos);
   end;

procedure SetInLinePos(Pos : integer);
   begin
   InLinePos  := Pos;
   InLineMode := InLineNormal;
   InLineUpdate;
   end;



procedure SetInline(i:integer);
   begin
   InlineLine:=i;
   end;


procedure WriteInline(st,sl:integer);
   var i : integer;
```



```
   begin
   if st <= sl then
      begin
      TermSetCursor(InlineLine,st);
      for i:=st to sl do
         WriteCh(InlineImage[i]);
      end;

   for i:=1 to InlineLast do
      ScanData.ScanImage[i]:=InlineImage[i];
   ScanData.length:=InlineLast;
   ScanData.pos:=1;
   ScanData.more:=true;
   ScanData.cursorpos:=InlinePos;
   TermSetCursor(InlineLine,InlinePos);
   end;

procedure InlineUpdate;
   var i : integer;
   begin
   TermSetCursor(InlineLine,0);
   EraseEndOfScreen;
   TermSetCursor(InlineLine,0);
   for i:=1 to InlineLast do
      Writech(InlineImage[i]);
   for i:=1 to InlineLast do
      ScanData.ScanImage[i]:=InlineImage[i];
   ScanData.length:=InlineLast;
   ScanData.pos:=1;
   ScanData.cursorpos:=InlinePos;
   ScanData.more:=true;
   TermSetCursor(InlineLine,InlinePos);
   end;

procedure InlineInit;
   var i :integer;
   begin
   for i:=1 to ImageLength do
      InlineImage[i]:=' ';
   InlineLast:=0; { last character }
   InlinePos:=1;
   InlineMode:=InlineNormal;
   TermSetCursor(InlineLine,0);
   EraseEndOfScreen;
   end;

function Inline((var ch : char): InlineType);
   const  DEL    =127;
          CntrlB =  2;
          CntrlC =  3;
          CntrlD =  4;
          CntrlE =  5;
          CntrlF =  6;
          CntrlR = 18;
          CntrlW = 23;
          BS     =  8;
          LF     = 10;    { Line feed }
          CR     = 13;
```

```
        ESC    = 27;

  var  i, j : integer;
       res  : InlineType;

  { Inline } procedure Forward;
     begin
     if (InlinePos <= InlineLast) then
        InlinePos:=InlinePos+1;
     WriteInline(0,-10);
     end;

  { Inline } procedure BackSpace;
     begin
     if (InlinePos <> 1) then
        InlinePos:=InlinePos-1;
     WriteInline(0,-10);
     end;

  begin
  if InlineMode = EscMode then
     begin
     if ch = '[' then
        InlineMode:=CSIMode
     else
        InlineMode:=InlineNormal;
     res:=InlineNothing;
     end
  else if InlineMode = CSIMode then
     begin
     if ch = 'A' then
        begin
           res:=InlineUp;
           WriteInline(0,-10);
           end
     else if ch = 'B' then
        begin
        res:=InlineDown;
        WriteInline(0,-10);
        end
     else if ch = 'C' then
        begin
        Forward;
        res:=InlineChar;
        end
     else if ch = 'D' then
        begin
        BackSpace;
        res:=InlineChar;
        end
     else
        res:=InlineNothing;
     InlineMode:=InlineNormal;
     end
  else if ch = chr(ESC) then
     begin
     res:=InlineNothing;
     InlineMode:=ESCMode;
```



```
     end
  else if ch in [' '..'~'] then    { printable char }
     begin
     if InlineLast < 78 then
        begin
        res:=InlineChar;
        InlineLast:=InlineLast+1;
        for i:=InlineLast downto InlinePos+1 do
            InlineImage[i]:=InlineImage[i-1];

        InlineImage[InlinePos]:=ch;
        InlinePos:=InlinePos+1;
        WriteInline(InlinePos-1,InlineLast);
        end
     else
        res:=InlineNothing;
     end
  else if ch = chr(CR) then
     begin
     res:=InlineChar;
     InlinePos:=1;
     WriteInline(0,-10);
     end
  else if ch = chr(DEL) then
     begin
     if InlinePos <> 1 then
        begin
        res:=InlineChar;
        for i:=InlinePos to InlineLast do
           InlineImage[i-1] := InlineImage[i];
        InlineImage[InlineLast]:=' ';
        InlineLast :=InlineLast - 1;
        InlinePos  :=InlinePos  - 1;
        WriteInline(InlinePos,InlineLast+1);
        end
     else
        res:=InlineNothing;
     end
  else if ch = chr(CntrlR) then
     begin
     if InlinePos <> (InlineLast+1) then
        begin
        res:=InlineChar;
        for i:=InlinePos to InlineLast-1 do
           InlineImage[i] := InlineImage[i+1];
        InlineImage[InlineLast]:=' ';
        InlineLast :=InlineLast - 1;
        WriteInline(InlinePos,InlineLast+1);
        end
     else
        res:=InlineNothing;
     end
  else if (ch = chr(CntrlB)) then
     begin
     res:=InlineChar;
     InlinePos:=1;
     WriteInline(0,-10);
     end
```

```
   else if (ch = Chr(CntrlD)) and (InlinePos=1) then
      begin
      res:=InlineChar;
      for i:=1 to ImageLength do
         InlineImage[i]:=' ';
      InlineLast:=0; { last character }
      InlinePos:=1;
      TermSetCursor(InlineLine,0);
      WriteInline(1,ImageLength);
      end
   else if (ch = chr(BS)) then
      begin
      BackSpace;
      res:=InlineChar;
      end
   else if (ch = chr(CntrlF)) then
      begin
      res:=InlineChar;
      Forward;
      end
   else if ch = chr(CntrlE) then
      begin
      res:=InlineChar;
      InlinePos:=InlineLast+1;
      WriteInline(0,-10);
      end
   else if ch = chr(CntrlW) then    { redraw the line }
      begin
      res:=InlineNothing;
      WriteInline(1,ImageLength);
      end
   else
      res:=InlineNothing;

   Inline:=res;
   end;

.INIT
SetInline(22);
InLineInit;
.END
```

use:[magnus.exarb.exprlexpr.pak

```
{ PACKAGE EXPR is
{
{-- Parse tree with attributes for expressions.
{
{-- Author: Hilding Elmqvist
{-- Date:   1981-07-26
{-- changed Magnus Taube   --  cursor handling in
{                              Expression editor
{}
.TYPE
   string80 = packed array [1..80]of char;
   pexpr    = ↑expr;
   parglist = ↑arglist;

   arglist  = record              { used for argumentlist to functions      }
      next        : parglist;     { pointer to nex argument                 }
      ex          : pexpr;        { The argument.                           }
      Comma,                      { true if a ,-char follows the expression }
      CommaCursor : boolean;      { true is the cusor is at the ,-char      }
      end;  { record arglist }

   texprcursor = record
      iscursor : boolean;   { true if this node contains the text cursor  }
      pos      : integer;   { The position of the cursor (if any)         }
      StartPos : integer;   { The first characters place in the input line}
      end;


   texprtype = (exprvariable, exprfunction,     { Valid types of nodes in }
                exprnumber, exprminus, exprnot, { expression-trees.       }
                exprifthenelse, exprbinary,
                exprerror, exprchinfo);

   tbintype = (binaddop, binsubop, binmultop, bindivop,    { valid type of}
               binpowerop, binequalop, binandop, binorop,  { binary nodes }
               bingreaterop, binlessop);

   texprvartype = (realexpr, booleanexpr);

   expr = record
      exprvartype : texprvartype;
      pgfig       : ↑gfig;                    { used by graphic routines }
      cursor      : texprcursor;

      case exprtype: texprtype of
         exprvariable:      (varid           : identifier );
         exprfunction:      (func            : identifier;
                             NrOfSpaces,        { used for spaces after  }
                                                { function identifier but}
                                                { before the ( -char     }
                             SpaceCursor        { If the cursor is placed}
                                                { under one of the spaces}
                            : integer;
                             LeftParanthes,
```

```
                        RightParanthes  : boolean;
                        ParanthesCursor : integer;  { 0 n:e,1 l,2 r}
                        argumentlist    : parglist;
                        nargs           : integer );
        exprnumber:     (numid          : identifier;
                        val             : real );
        exprminus:      (minusexpr      : pexpr );
        exprnot:        (notexpr        : pexpr;
                        NotLength       : Integer );
        exprifthenelse: (ifexpr,
                        thenexpr,
                        elseexpr        : pexpr;
                        ThenPos,
                        ElsePos,                    { pos of first ch }
                                                    { in then (else)  }
                        IfLength        : integer);{ if only part of }
                                                    { text is entered }
        exprbinary:     (expr1, expr2   : pexpr;
                        bintype         : tbintype );
        exprerror:      (s              : string80;
                        ErrLength       : integer;
                        ErrorNumber     : integer;
                        NextExpr        : pexpr);
        exprchinfo:     (ch             : char;
                        before          : boolean;
                        chexpr          : pexpr);
        end; { record expr }

.VAR
  zero, one, two: pexpr;

.FORWARD

function makeexpr(exprtype: texprtype): pexpr; forward;

procedure destroyexpr(var e: pexpr); forward;

function kindexpr(e: pexpr): texprtype; forward;

procedure initexpr; forward;

.END



{ ------------------------------------------------------------}

{ PACKAGE BODY EXPR is }

.PROCEDURE

function makeexpr { (exprtype: texprtype): pexpr } ;
```



```
   var e       : pexpr;
       blankid : identifier;
       i       : integer;
   begin
   for i:= 1 to idlength do
      blankid[i]:=' ';
   new(e);
   e↑.pgfig           := nil;
   e↑.exprtype        := exprtype;
   e↑.exprvartype     := realexpr;
   e↑.cursor.iscursor := false;
   e↑.cursor.pos:=0;
   with e↑ do
      case exprtype of
         exprvariable:   varid           :=blankid;
         exprfunction:   begin
                         func            :=blankid;
                         argumentlist    :=nil;
                         NrOfSpaces      :=0;
                         SpaceCursor     :=0;
                         LeftParanthes   :=false;
                         RightParanthes  :=false;
                         ParanthesCursor:=0;
                         nargs           :=0;
                         end;
         exprnumber:     begin
                         numid           :=blankid;
                         val             :=0.0;
                         end;
         exprminus:      minusexpr       :=nil;
         exprnot:        notexpr         :=nil;
         exprifthenelse: begin
                         ifexpr          :=nil;
                         thenexpr        :=nil;
                         elseexpr        :=nil;
                         end;
         exprbinary:     begin
                         expr1           :=nil;
                         expr2           :=nil;
                         bintype         :=binaddop;
                         end;
         exprerror:      begin
                         for i:=1 to 80 do
                            s[i]:=' ';
                         ErrorNumber     :=0;
                         NextExpr        :=nil;
                         end;
         exprchinfo:     begin
                         ch              :=' ';
                         before          :=false;
                         chexpr          :=nil;
                         end;
         end; { case }

   makeexpr := e;
   end; { MakeExpr }

procedure destroyexpr { (var e: pexpr) } ;
```

```
   begin
   DisposeExpr(e);
   end;

function kindexpr ( (e: pexpr) ) ;
   begin
   kindexpr := e↑.exprtype;
   end;

procedure initexpr;
   begin
   zero := makeexpr(exprnumber);
   zero↑.numid := '0               ';
   zero↑.val := 0.0;

   one := makeexpr(exprnumber);
   one↑.numid := '1                ';
   one↑.val := 1.0;

   two := makeexpr(exprnumber);
   two↑.numid := '2                ';
   two↑.val := 2.0;
   end;

.INIT
initexpr;
.END
{ PACKAGE DisposeExpr is
{
{-- Dispose an expression
{
{-- Author: Magnus Taube
{-- Date:   1982-05-23
{}

.FORWARD
procedure DisposeExpr(var ex:pexpr);forward;

.PROCEDURE
procedure DisposeExpr(var ex:pexpr);
   var arg, argold : parglist;
   begin
   if ex <> nil then
      begin
      with ex↑ do
         begin
         case exprtype of
            exprvariable:;
            exprfunction:  begin
                           arg   :=argumentlist;
                           while arg <> nil do
                              begin
                              DisposeExpr(arg↑.ex);
                              argold:=arg↑.next;
                              dispose(arg);
                              arg:=argold;
                              end;
                           end;
```



```
            exprnumber:;
            exprminus:     DisposeExpr (minusexpr);
            exprnot:       DisposeExpr (notexpr);
            expriftheneIse:begin
                           DisposeExpr (ifexpr);
                           DisposeExpr (thenexpr);
                           DisposeExpr (elseexpr);
                           end;
            exprbinary:    begin
                           DisposeExpr(expr1);
                           DisposeExpr(expr2);
                           end;
            exprerror:     DisposeExpr(NextExpr);
            exprchinfo:    DisposeExpr(chexpr);
            otherwise      begin new(ex); ex↑.pgfig:=nil; end;
         end; { case }

         if pgfig <> nil then
            dispose(pgfig);
         end ; { with }
      Dispose(ex);
      ex:=nil;
      end ; { if }
   end ; { DisposeExpr }
.END
```

```
.END
 PACKAGE EXPROPER is

-- Routines for operations on expression nodes.

-- Author: Hilding Elmqvist
-- Date:   1981-07-27

Modified : Magnus Taube    Removed simplification

uses SETEXPR;


.FORWARD

function Addop(x, y: pexpr): pexpr; forward;

function Subop(x, y: pexpr): pexpr; forward;

function Multop(x, y: pexpr): pexpr; forward;

function Divop(x, y: pexpr): pexpr; forward;

function Powerop(x, y: pexpr): pexpr; forward;

function Equalop(x, y: pexpr): pexpr; forward;

function Minusop(x: pexpr): pexpr; forward;

function Ifthenelseop(x, y, z: pexpr): pexpr; forward;

function Notop(x: pexpr): pexpr; forward;

function Andop(x, y: pexpr): pexpr; forward;

function Orop(x, y: pexpr): pexpr; forward;

function Lessop(x, y: pexpr): pexpr; forward;

function Greaterop(x, y: pexpr): pexpr; forward;

.END
 ------------------------------------------------


 PACKAGE BODY EXPROPER is

.PROCEDURE

  function Addop(x, y: pexpr): pexpr};
```



```
    var   exprtemp: pexpr;
    begin
{    if y↑.exprtype=exprminus then
{      Addop:=Subop(x, y↑.minusexpr)
{    else if x↑.exprtype=exprminus then
{      Addop:=Subop(y, x↑.minusexpr)
{    else
{}
      begin
      exprtemp := makeexpr(exprbinary);
      setexprbinary(exprtemp, binaddop, x, y);
      Addop:=exprtemp;
      end;
    end;


  function Subop(x, y: pexpr): pexpr};

    var   exprtemp: pexpr;
    begin
{   if y↑.exprtype=exprminus then
{     Subop:=Addop(x, y↑.minusexpr)
{   else
{}
      begin
      exprtemp := makeexpr(exprbinary);
      setexprbinary(exprtemp, binsubop, x, y);
      Subop:=exprtemp;
      end;
    end;


  function Multop(x, y: pexpr): pexpr};

    var   exprtemp: pexpr;
    begin
{    if y↑.exprtype=exprminus then
{      Multop:=Minusop(Multop(x, y↑.minusexpr))
{    else if x↑.exprtype=exprminus then
{      Multop:=Minusop(Multop(x↑.minusexpr, y))
{    else
{}
      begin
      exprtemp := makeexpr(exprbinary);
      setexprbinary(exprtemp, binMultop, x, y);
      Multop:=exprtemp;
      end;
    end;


  function Divop(x, y: pexpr): pexpr};

    var   exprtemp: pexpr;
    begin
      begin
      exprtemp := makeexpr(exprbinary);
```

```
    setexprbinary(exprtemp, bindivop, x, y);
    Divop:=exprtemp;
    end;
  end;


 function Powerop(x, y: pexpr): pexpr);

  var   exprtemp: pexpr;
  begin
    exprtemp := makeexpr(exprbinary);
    setexprbinary(exprtemp, binPowerop, x, y);
    Powerop:=exprtemp;
  end;


 function Equalop(x, y: pexpr): pexpr);
  var   exprtemp: pexpr;
  begin
  exprtemp := makeexpr(exprbinary);
  setexprbinary(exprtemp, binEqualop, x, y);
  Equalop:=exprtemp;
  end;



function Minusop(x: pexpr): pexpr);
  var   exprtemp: pexpr;
  begin
{  if x↑.exprtype=exprminus then
{    Minusop:=x↑.minusexpr
{  else if (x↑.exprtype=exprbinary) and
{{***   (x↑.bintype=binsubop) then
{    Minusop:=Subop(x↑.expr2, x↑.expr1)
{  else
{}
  begin
  exprtemp := makeexpr(exprminus);
  setexprminus(exprtemp, x);
  Minusop:=exprtemp;
  end;
 end;


function Ifthenelseop(x, y, z: pexpr): pexpr);

 {     Simplification rule:
   if x then y else y = y }

 var   exprtemp: pexpr;
 begin
{  if y=z then
{    Ifthenelseop:=y
{  else
{}
  begin
  exprtemp := makeexpr(exprifthenelse);
  setexprif(exprtemp, x, y, z);
```



```
    Ifthenelseop:=exprtemp;
    end;
 end;


function Notop(x: pexpr): pexpr);
 var   exprtemp: pexpr;
 begin
 exprtemp := makeexpr(exprnot);
 setexprnot(exprtemp, x);
 Notop:=exprtemp
 end;


function Andop(x, y: pexpr): pexpr);
 var   exprtemp: pexpr;
 begin
 exprtemp := makeexpr(exprbinary);
 setexprbinary(exprtemp, binandop, x, y);
 Andop:=exprtemp
 end;


function Orop(x, y: pexpr): pexpr);
 var   exprtemp: pexpr;
 begin
 exprtemp := makeexpr(exprbinary);
 setexprbinary(exprtemp, binorop, x, y);
 Orop:=exprtemp
 end;


function Lessop(x, y: pexpr): pexpr);
 var   exprtemp: pexpr;
 begin
 exprtemp := makeexpr(exprbinary);
 setexprbinary(exprtemp, binlessop, x, y);
 Lessop:=exprtemp
 end;


function Greaterop(x, y: pexpr): pexpr);
 var   exprtemp: pexpr;
 begin
 exprtemp := makeexpr(exprbinary);
 setexprbinary(exprtemp, bingreaterop, x, y);
 Greaterop:=exprtemp
 end;

.END
```

```
==============================================================

                  use:[magnus.exarb.expr]exprset.pak

==============================================================


{ PACKAGE EXPRSET is

-- Routines for assignment of attributes in
-- expression nodes.

-- Author: Hilding Elmqvist
-- Date:   1981-07-27

uses EXPR;

}


.FORWARD
procedure setexprvar(ex: pexpr;  varid: identifier); forward;

procedure setexprfunc(ex: pexpr;  id: identifier; argumentlist: parglist;
                      nargs: integer); forward;

procedure setexprnum(ex: pexpr;  numid: identifier;
                     val: real); forward;

procedure setexprminus(ex: pexpr;  minusexpr: pexpr); forward;

procedure setexprnot(ex: pexpr;  notexpr: pexpr); forward;

procedure setexprif(ex: pexpr;
                    ifexpr, thenexpr, elseexpr: pexpr); forward;

procedure setexprbinary(ex: pexpr; bintype: tbintype;
                        expr1, expr2: pexpr); forward;

.END
{ end EXPRSET; }


{-------------------------------------------------------------}



{ PACKAGE BODY EXPRSET is
}
.PROCEDURE

procedure setexprerror(errnum: integer);
  begin
  writeln(chr(7),'SETEXPR ERROR ',errnum:1);
  end;


procedure setexprvar(ex: pexpr;  varid: identifier);
  begin
```



```
  if ex↑.exprtype <> exprvariable then
    setexprerror(1)
  else
    begin
    ex↑.varid := varid;
    end;
  end;


procedure setexprfunc(ex: pexpr;  id: identfier; argumentlist: parglist;
                      nargs: integer);
   var i : integer;
       p : parglist;
   begin
   if ex↑.exprtype <> exprfunction then
      setexprerror(1)
   else
      begin
      i:=0;
      p:=argumentlist;
      while p<>nil do
         begin
         i:=i+1;
         p:=p↑.next;
         end;
      if i <> nargs then
         setexprerror(2);
      ex↑.func:=id;
      ex↑.argumentlist := argumentlist;
      ex↑.nargs := nargs;
      end;
  end;


procedure setexprnum(ex: pexpr;  numid: identifier;
                     val: real);
  begin
  if ex↑.exprtype <> exprnumber then
    setexprerror(1)
  else
    begin
    ex↑.numid := numid;
    ex↑.val := val;
    end;
  end;


procedure setexprminus(ex: pexpr;  minusexpr: pexpr);
  begin
  if ex↑.exprtype <> exprminus then
    setexprerror(1)
  else
    begin
    ex↑.minusexpr := minusexpr;
    end;
  end;
```

```
procedure setexprnot(ex: pexpr;  notexpr: pexpr);
  begin
  if ex↑.exprtype <> exprnot then
    setexprerror(1)
  else
    begin
    ex↑.notexpr := notexpr;
    end;
  end;


procedure setexprif(ex: pexpr;
                    ifexpr, thenexpr, elseexpr: pexpr);
  begin
  if ex↑.exprtype <> exprifthenelse then
    setexprerror(1)
  else
    begin
    ex↑.ifexpr := ifexpr;
    ex↑.thenexpr := thenexpr;
    ex↑.elseexpr := elseexpr;
    end;
  end;


procedure setexprbinary(ex: pexpr; bintype: tbintype;
                        expr1, expr2: pexpr);
  begin
  if ex↑.exprtype <> exprbinary then
     setexprerror(1)
  else
     begin
     ex↑.expr1:= expr1;
     ex↑.expr2 := expr2;
     ex↑.bintype := bintype;
     end;
{ *** }
  if bintype in [binequalop,binandop,binorop,bingreaterop,binlessop] then
     ex↑.exprvartype:=booleanexpr;
{ *** inserted by MT }
  end;


.END
```



```
.END
Author      : Magnus Taube
Copies an expression, including information for the exression-editor.

Removes  redundant charater information in an expr.

.FORWARD
function CopyExpr(ex:pexpr):pexpr;     forward;
procedure RemoveChInfo(var ex : pexpr);forward;

.PROCEDURE
function CopyExpr(ex:pexpr):pexpr);
var res          : pexpr;
    arg, resarg : parglist;
   begin
   res:=nil;
   if ex <> nil then
      begin
      new(res);
      res↑:=ex↑;
      if ex↑.pgfig<>nil then
         begin
         new(res↑.pgfig);
         res↑.pgfig↑:=ex↑.pgfig↑;
         end;
      with ex↑ do
         case exprtype of
            exprvariable:;
            exprfunction:  begin
                           arg    := argumentlist;
                           resarg:=nil;
                           while arg <> nil do
                              begin
                              if resarg = nil then  { first element in the}
                                                    { argument-list }
                                 begin
                                 new(resarg);
                                 res↑.argumentlist:=resarg;
                                 end
                              else
                                 begin
                                 new(resarg↑.next);
                                 resarg:=resarg↑.next;
                                 end;
                              resarg↑.ex:=CopyExpr(arg↑.ex);
                              arg       :=arg↑.next;
                              end;
                           end; { exprFunction }
            exprnumber:;
            exprminus:     res↑.minusexpr:=CopyExpr (minusexpr);
            exprnot:       res↑.notexpr  :=CopyExpr (notexpr);
            exprifthenelse:begin
                           res↑.ifexpr   :=CopyExpr (ifexpr);
                           res↑.thenexpr :=CopyExpr (thenexpr);
```

```
                        res↑.elseexpr :=CopyExpr (elseexpr);
                        end;
            exprbinary: begin
                        res↑.expr1    :=CopyExpr(expr1);
                        res↑.expr2    :=CopyExpr(expr2);
                        end;
            exprerror:  res↑.NextExpr :=CopyExpr(NextExpr);
            exprchinfo: res↑.chexpr   :=CopyExpr(chexpr);
         end; { case }
      end ; { if }
   CopyExpr:=res;
   end ; { CopyExpr }


procedure RemoveChInfo(var ex : pexpr);
   var hlp : pexpr;
       arg : parglist;
   begin
   if ex <> nil then
      begin
      with ex↑ do
         case exprtype of
            exprvariable:  ;
            exprfunction:  begin
                           NrOfSpaces:=0;
                           arg:=argumentlist;
                           while arg <> nil do
                              begin
                              RemoveChInfo(arg↑.ex);
                              arg:=arg↑.next;
                              end;
                           end;
            exprnumber:    ;
            exprminus:     RemoveChInfo(minusexpr);
            exprnot:       RemoveChInfo(notexpr);
            exprifthenelse:begin
                           RemoveChInfo(ifexpr);
                           RemoveChInfo(thenexpr);
                           RemoveChInfo(elseexpr);
                           end;
            exprbinary:    begin
                           RemoveChInfo(expr1);
                           RemoveChInfo(expr2);
                           end;
            exprerror:     if ErrorNumber = -1 then
                              begin
                              RemoveChInfo(nextexpr);
                              hlp:=nextexpr;
                              ex↑.nextexpr:=nil;
                              DisposeExpr(ex);
                              ex:=hlp;
                              end
                           else
                              RemoveChInfo(NextExpr);

            exprchinfo:    begin
                           RemoveChInfo(chexpr);
```



```
                           hlp:=chexpr;
                           chexpr:=nil;
                           DisposeExpr(ex);
                           ex:=hlp;
                           end;
         end; { case }
      end;

   end ; { RemoveChInfo }

.END
```

```
=============================================================

                    use:[magnus.exarb.exprJdemorgan.pak

=============================================================

{ PACKAGE deMorgan  is
{
{-- Converts logical expressions through de Morgans theorem
{
{-- NOT(x and y)=NOT x or  NOT y
{-- NOT(x or  y)=NOT x and NOT y
{
{-- Author: Magnus Taube
{-- Date:   1982-06-11
{}

.FORWARD
function deMorgan(ex:pexpr):pexpr;forward;

.PROCEDURE
function deMorgan{ex:pexpr):pexpr};
   var res, left, right : pexpr;
       flag             : boolean;

   begin
   if ex = nil then
      res:=nil
   else
      with ex↑ do
         case exprtype of
            exprvariable,
            exprfunction,
            exprnumber,
            exprminus,
            exprifthenelse,
            exprerror:   res:=ex; { not relevant types of expressions }
            exprnot:     begin
                         flag:=false;
                         if notexpr <> nil then
                            if notexpr↑.exprtype = exprbinary then
                               begin
                               if notexpr↑.bintype in
                                    [binandop, binorop] then
                                  begin
                                  left :=MakeExpr(exprnot);
                                  right:=MakeExpr(exprnot);
                                  left↑.notexpr :=notexpr↑.expr1;
                                  right↑.notexpr:=notexpr↑.expr2;
                                  if notexpr↑.bintype = binandop then
                                     notexpr↑.bintype:= binorop
                                  else
                                     notexpr↑.bintype:= binandop;
                                  notexpr↑.expr1:= left;
                                  notexpr↑.expr2:=right;
                                  notexpr↑.expr1:=deMorgan(notexpr↑.expr1);
                                  notexpr↑.expr2:=deMorgan(notexpr↑.expr2);
                                  res:=notexpr;
                                  dispose(ex);
                                  flag:=true;
```



```
                                  end;
                               end
                            else if notexpr↑.exprtype = exprnot then
                               begin
                               res:=deMorgan(notexpr↑.notexpr);
                               dispose(notexpr);
                               dispose(ex);
                               flag:=true;
                               end;
                         if not flag then
                            res:=ex;
                         end;

            exprbinary:  begin
                         expr1:=deMorgan(expr1);
                         expr2:=deMorgan(expr2);
                         res:=ex;
                         end;
         end; { case, with and if }
      deMorgan:=res;
   end ; { deMorgan }
.END
```

```
=============================================================

                use:[magnus.exarb.exprJscanner.PAK

=============================================================

.END

Author : ?
Modified : Magnus Taube. SetScannerData and som other modifications.
.VAR
       nextitem: identifier;
       nexttype: (idtype,deltype,numbtype,eolntype);
       nextnumb: real;

       nullchar : char;
.FORWARD


procedure InLineScan;                                    forward;
procedure SetScannerData(im : image; Pos : integer);     forward;

.END

{ ------------------------------------------------------------------------ }

{ PACKAGE BODY INOUT is
}

.TYPE
       tsys=record
         ScanImage        : image; { packed array[1..ImageLength] of char;}
         cursorpos,                { cursor position on line }
         pos, length      : integer;
         more             : boolean;
         end;

.VAR
       ScanData       : tsys;
       LengthNextItem : integer;

.PROCEDURE

procedure SetScannerData(im : image; Pos : integer);
   begin
   with ScanData do
      begin
      ScanImage := im;
      CursorPos := Pos;
      Pos       := 1;
      More      := true;
      Length    := LastNonSpace(im);
      end;
   end;

{ ------------------------------------------------------------------------ }
procedure InLineScan;
label 1;
var   ipos : integer;
      ch   : char;
```



```
      skip : boolean;


{
  Gets next item: identifier, number, delimiter or end-of-line.
  Delimiters are everything that not is something else, they have
  a length of 1, except the power-operator **.
  The next item is put in nextitem, its type in nexttype, the value
  of a number in nextnumb and the items length in LengthNextItem.

{ ------------------------------------------------------------------------ }


procedure EndOfLine;
{  called when there is no more input.
{  This scanner can only handle one line.
{}
   var i:integer;
   begin
   NextType:=EolnType;
   for i:=1 to idlength do NextItem[i]:=nullchar;
   goto 1;
   end;


function InChar:char;
   {
   {  Gets next character from input stream
   {  Globals: ScanData
   {}
   begin
   with ScanData do
      begin
      if (not more) or (pos > length) then
         begin
         if pos = length+1 then
            pos:=pos+1;
         inchar:=NullChar;
         more:=false;
         end
      else
         begin
         inchar:=ScanImage[pos];
         pos:=pos+1;
         more:= pos <= length;
         end;
      end; { with ScanData }
   end; { InChar }

{ ------------------------------------------------------------------------ }

function NextChar:char;
{ Returns next character in ScanData. Does not change pos in ScanData,
{ if eoln NullChar is returned. }
   begin
   with ScanData do
      if more then
         NextChar:=ScanImage[pos]
```

```
      else
         NextChar:=Nullchar;
   end; { NextChar }

{ ------------------------------------------------------------------------ }
procedure SetPos(newpos:integer);
{  Sets position pointer in image
{  Globals: ScanData
{}
begin
ScanData.pos:=newpos;
ScanData.more := newpos <= ScanData.length;
end { SetPos };

{ ------------------------------------------------------------------------ }

function InReal:real;
{
  Decodes a real number
  Calls: InChar, Digit, SetPos
  Globals: ScanData
}
var   xreal,fracdiv:real;
      tenexp,i:integer;
      c:char;
      neg:boolean;
begin
xreal:=0;

{ integer part }
while Digit(NextChar) do
  begin
  c:=InChar;
  xreal:=xreal*10+ord(c)-ord('0');
  end;

{ fraction part }
if NextChar='.' then
  begin
  c:=InChar;       { the . }
  fracdiv:=1;
  while Digit(NextChar) do
    begin
    c:=InChar;
    fracdiv:=fracdiv/10;
    xreal:=xreal+(ord(c)-ord('0'))*fracdiv;
    end;
  end;

{ exponent }
if NextChar='E' then
  begin
  c:=InChar;
  if NextChar in ['+','-'] then
    begin
    c:=InChar;
    neg:= c = '-';
    end;
```



```
  tenexp:=0;
  while Digit(NextChar) do
    begin
    c:=InChar;
    tenexp:=tenexp*10 +ord(c)-ord('0');
    end;
  for i:=1 to tenexp do
     if neg then
        xreal:=xreal/10.0
     else
        xreal:=xreal*10.0;
  end;

InReal:=xreal;
end { InReal };

{ ------------------------------------------------------------------------ }

procedure ImageSub(start,lght:integer;var result:identifier);
   {  Extracts substring from image
   {  Globals: ScanData
   {}
   var i:integer;
   begin
   result:='                ';
   for i:=1 to lght do
      result[i]:=ScanData.ScanImage[start+i-1];
   end { ImageSub };

{ ------------------------------------------------------------------------ }

begin { InLineScan }
with ScanData do
   begin
   more:=length >= pos;
   ipos:=pos;
   ch:=NextChar;

   if ch = nullchar then
      EndOfLine
   else if Digit(ch) then
      begin
      nexttype:=numbtype;
      nextnumb:=InReal;
      end
   else if Letter(ch) then
      begin
      nexttype:=idtype;
      while Letter(NextChar) or Digit(Nextchar) do
         ch:=InChar ;
      end

   else if length = 0 then
      EndOfLine
   else
      begin
      ch:=Inchar;
      nexttype:=deltype;
```

```
      if (ch='*') and (NextChar = '*') then  { PowerOp - symbol }
         ch:=InChar;
      end;

   ImageSub(ipos,pos-ipos,nextitem);
   LengthNextItem:=pos-ipos;
   end; { with ScanData }
1:
end (* InLineSCAN *) ;

{ ---------------------------------------------------------------------- }


.INIT
nextitem:='                ';
ScanData.more:=false;
nullchar:=chr(00);

.END
```



```
{ Package Equation

-- Routines to parse equations.
Author   : ?
Modified :  Magnus Taube.  Heavily modified to create error-nodes
                           in expression.

}
.FORWARD
function Equation  :pexpr;forward;
function Expression:pexpr;forward;
function Assignment:pexpr;forward;
{ body is... }


.FORWARD
function DoEquation(Equat:integer):pexpr;                forward;


.PROCEDURE


function DoEquation(Equat:boolean):pexr};
   var help, equ              : pexpr;
       i, CursorPlace, OldPos : integer;
       CursorHere, EquationOK : boolean;

   { Equation } function expression: pexpr;forward;

{ ---------------------------------------------------------------------- }

   procedure Scan;
      begin
      OldPos :=ScanData.pos;
      InLineScan;

      if (NextType = EolnType) and (ScanData.cursorpos ) ScanData.length)
               then
         begin
         CursorHere:=true;
         CursorPlace:=1;
         end
      else if ((NextType () EolnType) and
               (ScanData.cursorpos )= ScanData.pos-LengthNextItem) and
               (ScanData.cursorpos (  ScanData.pos))                then
         begin
         CursorHere:=true;
         CursorPlace:=LengthNextItem-(ScanData.pos-ScanData.cursorpos)+1;
         end
      else
         CursorHere:=false;
      end;
```

```
{ ------------------------------------------------------------------------ }

   function ConcExpr(chex,ex:pexpr):pexpr;
      { Adds ex to the list of ch-ex. If chexpr is nil, the
      { result is ex.
      {}
      var help : pexpr;
      begin
      if chex = nil then
         ConcExpr:=ex
      else
         begin
         help:=chex;
         while help^.chexpr <> nil do
            help:=help^.chexpr;
         help^.chexpr:=ex;
         ConcExpr:=chex;
         end;
      end; { ConcExpr }

{ ------------------------------------------------------------------------ }

   function CreateChInfo(ch:char;before:boolean):pexpr;
      { Creates an expression-node with type
      { exprchinfo and value ch & before.
      {}
      var ex : pexpr;
      begin
      ex:=MakeExpr(exprchinfo);
      ex^.before:=before;
      ex^.ch:=ch;
      ex^.chexpr:=nil;
      ex^.cursor.iscursor:=CursorHere;
      ex^.cursor.pos:=1;
      ex^.cursor.StartPos:=OldPos;
      CreateChInfo:=ex;
      end;

{ ------------------------------------------------------------------------ }

   function ScanSpace(before:boolean):pexpr;
      var ex, help : pexpr;
      begin
      ex:=nil;
      while (LengthNextItem=1) and (nextitem[1]=' ') do
         begin
         if before then
            ex:=ConcExpr(ex,CreateChInfo(' ',before))
         else
            begin
            help:=CreateChInfo(' ',before);
            help^.chexpr:=ex;
            ex:=help;
            end;
         Scan;
         end;

      ScanSpace:=ex;
```



```
      end; { ScanSpace }

{ ------------------------------------------------------------------------ }

   { Equation } function NextCh:char;
      begin
      if NextType in [idtype,deltype] then
         NextCh:=NextItem[1]
      else
         NextCh:=chr(00);
      end; { NextCh }

{ ------------------------------------------------------------------------ }

    function BigLetter(c:char):char;
      begin
      if c in ['a'..'z'] then BigLetter:=chr(ord(c)-32) { ASCII }
                         else BigLetter:=c;             { assumed }
      end; { BigLetter }

   { Equation } function NextString(s:text4):boolean;
      var i  : integer;
          eq : boolean;

      begin { NextString }
      eq:=NextType in [idtype,deltype];
      i:=0;
      while (i<4) and eq do
         begin
         i:=i+1;
         eq:=s[i]=BigLetter(NextItem[i]);
         end;
      NextString:=eq;
      end; { NextString }

 { ------------------------------------------------------------------------ }

   { Equation } function Error(ErrNr:integer):pexpr;

   { Error gives a pointer to an errordescription, containing
   { the rest of the input line, the errornumber and, if relevant, a
   { pointer to an expression.
   { Errornumber  0 (zero) signals an empty record.
   {   -- " --   -1 signals eoln-record
   { }
      var s : string80;
          i : integer;
          ex: pexpr;

      { Error } procedure Konk(var s:string80;var i:integer;id:identifier);
         var k : integer;
         begin
         for k:=1 to LengthNextItem do
            begin
            s[i]:=id[k];
            if i < 80 then
               i:=i+1;
            end;
         end; { Konk }
```

```
      begin { Error }
      ex:=MakeExpr(ExprError);
      ex↑.cursor.StartPos:=ScanData.pos;
      for i:=1 to 80 do
         s[i]:=' ';
      i:=1;
      if (ErrNr > 0) and EquationOK then
         begin
         while NextType <> EolnType do
            begin
            Konk(s,i,NextItem);
            if CursorHere then
               begin
               ex↑.cursor.iscursor :=true;
               ex↑.cursor.pos      :=i-LengthNextItem+CursorPlace-1;
               end;
            Scan;
            end;
         if CursorHere then
            begin
            ex↑.cursor.iscursor :=true;
            ex↑.cursor.pos      :=i;
            end;
         end
      else if ErrNr < 0 then            { end of line }
         begin
         ex↑.cursor.iscursor:=CursorHere;
         ex↑.cursor.pos:=1;
         end;

      ex↑.s:=s;
      ex↑.ErrorNumber:=ErrNr;
      ex↑.NextExpr:=nil;
      ex↑.ErrLength:=i;
      Error:=ex;
      EquationOK:=false;
      end; { Error }

{ ------------------------------------------------------------------------ }

   { Equation } function PRIMARY: pexpr;
      var ex, help : pexpr;

      { Primary } function Paranthes:pexpr;
          var ex1, ex2, help : pexpr;
                cur  : boolean;
          begin
          ex1 :=CreateChInfo('(',true);{ true (-) before the expr follow}
          Scan;                        { skip the '('  in nextitem    }
          if (NextType = EolnType) and CursorHere then
             begin
             ex1↑.cursor.iscursor:=true;
             ex1↑.cursor.pos:=2;
             CursorHere:=false;
             end;
          help:=ScanSpace(true);
          help:=ConcExpr(ex1,help);
```



```
          if NextType = EolnType then
             begin
             if not EquationOK then
                begin
                ex1:=Error(-2);
                ex1↑.cursor.iscursor:=CursorHere;
                Paranthes:=ConcExpr(help,ex1);
                end
             else
                Paranthes:=help;
             end
          else
             begin
             ex1:=Expression;
             ex1:=ConcExpr(ScanSpace(false),ex1);  { if spaces after expr }

             if NextCh <> ')' then
                begin
                if EquationOk then
                   begin
                   if NextType = EolnType then
                      ex2:=Error(-2)
                   else
                      ex2:=Error(1);
                   ex1:=ConcExpr(help,ex1);
                   ex2↑.NextExpr:=ex1;
                   Paranthes:=ex2;
                   end
                else
                   Paranthes:=ConcExpr(help,ex1);
                end
             else
                begin
                ex1:=ConcExpr(help,ex1);
                Paranthes:=ConcExpr(CreateChInfo(')',false),ex1);
                Scan;
                end;
             end;
          end ; { Paranthes }

       { Primary } function Ident:pexpr;
          var ex1, ex2, help, h : pexpr;
              id                : identifier;
              narg              : integer;
              OK                : boolean;
              arglist,
              arg               : parglist;
              cur               : boolean;
              curpos, op        : integer;
          begin
          cur:=CursorHere;
          curpos:=CursorPlace;
          op:=OldPos;
          id:=NextItem;
          Scan;
          help:=ScanSpace(false);     { Spaces after the identifier }
          if NextCh <> '(' then       { it is not a function }
             begin
```

```
            ex1:= makeexpr(exprvariable);
            setexprvar(ex1, id);
            ex1↑.cursor.iscursor:=cur;
            ex1↑.cursor.pos:=curpos;
            ex1↑.cursor.StartPos:=op;
            Ident:=ConcExpr(help,ex1);
            end
         else                          { a function and it's argument }
            begin
            ex1:=Makeexpr(exprfunction);
            ex1↑.NrOfSpaces:=0;
            h:=help;                   { The spaces after the funtion }
            while h <> nil do          { identifier but before the    }
               begin                   { (-char is counted.           }
               ex1↑.NrOfSpaces:=ex1↑.NrOfSpaces + 1;
               if h↑.cursor.iscursor then
                  ex1↑.SpaceCursor:=ex1↑.NrOfSpaces;
               h:=h↑.chexpr;
               end;
            if ex1↑.SpaceCursor <> 0 then
               ex1↑.SpaceCursor:=1 + ex1↑.NrOfSpaces - ex1↑.SpaceCursor;
            DisposeExpr(help);
            ex1↑.cursor.iscursor:=cur;
            ex1↑.cursor.pos:=curpos;
            ex1↑.cursor.StartPos:=OldPos;
            ex1↑.LeftParanthes:=true;
            if CursorHere then
               ex1↑.ParanthesCursor:=1;
            Scan;                      { skip the ( -character        }
            arg:=nil;
            arglist:=nil;
            with ex1↑ do
               begin
               OK:=true;
               narg:=0;
               while (NextCh <> ')')  and OK and EquationOK do
                  begin
                  if (narg > 0) and (NextCh <> ',') then
                     begin
                     if NextType = EolnType then
                        ex2:=Error(-1)
                     else
                        ex2:=Error(10);
                     OK:=false;
                     end
                  else
                     begin
                     if narg = 0 then
                        begin
                        new(arg);
                        arg↑.next:=nil;
                        arg↑.Comma:=false;  { No ,-char after the expr }
                        arglist:=arg;
                        end
                     else
                        begin
                        new(arg↑.next);
                        arg:=arg↑.next;
```



```
                        arg↑.next:=nil;
                        arg↑.Comma:=false;
                        Scan;
                        end;
                     arg↑.ex:=Expression;
                     arg↑.Comma:=NextCh = ',';
                     arg↑.CommaCursor:=(NextCh  = ',') and CursorHere;
                     narg:=narg+1;
                     end;
                  end; { while }
               end; { with }

            setexprfunc(ex1,id,arglist,narg);
            if OK then
               begin
               if EquationOK then
                  begin
                  if CursorHere then
                     ex1↑.ParanthesCursor:=2;
                  ex1↑.RightParanthes:=true;
                  Scan;                          { skip the )-character }
                  Ident:=ConcExpr(ScanSpace(false),ex1);
                  end
               else
                  Ident:=ex1;
               end
            else
               begin
               ex2↑.nextexpr:=ex1;
               Ident:=ex2;
               end;
            end; { function }
         end; { Ident }

      { Primary } function Number:pexpr;
         var ex1 : pexpr;
         begin
         ex1:=makeexpr(exprnumber);
         setexprnum(ex1, Nextitem, Nextnumb);
         ex1↑.cursor.iscursor:=CursorHere;
         ex1↑.cursor.pos:=CursorPlace;
         ex1↑.cursor.StartPos:=OldPos;
         Number:=ConcExpr(ScanSpace(false),ex1);
         Scan;
         end; { Number }


      begin { Primary }
         help:=ScanSpace(true);
         if NextCh = '(' then
            ex:=Paranthes
         else if Nexttype =  IdType then
            ex:=Ident
         else if Nexttype=Numbtype then
            ex:=Number
         else if NextType = EolnType then
            ex:=Error(-2)
         else
```

```
            ex:=Error(3);
         Primary:=ConcExpr(help,ex);
      end; { PRIMARY }

{ ------------------------------------------------------------------------ }

   { Equation } function Factor:pexpr;
      var ex          : pexpr;
          cur         : boolean;
          curpos, op  : integer;
      begin
      ex:=Primary;
      ex:=ConcExpr(ScanSpace(false),ex);
      while NextString('**  ') do
         begin
         op:=OldPos;
         cur:=CursorHere;
         curpos:=CursorPlace;
         Scan;
         ex:= Powerop(ex,Primary);
         ex:=ConcExpr(ScanSpace(false),ex);
         ex↑.cursor.iscursor:=cur;
         ex↑.cursor.pos:=curpos;
         ex↑.cursor.StartPos:=OldPos;
         end;
      Factor:=ConcExpr(ScanSpace(false),ex);
   end; { Factor }

{ ------------------------------------------------------------------------ }

   { Equation } function Term: pexpr;
      var   ex, help : pexpr;
            t        : boolean;
            op       : integer;
      begin
      ex:=Factor;
      ex:=ConcExpr(ScanSpace(false),ex);
      while (NextCh='*') or (NextCh='/') do
         begin
         t:=CursorHere;
         op:=OldPos;
         if NextCh = '*' then
            begin
            Scan;
            help:=Factor;
            help:=ConcExpr(ScanSpace(false),help);
            ex:=Multop(ex,help);
            end
         else
            begin
            Scan;
            help:=Factor;
            help:=ConcExpr(ScanSpace(false),help);
            ex:=Divop(ex,help);
            end;
         ex↑.cursor.iscursor:=t;
         ex↑.cursor.pos:=1;
         ex↑.cursor.StartPos:=op;
```



```
         end; { while }
      Term:=ex;
      end;

{ ------------------------------------------------------------------------ }

   { Equation } function Simpexpr: pexpr;
      var   ex, help       : pexpr;
            t, OMore       : boolean;
            i, OPos, p, op : integer;
      begin
      help:=ScanSpace(true);
      t:=CursorHere;
      p:=CursorPlace;
      op:=OldPos;
      if NextCh = '+' then
         begin
         help:=ConcExpr(help,CreateChInfo('+',true));
         Scan;
         ex:=Term;
         end
      else if NextCh = '-' then
         begin
         Scan;
         ex:=MinusOp(term);
         ex↑.cursor.iscursor:=t;
         ex↑.cursor.pos:=1;
         ex↑.cursor.StartPos:=op;
         end
      else if NextString('NOT ') then
         begin
         Scan;
         ex:=Notop(Term);
         ex↑.cursor.iscursor:=t;
         ex↑.cursor.pos:=p;
         ex↑.cursor.StartPos:=op;
         ex↑.NotLength:=3;
         end
      else if NextString('N   ') or NextString('NO  ') then
         begin
         if NextString('N   ') then
            i:=1
         else
            i:=2;
         OMore:=ScanData.more;
         OPos :=OldPos;
         Scan;
         if NextType = EolnType then
            begin
            ex:=NotOp(Error(-1));
            ex↑.cursor.iscursor:=t;
            ex↑.cursor.pos:=p;
            ex↑.cursor.StartPos:=op;
            ex↑.NotLength:=i;
            end
         else   { Undo the Scan-operation }
            begin
```

```
            ScanData.Pos:=OldPos;
            OldPos:=OPos;
            ScanData.more:=OMore;
            end
         end
      else
         begin
         ex:=Term;
         end;

      ex:=ConcExpr(help,ex);

      ex:=ConcExpr(ScanSpace(false),ex); { spaces before the +(-) char }
      while (NextCh = '+') or (NextCh = '-') do
         begin
         t:=cursorhere;
         p:=cursorplace;
         op:=OldPos;
         if NextCh = '+' then
            begin
            Scan;
            help:=Term;
            help:=ConcExpr(ScanSpace(false),help);   { spaces before next }
            ex:=AddOp(ex,help);                      { + (-) char         }
            end
         else
            begin
            Scan;
            help:=Term;
            help:=ConcExpr(ScanSpace(false),help);   { spaces before next }
            ex:=SubOp(ex,help);                      { + (-) char         }
            end;
         ex^.cursor.iscursor:=t;
         ex^.cursor.pos:=1;
         ex^.cursor.StartPos:=op;
         end;

      Simpexpr:=ex;
      end; { Simpexpr }

{ ------------------------------------------------------------------------ }

   { Equation } function Relation:pexpr;
      var ex, help : pexpr;
          t        : boolean;
          op       : integer;
      begin
      ex:=Simpexpr;
      ex:=ConcExpr(ScanSpace(false),ex); { spaces before ( and ) operator }
      t :=CursorHere;
      op:=OldPos;
      if NextCh in ['(',')'] then
         begin
         if NextCh = '(' then
            begin
            Scan;
            ex:=Lessop(ex,Simpexpr);
            end
```



```
         else
            begin
            Scan;
            ex:=Greaterop(ex,Simpexpr);
            end;

         ex^.cursor.iscursor:=t;
         ex^.cursor.pos:=1;
         ex^.cursor.StartPos:=op;
         end;
      Relation:=ex;
      end; { Relation }

{    ------------------------------------------------------------    }
 function ANDexpr:pexpr;
      var   ex, help : pexpr;
            t        : boolean;
            p, op    : integer;
      begin
      ex:=Relation;
      ex:=ConcExpr(ScanSpace(false),ex);
      while NextString('AND ') do
         begin
         t:=CursorHere;
         p:=CursorPlace;
         op:=OldPos;
         Scan;
         ex:=Andop(ex,Relation);
         ex^.cursor.iscursor:=t;
         ex^.cursor.pos:=p;
         ex^.cursor.StartPos:=op;
         ex:=ConcExpr(ScanSpace(false),ex);
         end;
      ANDexpr:=ex;
      end; { ANDexpr }


 function ORexpr:pexpr;
      var   ex, help : pexpr;
            t        : boolean;
            p, op    : integer;
      begin
      ex:=ANDexpr;
      ex:=ConcExpr(ScanSpace(false),ex);
      while NextString('OR  ') do
            begin
            t:=CursorHere;
            p:=CursorPlace;
            op:=OldPos;
            Scan;
            ex:=Orop(ex,ANDexpr);
            ex^.cursor.iscursor:=t;
            ex^.cursor.pos:=p;
            ex^.cursor.StartPos:=op;
            ex:=ConcExpr(ScanSpace(false),ex);
            end;
      ORexpr:=ex;
      end; { ORexpr }
```

```
{ ------------------------------------------------------------------------ }

   { Equation } function expression{ : pexpr  };
      var ex, expr1, expr2, expr3, help : pexpr;
      { expression } function PartOfNext(t : text4):integer;
      var i, OPos, CP  : integer;
          txt          : text4;
          OldMore, CH  : boolean;

         begin
         if NextType = EolnType then
            PartOfNext:=0
         else if (LengthNextItem > 3) or (NextType <> IdType) then
            PartOfNext:=-1              { Error }
         else
            begin
            txt:='    ';
            for i:=1 to LengthNextItem do
               txt[i]:=BigLetter(NextItem[i]);
            for i:=LengthNextItem+1 to 4 do
               t[i]:=' ';
            if (t <> txt) then
               PartOfNext:=-1
            else
               begin
               OldMore:=ScanData.More;
               Opos:=OldPos;
               CH:=CursorHere;
               CP:=CursorPlace;
               Scan;
               if NextType = EolnType then
                  PartOfNext:=LengthNextItem
               else
                  PartOfNext:=-1;
               ScanData.Pos:=OldPos;            { Undo the Scan-operation }
               ScanData.More:=OldMore;
               OldPos:=Opos;
               CursorHere:=CH;
               CursorPlace:=CP;
               end;
            end;
         end; { PartOfNext }

      begin
      help:=ScanSpace(true);         { spaces before IF }
      if NextString('IF  ') then
         begin
         ex:=makeexpr(ExprIfThenElse);
         ex^.cursor.iscursor:=CursorHere;
         ex^.cursor.pos     :=CursorPlace;
         Scan;
         ex^.cursor.StartPos:=OldPos;
         expr1:=Expression;
         expr1:=ConcExpr(ScanSpace(false),expr1);
         if NextString('THEN') then
            begin
            if CursorHere then
```



```
               begin
               ex^.cursor.iscursor:=true;
               ex^.cursor.pos     :=CursorPlace+2;
               end;
            ex^.thenpos:=OldPos;
            Scan;
            expr2:=ORexpr;
            expr2:=ConcExpr(ScanSpace(false),expr2);
            if NextString('ELSE') then
               begin
               if CursorHere then
                  begin
                  ex^.cursor.iscursor:=true;
                  ex^.cursor.pos     :=CursorPlace+6;
                  end;
               ex^.ElsePos:=OldPos;
               ex^.IfLength:=10;
               Scan;
               SetExprIf(ex,expr1,expr2,Expression);
               end
            else
               begin
               i:=PartOfNext('ELSE');
               if i <= 0 then
                  begin
                  ex^.IfLength:=6;
                  if (NextType = EolnType) and EquationOK then
                     SetExprIf(ex,expr1,expr2,Error(-2))
                  else if NextType <> EolnType then
                     SetExprIf(ex,expr1,expr2,Error(21))
                  else { not EquationOK }
                     SetExprIf(ex,expr1,expr2,Error(0));
                  end
               else
                  begin
                  ex^.IfLength:=6+i;
                  if CursorHere then
                     begin
                     ex^.cursor.iscursor:=true;
                     ex^.cursor.pos     :=CursorPlace+6;
                     end;
                  ex^.ElsePos:=OldPos;
                  Scan;
                  SetExprIf(ex,expr1,expr2,Error(-1));
                  end;
               end;
            end
         else { NextItem <> THEN }
            begin
            i:=PartOfNext('THEN');
            if i <= 0 then
               begin
               ex^.IfLength:=2;
               if (NextType = EolnType) and EquationOK then
                  SetExprIf(ex,expr1,Error(-2),Error(0))
               else if NextType <> EolnType then
                  SetExprIf(ex,expr1,Error(21),Error(0))
               else { not EquationOK }
```

```
                SetExprIf(ex,expr1,Error(0),Error(0));
             end
          else
             begin
             ex↑.IfLength:=2+i;
             if CursorHere then
                begin
                ex↑.cursor.iscursor:=true;
                ex↑.cursor.pos      :=CursorPlace+2;
                end;
             ex↑.thenpos:=OldPos;
             Scan;
             SetExprIf(ex,expr1,Error(-1),Error(0));
             end;
          end;
       ex:=ConcExpr(help,ex);
       end
    else        { NextItem () IF }
       ex:=ConcExpr(help,ORexpr);
    Expression:=ex;
 end; { Expression }

{ ------------------------------------------------------------------------ }

 { Equation } function Equation : pexpr ;
    var  ex, ex2, errex, exlocal, help : pexpr;
         t                             : boolean;
         op                            : integer;
    begin
    ex:= Expression;
    ex:= ConcExpr(ScanSpace(false),ex);
    t := CursorHere;
    op:=OldPos;
    if NextType = EolnType then
       begin
       if EquationOK then
          begin
          ex2:=Error(-2);
          ex2↑.nextexpr:=ex;
          exlocal:=ex2;
          end
       else
          exlocal:=ex;
       end
    else if not(NextCh = '=')then
       begin
       ex2:=Error(6);
       ex2↑.nextexpr:=ex;
       exlocal:=ex2;
       end
    else
       begin
       Scan;
       ex2:=Expression;
       ex2:=ConcExpr(ScanSpace(false),ex2); { take spaces after expr }
       if Nexttype = EolnType then
          begin
          if EquationOK then
```



```
             begin
             exlocal:=Error(-1);
             help:=EqualOp(ex,ex2);
             help↑.cursor.iscursor:=t;
             help↑.cursor.pos:=1;
             help↑.cursor.StartPos:=op;
             exlocal↑.nextexpr:=help;
             end
          else
             begin
             exlocal:=Equalop(ex,ex2);
             exlocal↑.cursor.iscursor:=t;
             exlocal↑.cursor.pos:=1;
             exlocal↑.cursor.StartPos:=op;
             end
          end
       else
          begin
          errex:=Error(8);
          errex↑.nextexpr:=ex2;
          help:=EqualOp(ex,errex);
          help↑.cursor.iscursor:=t;
          help↑.cursor.pos:=1;
          help↑.cursor.StartPos:=op;
          exlocal:=help;
          end;
       end;
    Equation :=exlocal;
    end; { Equation  }

{ ------------------------------------------------------------------------ }

 { Equation } function Assignment: pexpr ;
    var  ex, ex2, errex, exlocal, help : pexpr;
         t                             : boolean;
         op                            : integer;
    begin
    ex:= Primary;
    ex:= ConcExpr(ScanSpace(false),ex);
    t := CursorHere;
    op:=OldPos;
    if NextType = EolnType then
       begin
       if EquationOK then
          begin
          ex2:=Error(-2);
          ex2↑.nextexpr:=ex;
          exlocal:=ex2;
          end
       else
          exlocal:=ex;
       end
    else if not(NextCh = '=')then
       begin
       ex2:=Error(6);
       ex2↑.nextexpr:=ex;
       exlocal:=ex2;
       end
```

```
      else
         begin
         Scan;
         ex2:=Expression;
         ex2:=ConcExpr(ScanSpace(false),ex2); { take spaces after expr }
         if Nexttype = EolnType then
            begin
            if EquationOK then
               begin
               exlocal:=Error(-1);
               help:=EqualOp(ex,ex2);
               help↑.cursor.iscursor:=t;
               help↑.cursor.pos:=1;
               help↑.cursor.StartPos:=op;
               exlocal↑.nextexpr:=help;
               end
            else
               begin
               exlocal:=Equalop(ex,ex2);
               exlocal↑.cursor.iscursor:=t;
               exlocal↑.cursor.pos:=1;
               exlocal↑.cursor.StartPos:=op;
               end
            end
         else
            begin
            errex:=Error(8);
            errex↑.nextexpr:=ex2;
            help:=EqualOp(ex,errex);
            help↑.cursor.iscursor:=t;
            help↑.cursor.pos:=1;
            help↑.cursor.StartPos:=op;
            exlocal:=help;
            end;
         end;
      Assignment:=exlocal;
      end; { Assignment }

  { ------------------------------------------------------------------ }

   begin { DoEquation }

   EquationOK:=true;
   equ:=makeexpr(exprerror);
   Scan;
   if equat=0 then
      Equ:=Equation
   else if equat=1 then
      begin
      Equ:=Assignment;
{      Equ:=ConcExpr(ScanSpace(false),Equ);
{      if NextType <> EolnType then
{         begin
{         help:=Error(9);
{         help↑.nextexpr:=Equ;
{         Equ:=help;
{         end
{      else
```



```
{           begin
{           if EquationOK then
{              begin
{              help:=Error(-1);
{              help↑.nextexpr:=Equ;
{              Equ:=help;
{              end;
{           end;
{>       end
   else if equat=2 then
      begin
      Equ:=Expression;
      Equ:=ConcExpr(ScanSpace(false),Equ);
      if NextType <> EolnType then
         begin
         help:=Error(9);
         help↑.nextexpr:=Equ;
         Equ:=help;
         end
      else
         begin
         if EquationOK then
            begin
            help:=Error(-1);
            help↑.nextexpr:=Equ;
            Equ:=help;
            end;
         end;
      end
   else
      Equ:=nil;

   DoEquation:=equ;
   end; { DoEquation }


{ ====================END  EQUATION  ============================== }
function Equation{:pexpr};
   begin
   Equation:=DoEquation(0);
   end;

function Assignment{:pexpr};
   begin
   Assignment:=DoEquation(1);
   end;

function Expression{:pexpr};
   begin
   Expression:=DoEquation(2);
   end;
.END
```

```
=============================================================

                  use:[magnus.exarb.prettyl]Pretty.dfs

=============================================================

.END
Contains declarations for expression-editor.
Author : Magnus Taube
.TYPE

Vector        = record
   x,y  : real;
   end;

Paranthes     = record
   exist                      : boolean;
   LeftPar,RightPar,ParScale  : Vector;
   end;


Gfig          = record
   Size            : Vector ;  { total need of space for the expression  }
   par             : Paranthes;{ used in non-interactive mode to store   }
                               { parantheses (if any) place and size     }
   ParameterPlace : Vector;    { used when expression is a parameter to  }
                               { a function. Contains the place from the }
                               { functions point (0,0) to the parameters }
                               { point (0,0)                             }
   case texprtype of
      exprminus,
      exprnot       : (negationvect,negvect   : Vector);
      exprifthenelse: (ifvect,ifexvect,thenvect,thenexvect,
                       elsevect,elseexvect    : Vector);
      exprbinary    : (left,op,right,DivLine,
                       sqrtV,sqrtOver         : Vector);
      exprfunction  : (FunctionVect           : Vector);
      exprerror     : (NextVect,ErrorVect     : Vector;
                       SLength                : integer);
      exprchinfo    : (chvect,chexprvect      : Vector);
   end; { Gfig }
.END
```

```
=============================================================

                  use:[magnus.exarb.prettyl]Pretty.PAK

=============================================================

{ Module for "Pretty-print" of expressions.

Written by Magnus Taube
}

.CONST
  PrettyPowerScale = 0.625;    { Scale for right part of power-operation }
                               { expressions.                            }
  ERRORColor       = 1;        { Red.  Color for error-part of expression}
  CursorColor      = 5;        { Yellow. }
  ExColor          = 2;        { Blue. Normal expression color.          }
  PrettyCharWidth  = 23.4;     { Space between two characters. See also  }
  PrettyCharHeight = 26;       { constants in procedure Pretty.          }

.FORWARD

procedure PrePretty(var ex:pexpr;PrettyInter:boolean);     forward;
procedure    Pretty(    ex:pexpr;PrettyInter:boolean);     forward;
procedure ReturnExprSpace(ex : pexpr; var x, y : real); forward;
.PROCEDURE

{ ========================================================================== }
procedure ReturnExprSpace(ex : pexpr; var x, y : real);
   begin
   if ex <> nil then
      if ex↑.pgfig <> nil then
         begin
         x := ex↑.pgfig↑.Size.x;
         y := ex↑.pgfig↑.Size.y;
         end;
   end;

procedure PrePretty(ex:pexpr;PrettyInter:boolean);
   var b: real;
   procedure PrePrettyDo(var ex:pexpr;pri:integer;var base:real); forward;

{ -------------------------------------------------------------------------- }

procedure ZeroVect(var v:Vector);
   begin
   v.x:=0;
   v.y:=0;
   end;



{ -------------------------------------------------------------------------- }

function PrePrettyPar(p1,p2:integer):boolean;
   begin
   PrePrettyPar:=(p1 < p2)and not PrettyInter;
   end;

{ -------------------------------------------------------------------------- }
```

```
procedure VarPrePretty(ex:pexpr;var base:real);
   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      Size.x:=StringLength(varid) * PrettyCharWidth;
      Size.y:=PrettyCharHeight;
      end;
   base:=0.0;
   end;

{ ------------------------------------------------------------------------ }

procedure ErrorPrePretty(ex:pexpr;var base:real);
   var i  : integer;
       Sx : real;
       b  : boolean;
   begin
   with ex↑, pgfig↑ do
      begin
      if nextexpr <> nil then
         begin
         PrePrettyDo(nextexpr,100,base);
         ZeroVect(NextVect);
         ErrorVect.x:=nextexpr↑.pgfig↑.Size.x;
         ErrorVect.y:=base;
         Size.y:=MaxOfReals(NextExpr↑.pgfig↑.Size.y , PrettyCharHeight);
         Sx:=nextexpr↑.pgfig↑.Size.x;
         end
      else
         begin
         base:=0;
         ZeroVect(ErrorVect);
         ZeroVect(NextVect);
         Size.y:=PrettyCharHeight;
         Sx:=0;
         end;

      if ErrorNumber > 0 then
         begin
         i:=80;
         while (i > 1) and (s[i] = ' ') do
            i:=i-1;
         if ( i=1 ) and ( s[1]=' ') then
            i:=0;
         end
      else
         i:=0;

      Size.x:=Sx + i * PrettyCharWidth;
      Slength:=i;
      end; { with }
   end; { ErrorPrePretty }

{ ------------------------------------------------------------------------ }

procedure ChInfoPrePretty(ex:pexpr;pri:integer;var base : real);
   var SizeRight : Vector;
```



```
   begin
   if PrettyInter then
      begin
      with ex↑, pgfig↑ do
         begin
         PrePrettyDo(chexpr,pri,base);
         SizeRight:=chexpr↑.pgfig↑.Size;
         if before then      { character left of expression following }
            begin
            chvect.x:=0;
            chvect.y:=base;
            ChExprVect.x:=PrettyCharWidth;
            ChExprVect.y:=0;
            Size.x:=ChExprVect.x + SizeRight.x;
            Size.y:=SizeRight.y;
            end
         else                          { character right of the expression }
            begin
            ChExprVect.x:=0;
            ChExprVect.y:=0;
            chvect.x:=SizeRight.x;
            chvect.y:=base;
            Size.x:=chvect.x + PrettyCharWidth;
            Size.y:=SizeRight.y;
            end;
         end
      end
   else
      with ex↑, pgfig↑ do
       begin
       PrePrettyDo(chexpr,pri,base);
       Size:=chexpr↑.pgfig↑.Size;
       ZeroVect(chvect);
       ZeroVect(chexprVect);
       end;
   end; { ChInfo PrePretty }

{ ------------------------------------------------------------------------ }

procedure NumPrePretty(ex:pexpr;var base:real);

   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      Size.x:=StringLength(numid) * PrettyCharWidth;
      Size.y:=PrettyCharHeight;
      end;
   base:=0.0;
   end;

{ ------------------------------------------------------------------------ }

procedure FuncPrePretty(ex: pexpr; var base : real);
   var b, HigherY, LowerY, NextX : real;
       i                         : integer;
       Parameter                 : pexpr;
       ParAkt                    : pArgList;
   begin
```

```
   with ex↑, ex↑.pgfig↑ do
      begin

      if nargs = 0 then
         begin
         if PrettyInter then
            begin
            ZeroVect(FunctionVect);
            if LeftParanthes then
               Size.x:=(NrOfSpaces+StringLength(func)+1) * PrettyCharWidth
            else
               Size.x:= (NrOfSpaces+StringLength(func))  * PrettyCharWidth;
            Size.y:=PrettyCharHeight;
            par.exist:=false;
            end
         else { not Prettyinter }
            begin
            with par do
               begin
               exist:=true;
               LeftPar.x:=StringLength(func) * PrettyCharWidth;
               LeftPar.y:=0.0;
               Rightpar.x:=(StringLength(func)+1) * PrettyCharWidth;
               Rightpar.y:=0.0;
               ParScale.x:=1.0;
               ParScale.y:=1.0;
               end;
            Size.x:=(StringLength(func)+2)*PrettyCharWidth;
            Size.y:=PrettyCharHeight;
            end;
         base:=0.0;
         end
      else { nargs > 0 }
         begin
         if PrettyInter then
            NextX:=(NrOfSpaces+StringLength(func)+1) * PrettyCharWidth
         else
            NextX:=(StringLength(func)+1) * PrettyCharWidth;
                  { +1 gives space for (-char }
         if PrettyInter then
            par.exist:=false
         else
            begin
            with par do
               begin
               exist:=true;
               LeftPar.x:=StringLength(func) * PrettyCharWidth;
               ParScale.x:=1;
               end;
            end;

         HigherY:=0.0;
         LowerY :=0.0;
         ParAkt :=ArgumentList; { First parameter to the function }
         for i :=1 to nargs do
            begin
            Parameter:=ParAkt↑.ex;
            PrePrettyDo(Parameter,100,b);
```



```
            with Parameter↑, Parameter↑.pgfig↑ do
               begin
               LowerY :=MaxOfReals(b,LowerY);
               HigherY:=MaxOfReals(Size.y-b,HigherY);
               ParameterPlace.x:=NextX;
               ParameterPlace.y:=b;
               if ParAkt↑.Comma then
                  NextX:=NextX + Size.x + PrettyCharWidth
                     { space for ,-char }
               else
                  NextX:=NextX + Size.x;
               end;
            ParAkt:=ParAkt↑.next;
            end; { for i:= }

         if PrettyInter then
            if RightParanthes then
               Size.x:=NextX + PrettyCharWidth
            else
               Size.x:=NextX
         else
            begin
            Size.x:=NextX + PrettyCharWidth;
            par.RightPar.x:=NextX;
            par.ParScale.x:=1.0;
            par.ParScale.y:=(HigherY+LowerY) /
                                 PrettyCharHeight;
            end;
         base:=LowerY;
         Size.y:=LowerY + HigherY;
         FunctionVect.x:=0.0;
         FunctionVect.y:=LowerY;
         ParAkt:=Argumentlist;
         for i:=1 to nargs do
            begin
            Parameter:=ParAkt↑.ex;
            Parameter↑.pGfig↑.ParameterPlace.y:=
                  LowerY - Parameter↑.pGfig↑.ParameterPlace.y;
            ParAkt:=ParAkt↑.next;
            end;
         end; { if nargs = 0 }
      end; { with }
   end;

{ ------------------------------------------------------------------------ }

procedure NotPrePretty(ex:pexpr;pri:integer;var base:real);
   var spa : integer;
   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      if PrettyInter or (NotLength <> 3) then
         spa:=NotLength
      else
         spa:=4;
      PrePrettyDo(notexpr,5,base);
      negationvect.y:=base;
      negvect.y:=0;
```

```
      Size.y:=notexpr↑.pgfig↑.Size.y;
      if PrePrettyPar(pri,5) then
         with par do                { approx the same code as in }
            begin                   { MinusPrePrettyDo            }
            exist:=true;
            ZeroVect(leftPar);
            ParScale.x:=1;
            ParScale.y:=Size.y/PrettyCharHeight; { height of parantheses }
            negationvect.x:=PrettyCharWidth * ParScale.x;
            negvect.x:=negationvect.x+PrettyCharWidth * spa;
                                                 { space for 'not' }
            rightpar.y:=0;
            rightpar.x:=negvect.x+notexpr↑.pgfig↑.Size.x;
            Size.x:=rightpar.x + PrettyCharWidth * ParScale.x;
            end
      else
         begin
         Par.exist:=false;
         negationvect.x:=0;
         negvect.x:= PrettyCharWidth * spa; { space for 'not' }
         Size.x:=negvect.x + notexpr↑.pgfig↑.Size.x;
         end;
      end { with };
   end    { NotPrePretty };

{ ------------------------------------------------------------------- }

procedure IfthenelsePrePretty(ex:pexpr;pri:integer;var base:real);

var   Sizeif,Sizethen,Sizeelse : Vector;
      baseif,basethen,baseelse : real;
      spa                      : real;
   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      PrePrettyDo(  ifexpr,pri,baseif  );
      PrePrettyDo(thenexpr,pri,basethen);
      PrePrettyDo(elseexpr,pri,baseelse);

      Sizeif  :=  ifexpr↑.pgfig↑.Size;
      Sizethen:=thenexpr↑.pgfig↑.Size;
      Sizeelse:=elseexpr↑.pgfig↑.Size;

      if PrettyInter then
         spa:=2                 { when all character is typed }
      else
         spa:=3;                { normal space for "if" }
      ifVect.x:=0;                                          { if }
      ifVect.y:=MaxOfReals(baseif,MaxOfReals(basethen,baseelse));
      ifexVect.x:=spa * PrettyCharWidth;
      ifexVect.y:=ifVect.y-baseif;

      if PrettyInter then
         spa:=MinOfInts(4,IfLength-2)  { when all character is typed }
      else
         spa:=MinOfInts(5,IfLength-2); { normal space for "then"&"else" }

      if PrettyInter then
```



```
         thenVect.x:=ifexVect.x+Sizeif.x                        { then }
      else
         thenVect.x:=ifexVect.x+Sizeif.x+1.0*PrettyCharWidth;   { then }
      thenVect.y:=ifVect.y;
      thenexVect.x:=thenVect.x + spa * PrettyCharWidth;
      thenexVect.y:=ifVect.y-basethen;

      if PrettyInter then
         spa:=MinOfInts(4,IfLength-6)    { when all character is typed }
      else
         spa:=MinOfInts(5,IfLength-6);
                                  { normal space for "then" & "else" }

      if PrettyInter then
         elseVect.x:=thenexVect.x+Sizethen.x                    { else }
      else
         elseVect.x:=thenexVect.x+Sizethen.x+1.0*PrettyCharWidth; { else }
      elseVect.y:=ifVect.y;
      elseexVect.x:=elseVect.x+spa * PrettyCharWidth;
      elseexVect.y:=ifVect.y-baseelse;


      Size.x:=elseexVect.x+Sizeelse.x;
      Size.y:=MaxOfReals(ifexVect.y+Sizeif.y,
            MaxOfReals(thenexVect.y+Sizethen.y,elseexVect.y+Sizeelse.y));
      base:=ifVect.y;

      if PrePrettyPar(pri,14) then
         with Par do
         begin
         exist:=true;
         ZeroVect(LeftPar);
         ParScale.x:=1;
         ParScale.y:=Size.y/PrettyCharHeight;

         ifVect.x    :=ifVect.x     + ParScale.y * PrettyCharWidth;
         ifexVect.x  :=ifexVect.x   + ParScale.y * PrettyCharWidth;

         thenVect.x  :=thenVect.x   + ParScale.y * PrettyCharWidth;
         thenexVect.x:=thenexVect.x + ParScale.y * PrettyCharWidth;

         elseVect.x  :=elseVect.x   + ParScale.y * PrettyCharWidth;
         elseexVect.x:=elseexVect.x + ParScale.y * PrettyCharWidth;

         Size.x:=Size.x + 2 * ParScale.y * PrettyCharWidth;
         end
      else
         Par.exist:=false;
      end { with };
   end ;

{ ------------------------------------------------------------------- }

procedure MinusPrePretty(ex:pexpr;pri:integer;var base:real);

var SizeRight : Vector;
   begin
   with ex↑, ex↑.pgfig↑ do
```

```
      begin
      PrePrettyDo(minusexpr,5,base);
      SizeRight:=minusexpr↑.pgfig↑.Size;
      negationvect.y:=base;
      negvect.y:=0;
      Size.y:=SizeRight.y;
      if PrePrettyPar(pri,6) then
         with par do
            begin
            exist:=true;
            ZeroVect(leftPar);
            ParScale.x:=1;
            ParScale.y:=Size.y/PrettyCharHeight;        { height of par }
            negationvect.x:=PrettyCharWidth * ParScale.x;
            negvect.x:=negationvect.x+PrettyCharWidth;
            rightpar.y:=0;
            rightpar.x:=negvect.x+SizeRight.x;
            Size.x:=rightpar.x+PrettyCharWidth*ParScale.x;
            end
      else
         begin
         Par.exist:=false;
         negationvect.x:=0;
         negvect.x:= PrettyCharWidth; { space for minus sign }
         Size.x:=negvect.x+SizeRight.x;
         end;
      end { with };
   end    { MinusPrePretty };

{ ------------------------------------------------------------------------ }

procedure BinPrePretty(ex:pexpr;pri,prior1,prior2:integer;
                                        var base:real);
var  SizeLeft,SizeRight  : Vector;
     bL,bR :real;
     squarerot: boolean;

     { BinPrePretty } function spacebefore(b:tbintype):real;
        begin
        if not PrettyInter and ((b = binandop) or (b = binorop))then
           spacebefore := 1
        else
           spacebefore:=0;
        end;

     { BinPrePretty } function oplength(b:tbintype):real;
        begin
        if b=binandop then
           if PrettyInter then
              oplength:=3
           else
              oplength:=4
        else if b=binorop then
           if PrettyInter then
              oplength:=2
           else
              oplength :=3
        else
```



```
           oplength:=1;
        end;

   begin
   with ex↑, ex↑.pgfig↑ do
   case bintype of
      binandop,binorop,bingreaterop,binlessop,
      binaddop, binsubop, binmultop,
      binequalop :     begin
                       PrePrettyDo(expr1,prior1,bL);
                       PrePrettyDo(expr2,prior2,bR);
                       SizeLeft :=expr1↑.pgfig↑.Size;
                       SizeRight:=expr2↑.pgfig↑.Size;

                       if bL > bR then
                       { Center around operator}
                          begin
                          left.y:=0;
                          op.y:=bL;
                          right.y:=op.y-bR;
                          end
                       else
                          begin
                          op.y:=bR;
                          left.y:=op.y-bL;
                          right.y:=0;
                          end;
                       left.x:=0;
                       op.x:=SizeLeft.x+
                          spacebefore(bintype)*PrettyCharWidth;
                       right.x:=op.x   +
                          oplength(bintype)*PrettyCharWidth;
                       Size.y:=MaxOfReals( left.y+ SizeLeft.y,
                                           right.y+SizeRight.y);
                       Size.x:=right.x+SizeRight.x;
                       base:=op.y;
                       if PrePrettyPar(pri,prior1) then
                          with par do
                             begin
                             exist:=true;
                             ZeroVect(leftPar);
                             ParScale.x:=1;
                             ParScale.y:=Size.y/PrettyCharHeight;
                                                {height of par}
                             left.x:=left.x+PrettyCharWidth;
                             op.x:=op.x+PrettyCharWidth;
                             right.x:=right.x+PrettyCharWidth;
                             rightPar.x:=right.x+SizeRight.x;
                             rightPar.y:=0;
                             Size.x:=Size.x + 2 * PrettyCharWidth;
                          end
                       else
                          par.exist:=false;
                       end;

      bindivop:           begin
                          PrePrettyDo(expr1,prior1,bL);
                          PrePrettyDo(expr2,prior2,bR);
```

```
                        SizeLeft :=expr1↑.pgfig↑.Size;
                        SizeRight:=expr2↑.pgfig↑.Size;
                        par.exist:=false;
                        op.x:=0;
                        if SizeLeft.x ) SizeRight.x then
                           begin
                           right.x:=(SizeLeft.x-SizeRight.x)/2;
                           left.x:=0;
                           divline.x:=SizeLeft.x;
                           end
                        else
                           begin
                           right.x:=0;
                           left.x:=(SizeRight.x-SizeLeft.x)/2;
                           DivLine.x:=SizeRight.x;
                           end;
                        left.y:=SizeRight.y+PrettyCharHeight*0.7;
                        op.y:=SizeRight.y+PrettyCharHeight*0.35;
                        divline.y:=op.y;
                        right.y:=0;
                        Size.x:=MaxOfReals(SizeLeft.x,SizeRight.x);
                        Size.y:=left.y+SizeLeft.y;
                        base:=SizeRight.y; {+0.5*PrettyCharHeight; }
                        end;

     binpowerop:        begin
                        squarerot:=(expr2↑.exprtype = exprnumber) and
                                   (expr2↑.val = 0.5) and not PrettyInter;
                        if not squarerot then
                           begin
                           PrePrettyDo(expr1,prior1,bL);
                           PrePrettyDo(expr2,prior2,bR);
                           SizeLeft :=expr1↑.pgfig↑.Size;
                           SizeRight:=expr2↑.pgfig↑.Size;
                           end
                        else
                           begin
                           PrePrettyDo(expr1,100,bL);
                           SizeLeft :=expr1↑.pgfig↑.Size;
                           end;

                        Par.exist:=false;
                        if not squarerot then
                              { Adjust for scalediff }
                           begin
                           SizeRight.x:=SizeRight.x*PrettyPowerScale;
                           SizeRight.y:=SizeRight.y*PrettyPowerScale;
                           bR:=bR*PrettyPowerScale;
                           ZeroVect(left);
                           right.x:=SizeLeft.x;
                           right.y:=SizeLeft.y-PrettyCharHeight*0.5;
                           op:=right;
                           Size.x:=SizeLeft.x+SizeRight.x;
                           Size.y:=MaxOfReals(SizeLeft.y,
                                              right.y+SizeRight.y);
                           base:=bL;
                           end
{ Special for squarerotchar}
```



```
                        else { if squarerot }
                           begin
                           ZeroVect(left);
                           Left.x:=PrettyCharWidth;
                           ZeroVect(right);
                           ZeroVect(op);
                           Size.x:=Left.x + SizeLeft.x
                                   + PrettyCharWidth/2;
                           Size.y:=SizeLeft.y;
                           base:=bl;
                           SqrtV.x:=Left.x;
                           SqrtV.y:=SizeLeft.y;
                                   { size of  ------\ in Squarerotchar }
                           SqrtOver.x:=SizeLeft.x;
                           SqrtOver.y:=PrettyCharHeight/4;
                           end;
                        end { binpowerop };

        end {case};
   end {BinPrePretty};

{ ------------------------------------------------------------------------ }

procedure PrePrettyDo(var ex:pexpr;pri:integer;var base:real);
   var i : integer;
   begin
   if ex <> nil then
      begin
      if ex↑.pgfig = nil then new(ex↑.pgfig);
      case ex↑.exprtype of
         exprvariable:     VarPrePretty(ex,base);
         exprnumber:       NumPrePretty(ex,base);
         exprfunction:     FuncPrePretty(ex,base);
         exprminus:        MinusPrePretty(ex,pri,base);
         exprnot:          NotPrePretty  (ex,pri,base);
         exprifthenelse:   IfthenelsePrePretty(ex,pri,base);
         exprbinary:
            case ex↑.bintype of
               binaddop:      BinPrePretty(ex,pri, 8, 8,base);
               binsubop:      BinPrePretty(ex,pri, 8, 7,base);
               binmultop:     BinPrePretty(ex,pri, 4, 4,base);
               bindivop:      BinPrePretty(ex, 18,15,15,base);
               binpowerop:    BinPrePretty(ex,pri, 2, 1,base);
               binequalop:    BinPrePretty(ex, 18,16,16,base);
               binandop:      BinPrePretty(ex,pri,11,11,base);
               binorop:       BinPrePretty(ex,pri,12,12,base);
               bingreaterop:  BinPrePretty(ex,pri,10,10,base);
               binlessop:     BinPrePretty(ex,pri,10,10,base);
               end; { case ex↑.bintype }

         exprError:        ErrorPrePretty(ex,base);
         exprchinfo:       ChInfoPrePretty(ex,pri,base);
         end; { case ex↑.exprtype }
      end
   else
      begin
      ex:=MakeExpr(exprerror);
      for i:=1 to 80 do
```

```
        ex↑.s[i]:=' ';
      ex↑.ErrorNumber:=0;
      ex↑.nextexpr:=nil;
      ex↑.cursor.iscursor:=false;
      prePrettydo(ex,pri,base);
      end;

   end {PrePrettyDo};

{ ------------------------------------------------------------------------ }

   begin      { PrePretty }
   PrePrettyDo(ex,100,b);
   end;



{ ================ Pretty PART START ====================== }

procedure Pretty(ex:pexpr;PrettyInter:boolean);
   const CharWidth  = PrettyCharWidth / 1.3;
                      { Physical size for characters }
         CharHeight = PrettyCharHeight;
                      { Used by CharSize-procedure   }
   var T : TransfDescr;
   procedure PrettyDo(ex:pexpr); forward;

function CS(c:texprcursor):boolean;
   begin
   CS:=c.iscursor and PrettyInter;    { The cursor is displayed only when }
                                      { the module is in interactive mode }
   end;

{ ------------------------------------------------------------------------ }

procedure CursorDraw(x,y,l:real;typ:integer);
   { Draws a cursor at specified position, with 'length' l and type typ }
   { typ = 0 is a normal textcursor, a line horizontal line             }
   {     = 1 is an arrow pointing up and left                           }
   {     = 2 is a horizontal arrow pointing left.                       }
   var t:TransfDescr;
       h:real;
   begin
   case typ of
   0:   begin    { Line under current character }
{       SaveTransform(t);       { Just save Color }
        SetColor(CursorColor);
        MoveTo(x,y-PrettyCharWidth*0.1);
        LineTo(x+PrettyCharWidth,y-PrettyCharWidth*0.1);
{       SetTransform(t);   }
        SetColor(ExColor);
        end;

   1:   begin    { Arrow pointing upp-left }
{       SaveTransform(t);   { Just save Color }
        SetColor(CursorColor);
        h:=PrettyCharWidth * 1;
```



```
        MoveTo(x,y);
        LineTo(x+h*3,y-h*3);
        MoveTo(x,y-h);
        LineTo(x,y);
        LineTo(x+h,y);
{       SetTransform(t);    }
        SetColor(ExColor);
        end;
   2:   begin
        SetColor(CursorColor);
        h:=PrettyCharWidth * 1;
        MoveTo(x,y);
        LineTo(x+h*3,y);
        MoveTo(x+h,y+h);
        LineTo(x,y);
        LineTo(x+h,y-h);
        SetColor(ExColor);
        end;
      end { case };
   end;

{ ------------------------------------------------------------------------ }

procedure ChangeScale(f:real);
   begin
   Scale(f,f);
   CharSize(CharWidth,CharHeight); { InitDrawChar }
   end;

{ ------------------------------------------------------------------------ }

procedure TranslateVector(v:Vector);
   begin
   Translate(v.x,v.y);
   end;

{ ------------------------------------------------------------------------ }

procedure DrawParanthes(p:Paranthes);
var t:TransfDescr;
   begin
   with p do
      begin
      SaveTransform(t);
      TranslateVector(LeftPar);
      Scale(ParScale.x,ParScale.y);
      CharSize(CharWidth,CharHeight); { InitDrawChar }
      SetCursor(0,0);
      DrawChar('(');
      SetTransform(t);

      TranslateVector(RightPar);
      Scale(ParScale.x,ParScale.y);
      CharSize(CharWidth,CharHeight); { InitDrawChar }
      SetCursor(0,0);
      DrawChar(')');
      SetTransform(t);
      CharSize(CharWidth,CharHeight); { InitDrawChar }
```

```
      end;
   end { DrawParanthes };

{ ------------------------------------------------------------------------ }


   procedure PrettyOp(t:text4;op:vector);
   var i :integer;
       begin
       for i:= 1 to 4 do
          if t[i] <> ' ' then
             begin
             SetCursor(op.x+(i-1)*PrettyCharWidth,op.y);
             DrawChar(t[i]);
             end;
       end;

{ ------------------------------------------------------------------------ }

procedure VarPretty(ex:pexpr);
var i :integer;
    t :TransfDescr;
    s : text4;
   begin
   for i:=1 to StringLength(ex^.varid) do
      begin
      SetCursor((i-1)*PrettyCharWidth,0);
      DrawChar(ex^.varid[i]);
      end;

   if CS(ex^.cursor) then
      CursorDraw((ex^.cursor.pos-1)*PrettyCharWidth,0,1,0);

   end;

{ ------------------------------------------------------------------------ }

procedure NumPretty(ex:pexpr);
var i :integer;
    t :TransfDescr;
   begin
   for i:=1 to StringLength(ex^.numid) do
      begin
      SetCursor((i-1)*PrettyCharWidth,0);
      DrawChar(ex^.numid[i]);
      end;

   if CS(ex^.cursor) then
      CursorDraw((ex^.cursor.pos-1)*PrettyCharWidth,0,1,0);
   end;

{ ------------------------------------------------------------------------ }

procedure FuncPretty(ex:pexpr);
   var i         : integer;
       t         : TransfDescr;
       ParAkt    : parglist;
       Parameter : pexpr;
```



```
   begin
   with ex^, ex^.pgfig^ do
      begin
      for i:=1 to StringLength(func) do
         begin
         SetCursor((i-1) * PrettyCharWidth + FunctionVect.x,
                   FunctionVect.y);
         DrawChar(func[i]);
         end;
      if par.exist then
         DrawParanthes(par);
      ParAkt:=ArgumentList;
      for i:=1 to nargs do
         begin
         Parameter:=ParAkt^.ex;
         if (i <> 1) and not PrettyInter then
            begin
            SetCursor(Parameter^.pgfig^.ParameterPlace.x - PrettyCharWidth,
                      FunctionVect.y);
            DrawChar(',');
            end;
         if PrettyInter and ParAkt^.Comma then
            with Parameter^.pgfig^ do
               begin
               SetCursor(ParameterPlace.x + Size.x,
                         ex^.pgfig^.FunctionVect.y);
               DrawChar(',');
               if ParAkt^.CommaCursor then
                  CursorDraw(ParameterPlace.x + Size.x,
                                        ex^.pgfig^.FunctionVect.y,1,0);
               end;

            SaveTransform(T);
            TranslateVector(Parameter^.pgfig^.ParameterPlace);
            PrettyDo(Parameter);
            SetTransform(T);
            ParAkt:=ParAkt^.next;
            end;

               { Cursor under function-identifier ? }
         if CS(cursor) then
            CursorDraw((cursor.pos-1)*PrettyCharWidth+FunctionVect.x,
                       FunctionVect.y,1,0);
               { Cursor under spaces following function-identifier ? }

         if PrettyInter then
            begin
            if (SpaceCursor <> 0) then
               CursorDraw((SpaceCursor-1+StringLength(func))*PrettyCharWidth+
                          FunctionVect.x,FunctionVect.y,1,0);

            if LeftParanthes then
               begin
               SetCursor((NrOfSpaces + StringLength(func)) * PrettyCharWidth +
                         FunctionVect.x,FunctionVect.y);
               DrawChar('(');
               end;
            if RightParanthes then
```

```
            begin
            SetCursor(Size.x - PrettyCharWidth + FunctionVect.x,
                      FunctionVect.y);
            DrawChar(')');
            end;

            { Cursor under some of the parantheses }
         if ParanthesCursor = 1 then              { Left paranthes }
            CursorDraw((NrOfSpaces + StringLength(func)) *
                       PrettyCharWidth +
                       FunctionVect.x,FunctionVect.y,1,0)
         else if ParanthesCursor = 2 then         { Right paranthes }
            CursorDraw(Size.x - PrettyCharWidth +
                      FunctionVect.x,FunctionVect.y,1,0);
         end; { if PrettyInter }

      end; { with }
   end;

{ ------------------------------------------------------------------------ }

procedure MinusPretty(ex:pexpr);
   begin
   with ex^, ex^.pgfig^ do
      begin
      if par.exist then
         DrawParanthes(par);
      SetCursor(NegationVect.x,NegationVect.y);
      DrawChar('-');

      if CS(cursor) then
         CursorDraw((cursor.pos - 1)*PrettyCharWidth+NegationVect.x,
                                                  NegationVect.y,1,0);
      TranslateVector(negVect);
      PrettyDo(minusexpr);
      end;
   end;

{ ------------------------------------------------------------------------ }

procedure ErrorPretty(ex:pexpr);
var  t :TransfDescr;
     i : integer;
     p : real;
   begin
   with ex^, pgfig^ do
      begin
      if NextExpr () nil then
         begin
         SaveTransform(t);
         TranslateVector(NextVect);
         PrettyDo(NextExpr);
         SetTransform(t);
         CharSize(CharWidth,CharHeight); { InitDrawChar }
         end;
      TranslateVector(ErrorVect);
      SetColor(ERRORColor);
```



```
      for i:=1 to Slength do
         begin
         SetCursor((i-1)*PrettyCharWidth,0);
         DrawChar(ex^.s[i]);
         end;

      if CS(ex^.cursor) then
         CursorDraw((ex^.cursor.pos-1)*PrettyCharWidth,0,1,0);
      if ((ErrorNumber=-2)or(ErrorNumber=0)) and not PrettyInter then
         begin
         p:=PrettyCharWidth/4;
         moveto(0,0);
         LineTo(p,0);
         LineTo(p,p);
         LineTo(0,p);
         LineTo(0,0);
         end;
      end; { with }
   SetColor(ExColor);
   end; { ErrorPretty }

{ ------------------------------------------------------------------------ }

procedure chInfoPretty(ex:pexpr);
   begin
   if PrettyInter then
      begin
      with ex^ , pgfig^ do
         begin
         SetCursor(chvect.x,chvect.y);
         DrawChar(ch);
         if CS(cursor) then
            CursorDraw(chvect.x+(cursor.pos-1)*PrettyCharWidth,
                       chvect.y,1,0);
         TranslateVector(chexprvect);
         PrettyDo(chexpr);
         end;
      end
   else
      PrettyDo(ex^.chexpr);
   end;

{ ------------------------------------------------------------------------ }

procedure NotPretty(ex:pexpr);
   begin
   with ex^, ex^.pgfig^ do
      begin
      if par.exist then DrawParanthes(par);
      case NotLength of
         1 : PrettyOp('n   ',NegationVect);
         2 : PrettyOp('no  ',NegationVect);
         3 : PrettyOp('not ',NegationVect);
         end;
      if CS(cursor) then
         CursorDraw((cursor.pos-1)*PrettyCharWidth+NegationVect.x,
                                                  NegationVect.y,1,0);
      TranslateVector(negVect);
```

```
      PrettyDo(notexpr);
      end;
   end;

procedure IfthenelsePretty(ex:pexpr);      { IfthenelsePretty }
var  t        : TransfDescr;
     NullVect : Vector;
   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      NullVect.x:=0.0;
      NullVect.y:=0.0;
      if Par.exist then DrawParanthes(par);
      SaveTransform(t);
      TranslateVector(ifvect);
      PrettyOp('if  ', NullVect);        { if }
      if CS(Cursor) and (Cursor.pos (= 2) then
         CursorDraw((Cursor.pos-1) * PrettyCharWidth, 0, 1, 0);
      SetTransform(t);
      TranslateVector(ifexVect);
      PrettyDo(ifexpr);

      SetTransform(t);                   { then }
      CharSize(CharWidth,CharHeight); { InitDrawChar }
      TranslateVector(thenvect);
      if IfLength )=6 then
         PrettyOp('then',NullVect)
      else if IfLength )=3 then
         case IfLength of
            3 : PrettyOp('t   ',NullVect);
            4 : PrettyOp('th  ',NullVect);
            5 : PrettyOp('the ',NullVect);
            end;
      if CS(Cursor) and (Cursor.pos in [3..6]) then
         CursorDraw((Cursor.pos-3) * PrettyCharWidth, 0, 1, 0);

      SetTransform(t);
      TranslateVector(thenexVect);
      PrettyDo(thenexpr);

      SetTransform(t);                   { else }
      CharSize(CharWidth,CharHeight); { InitDrawChar }
      TranslateVector(elsevect);
      if IfLength )=10 then
         PrettyOp('else',NullVect)
      else if IfLength )=7 then
         case IfLength of
            7 : PrettyOp('e   ',Nullvect);
            8 : PrettyOp('el  ',Nullvect);
            9 : PrettyOp('els ',Nullvect);
            end;
      if CS(Cursor) and (Cursor.pos )=7) then
         CursorDraw((Cursor.pos-7) * PrettyCharWidth, 0, 1, 0);

      SetTransform(t);
      TranslateVector(elseexVect);
      PrettyDo(elseexpr);
      end {with };
```



```
   end;

procedure BinPretty(ex:pexpr);                              { BinPretty }
var t:TransfDescr;
    squarerot:boolean;


   begin
   with ex↑, ex↑.pgfig↑ do
      begin
      if par.exist then
         DrawParanthes(par);
      SaveTransform(t);
      TranslateVector(left);
      PrettyDo(expr1);
      SetTransform(t);
      CharSize(CharWidth,CharHeight); { InitDrawChar }
      case bintype of
         binaddop:      begin
                        PrettyOp('+   ',op);
                        if CS(cursor) then
                           CursorDraw(op.x,op.y,1,0);
                        end;
         binsubop:      begin
                        PrettyOp('-   ',op);
                        if CS(cursor) then
                           CursorDraw(op.x,op.y,1,0);
                        end;
         binmultop:     begin
                        PrettyOp('*   ',op);
                        if CS(cursor) then
                           CursorDraw(op.x,op.y,1,0);
                        end;

         bindivop:      begin
                        MoveTo(op.x,op.y);
                        LineTo(DivLine.x,DivLine.y);
                        if CS(cursor) then
                           CursorDraw(DivLine.x,DivLine.y,0.4,1);
                        end;
         binpowerop:    squarerot:= (expr2↑.exprtype = exprnumber) and
                                    (expr2↑.val = 0.5) and not PrettyInter;
         binequalop:    begin
                        PrettyOp('=   ',op);
                        if CS(cursor) then
                           CursorDraw(op.x,op.y,1,0);
                        end;
         binandop:      begin
                        PrettyOp('and ',op);
                        if CS(cursor) then
                           CursorDraw
                              (op.x+(cursor.pos-1)*PrettyCharWidth,
                               op.y,1,0);
                        end;
         binorop:       begin
                        PrettyOp('or  ',op);
                        if CS(cursor) then
```

```
                        CursorDraw
                           (op.x+(cursor.pos-1)*PrettyCharWidth,
                            op.y,1,0);
                     end;
         bingreaterop: begin
                     PrettyOp(')   ',op);
                     if CS(cursor) then
                        CursorDraw(op.x,op.y,1,0);
                     end;
         binlessop:  begin
                     PrettyOp('(   ',op);
                     if CS(cursor) then
                        CursorDraw(op.x,op.y,1,0);
                     end;
      end;
      if (bintype = binpowerop) then
         if not squarerot then
            begin
            if CS(Cursor) then
               if Cursor.pos = 1 then
                  CursorDraw(op.x, op.y-PrettyCharHeight*0.2, 0.3, 2)
               else
                  CursorDraw(op.x, op.y-PrettyCharHeight*0.1, 0.3, 1);
            TranslateVector(right);
            ChangeScale(PrettyPowerScale);
            PrettyDo(expr2);
            ChangeScale(1/PrettyPowerScale);
            end
         else
{ Special for squarerotchar}
            with par do
               begin

               MoveTo(0,SqrtV.y*0.7);
               LineTo(SqrtV.x*0.5,-2);
               LineTo(SqrtV.x,SqrtV.y);
               LineTo(SqrtV.x+SqrtOver.x,SqrtV.y);
               LineTo(SqrtV.x+SqrtOver.x,SqrtV.y-SqrtOver.y);
               end
      else
         begin
         TranslateVector(right);
         PrettyDo(expr2);
         end;
      end { with };
   end { BinPretty };

procedure PrettyDo(*ex:pexpr*);
   begin
   if ex <> nil then
   case ex^.exprtype of
      exprvariable:   VarPretty(ex);
      exprfunction:   FuncPretty(ex);
      exprnumber:     NumPretty(ex);
      exprminus:      MinusPretty(ex);
      exprnot:        NotPretty(ex);
      exprifthenelse: IfthenelsePretty(ex);
      exprbinary:     BinPretty(ex);
```



```
      exprError:      ErrorPretty(ex);
      exprchinfo:     ChInfoPretty(ex);
      end;
   end;

{ ------------------------------------------------------------------------}

   begin       { Pretty }
   SaveTransform(T);
   SetColor(ExColor);
   CharSize(CharWidth,CharHeight); { InitDrawChar }
   PrettyDo(ex);
   SetTransform(T);
   CharSize(CharWidth,CharHeight); { InitDrawChar }
   end;

.END
```

```
use:[magnus.exarb.prettylupdown.pak

.END
Author Magnus Taube

Handles moving up/down in an expression

.FORWARD

procedure MoveUpOrDown(ex : pexpr; up : boolean);       forward;
.PROCEDURE
procedure HandleUpDown(var ready, csfound : boolean;up : boolean;
                       hex : pexpr; var divnode : pexpr; var x : real);
   var opx : real;
   begin
   if hex () nil then
      with hex↑ do
         begin
         case exprtype of
            exprvariable:
               begin
               csfound:=cursor.iscursor;
               if csfound then
                  x:=pgfig↑.Size.x/StringLength(varid) * (cursor.pos - 1);
               end;
            exprnumber:
               begin
               csfound:=cursor.iscursor;
               if csfound then
                  x:=pgfig↑.Size.x/StringLength(numid) * (cursor.pos - 1);
               end;
            exprchinfo:
               begin
               csfound:=cursor.iscursor;
               if csfound then
                  begin
                  if before then
                     x:=0
                  else
                     x:=pgfig↑.chvect.x;
                  end
               else
                  begin
                  HandleUpDown(ready, csfound, up, chexpr, divnode, x);
                  if csfound and not ready then
                     if before then
                        x:=pgfig↑.chexprvect.x + x;
                  end;
               end;
            exprnot,
            exprminus:
               begin
               csfound:=cursor.iscursor;
               if csfound then
                  begin
                  if exprtype = exprnot then
```



```
                     x:=(pgfig↑.negationvect.x - pgfig↑.negvect.x) / 3  *
                        (cursor.pos - 1)
                  else
                     x:=pgfig↑.negationvect.x;
                  end
               else
                  begin
                  if exprtype = exprnot then
                     HandleUpDown(ready, csfound, up, minusexpr,
                                  divnode, x)
                  else
                     HandleUpDown(ready, csfound, up, notexpr, divnode, x);
                  if csfound and not ready then
                     x:=pgfig↑.negationvect.x + x;
                  end;
               end;
            exprbinary:
               begin
               if (bintype () bindivop) then
                  begin
                  csfound:=cursor.iscursor;
                  if csfound then
                     begin
                     opx:=pgfig↑.Right.x - pgfig↑.op.x;
                     if bintype = binorop then
                        x:=pgfig↑.op.x + opx/2 * (cursor.pos -1)
                     else if bintype = binandop then
                        x:=pgfig↑.op.x + opx/3 * (cursor.pos -1)
                     else
                        x:=pgfig↑.op.x;
                     end
                  else { No cursor at the operator, check sons }
                     begin
                     HandleUpDown(ready, csfound, up, expr1, divnode, x);
                     if csfound and not ready then
                        x:=pgfig↑.Left.x + x
                     else if not ready then
                        begin
                        HandleUpDown(ready, csfound, up, expr2,
                                     divnode, x);
                        if csfound and not ready then
                           if bintype = binpowerop then
                              x:=pgfig↑.Right.x + PrettyPowerScale * x
                           else
                              x:=pgfig↑.Right.x + x;
                        end;
                     end
                  else { bintype is bindivop }
                     begin
                     if up then
                        begin
                        HandleUpDown(ready, csfound, up, expr2, divnode,  x);
                        if csfound and not ready then
                           begin
                           ready:=true;
                           x:=pgfig↑.Right.x + x;
                           divnode:=hex;
```

```
                end
             else
                begin
                HandleUpDown(ready, csfound, up, expr1,
                             divnode, x);
                if csfound and not ready then
                   x:=pgfig↑.Left.x + x;
                end;
             end
          else { not up ==) down }
             begin
             HandleUpDown(ready, csfound, up, expr1, divnode, x);
             if csfound and not ready then
                begin
                ready:=true;
                x:=pgfig↑.Left.x + x;
                divnode:=hex;
                end
             else
                begin
                HandleUpDown(ready, csfound, up, expr2,
                             divnode, x);
                if csfound and not ready then
                   x:=pgfig↑.Right.x + x;
                end;
             end;
          end;
       end; { exprbinary }
    exprfunction: ;
    exprifthenelse:
       with pgfig↑ do
          begin
          csfound:=cursor.iscursor;
          if csfound then
             begin
             if cursor.pos (= 2 then       { cs under if   }
                x:=ifvect.x + (ifexvect.x - ifvect.x) / 2 *
                   (cursor.pos - 1)
             else if cursor.pos (= 6 then  { cs under then }
                x:=thenvect.x + (thenexvect.x - thenvect.x) / 4 *
                   (cursor.pos - 3)
             else                          { cs under else }
                x:=elsevect.x + (elseexvect.x - elsevect.x) / 4 *
                   (cursor.pos - 7);
             end
          else { not csfound }
             begin
             HandleUpDown(ready, csfound, up, ifexpr, divnode, x);
             if csfound and not ready then
                x:=x + ifexvect.x
             else
                begin
                HandleUpDown(ready, csfound, up, thenexpr,
                             divnode, x);
                if csfound and not ready then
                   x:=x + thenexvect.x
                else
                   begin
```



```
                   HandleUpDown(ready,csfound,up,elseexpr,
                                divnode, x);
                   if csfound and not ready then
                      x:=x + elseexvect.x;
                      end;
                end;
             end; { if csfound .. }
          end; { ifthenelse }

       exprerror:
          begin
          csfound:=cursor.iscursor;
          if csfound then
             begin
             if pgfig↑.SLength = 0 then
                x:=0
             else
                x:=pgfig↑.Size.x / pgfig↑.SLength * (cursor.pos - 1);
             end
          else
             begin
             HandleUpDown(ready, csfound, up, nextexpr, divnode, x);
             if csfound and not ready then
                x:=pgfig↑.nextvect.x + x;
             end;
          end;
       end; { case }
    end; { if and with }
 end; { HandleUpDown }

procedure MoveUpOrDown(ex : pexpr; up : boolean);

var ready, csfound       : boolean;
    divnode, nod         : pexpr;
    x                    : real;
    pos                  : integer;



procedure GetActualPos(var ready : boolean; up : boolean; ex : pexpr;
                       x : real);
   var nod : pexpr;

   { GetActualPos } function FixaPos(totlength : real; chars: integer)
                                     : integer;
      var res : integer;
      begin
      res:=round(x / totlength * chars );
      res:=maxofints(0,res);
      res:=MinOfInts(chars - 1,res) + ex↑.Cursor.StartPos;
      FixaPos:=res;
      end; { FixaPos }

   begin
   if ex <> nil then
      with ex↑, pgfig↑ do
         begin
         case exprtype of
```

```
        exprvariable:
           begin
           if x (= Size.x then
              begin
              ready:=true;
              pos:=FixaPos(ex↑.pgfig↑.Size.x, StringLength(varid));
              end;
           end;
        exprnumber:
           if x (= Size.x then
              begin
              ready:=true;
              pos:=FixaPos(ex↑.pgfig↑.Size.x, StringLength(numid));
              end;
        exprchinfo:
              begin
              if before then
                 begin
                 if x (= chexprvect.x then
                    begin
                    ready:=true;
                    pos:=cursor.StartPos;
                    end
                 else if (x )= chexprvect.x) and (x (= Size.x) then
                    GetActualPos(ready, up, chexpr, x - chexprvect.x)
                 end
              else { not before }
                 begin
                 if (x )= chvect.x) and (x (= Size.x) then
                    begin
                    ready:=true;
                    pos:=cursor.StartPos;
                    end
                 else if (x (= chvect.x) then
                    GetActualPos(ready, up, chexpr, x);
                 end
              end;
        exprnot:
           if ( x )= negvect.x ) and ( x (= negationvect.x) then
              begin
              ready:=true;
              pos:=FixaPos(ex↑.pgfig↑.Size.x, 3);
              end
           else if ( x )= negationvect.x ) and ( x (= Size.x ) then
              GetActualPos(ready, up, notexpr, negationvect.x - x);
        exprminus:
           if ( x )= negvect.x ) and ( x (= negationvect.x) then
              begin
              ready:=true;
              pos:=cursor.StartPos;
              end
           else if ( x )= negationvect.x ) and ( x (= Size.x ) then
              GetActualPos(ready, up, minusexpr, negationvect.x - x);

        exprifthenelse:
           begin
           if (x )= ifvect.x) and (x (= ifexvect.x) then
              begin
```



```
              ready:=true;
              pos:=round((x - ifvect.x) / (ifexvect.x-ifvect.x) * 2);
              pos:=MaxOfInts(0,pos);
              pos:=MinOfInts(1,pos) + cursor.startpos;
              end
           else if (x )= ifexvect.x) and (x (= thenvect.x) then
              GetActualPos(ready, up, ifexpr, x - ifexvect.x)
           else if (x )= thenvect.x) and (x (= thenexvect.x) then
              begin
              ready:=true;
              pos:=round((x - thenvect.x) /
                         (thenexvect.x-thenvect.x) * 4);
              pos:=MaxOfInts(0,pos);
              pos:=MinOfInts(3,pos);
              pos:=MinOfInts(IfLength-3,pos) + ThenPos;
              end
           else if (x )= thenexvect.x) and (x (= elsevect.x) then
              GetActualPos(ready, up, thenexpr, x - thenexvect.x)
           else if (x )= elsevect.x) and (x (= elseexvect.x) then
              begin
              ready:=true;
              pos:=round((x - elsevect.x) /
                         (elseexvect.x-elsevect.x) * 4);
              pos:=MaxOfInts(0,pos);
              pos:=MinOfInts(3,pos);
              pos:=MinOfInts(IfLength-7,pos) + ElsePos;
              end
           else if (x )= elseexvect.x) and (x (= size.x) then
              GetActualPos(ready, up, elseexpr, x - elseexvect.x);
           end; { ifthenelse }

        exprfunction:;

        exprerror:
           if (x )= ErrorVect.x ) and ( x (= NextVect.x ) then
              begin
              ready:=true;
              pos:=FixaPos(NextVect.x-ErrorVect.x,SLength);
              end
           else
              begin
              GetActualPos(ready, up, NextExpr, NextVect.x - x);
              if not ready or (pos = -1) then
                 writeln(chr(7),'Error in GetActualPos 0');
              end;
        exprbinary:
           begin
           if bintype () bindivop then
              begin
              if ( x )= Left.x ) and ( x (= op.x ) then
                 begin
                 GetActualPos(ready, up, expr1, x - Left.x);
                 if not ready or (pos = -1) then
                    writeln(chr(7),' Error in GetActualPos 1');
                 end
              else if ( x )= op.x ) and ( x (= Right.x ) then
                 begin
                 ready:=true;
```

```
                    if bintype = binorop then
                       pos:=FixaPos(Right.x - op.x, 2)
                    else if bintype = binandop then
                       pos:=FixaPos(Right.x - op.x, 3)
                    else
                       pos:=Cursor.StartPos;
                    end
                 else if ( x )= Right.x ) and ( x (= Size.x ) then
                    begin
                    if bintype = binpowerop then
                       GetActualPos(ready, up, expr2,
                                 (x - Right.x) * PrettyPowerScale)
                    else
                       GetActualPos(ready, up, expr2, x - Right.x);
                    if not ready or (pos = -1) then
                       writeln(chr(7),'error in getacualpos 2');
                    end
                 end
              else { bintype = bindivop }
                 begin
                 if up then
                    begin
                    nod:=expr2;
                    if nod () nil then
                       begin
                       x:=MaxOfReals(x,  Right.x*1.01);
                       x:=MinOfReals(x,
                           (Right.x + nod↑.pgfig↑.Size.x)*0.99);
                       GetActualPos(ready, up, nod, x - Right.x);
                       if not ready or (pos = -1) then
                          writeln(chr(7),'Error in GetActualPos 3');
                       end;
                    end
                 else { not up ==) down }
                    begin
                    nod:=expr1;
                    if nod () nil then
                       begin
                       x:=MaxOfReals(x,  Left.x*1.01);
                       x:=MinOfReals(x,
                           (Left.x + nod↑.pgfig↑.Size.x)*0.99);
                       GetActualPos(ready, up, nod, x - Left.x);
                       if not ready or (pos = -1) then
                          writeln(chr(7),'Error in GetActualPos 4');
                       end;
                    end;
                 end; { if bintype () bindivop }
              end;
           end ; { case }
        end; { if and with }
   end; { GetActualPos }

begin { MoveUpOrDown }
pos    :=-1;
x      :=0.0;
divnode:=nil;
ready  :=false;
HandleUpDown(ready, csfound, up, ex, divnode, x);
```



```
if ready and (divnode () nil) then
   begin
   if up then
      begin
      nod:=divnode↑.expr1;
      if nod () nil then
         begin
         x:=MaxOfReals(x, divnode↑.pgfig↑.Left.x*1.01);
         x:=MinOfReals(x,
               (divnode↑.pgfig↑.Left.x + nod↑.pgfig↑.Size.x)*0.99);
         x:=x - divnode↑.pgfig↑.Left.x;
         end;
      end
   else { down }
      begin
      nod:=divnode↑.expr2;
      if nod () nil then
         begin
         x:=MaxOfReals(x, divnode↑.pgfig↑.Right.x*1.01);
         x:=MinOfReals(x,
               (divnode↑.pgfig↑.Right.x + nod↑.pgfig↑.Size.x)*0.99);
         x:=x - divnode↑.pgfig↑.Right.x;
         end;
      end;
      ready:=false;
      GetActualPos(ready, up, nod, x);
      if ready and (pos () -1) then
         InLinePos:=Pos;
      end;
   end; { MoveUpOrDown }
.END
```

```
==========================================================

                use:[magnus.exarb.prettylmtlib.pak

==========================================================

.END
Some useful routines.

Author Magnus Taube
.FORWARD
function MinOfInts(a,b:integer):integer;                    forward;
function MinOfReals(a,b:real):real;                         forward;
function MaxOfInts(a,b:integer):integer;                    forward;
function MaxOfReals(a,b:real):real;                         forward;
function StringLength(id:identifier):integer;               forward;
procedure ErrorAndQuit(str:packed array[1..12:integer] of char);
                                                            forward;
function LastNonSpace(t:image):integer;                     forward;
.PROCEDURE

function LastNonSpace{ (t:image):integer };
   var i  : integer;
       OK : boolean;
   begin
   i:=ImageLength;
   OK:=true;
   while (i>0) and OK do
      begin
      OK:=t[i] = ' ';
      if OK then
         i:=i-1;
      end;
   LastNonSpace:=i;
   end; { LastNonSpace }

function MinOfInts{(a, b : integer) : integer};
   begin
   if a<b then MinOfInts:=a
          else MinOfInts:=b;
   end;

function MinOfReals{(a, b : real) : real};
   begin
   if a<b then MinOfReals:=a
          else MinOfReals:=b;
   end;

function MaxOfInts{(a, b : integer) : integer};
   begin
   if a>b then MaxOfInts:=a
          else MaxOfInts:=b;
   end;

function MaxOfReals{(a, b : real) : real};
   begin
   if a>b then MaxOfReals:=a
          else MaxOfReals:=b;
   end;
```



```
function StringLength{ (id:identifier):integer };
{ returns the length of an indentifier.       }
{ I.e the position for the first space-1      }
   var  i :integer;
   begin
   if id[1] = ' ' then
      i:=0
   else
      begin
      i:=1;
      while (i < idlength-1) and (id[i+1] <> ' ') do
         i:=i+1;
      if (i = idlength-1) and (id[idlength] <> ' ') then
         i:=idlength;
      end;
   StringLength:=i;
   end;

procedure ErrorAndQuit{str:packed array[1..12:integer] of char};
   begin
   writeln(str);
   HALT;
   end;

.END
```

```
use:[magnus.exarb.ladder]ladder.pak

.END
Autho : Magnus Taube


.TYPE
Point       = Vector;

LDItemType = (LDLine, LDMacro, LDText, LDHead);

pConnect    = ↑Connect;

pLDItem     = ↑LDItem;

LDItem      = record
   next, previos : pLDItem;
   Size          : Vector; { Total space used by the list or the macro, }
                           { not valid for lines and texts             }
   case LDIt : LDItemType of
      LDHead : ( exprPlace,
                 exprScale   : Vector  ;
                 expr        : pexpr)  ;
      LDLine : ( Lp1,Lp2     : point   ;
                 used        : boolean);
      LDMacro: ( place       :  point  ; { virtuall  (0,0) in the macro }
                 State       : 0..3    ; { Color code                   }
                 ex          : pexpr   ; { Graphic tree for expression  }
                 explace     : vector  ; { relative from place          }
                 exscale     : vector  ; { Scale for expression         }
                 first       : pLDItem ; { first line in macro.         }
                                         { Only lines allowed in macros }
                 conn        : pConnect);{ Connect points in the macro  }
      LDText:  ( LowerLeft,
                 UpperRight  : point   ;
                 Txt         : image   ; { The text-info. Only one line}
                 Length      : integer);
   end; { LDItem }

Connect = record
   next : pConnect;
   p:point;
end; { Connect }


.VAR
Ladder,InvLadder : pLDItem;

.FORWARD
procedure SubtractPoint(a,b:point;var c:point);    forward;
procedure AddPoint(a,b:point;var c:point);         forward;
procedure InitLadder;                              forward;
procedure CopyConn(Src:pConnect;var Dest:pConnect);forward;

function  CreateLDItem(t:LDItemType):pLDItem;      forward;
procedure DestroyLDItem(var p:pLDItem);            forward;
```



```
procedure DestroyLDItemhead(var p:pLDItem);        forward;
procedure DisposeLadder(var p:pLDItem);            forward;
function  CreateConnect:pConnect;                  forward;
procedure DestroyConnect(var p:pconnect);          forward;
procedure AddToLDMacro(A, Macro : pLDItem);        forward;
procedure CopyLDMacro(Src : pLDItem; var Dest : pLDItem);   forward;

function First(list:pLDItem):pLDItem;      forward;
procedure Out(A:pLDItem);                  forward;
procedure Into(A,list:PLDItem);            forward;
function follow(A,list:PLDItem):pLDItem;   forward;

procedure CheckIfList(l : pLDItem;
                      t : packed array [l1..l2:integer]of char);
                                  forward;

.PROCEDURE
procedure CheckIfList(l : pLDItem;
                      t : packed array [l1..l2:integer]of char);
   begin
   if l = nil then
      writeln(t,' called with list = nil',chr(7))
   else if l↑.LDIt <> LDHead then
      writeln(t,' called with list not of type LDHead',chr(7));
   end;


function First(*list:pLDItem):pLDItem*);
   begin
   CheckIfList(list,'First');
   if list↑.next=list then
      First:=nil
   else
      First:=list↑.next;
   end; { First }

procedure Out(*A:pLDItem*);
   begin
   if A↑.next <> nil then
      A↑.next↑.previos:=A↑.previos;
   if A↑.previos <> nil then
      A↑.previos↑.next:=A↑.next;
   A↑.previos:=nil;
   A↑.next:=nil;
   end; { Out }

procedure Into(*A,list:PLDItem*);
   begin
   if (A↑.previos <> nil) or (A↑.next <> nil) then
      writeln('INTO   A:s pointers is not nil',chr(7));
   CheckIfList(list,' Into');

   A↑.previos:=list↑.previos;
   A↑.next:=list;
   A↑.previos↑.next:=A;
   list↑.previos:=A;
   end; { Into }
```

```
function Follow(*A,list:PLDItem):pLDItem*);
   begin
   CheckIfList(list,' Follow');
   if A↑.next <> list then
      Follow:=a↑.next
   else
      Follow:=nil;
   end; { Follow }


function CreateLDItem(t:LDItemType):pLDItem);
   var p : pLDItem;
       i : integer;
   begin
   new(p);
   with p↑ do
      begin
      next   :=nil;
      previos:=nil;
      Size.x :=0;
      Size.y :=0;
      LDIt   :=t;
      case t of
         LDHead : begin
                  expr      :=nil;
                  ExprPlace :=Size;
                  ExprScale :=Size;
                  next      :=p;      { The head↑.next = head }
                  previos   :=p;
                  end;
         LDLine : begin
                  Lp1:=Size;          { (0,0) }
                  Lp2:=lp1;
                  used:=false;
                  end;
         LDMacro: begin
                  place    :=Size;    { (0,0) }
                  State    :=0;
                  ex       :=nil;
                  explace  :=place;
                  exscale  :=place;
                  first    :=nil;
                  conn     :=nil;
                  end;
         LDText:  begin
                  LowerLeft :=Size;   { (0,0) }
                  UpperRight:=Size;
                  length:=0;
                  for i:=1 to 79 do
                     Txt[i]:=' ';
                  end;
      end ; { case }
   end ; { with }
   CreateLDItem:=p;
   end ; { CreateLDItem }

procedure DestroyLDItem(var p:pLDItem);
   var l, help : pLDItem;
```



```
   begin
   if p <> nil then
      begin
      with p↑ do
         begin
         case LDIt of
            LDHead   : DestroyLDItemHead(p);
            LDText,
            LDLine   : ;
            LDMacro  : begin
                       DisposeExpr(ex);
                       l:=first;
                       while l <> nil do
                          begin
                          help:=l↑.next;
                          dispose(l);      { Only lines allowed !! }
                          l:=help;
                          end;
                       end;
         end; { case }
         if next <> nil then
            next↑.previos:=previos;
         if previos<> nil then
            previos↑.next:=next;
         end; { with p }
      if p <> nil then   { = nil only if DestoyLDItemHead is called }
         dispose(p);
      p:=nil;
      end;
   end ; { DestroyLDItem }


procedure DestroyLDItemhead(*var p:pLDItem*);
   var l:pLDItem;
   begin
   l:=First(p);
   while l <> nil do
      begin
      Out(l);
      DestroyLDItem(l);
      l:=First(p);
      end;
   DisposeExpr(p↑.expr);
   dispose(p);
   p:=nil;
   end;

procedure DestroyConnect(var p:pconnect);
   var h:pconnect;
   begin
   while p <> nil do
      begin
      h:=p↑.next;
      dispose(p);
      p:=h;
      end;
   end;
```

```
procedure DisposeLadder(var p:pLDItem);
   begin
   if p <> nil then
      DestroyLDItemHead(p);
   end;

function CreateConnect(:pConnect);
   var c:pConnect;
   begin
   new(c);
   with c^ do
      begin
      next:=nil;
      p.x:=0;
      p.y:=0;
      end;
   CreateConnect:=c;
   end; { CreateConnect }


procedure AddToLDMacro(A, Macro : pLDItem);
   begin
   if A^.LDIt <> LDLine then
      writeln
        (chr(7),
         'AddLDToMacro called with element which is not a line. Ignored')
   else if Macro^.LDIt <> LDMacro then
      writeln
        (chr(7),
         'AddLDToMacro called with head which is not a macro. Ignored')
   else
      begin
      A^.next:=Macro^.first;
      Macro^.first:=A;
      A^.previos:=nil;
      if A^.next <> nil then
         A^.next^.previos:=A;
      end;
   end; { AddToLDMacro }


procedure CopyLDMacro(Src : pLDItem; var Dest : pLDItem);
   var l, lDest : pLDItem;
   begin
   if Src = nil then
      Dest:=nil
   else
      begin
      Dest:=CreateLDItem(LDMacro);
      Dest^:=Src^;
      Dest^.previos:=nil;
      Dest^.next   :=nil;
      Dest^.first  :=nil;
      CopyConn(Src^.conn, Dest^.conn);
      l:=Src^.First;
      while l <> nil do
         begin
         lDest:=CreateLDItem(LDLine);{ Only lines allowed. No check !! }
```



```
         lDest^:=l^;
         lDest^.previos:=nil;
         lDest^.next   :=nil;
         AddToLDMacro(lDest,Dest);
         l:=l^.next;
         end;
      end;
   end; { CopyLDMacro }


procedure CopyConn(Src:pConnect;var Dest:pConnect);
   var c,last:pConnect;
   begin
   if Src=nil then
      Dest:=nil
   else
      begin
      c:=CreateConnect;
      c^:=src^;
      Dest:=c;
      last:=c;
      Src:=Src^.next;
      while Src <> nil do
         begin
         c:=CreateConnect;
         c^:=src^;
         last^.next:=c;
         last:=c;
         Src:=Src^.next;
         end;
      end;
   end;


procedure  InitLadder;
   var p1, p2 : point;
       l      : pLDItem;

   begin
{
{
{
{
{              4,4   6,4
{                !   !
{    0,2      4,2!   !6,2       10,2
{     -----------!   !-----------
{                !   !
{                !   !
{     x        4,0   6,0
{    0,0
{}
   p1.x:=0;
   p1.y:=2;
   p2.x:=10;
   p2.y:=2;
   Ladder   :=CreateLDItem(LDMacro);
   Ladder^.Size.x:=10;
```

```
   Ladder↑.Size.y:=4;
   Ladder↑.conn:=CreateConnect;
   Ladder↑.conn↑.p:=p1;
   Ladder↑.conn↑.next:=CreateConnect;
   Ladder↑.conn↑.next↑.p:=p2;              { connects ready }

   l:=createLDItem(LDLine);
   l↑.Lp1.x:=0;
   l↑.Lp1.y:=2;
   l↑.Lp2.x:=4;
   l↑.Lp2.y:=2;
   AddToLDMacro(l,Ladder);
   l:=createLDItem(LDLine);
   l↑.Lp1.x:=6;
   l↑.Lp1.y:=2;
   l↑.Lp2.x:=10;
   l↑.Lp2.y:=2;
   AddToLDMacro(l,Ladder);
   l:=createLDItem(LDLine);
   l↑.Lp1.x:=6;
   l↑.Lp1.y:=0;
   l↑.Lp2.x:=6;
   l↑.Lp2.y:=4;
   AddToLDMacro(l,Ladder);
   l:=createLDItem(LDLine);
   l↑.Lp1.x:=4;
   l↑.Lp1.y:=0;
   l↑.Lp2.x:=4;
   l↑.Lp2.y:=4;
   AddToLDMacro(l,Ladder);

   CopyLDMacro(Ladder,InvLadder);
   l:=createLDItem(LDLine);
   l↑.Lp1.x:=3;
   l↑.Lp1.y:=0;
   l↑.Lp2.x:=7;
   l↑.Lp2.y:=4;
   AddToLDMacro(l,InvLadder);

   end;

procedure SubtractPoint(a,b:point;var c:point);
   begin
   c.x:=a.x-b.x;
   c.y:=a.y-b.y;
   end;

procedure AddPoint(a,b:point;var c:point);
   begin
   c.x:=a.x+b.x;
   c.y:=a.y+b.y;
   end;


.INIT
INITLadder;
.END
```



```
.END
Author Magnus Taube
Makes a list defining a ladder diagram to an expression
.FORWARD
procedure MakeLadder(mex: pexpr; var LDList: pLDItem);   forward;

.VAR
diff : point;

.FORWARD
procedure MakeLadderDo
   (ex:pexpr;pstart:point;var pend,plower:point;var LDlist:pLDItem);
      forward;
.PROCEDURE

procedure MakeLadder(mex: pexpr; var LDList: pLDItem);
   var e, eleft         : pexpr;
       pstart, pend,
       plower           : point;
       binop            : boolean;
   begin
   pend.x :=0;
   pend.y :=0;
   pstart :=pend;
   plower :=pend;

   LDList:=CreateLDItem(LDHead);

   e:=deMorgan(CopyExpr(mex)); { Create temporary expression, passed }
                               { thru deMorgans algoritm.            }
   RemoveChInfo(e);
   if e <> nil then
      begin
      binop:=false;
      if e↑.exprtype = exprbinary then
         if e↑.bintype = binequalop then
            binop:=true;
      if binop then
         begin
         MakeLadderDo(e↑.expr2,pstart,pend,plower,LDList);
         eleft:=CopyExpr(e↑.expr1);
         PrePretty(eleft,false);
         LDList↑.expr:=eleft;
         LDList↑.exprPlace.x:=pend.x + diff.x/2;
         LDList↑.exprPlace.y:=0;
         with LDList↑.exprScale do
            begin
            x:=diff.x/eleft↑.pgfig↑.Size.x;
            y:=diff.y * 0.3/eleft↑.pgfig↑.Size.y;
            end;
         LDList↑.Size.x:=pend.x + diff.x/2 + diff.x;
         LDList↑.Size.y:=abs(plower.y) { the lowest connect point        }
                         +2            { space for the lowest Ladder     }
                         +diff.y/2     { base of expression over Ladder  }
```

```
                        +diff.y*0.3;  { space for it                    }
         end
      else { not binop }
         MakeLadderDo(e,pstart,pend,plower,LDList);
      DisposeExpr(e);      { Dispose temporary expression }
      end
   else { ex = nil }
      begin
      end;

   end;

procedure MakeLadderDo
    {ex:pexpr;pstart:point;var pend,plower:point;var LDlist:pLDItem};

   var Local          : pLDItem;
       lower,
       point1,point2 : point;
       xscale,yscale : real;

   procedure MLError(i:integer);
      begin
      writeln(i:2,'MakeLadder ERROR   ******************');
      ErrorAndQuit('Leaving program due to fatal error');
      end;

   procedure ML(p:pLDItem;e:pexpr);
      var L     : pLDItem;
          MLex : pexpr;
      begin
      CopyLDMacro(p,l);
      SubtractPoint(pstart,L↑.conn↑.p,l↑.place);
      AddPoint(L↑.place,l↑.conn↑.next↑.p,pend);
      plower:=l↑.place;
      MLex:=CopyExpr(e);
      PrePretty(MLex,false);
      l↑.ex:=MLex;
      l↑.explace.x:=diff.x/2;
      l↑.explace.y:=diff.y/3;
      if MLex↑.pgfig↑.size.x <> 0 then
         xscale:=Ladder↑.Size.x * 0.5 / MLex↑.pgfig↑.size.x
      else
         xscale:=1;
      if MLex↑.pgfig↑.size.y <> 0 then
         yscale:=diff.y * 0.3 / MLex↑.pgfig↑.size.y
      else
         yscale:=1;
      with l↑.exscale do
         begin
         x:=xscale;     { max (--) min etc }
         y:=yscale;
         end;
      Into(l,LDList);
      end; { ML }

   begin

   if ex = nil then
```



```
      MLerror(1)
   else
      with ex↑ do
         begin
         case exprtype of
            exprvariable:       ML(Ladder,ex);
            exprfunction:       ML(Ladder,ex);
            exprnumber:         ML(Ladder,ex);
            exprminus:          ML(Ladder,ex);
            exprnot:
               begin
               ML(InvLadder,ex↑.notexpr);
               end;
            exprifthenelse:     ML(Ladder,ex);
            exprerror:          ML(Ladder,ex);
            exprbinary:
               begin
               case bintype of
                  binaddop,
                  binsubop,
                  binmultop,
                  bindivop,
                  binpowerop,
                  binequalop:   ML(Ladder,ex);

                  binandop:
                     begin
                     lower:=plower;
                     MakeLadderDo(expr1,pstart,point1,lower,LdList);
                     if lower.y < plower.y then
                        plower:=lower;
                     MakeLadderDo(expr2,point1,pend,lower,LdList);
                     if lower.y < plower.y then
                       plower:=lower;
                     end;

                  binorop:
                     begin
                     MakeLadderDo(expr1,pstart,point1,plower,LDList);
                     Local:=CreateLDItem(LDLine);
                     Local↑.lp1:=pstart;

                     Local↑.lp2.x:=pstart.x;
                     Local↑.lp2.y:=plower.y-diff.y/2;
                     Into(Local,LDList);
                     MakeLadderDo(expr2,local↑.lp2,point2,plower,LDList);
                     if point1.x <> point2.x then
                        begin
                        local:=CreateLDItem(LDLine);
                        if point1.x < point2.x then
                           begin
                           local↑.lp1  :=point1;
                           Local↑.lp2.x:=point2.x;
                           Local↑.lp2.y:=point1.y;
                           point1.x    :=point2.x;
                           end
                        else
                           begin
```

```
                        local↑.lp1  :=point2;
                        Local↑.lp2.x:=point1.x;
                        Local↑.lp2.y:=point2.y;
                        point2.x    :=point1.x;
                        end;
                     Into(Local,LDList); { horisontal line ready }
                     end;
                  local:=CreateLDItem(LDLine);
                  local↑.lp1:=point1;
                  local↑.lp2:=point2;
                  pend:=point1;
                  Into(local,LDList);      { Vertical line ready }
                  end;

               bingreaterop,
               binlessop:      ML(Ladder,ex);

               end; { case bintype }
            end;{ exprbinary }

         end; { case exprtype }
      end; { with and if }
   end;
.INIT


   diff.x:=5;
   diff.y:=15;

.END
```



```
.END
Author Magnus Taube
Draws a ladder on screen
.CONST
LDtrueColor  = 3;
LDfalseColor = 7;
LDexprColor  = 2;

.FORWARD
procedure DrawLDList(list:pLDItem);                    forward;
procedure DrawLadder(list : pLDItem);                  forward;
.PROCEDURE

procedure DrawLadder(* list : pLDItem *);
   const LadderScale = 4.0;
   var T : TransfDescr;
   begin
   SaveTransform(T);
   Scale(LadderScale, LadderScale);
   SetColor(6);
   DrawLDList(list);
   SetTransform(T);
   end;

procedure DrawLDList(h:pLDItem);

   { DrawLDList } procedure DrawLDHead(h:pLDItem);
      var t : TransfDescr;
      begin
      if h <> nil then
         begin
         SaveTransform(t);
         Translate(h↑.exprPlace.x, h↑.exprPlace.y);
         Scale(h↑.exprScale.x,h↑.exprScale.y);
         if h↑.expr <> nil then
            Pretty(h↑.expr,false);
         SetTransform(t);
         end;
      end; { DrawLDHead }

   { DrawLDList } procedure DrawLDItem(i:pLDItem);
      var t : TransfDescr;
          l : pLDItem;
      begin
      if i <> nil then
         with i↑ do
         begin
            case LDIt of
               LDLine : begin
                        MoveTo(lp1.x,lp1.y);
                        LineTo(lp2.x,lp2.y);
                        end;
               LDMacro: begin
                        if ex <> nil then
```

```
                        begin
                        SaveTransform(t);
                        Translate(  place.x,  place.y);
                        l:=first;
                        while l <> nil do
                           begin
                           DrawLDItem(l);   { if Status = 1,2,3, ?? }
                           l:=l^.next;
                           end;
                        Translate(explace.x,explace.y);
                        Scale(exscale.x,exscale.y);
                        SetColor(LDexprColor);
                        Pretty(ex,false);
                        SetTransform(t);
                        end;
                     end;
            Otherwise writeln('ERROR CASE IN DRAWLDITEM LDIT=',ldit);
            end; { case }
         end ; {with}
      end; { DrawLDItem }

   { DrawLDList } procedure DrawLDListDo(list:pLDItem);
      var l:pLDItem;
      begin
      if list <> nil then
         begin
         l:=list;
         while l^.LDIt <> LDHead do
            begin
            DrawLDItem(l);
            l:=l^.next;
            end;
         end;
      end; { DrawLDListDo }

   begin
   if list <> nil then
      begin
      DrawLDHead(list);
      DrawLDListDo(list^.next);
      end;
   end; { DrawLDList }


.END
```

use:[magnus.exarb.pretty]Prettyini.pak

```
.INIT
GetFont;
outinit;
TwoPlanes;
.END
```

```
use:[magnus.exarb.ladder]laddbasic.pak
```

```
.END
This file contains some useful declarations for the picturecompilation
of ladderdiagrams in LICS.

Author   : Magnus Taube
.CONST
ImageLength = 79;
.TYPE
gfig   = record { Dummy } end;
list   = record { Dummy } end;
text3  = packed array[1.. 3] of char;
text4  = packed array[1.. 4] of char;
image  = packed array[1..ImageLength] of char;
Vector = record x,y : real ; end;
InteractType =(InteractChar);

.FORWARD
procedure prepretty(e : pexpr; b : boolean); forward;
function Interact(var c : char): Interacttype;
begin
writeln('Interact');
Interact:=InteractChar;
end;

.FORWARD
function Digit(c:char):boolean;    forward;
function Letter(c:char):boolean;   forward;

.PROCEDURE
procedure prepretty(* e : pexpr; b : boolean* );
   begin
   { Never called }
   end;

function Digit{(c:char):bollean};
   begin
   Digit:=c in ['0'..'9'];
   end;

function Letter{(c:char):boolean};
   begin
   Letter:=c in ['a'..'z','A'..'Z'];
   end;

.END
```

```
use:[magnus.exarb.exprlexprinf.pak
```

```
{ PACKAGE EXPRINF is

-- Routines for producing infix form of assignment statements
-- from the tree representation

-- Author: Hilding Elmqvist
-- Date:   1981-07-26
}

.VAR
   charcount :integer;


.FORWARD

procedure countchar(nrchar: integer); forward;
procedure WriteIdent(ident: identifier); forward;



.PROCEDURE

procedure countchar(nrchar: integer);
  begin
  charcount := charcount+nrchar;
  end;


procedure WriteIdent(ident: identifier);
  var i: integer;
  begin
  if charcount ) 70 then
     begin
     writeln(output);
     charcount:=0;
     end;
  for i:=1 to 15 do
    if ident[i]()' ' then
      begin
      write(output,ident[i]);
      charcount := charcount + 1;
      end
  end;

.END
.FORWARD
procedure infix(ex: pexpr);    forward;
procedure WriteInfix(e:pexpr); forward;
{ end exprinf; }

.END
```

```
 PACKAGE BODY EXPRINF is

.FORWARD
  procedure infixpri(ex: pexpr; pri: integer); forward;

.PROCEDURE

  procedure leftpar(pri, priority: integer);
    begin
    if pri<priority then
    WriteIdent('(              ')
    end;


  procedure rightpar(pri, priority: integer);
    begin
    if pri<priority then
    WriteIdent(')              ')
    end;


{     ------------------------------------------------------     }

  procedure varinfix(ex: pexpr);
    begin
    WriteIdent(ex^.varid);
    end;

{     ------------------------------------------------------     }

  procedure funcinfix(ex: pexpr);
    var   i: integer;
          arg: parglist;
    begin
    with ex^ do
      begin
      WriteIdent(func);
      write(output,'(');
      countchar(1);
      arg := argumentlist;
      for i:=1 to nargs do
         begin
         if i>1 then
            begin
            write(output,',');
            countchar(1);
            end;
         infix(arg^.ex);
         arg:=arg^.next;
         end;

      write(output,')');
      countchar(1);
      end;
    end;
```



```
  procedure numinfix(ex: pexpr);
    begin
    WriteIdent(ex^.numid);
    end;


  procedure minusinfix(ex: pexpr; pri: integer);
    const minuspri = 6;
    begin
    leftpar(pri, minuspri);
    write(output, '-');
    countchar(1);
    infixpri(ex^.minusexpr, minuspri-1);
    rightpar(pri, minuspri);
    end;


  procedure notinfix(ex: pexpr; pri: integer);
    const notpri = 6;
    begin
    leftpar(pri, notpri);
    write(output, ' NOT ');
    countchar(5);
    infixpri(ex^.notexpr, notpri-1);
    rightpar(pri, notpri);
    end;


  procedure ifinfix(ex: pexpr; pri: integer);
    const ifpri = 14;
    begin
    with ex^ do
      begin
      leftpar(pri, ifpri);
      write(output, 'if ');
      countchar(3);
      infixpri(ifexpr, ifpri);
      write(output,' then ');
      countchar(6);
      infixpri(thenexpr, ifpri-1);
      write(output,' else ');
      countchar(6);
      infixpri(elseexpr, ifpri);
      rightpar(pri, ifpri);
      end;
    end;


  procedure bininfix(ex: pexpr; pri,prior1:integer;
            op:text3; prior2:integer);
    var   i:integer;
    begin
    with ex^ do
      begin
```

```
      leftpar(pri, prior1);
      infixpri(expr1, prior1);
      if op[1] = ' ' then
        begin
        write(output, op);
        countchar(3);
        end
      else if op = 'AND' then
        begin
        write(output,' AND ');
        countchar(5);
        end
      else if op = 'OR ' then
        begin
        write(output,' OR ');
        countchar(4);
        end
      else
        begin
        i:=1;
        while (i <= 3) and (op[i] <> ' ') do
          begin
          write(output,op[i]);
          i:=i+1;
          countchar(1);
          end
        end;
      infixpri(expr2, prior2);
      rightpar(pri, prior1);
      end;
    end;



  procedure addinfix(ex: pexpr;  pri: integer);
    begin
    bininfix(ex, pri, 8, ' + ', 8);
    end;


  procedure subinfix(ex: pexpr;  pri: integer);
    begin
    bininfix(ex, pri, 8, ' - ', 7);
    end;


  procedure multinfix(ex: pexpr;  pri: integer);
    begin
    bininfix(ex, pri, 4, '*  ', 4);
    end;


  procedure divinfix(expr: pexpr;  pri: integer);
    begin
    bininfix(expr, pri, 4, '/  ', 3);
    end;
```



```
  procedure powerinfix(ex: pexpr;  pri: integer);
    begin
    bininfix(ex, pri, 2, '** ', 1);
    end;


  procedure equalinfix(ex: pexpr);
    begin
    bininfix(ex, 18, 16, ' = ', 16);
    end;


procedure andinfix(ex: pexpr;  pri: integer);
  begin
  bininfix(ex, pri, 11, 'AND', 11)                { N B   11 / MT }
  end;


procedure orinfix(ex: pexpr;  pri: integer);
  begin
  bininfix(ex, pri, 12, 'OR ', 12)
  end;


procedure greaterinfix(ex: pexpr;  pri: integer);
  begin
  bininfix(ex, pri, 10, ' > ', 10)
  end;


procedure lessinfix(ex: pexpr;  pri: integer);
  begin
  bininfix(ex, pri, 10, ' < ', 10)
  end;


procedure infixpri { ex: pexpr;  pri: integer } ;
   var i : integer;
   begin
   if ex <> nil then
      begin
      if ex↑.cursor.iscursor then
         write ('# pos=',ex↑.cursor.pos:1);
      case ex↑.exprtype of
        exprvariable:   varinfix(ex);
        exprfunction:   funcinfix(ex);
        exprnumber:     numinfix(ex);
        exprminus:      minusinfix(ex, pri);
        exprnot:        notinfix(ex, pri);
        exprifthenelse: ifinfix(ex, pri);
        exprchinfo:     if ex↑.before then
                           begin
                           write('!',ex↑.ch,'!');
                           infixpri(ex↑.chexpr,pri);
                           end
                        else
                           begin
```

```
                        infixpri(ex↑.chexpr,pri);
                        write('!',ex↑.ch,'!');
                        end;
      exprerror:      begin
                      if ex↑.nextexpr <> nil then
                         infixpri(ex↑.nextexpr,pri);
                      writeln('ERROR');
                      write('#');
                      for i:=1 to ex↑.ErrLength do
                         write(ex↑.s[i]);
                      write('#',ex↑.ErrorNumber:1,'#');
                      end;

      exprbinary:
         case ex↑.bintype of
            binaddop:     addinfix(ex, pri);
            binsubop:     subinfix(ex, pri);
            binmultop:    multinfix(ex, pri);
            bindivop:     divinfix(ex, pri);
            binpowerop:   powerinfix(ex, pri);
            binequalop:   equalinfix(ex);
            binandop:     andinfix(ex, pri);
            binorop:      orinfix(ex, pri);
            bingreaterop: greaterinfix(ex, pri);
            binlessop:    lessinfix(ex, pri);
            end; { case ex↑.bintype }
       end; { case }
     end; { if }
  end;


procedure infix { ex: pexpr } ;
  begin
  infixpri(ex, 100);
  end;

procedure WriteInfix(e:pexpr);
   begin
   charcount:=0;
   infix(e);
   writeln;
   end;


.END
```



```
.END
Author Magnus Taube
.TYPE
LDIdType = (LDAnd, LDOr, LDLeaf);
pLDtree = ↑LDTree;

LDTree  = record
   number             : integer;
   id                 : LDIdType;
   Pict               : pLDItem;
   next, previos,
   left, right        : pLDTree;
   ex                 : pexpr;
   extext             : image;
   extextlength       : integer;
   cpleft, cpright    : Point;
   end;

.VAR
treenumber:integer;
.INIT
treenumber:=1;
.FORWARD

function  LDLineLength(p:pLDItem):real;                                  forward;
procedure LDanalys(unused:pLDItem; var used:pLDItem);                    forward;
function  FindParallellvert(Akt,list:pLDItem):pLDItem;                   forward;
procedure FindHorizontell
     (V1,V2:pLDItem;var H1,H2:pLDItem;list:pLDItem);                     forward;
function  FindDiagLine(V1,V2,list:pLDItem):pLDItem;                      forward;
procedure CreateLadder(var l1,l2,l3,l4,l5:pLDItem;list:pLDItem);         forward;
function Intervall(a,b,c:real):boolean;                                  forward;



procedure ConnectPoint(dir : char; it : pLDItem; var res : point); forward;

function ConnectedToTree
   (side : char; p : point; list : pLDItem; tree : pLDTree;
                                        var node : pLDTree):boolean;forward;
function ConnectedToList(side : char; p : point; list : pLDItem;
                           var it : pLDItem):boolean;                    forward;

function InContact(list : pLDItem; p1, p2 : point):boolean;              forward;
procedure MarkAsUsed(list : pLDItem; p1,p2 : point);                     forward;
function EqvPoint(p1,p2:point):boolean;                                  forward;

function ConvertToExpr(tree : pLDTree):pexpr;                            forward;
function MakeNewNode(ty : LDIdType):pLDTree;                             forward;

function FollowTree(Item:pLDTree):pLDTree;                               forward;
procedure OutTree(Item:pLDTree);                                         forward;
procedure IntoTree(Item:pLDTree);                                        forward;
.PROCEDURE
```

```
procedure TreeOutdo(t:pLDTree);
   var i : integer;
   begin
   if t = nil then
      write(' NIL ')
   else
      with t↑ do
         case id of
            LDLeaf  : begin
                      write ('Leaf',number:1,' ');
                      write('extextlength',extextlength:1,'  !');
                      for i:=1 to extextlength do
                         write(extext[i]);
                      if extextlength <> 0 then
                         write('!  ');
                      end;
            Otherwise begin
                      TreeOutDo(left);write(' ');
                      Write(' ',id,number:1,' ');
                      TreeOutDo(right);write(' ');
                      end;
            end;
   end;

procedure TreeOut(t:pLDTree);
   begin
   write('treeOut:');
   TreeOutDo(t);
   writeln;
   end;

procedure TreeListOut;
   var t : pLDTree;
       i : integer;
   begin
   t:=TreeList;
   i:=1;
   writeln('TreeListOut');
   while t <> nil do
      begin
      writeln('Node nr ',i:1);
      TreeOutDo(t);
      i:=i+1;
      t:=FollowTree(t);
      end;
   end; { TreeListOut }

 procedure ListOut(l:pLDItem);  { Debug routine }
   var a : pLDItem;
   begin
   a:=first(l);
   while a <> nil do
      begin
      with a↑ do
         case LDIt of
            LDLine:
```



```
               writeln
                  ('ListOut      LINE p1,p2= (',lp1.x:5:1,',',lp1.y:5:1,
                       ')(',lp2.x:5:1,',',lp2.y:5:1,')');
            LDMacro:
               writeln('ListOut   LDMacro c1,c2= (',conn↑.p.x:5:1,',',
                       conn↑.p.y:5:1,')(',conn↑.next↑.p.x:5:1,',',
                       conn↑.next↑.p.y:5:1,')');
            LDText: begin
               writeln('ListOut   LDText. ll,ur = (',LowerLeft.x:5:1,',',
                     LowerLeft.y:5:1,')(',UpperRight.x:5:1,',',
                     UpperRight.y:5:1,')');
               writeln(txt);
               end;
            end; { case }
      a:=follow(a,l);
      end; { while }
   end;

procedure OutTree(Item : pLDTree);
   { Removes Item from the global list treelist }
   begin
   if Item = treelist then
      begin
      treelist:=Item↑.next;
      if treelist <> nil then
         treelist↑.previos:=nil;
      end
   else
      begin
      Item↑.previos↑.next:=Item↑.next;
      if Item↑.next <> nil then
         Item↑.next↑.previos:=Item↑.previos;
      end;
   Item↑.previos:=nil;
   Item↑.next:=nil;
   end; { OutTree }

function FollowTree((Item:pLDTree):pLDTree);
   begin
   FollowTree:=Item↑.next;
   end; { FollowTree }

procedure IntoTree(Item : pLDTree);
   { Puts Item into the global TreeList as first item }
   begin
   if TreeList <> nil then
      TreeList↑.previos:=Item;
   Item↑.previos:=nil;
   Item↑.next:=TreeList;
   TreeList:=Item;
   end; { IntoTree }

function MakeNewNode(ty : LDIdType):pLDTree);
   var res : pLDTree;
   begin
   new(res);
   with res↑ do
      begin
```

```
      number      :=treenumber;
      treenumber  :=treenumber+1;
      id          :=ty;
      Pict        :=nil;
      left        :=nil;
      right       :=nil;
      next        :=nil;
      previos     :=nil;
      ex          :=nil;
      cpleft.x    :=0;
      cpleft.y    :=0;
      cpright     :=cpleft;
      {  extext : not initialized }
      extextlength :=0;
      end;
   MakeNewNode:=res;
   end;

procedure ConvertToTreeList(list : pLDItem);
   var Akt,help    : pLDItem;
       last, ltree : pLDTree;
   begin
   treelist:=nil; { global }
   last:=nil;
   Akt:=First(list);
   while Akt <> nil do
      begin
      ltree:=MakeNewNode(LDLeaf);
      with ltree^ do
         begin
         pict        :=Akt;
         cpleft      :=Akt^.conn^.p;
         cpright     :=Akt^.conn^.next^.p;
         if last  = nil then
            begin
            treelist:=ltree;
            last    :=ltree;
            end
         else
            begin
            last^.next:=ltree;
            previos    :=last;
            last       :=ltree;
            end;
         help:=Follow(Akt,list);
         Out(Akt);
         Akt:=help;
         end; { with }
      end; { while }
   end; { ConvertToTreeList }


procedure BuildLadderExpression(connlist : pLDItem);
   var changed          : boolean;
       temp, Akt, help : pLDTree;
   { BuildLadderExpression } function OrConnected
                                (t1, t2 : pLDTree):boolean;
      begin
```



```
      OrConnected :=InContact(connlist,t1^.cpleft ,t2^.cpleft) and
                    InContact(connlist,t1^.cpright,t2^.cpright);
      end; { OrConnected }

   { BuildLadderExpression } function AndConnected
                                (t1, t2 : pLDTree):boolean;
      var help          : pLDTree;
          t1left, break : boolean;
      begin
      if  InContact(connlist,t1^.cpleft ,t2^.cpright) or
          InContact(connlist,t1^.cpright,t2^.cpleft) then
         begin
         t1left:=t1^.cpleft.x < t2^.cpleft.x;
                        { true (=) t1 is left ladder }
         help:=TreeList;
         break:=false;
         while not break and (help <> nil) do
            begin
            if (help <> t1) and (help <> t2) then
               begin
               if t1left then
                  break:=InContact(connlist,t1^.cpright,help^.cpleft) or
                         InContact(connlist,t2^.cpleft,help^.cpright)
               else
                  break:=InContact(connlist,t1^.cpleft,help^.cpright) or
                         InContact(connlist,t2^.cpright,help^.cpleft);
               end;
            help:=FollowTree(help);
            end; { while }
         AndConnected:=not break;
         end
      else
         AndConnected:=false;
      end; { AndConnected }

procedure Delay(t:real);extern;

   begin { BuildLadderExpression }
   changed:=true;
   while changed do
      begin
      changed:=false;
      Akt:=treelist;
      while not changed and (Akt <> nil) do
         begin
         temp:=FollowTree(Akt);
         while not changed and (temp <> nil) do
            begin
            if OrConnected(Akt,temp) then
               begin
               help:=MakeNewNode(LDOr);
               OutTree(Akt);
               OutTree(temp);
               help^.left:=Akt;
               help^.right:=temp;
               help^.cpleft:=Akt^.cpleft;
               help^.cpright:=Akt^.cpright;
               IntoTree(help);
```

```
            changed:=true;
            end
         else
            temp:=FollowTree(temp);
         end; { while not changed and (temp <> nil) }
      if not changed then
         Akt:=FollowTree(Akt);
      end; { while not changed and (Akt <> nil) }
   if not changed then
      begin
      Akt:=TreeList;
      while not changed and (Akt <> nil) do
         begin
         temp:=FollowTree(Akt);
         while not changed and (temp <> nil) do
            begin
            if AndConnected(Akt,temp) then
               begin
               help:=MakeNewNode(LDAnd);
               OutTree(Akt);
               OutTree(temp);
               if Akt^.cpleft.x < temp^.cpleft.x then
                  begin
                  help^.left:=Akt;
                  help^.right:=temp;
                  help^.cpleft:=Akt^.cpleft;
                  help^.cpright:=temp^.cpright;
                  end
               else
                  begin
                  help^.left:=temp;
                  help^.right:=Akt;
                  help^.cpleft:=temp^.cpleft;
                  help^.cpright:=Akt^.cpright;
                  end;
               changed:=true;
               IntoTree(help);
               end
            else
               temp:=FollowTree(temp);
            end; { while not changed and (temp <> nil) }
         if not changed then
            Akt:=FollowTree(Akt);
         end; { while not changed and (Akt <> nil) }
      end; { if changed }
   end; { while changed }
end; { BuildLadderExpression }


function EqvPoint(p1,p2:point):boolean);
   begin
   EqvPoint:=(p1.x=p2.x) and (p1.y=p2.y);
   end;

procedure ConnectPoint(dir : char; it : pLDItem; var res : point);
   begin
   case dir of
      'L' : res:=it^.conn^.p;      { Guaranted by CreateLadder }
```



```
      'R' : res:=it^.conn^.next^.p;
      otherwise
         writeln(chr(7),'ConnectPoint called with wrong direction !');
      end;
   end;


function NextOfType(p,list:pLDItem;t:LDItemType):pLDItem;
   var Akt   : pLDItem;
       found : boolean;
   begin
   if (p=nil) or ( list=nil) then
      NextOfType:=nil
   else
      begin
      found:=false;
      Akt:=Follow(p,list);
      while (Akt <> nil) and not found do
         begin
         found:=Akt^.LDIt=t;
         if not found then
            Akt:=Follow(Akt,list);
         end;
      if found then
         NextOfType:=Akt
      else
         NextOfType:=nil;
      end;
   end; {NextOfType }

function NextMacro(p,list:pLDItem):pLDItem;
   begin
   NextMacro:=NextOfType(p,list,LDMacro);
   end;

function NextLine1(p,list:pLDItem):pLDItem;
   begin
   NextLine1:=NextOfType(p,list,LDLine);
   end;

function FirstLine(list:pLDItem):pLDItem;
   begin
   FirstLine:=NextLine1(First(list),list);
   end;


function  LDLineLength(p:pLDItem):real);
   begin
   with p^ do
      LDLineLength:=sqrt(sqr(lp1.x-lp2.x)+sqr(lp1.y-lp2.y));
   end;

function  FindDiagLine(V1,V2,list:pLDItem):pLDItem);
   var xlow,xhigh,l,
       ylow,yhigh   : real   ;
       Akt          : pLDItem;
       found        : boolean;
   begin
```

```
   l:=( LDLineLength(V1) + LDLineLength(V2) )/2;
                              { length of vert lines }
   Akt:=First(list);
   if V1↑.Lp1.x ) V2↑.Lp1.x then
      begin
      xlow :=V2↑.Lp1.x;
      ylow :=minofreals(V2↑.Lp1.y,V2↑.Lp2.y);
      xhigh:=V1↑.Lp1.x;
      yhigh:=maxofreals(V1↑.Lp1.y,V1↑.Lp2.y);
      end
   else
      begin
      xlow:= V1↑.Lp1.x;
      ylow:= minofreals(V1↑.Lp1.y,V1↑.Lp2.y);
      xhigh:=V1↑.Lp2.x;
      yhigh:=maxofreals(V2↑.Lp1.y,V2↑.Lp2.y);
      end;
   found:=false;
   while ( Akt <> nil ) and not found do
      begin
      if Akt↑.LDIt = LDLine then
         with Akt↑ do
            if Interval1(Lp1.x,xlow-0.2*l,0.1*l)  and
               Interval1(Lp1.y,ylow       ,0.2*l) and
               Interval1(Lp2.x,xhigh+0.2*l,0.1*l) and
               Interval1(Lp2.y,yhigh      ,0.2*l)     or
               Interval1(Lp2.x,xlow-0.2*l ,0.1*l) and
               Interval1(Lp2.y,ylow       ,0.2*l) and
               Interval1(Lp1.x,xhigh+0.2*l,0.1*l) and
               Interval1(Lp1.y,yhigh      ,0.2*l)     then
               found:=true;
      if not found then
         Akt:=Follow(Akt,list)
      end;
   if found then
      FindDiagLine:=Akt
   else
      FindDiagLine:=nil;

   end { FindDiagLine } ;


procedure LDanalys(unused:pLDItem; var used:pLDItem);
   { Search thru the input list (unused) for ladder-macros. Creates a new}
   { list (used) with all found ladder-macros and all lines connected    }
   { directly or indirectly to any ladder.                               }

   var Akt, Vert, H1, H2, Inv, Next : pLDItem;

   { LDAnalys } procedure TakeOut(l : pLDItem);
      { Take out l from the list (unused) and check if l = next, then }
      { Next is uppdated. Next becomes the next member of unused.     }
      begin
      if l = Next then
         Next:=Follow(l, unused);
      Out(l);
      end; { TakeOut }
```



```
   { LDAnalys } function ConnToList(it : pLDItem; list : pLDItem):boolean;
      var dummy : pLDItem;
      begin
      if it↑.LDIt = LDLine then
         with it↑ do
            ConnToList:=ConnectedToList('L',Lp1,list,dummy) or
                        ConnectedToList('R',Lp1,list,dummy)
      else
         ConnToList:=false;
      end;

   begin
   used:=CreateLDItem(LDHead);
   Akt:=First(unused);
   while Akt <> nil do
      begin
      Next:=Follow(Akt, unused);
      Vert:=FindParallelVert(Akt,unused);
      if Vert <> nil then
         begin
         FindHorizontell(Akt,Vert,H1,H2,unused);
         if (H1 <> nil) and (H2 <> nil) then
            begin
            TakeOut(Akt);
            TakeOut(Vert);
            TakeOut(H1);
            TakeOut(H2);
            Inv:=FindDiagLine(Akt,Vert,unused);
            if Inv <> nil then
               TakeOut(Inv);
            CreateLadder(Akt,Vert,H1,H2,Inv,used);
            end;
         end;
      Akt:=Next;
      end;

   { Now: Try to pick up the rest of unused list and move them to
      the used list. We move every item which has at least 1
      connect to the used list }

   Akt:=First(unused);
   while Akt <> nil do
      if ConnToList(Akt,Used) then
         begin
         Out(Akt);
         Into(Akt,used);
         Akt:=First(unused);    { start from begining again, easy }
         end                    { but not very smart }
      else
         Akt:=Follow(Akt,unused);

   end { LDanalys } ;

function ConnectedToList((side : char; p : point; list : pLDItem;
                          var it : pLDItem):boolean);

   var Akt   : pLDItem;
       found : boolean;
```

```
      ploc  : point;
   begin
   Akt:=First(list);
   found:=false;
   while not found and (akt <> nil) do
      if Akt^.LDIt = LDMacro then
         begin
         ConnectPoint(side,Akt,ploc);
         found:=InContact(list,p,ploc);
         if not found then
            Akt:=Follow(Akt,list);
         end
      else
         Akt:=Follow(Akt,list);

   if found then
      it:=Akt
   else
      it:=nil;
   ConnectedToList:=found;
   end; { ConnectedToList }


procedure CreateLadder(var l1,l2,l3,l4,l5:pLDItem;list:pLDItem);
   { l1 and l2 is vertikal, l3 and l4 is horizontal,                }
   { l5 is inverterline or nil.                                     }
   { Creates a Ladder-macro of the 4 (5) lines. The lines should not }
   { be a part of any list. The created macro is added to the list   }
   { 'list'.                                                        }

   var con1, con2, pll, pur : point;
       LD                   : pLDItem;

   { CreateLadder } procedure Setp(p:pLDItem);   { very local procedure }
      begin
      if p <> nil then
         with p^ do
            begin
            pll.x:=minofreals(pll.x,lp1.x);
            pll.x:=minofreals(pll.x,lp2.x);
            pll.y:=minofreals(pll.y,lp1.y);
            pll.y:=minofreals(pll.y,lp2.y);
            pur.x:=maxofreals(pur.x,lp1.x);
            pur.x:=maxofreals(pur.x,lp2.x);
            pur.y:=maxofreals(pur.y,lp1.y);
            pur.y:=maxofreals(pur.y,lp2.y);
            end;
      end; { Setp }

   { CreateLadder } function Left(a,b:pLDItem):boolean;
      begin
      Left:=minofreals(a^.Lp1.x , a^.Lp2.x) <
            minofreals(b^.Lp1.x , b^.Lp2.x);
      end; { Left }

   { CreateLadder } procedure InLine(var p:pLDItem; mac : pLDItem);
                              { very local }
      begin
```



```
      if p <> nil then
         begin
         with p^ do
            begin
            SubtractPoint(lp1,pll,lp1);
            SubtractPoint(lp2,pll,lp2);
            end;
         AddToLDMacro(p,mac);
         end;
      end; { InLine }

   begin { CreateLadder }
   { first find lowerleft (pll) and upperright (pur) points }

   pll.x:=minofreals(l1^.lp1.x,l1^.lp2.x);
   pll.y:=minofreals(l1^.lp1.y,l1^.lp2.y);
   pur.x:=maxofreals(l1^.lp1.x,l1^.lp2.x);
   pur.y:=maxofreals(l1^.lp1.y,l1^.lp2.y);
   Setp(l2);
   Setp(l3);
   Setp(l4);
   Setp(l5);

   {
       Now find connects. There are 2 (exact) connects on
       a ladder. The leftmost is con1 and the rightmost con2  }

   if Left(l3,l4) then
      begin
      if l3^.Lp1.x < l3^.lp2.x then
         con1:=l3^.Lp1
      else
         con1:=l3^.Lp2;
      if l4^.Lp1.x > l4^.Lp2.x then
         con2:=l4^.lp1
      else
         con2:=l4^.lp2;
      end
   else
      begin
      if l4^.Lp1.x < l4^.lp2.x then
         con1:=l4^.Lp1
      else
         con1:=l4^.Lp2;
      if l3^.Lp1.x > l3^.Lp2.x then
         con2:=l3^.lp1
      else
         con2:=l3^.lp2;
      end;


   LD:=CreateLDItem(LDMacro);

   Subtractpoint(pur,pll,LD^.size);  { macro's size }
   InLine(l1,LD);                    { Puts line in macro }
   InLine(l2,LD);
   InLine(l3,LD);
   InLine(l4,LD);
```

```
   if 15 <> nil then
      InLine(15,LD);

   LD↑.conn:=CreateConnect;                { Now put in connects in macro }
   LD↑.conn↑.p:=con1;
   LD↑.conn↑.next:=CreateConnect;
   LD↑.conn↑.next↑.p:=con2;

   LD↑.place:=pll;
   Into(LD,list);
   end; { CreateLadder }


function FindParallellvert(Akt,list:pLDItem):pLDItem};
   var p     : pLDItem;
       found : boolean;
       l     : real;
   begin
   if Akt = nil then
      FindParallellvert:=nil
   else if Akt↑.LDIt <> LDLine then
      FindParallellvert:=nil
   else if Akt↑.lp1.x <> Akt↑.lp2.x then
      FindParallellvert:=nil
   else
      with Akt↑ do
         begin
         l:=abs(Lp1.y-Lp2.y);
         p:=Follow(Akt,list);
         found:=false;
         while (p<>nil) and not found do
            begin
            found:=(p↑.LDIt = LDLine) and
                   (p↑.Lp1.x=p↑.Lp2.x) and { Vertikal }
                   Intervall(p↑.Lp1.x,Lp1.x,l*0.5) and

                   (Intervall(p↑.Lp1.y,Lp1.y,l*0.2) and
                   Intervall(p↑.Lp2.y,Lp2.y,l*0.2) or       { Parallell}
                   Intervall(p↑.Lp1.y,Lp2.y,l*0.2) and
                   Intervall(p↑.Lp2.y,Lp1.y,l*0.2) );
            if not found then
               p:=Follow(p,list);
            end;
         if found then
            FindParallellvert:=p
         else
            FindParallellvert:=nil;
         end;
   end { FindParallellvert };


procedure FindHorizontell(V1,V2:pLDItem;var H1,H2:pLDItem;list:pLDItem};
   var Akt               : pLDItem;
       ymed,xhigh,xlow,l : real;
   begin
   H1:=nil;
   H2:=nil;
   ymed:=(V1↑.Lp1.y + V1↑.Lp2.y+
```



```
          V2↑.Lp1.y + V2↑.Lp2.y) / 4;            { medium of y-koords   }

   l:=( LDLineLength(V1)+LDLineLength(V2) )/2;    { length of vert lines }
   Akt:=First(list);
   xlow :=minofreals(V1↑.Lp1.x,V2↑.Lp1.x);
   xhigh:=maxofreals(V1↑.Lp1.x,V2↑.Lp1.x);
   while (Akt <> nil) and ((H1 = nil) or (H2 = nil)) do
      begin
      if Akt↑.LDIt = LDLine then
         begin
         if H1 = nil then      { look for left part }
            if (maxofreals(Akt↑.Lp1.x,Akt↑.Lp2.x) = xlow ) and
               Intervall(Akt↑.Lp1.y,ymed,l*0.2) and     { "horizontell" }
               Intervall(Akt↑.Lp2.y,ymed,l*0.2) then
                  H1:=Akt;
         if H2 = nil then      { look for right part }
            if (minofreals(Akt↑.Lp1.x,Akt↑.Lp2.x) = xhigh ) and
               Intervall(Akt↑.Lp1.y,ymed,l*0.2) and    { "horizontell" }
               Intervall(Akt↑.Lp2.y,ymed,l*0.2) then
                  H2:=Akt;

         end;
         Akt:=Follow(Akt,list);
      end;
   end { FindHorizontell };

function Intervall(a,b,c:real):boolean};
   { returns true if a in b +/- c }
   begin
   Intervall:=(a > (b-c) )and(a < (b+c) );
   end;


function ConnectedToTree
      ((side : char; p : point; list : pLDItem; tree : pLDTree;
                                                 var node : pLDTree):boolean};
   var OK : boolean;
       pl : point;
   begin
   if tree = nil then
      begin
      node:=nil;
      ConnectedToTree:=false;
      end
   else if tree↑.id = LDLeaf then
      begin
      ConnectPoint(side,Tree↑.pict,pl);
      if InContact(list,p,pl) then
         begin
         node:=tree;
         ConnectedToTree:=true;
         end
      else
         begin
         node:=nil;
         ConnectedToTree:=false;
         end;
      end
```

```
   else
      begin
      OK:=false;
      OK:=ConnectedToTree(side,p,list,tree↑.left,node);
      if not OK then
         OK:=ConnectedToTree(side,p,list,tree↑.right,node);
      if not OK then
         node:=nil;
      ConnectedToTree:=OK;
      end;
   end ; { ConnectedToTree }



function CheckContact(list : pLDItem; p1, p2 : point; mark : boolean)
                      : boolean;
   { ===== Used by MarkAsUsed and InConntact  ====== }

   function HC(p, lastp, goal : point):boolean;
      var Akt  : PLDItem;
          OK   : boolean;
          Locp : point;
      begin
      Akt:=First(list);
      OK:=false;
      while (Akt <> nil) and not OK do
         begin
         if Akt↑.LDIt = LDLine then
            OK:=EqvPoint(p,Akt↑.Lp1) or EqvPoint(p,Akt↑.lp2);
                     { Actual line is connected to the search-point }
         if OK then
            begin
            if EqvPoint(p,Akt↑.Lp1) then
               Locp:=Akt↑.Lp2
            else
               Locp:=Akt↑.Lp1;
            if not EqvPoint(Locp,goal) then
               if EqvPoint(Locp,lastp) then
                  OK:=false
               else
                  OK:=HC(locp,p,goal);
            end;
         if not OK then
            Akt:=Follow(Akt,list)
         else
            begin
            if mark then
               Akt↑.used:=true;
            end;
         end;
      HC:=OK;
      end; { HC }
   begin
   if EqvPoint(p1,p2) then
      CheckContact:=true
   else
      CheckContact:=HC(p1,p1,p2);
   end; { CheckContact }
```



```
function InContact((list : pLDItem; p1, p2 : point):boolean);
   begin
   InContact:=CheckContact(list,p1,p2,false);
   end;

procedure MarkAsUsed(list : pLDItem; p1,p2 : point);
   begin
   if not CheckContact(list,p1,p2,true) then
      writeln(chr(7),'Markasused found no way !');
   end;




function ConvertToExpr((tree : pLDTree):pexpr);
   var res, ex1, ex2 : pexpr;
   begin
   if tree = nil then
      res:=nil
   else
      with tree↑ do
         case id of
            LDAnd,
            LDOr    : begin
                      res:=MakeExpr(exprbinary);
                      ex1:=ConvertToExpr(left);
                      ex2:=ConvertToExpr(right);
                      if id = LDAnd then
                         setexprbinary(res,binandop,ex1,ex2)
                      else
                         setexprbinary(res,binorop,ex1,ex2);
                      end;
            LDLeaf:   res:=CopyExpr(ex);
            Otherwise begin
                      writeln
                        ('ConvertToExpr called with illegal type ',id);
                      end;
            end; { case & with & if }
   ConvertToExpr:=res;
   end; { ConvertToExpr }


.END
.VAR
TreeList : pLDTree;
.FORWARD
procedure LadderAnalys(list : pLDItem; var ex : pexpr);forward;
function Cardinal(p:pLDItem):integer;                    forward;

.PROCEDURE
function Cardinal((p:pLDItem):integer);
   var help : pLDItem;
       res  : integer;
   begin
```

```
   help:=First(p);
   res:=0;
   while help <> nil do
      begin
      res:=res+1;
      help:=Follow(help,p);
      end;
   Cardinal:=res;
   end; { Cardinal }

procedure LadderAnalys(list : pLDItem; var ex : pexpr);
   var ConnectList, t,
       txt,
       LadderList, TextList  : pLDItem;
       rigthmost             : real;
       helptree, tree        : pLDTree;
       i, nextnr             : integer;
       ex1, ex2              : pexpr;


   { LadderAnalys } function Rigthmostconnect(t:pLDTree):real;
      var Akt : pLDTree;
          res : real;
      begin
      Akt:=t;
      if Akt <> nil then
         begin
         res:=Akt^.cpright.x;
         while Akt <> nil do
            begin
            if Akt^.cpright.x>res then
               res:=Akt^.cpright.x;
            Akt:=Akt^.next;
            end;
         end;
      RigthMostConnect:=res;
      end; { RigthMostConnect }

   { LadderAnalys } procedure FindText(TextList : pLDItem);
      var t      : pLDItem;
          tree   : pLDTree;
          nextnr : integer;

      { FindText } function FindIt(l : pLDTree):pLDItem;
         var Akt   : pLDItem;
             found : boolean;
            begin
            Akt:=First(TextList);
            found:=false;
            while not found and (Akt <> nil) do
               with l^.pict^ do
                  begin
                  if Intervall(place.y + Size.y, Akt^.LowerLeft.y,
                               0.25 * Size.y) then
                     { Correct y-level. Is then text placed over       }
                     { the ladder ?                                    }
                     { It must be placed either                        }
                     { 1 : One x-koordinate inside the ladder's x-koord}
```



```
                     { 2 : Both text-x-limits outside the ladders x-   }
                     {     limits                                      }
                     found:=( Akt^.LowerLeft.x >= place.x) and
                            ( Akt^.LowerLeft.x <= (place.x + Size.x)) or
                            (Akt^.UpperRight.x >= place.x) and
                            (Akt^.UpperRight.x <= (place.x + Size.x)) or
                            ( Akt^.LowerLeft.x <= place.x) and
                            (Akt^.UpperRight.x >= (place.x + Size.x));
                  if not found then
                     Akt:=Follow(Akt,TextList);
                  end;
            if found then
               FindIt:=Akt
            else
               FindIt:=nil;
            end;

         begin
         nextnr:=1;
         tree:=treelist;
         while tree <> nil do
            begin
            if tree^.id = LDLeaf then
               begin
               t:=FindIt(tree);
               if t <> nil then
                  begin
                  Out(t);
                  tree^.extext:=t^.txt;
                  tree^.extextlength:=t^.length;
                  dispose(t);
                  end
               else
                  begin
                  tree^.extext[1]:='N';
                  tree^.extext[2]:='o';
                  tree^.extext[3]:='T';
                  tree^.extext[4]:='e';
                  tree^.extext[5]:='x';
                  tree^.extext[6]:='t';
                  tree^.extext[7]:=chr((nextnr div 10)+48);
                  tree^.extext[8]:=chr((nextnr mod 10)+48);
                  tree^.extextlength:=8;
                  nextnr:=nextnr+1;
                  end;
               end;
            tree:=FollowTree(tree);
            end; { while }

         end; { FindText }

      { LadderAnalys } procedure ParseTree(t : pLDTree);
         begin
         if t <> nil then
            with t^ do
               case id of
                  LDLeaf:    begin
                             ScanData.ScanImage:=extext;
```

```
                        ScanData.length:=extextlength;
                        ScanData.pos:=1;
                        ex:=expression;
                        end;
                Otherwise  begin
                        ParseTree(left);
                        ParseTree(right);
                        end;
            end; { case, with & if }
    end; { ParseTree }

{ LadderAnalys } function CountAllUnused(l : pLDItem):integer;
    var Akt : pLDItem;
        res : integer;
    begin
    Akt:=first(l);
    res:=0;
    while (Akt <> nil) do
       begin
       if (Akt^.LDIt = LDLine) and not Akt^.used then
          res:=res+1;
       Akt:=Follow(Akt,l);
       end;
    CountAllUnused:=res;
    end; { CountAllUnused }

{ LadderAnalys } procedure SetListUnused(l : pLDItem);
    { Sets the flag 'used' to false for all lines in the list l }
    var Akt : pLDItem;
    begin
    Akt:=First(l);
    while Akt <> nil do
       begin
       if Akt^.LDIt = LDLine then
          Akt^.used:=false;
       Akt:=Follow(Akt,l);
       end;
    end; { SetListUnused }

{ LadderAnalys } procedure CreateTextList(inlist : pLDItem;
                                          var textlist : pLDItem);
    var Akt, Next : pLDItem;
    begin
    textlist:=CreateLDItem(LDHead);
    Akt:=first(inlist);
    while Akt <> nil do
       begin
       Next:=Follow(Akt, inlist);
       if Akt^.LDIt = LDText then
          begin
          Out(Akt);
          Into(Akt, TextList);
          end;
       Akt:=Next;
       end;
    end; { CreateTextList }

{ LadderAnalys } procedure CreateConnectList(inlist : pLDItem;
```



```
                                          var connlist : pLDItem);
    var Akt, Next : pLDItem;
    begin
    connlist:=CreateLDItem(LDHead);
    Akt:=first(inlist);
    while Akt <> nil do
       begin
       Next:=Follow(Akt, inlist);
       if Akt^.LDIt = LDLine then
          begin
          Out(Akt);
          Into(Akt, connlist);
          end;
       Akt:=Next;
       end;
    end; { Createconnlist }

begin { LadderAnalys }
writeln('Entering LadderAnalys');
CreateTextList(list, TextList);
LDAnalys(list, LadderList);              { creating the ladders }
if First(list) <> nil then
   begin
   writeln
      ('List is not empty after creation of ladders and connections');
   end
else
   writeln('List is empty after LDAnalys');

CreateConnectList(Ladderlist, ConnectList);
   { Moving lines not in ladders to connections list. }
SetListUnused(ConnectList);

ConvertToTreeList(LadderList);

rigthmost:=RigthMostConnect(treelist);

if First(LadderList) <> nil then
   begin
   writeln('LadderList is not empty after ConvertToTreeList');
   end;
FindText(TextList);

BuildLadderExpression(ConnectList);

i:=0;
helptree:=treelist;
while helptree <> nil do
   begin
   i:=i+1;
   helptree:=FollowTree(helptree);
   end;

writeln(i:1,' expression(s) detected !');

if treelist <> nil then
   begin
   ParseTree(treelist);
```

```
      ex1:=ConvertToExpr(treelist);
      end
   else
      ex1:=nil;


   if First(TextList) <> nil then
      begin
      txt:=First(TextList);
      if txt^.UpperRight.x > rigthmost then
         begin
         Out(txt);
         ex2:=MakeExpr(exprvariable);
         for i:=1 to idlength do
            ex2^.varid[i]:=txt^.txt[i];
         ex:=MakeExpr(exprbinary);
         setexprbinary(ex,binequalop,ex2,ex1);
         dispose(txt);
         end
      else
         ex:=ex1;
      end
   else
      ex:=ex1;

   if Cardinal(TextList) <> 0 then
      begin
      writeln('All texts is not used !');
      t:=First(TextList);
      while t <> nil do
         begin
         writeln(t^.txt);
         t:=Follow(t,TextList);
         end;
      end;
    end;
.END
```



```
.FORWARD

function ConvertFromLICS(l : link):pLDItem;forward;

.PROCEDURE
function ConvertFromLICS{(l : link):pLDItem};
   { Convert a list from the LICS-program to the format used in Ladder. }
   { Supports Line and Text-type of records.                             }
   { Author Magnus Taube }
   var res, temp : pLDItem;
       lp        : link;
       i         : integer;
       str       : linetype;
   begin
   res:=CreateLDItem(LDHead);
   lp := l^.next;
   while lp^.itemtype <> noitem do
      begin
      with lp^ do
         case lp^.itemtype of
            LineItem : begin
                       temp:=CreateLDItem(LDLine);
                       with temp^ do
                          begin
                          lp1.x:=xl;
                          lp1.y:=yb;
                          lp2.x:=xr;
                          lp2.y:=yt;
                          end;
                       Into(temp,res);
                       end;
            TextItem:  begin
                       temp:=CreateLDItem(LDText);
                       with temp^ do
                          begin
                          LowerLeft.x:=xl;
                          LowerLeft.y:=yb;
                          UpperRight.x:=xr;
                          UpperRight.y:=yt;
                          length:=textobj.texthead^.next^.length;
                          str   :=textobj.texthead^.next^.text;
                          for i:=1 to length do
                             txt[i]:=str[i];
                          Into(temp,res);
                          end;
                       end;
            Otherwise  ; { No action }
            end; { case }
      lp:=lp^.next;
      end; { while }
   ConvertFromLICS:=res;
   end; { ConvertFromLICS }
.END
```

```
.END
LICS types

identifier = packed array [1..idlength] of char;
linetype = packed array [1..linelength] of char;
lineptr = ↑linerec;

linerec = record
  next, prec: lineptr;
  lineid, length: integer;
  text: linetype;
  end;

texttype = record
  textid: integer;
  xcharsize, ycharsize: real;
  nrlines: integer;
  texthead: lineptr;
  fnamelength: integer;
  writeflag: boolean;
  filename: linetype;
  end;

{ ============ Item types ============== }

link = ↑item;
normsel = (norm, selected);
itemkind = (noitem, lineitem, textitem);

item = record
  next, prec: link;
  xl, xr, yb, yt: real;
  color: integer;
  stat: normsel;
  case itemtype: itemkind of
    lineitem: ( );
    textitem: (textobj: texttype);
  end;

.FORWARD

procedure PictCompile2(var normal: link); forward;


.PROCEDURE

procedure PictCompile2((var normal: link));
{ Replaces dummy package with same name in LICS   }
   var nylist : pLDItem;
       ex : pexpr;
   begin
   writeln('Entering PictComp2');
   nylist:=ConvertFromLICS(normal);
   LadderAnalys(nylist,ex);
```



```
   removechinfo(ex);
   writeinfix(ex);
   end;

.END
```