DM 18 projekt - Maj 2005

Tony

Rapport fra

Group	13

Name	Bjørk Boye Busch
Birthday	05-04-1957
IMADA login	bjbu
Email	bjbu@tietgen.dk

Indhold

Forord

Denne rapport omhandler et compilerprojekt hørende til kurset DM18 på imada, sdu.
Jeg har fuldtidsarbejde og tager dette kursus på en tompladsordning.
Jeg har valgt (og fået accept) at udarbejde projektet alene, idet jeg ikke fra starten var sikker på at kunne få tid til at gennemføre, samt at jeg har været nødt til at arbejde med projektet i mine ledige stunder, hvilket ikke ville være tilfredsstillende vilkår for andre medlemmer af en gruppe.

1 Indledning

Værktøjer

De væsentligste dele har været udviklet på en windows-maskine med XP, henholdsvis hjemme og på mit arbejde.
Der har været brugt forskellige editorer Notepad, ConText og endelig editoren fra Microsoft Visual Studio, som jeg normal anvender i forbindelse med mit job. Hertil kommer gcc og de værktøjer der følger med gnu-pakken, som jeg har hentet ned til en windows maskine.
Jeg har desuden til aftestning anvendt et lille program xmlnotepad til windows fra Micro Soft. Programmet giver en god mulighed for at se træ-strukturen i en fil med XML struktur.
Rapporten er lavet med Microsoft Word og Acrobat Writer.
Den endelige test er foretaget på IMADA's linux-maskiner ved fjernopkobling.

Jeg har ud over ovenstående selv lavet et lille program ved brug af FLEX, der kan omsætte dele af en windows-bat fil til linux format, herunder ændring af format for linieskift. Jeg lavede dette for ikke at skulle lave en ekstra linux udgave. Dette lille program sammen med en linux komandofil **runbat** giver mig mulighed for, at bruge mine bat-filer på linux. På min windows har jeg tilsvarende en RM og MV for at slette og rename, så jeg kan bruge samme make-file på windows og linux platformen. Jeg har desuden lavet et lille hjælpeværktøj i Flex som kan sætte linienr på tekst-filer.

1.1 Afklaringer

Grammatikken beskriver ikke hele forståelsen af sproget og det har derfor været nødvendigt, at få fortaget en del præciseringer, der ikke fremgår at grammatikken.

Til denne tolkning ligger hvilke konstruktioner der er legale - om typer, funktioner og variable er defineret korrekt og sammen med de faste termer (num, true, false og null) bruges korrekt typemæssigt i de forskellige expression og statements.
Det kunne f.eks gælde udtrykket ” a < b < c ”, som i grammatikken f.eks. både kan tolkes som ” (a < b) < c ” og ” a < (b < c) ”. Det er dog ikke det eneste problem, idet udtrykket ikke beskriver hvad ”< ” betyder. I nogle situationer betyder < redirektion, men i programmeringssprog betyder ”<” normalt mindre end. Det vil selvfølgeligt være fornuftigt at fastholde den normale betydning, men hvad kan så sammenlignes?
Hvis a er af type int og b er type bool, kan man så sammenligne disse? Det kan man f.eks. i C, hvor en bool kan opfattes som en int, der har værdien 0 for false og 1 for true. En sådan tolkning kunne også tillade det beskrevne udtryk, hvor ” a < b ” kunne give 0 for falsk og 1 for sand og således stadig kunne give mulighed for at måle mod c, hvis det var en int eller bool. Men hvad hvis nu a var et array af int med et element og b en int?

Jeg vil her komme ind på nogle af de tolkninger, som jeg har lavet.

Præcisering af grammatikken, så den bliver entydig

Expression og præcedensregler

Problemet med udtrykket a< b < c der kan tolkes på to måder, skyldes at der kan laves 2 syntakstræer, ved at udledning enten til venste eller til højre. Her vil en parsing fra venste med lookahead på et symbol ”LR(1)” give en entydighed, idet der så væges en højremest udledning, hvilket vil give ”nedstigning” til venstre.
Denne tolkning giver (a<b) < c, hvis det ellers giver mening semantisk.
Tolkningen vil for et udtryk 4 + 5 +6 vil det blive tolket som (4 + 5) + 6. Hvis vi regner med at det er et almindeligt aritmetisk udtryk passer det fint, da operanterne er associative.
Tager vi i midlertidig udtrykket 4 + 5 * 6, vil en venstemest tolkning give (4 + 5) * 6, hvilket ikke svarer til den normale tolkning at regneudtrykket være 4 + (5 * 6), da * har højere præcedens end +. Samme problemstilling gør sig gældende for de øvrige aritmetiske operatorer.
Jeg har valgt at følge de normale præcedens-regler, og at der med samme præcedense skal vælges venstremest.
Præcendens regler, idet monadiske går forud for dyadiske:

(1) !
(2) * /
(3) + -
(4) < > <= =>
(5) == !=
(6) &&
(7) ||

Negationen (!) er løst i gramatikken, men de øvrige er ikke løst.
Løsningen kan foregå i grammatikken gennem en omskrivning (faktorisering).
Tager vi udtrykket som oftest bruges som eksempel i litteraturen:

S -> E
E -> E+T | E–T | E*T | E/T | T
T -> id | num | (E)

Kan det omskives til:

S -> E
E -> TE’
E’ -> +TE’ | -TE’
T -> F T’
T’ -> *FT’ | /FT’ | ε
F -> id | num | (E)

Teknikken er lettest ved at starte bottom up, med højeste præcedense først og så fortsætte med næste. Hvis sammenligningsoperatorerne < > <= => skulle med kunne de tilføjes med følgende tilføjelser i grammatikken:

Q -> EQ
Q’ -> <EQ’ | >EQ’ | <=EQ’ | >=EQ’

Samt ændre startsymbolet til:

S -> Q

Der vil altså ske en udvidelse af grammatikken med 2 nonterminaler for hvert ekstra præcedense niveau. Det bliver i midlertidig ikke nødvendig med en sådan omskrivning af grammatikken, da Bison tilbyder muligheder for at beskrive dette mere enkelt.

If–else-if problemet

Der er yderlig et problem i tony-grammatikken, som skal løses. Det handler om, at grammatikken heller ikke er entydig med hensyn til hvordan, det skal tolkes, når der ikke er lige mange if og else.
Hvordan skal f.eks. if E then if E’ then S else S’ tolkes.
Det kan tolkes på følgende 2 måder:

```
if E then {
   if E’ then S
   else S’
}
```

```
if E then {
   if E’ then S
} else S’
```

Den almindelig brugte tolkning er den første, hvor else ”bindes” til den nærmeste if. Jeg vil også foretage denne tolkning.
Løsningen kan her ligeledes være en omskrivning af grammatikken, hvor der skelnes mellem matchede og unmatchede if sætninger, og hermed kan vælges en tolkning.

En venstemest nedstigning (højremest udledning) med lookahead på en (LR(1)), vil dog umiddelbart pege på den første, som ønsket og Bison vil også vælge denne, da den virker efter ovenstående princip, så en omskrivning af grammatikken bliver ikke nødvendig.

Præcisering af øvrig syntaks.

Det er ikke nok at grammatikken kan tolkes entydigt, hvilket kan illustreres med følgende udtryk:

```
var x: int; y = x;
```

Grammatikken giver her lov til at bruge en variabel uden at den er erklæret. Grammatikken giver også lov til at regne på en record. Vi kan ikke beskrive disse regler med en kontekstfri grammatik, så der må andre redskaber til for, at udtrykke disse ting.
For at beskrive denne del af syntaksen anvender vi typereglerne og en fase, hvor der typecheckes.
Jeg har på det punkt behov for megen afklaring og præcisering. Dette er beskrevet under afsnit 4.2, hvorfor jeg vil henvise til dette afsnit for uddybning.

Præcisering af semantikken.

Når også typereglerne er fastlagt kan vi sige at sprogets syntaks beskrevet, men der rejser sig så et nyt spørgsmål.
Vi kan f.eks. tage følgende udtryk:

```
var x: int; x = x + 10;
```

Udtrykket vil her (med typereglerne) være syntaktisk korrekt, men hvad betyder det? Hvad er meningen? Er det en ligning med en ubekendt der skal løses eller et assign statement.
Hvis jeg skulle præcisere dette fra bunden, ville det være en uoverkommelig opgave, men det behøver jeg ikke, idet jeg vil tillade mig generelt at henvise til den tolkning, der normalt ligger i programmeringssprogene.

Jeg har kun i begrænset omfang brug for en letter præcision, enkelte steder. En del af præcisionen er indirekte beskrevet i mine typeregler, som netop underforstået bruger begreber som i C, Pascal, Java m.fl.

Jeg har præciseret min udvidelse af en for-konstruktion under afsnit 1.3 om udvidelser. Dertil kommer noget om bool i næste afsnit 1.2 om begrænsninger. Endelig vil jeg henvise til beskrivelserne om mine typeregler i afsnit 4.2

1.2 Begrænsninger

Jeg har valgt, ikke at lave begrænsninger i tony-sproget i forhold til oplæget.
Jeg har valg, at lade en del af tolkningerne være de samme som i C, java, pascal m.fl. med de begrænsninger, der heraf følger.

Jeg har således valgt at tolke && og || som bolske operatorer og ikke bitvise. Hermed kan de kun bruges på bool-typen og ikke int. Det vil dog være meget let, at ændre i min compiler, så der tillades bitvise operationer på int-typen. Det eneste der skal ændres, er at slække på mit typecheck, så int typen tillades. Måden jeg har realiseret resten af compileren på kan allerede håndtere bitvis and og or på typen int.

En anden begrænsning, som heller ikke er grammatisk, er at jeg kun tillader, at der er reference til typer defineret senere i koden, hvis disse ikke er en ny type. Denne begrænsning vil dog ikke have nogen praktisk betydning, men gør at jeg lettere kan sikre mod cykliske typereferencer.
Jeg kan håndtere typer af array og record med reference til andre typer og for record til sig selv, så der kan laves linkede lister m.m, hvilket også vil kunne ses dokumenteret i mine bilag.

Eksempel på tilladt declaration:

```
type tal = heltal;
type heltal = int;
```

Eksempel på ikke tilladt declaration:

```
type tal = heltal;
type heltal = etheltal;
type etheltal = int;
```

1.3 Udvidelser

Udvidelser af sproget

Der blev i projektformuleringen stillet krav om, at der skulle laves mindst en sproglig udvidelse.
Jeg har valgt at lave 2 typer udvidelser. Denne ene er af typen syntaktisk sukker, og den anden en egentlig ny sprogkonstruktion. Jeg har af tidsmæssige årsager valgt ikke at beskrive og implementere yderligere udvidelser af sproget.

Syntaktisk sukker

Jeg har valgt nogle meget simple udvidelser, der ikke har krævet andet end et par ændringer i Bison grammatikken med action, der kun brugte allerede skrevne AST strukturer og funktioner.

Det drejer sig om følgende:

- minus foran konstanter
- minus foran variable
- += i assignment som i C, java m.fl
- -= i assignment som i C, java m.fl

Begrundelsen er, at disse udvidelser kan gøre det lidt lettere / hurtigere at skrive kode, og de samtidig er så nemme at implementere, at de er oplagte udvidelser.
Jeg kunne også have medtaget *= og /= i assignment, men jeg har først tænkt på disse nu ved rapportskrivningen og efter mine bilag er udarbejdet. De vil dog meget let kunne implementeres, ved at kopier en af de andre og rette operationen til henholdsvis * og /.

For-loop-konstruktion

Jeg har valgt at udvide med en for-loop-konstruktion.

- for i to n do
- for i downto n do

Min begrundelse er her, at der er ofte i programmer brug for, at kunne lave et loop, som kører et bestemt antal gange. Konstruktionen er derfor oftest indbygget i programmeringssprogene.
Jeg har tænkt mig den skal virke på samme måde som den gjorde i pascal, hvor loop-variablen var en almindelig variabel, men hvor den ved gennemløb blev påtvunget næste værdi uanset om den blev ændret i loopet, ligesom stopværdien også blev fastholdt som beregnet med starten, uanset om en rekalkulation ved senere gennemløb vil give et andet resultat.
Man kan således ikke ”bryde” ud af et sådan loop. Konstruktionen giver mulighed for at lave et mere effektivt loop med disse begrænsninger.

Overvejelser der ikke er taget med i sproget

For en praktisk brug af sproget, ville det have været mere relevant med en indlæsningsrutine, men jeg har vurderet, at der er flere udfordringer i for-konstruktionen, hvorfor jeg har valgt denne.

Jeg har haft et par andre overvejelser til sprogudvidelse, men måtte begrænse mig på grund at tiden.

Initiering af variable sammen med declarationen.

Jeg havde også en overvejelse om, at udvide med muligheden for, at erklære variable med en startværdi. Jeg havde gjort lidt forarbejde, idet jeg på et tidspunkt initierede variable med 0. Jeg droppede det igen på et tidspunkt, da jeg ikke havde beskrevet det. Jeg har ikke overvejet det igen før nu, da der har været så mange andre mere interessante ting, at tage fat på.

Jeg kunne grammatiks løse initiering af variable med følgende udvidelse af tony:

```
<Var_type>          ……
                    | id : <Type>
                    | id : <Type> = <Expression>
```

Der skulle samtidig præciseres, at de allerede gældende præcision af typer for assignment, samt at det skulle sikres at Expression ikke indhold referencer til <Var_type>'er declareret senere og til sig selv.

Nedarvning på typer.

Jeg havde også en overvejelse om, at udvide med muligheden for at nedarve på en record-type.
Dette kunne også være en nyttig konstruktion, idet der så kunne laves mere generisk kode.

Jeg kunne grammatiks løse arv på record med følgende udvidelse af tony:

```
<Type>              ……
                    | extends id with { <Var_decl_list> }
```

Denne idé blev heller ikke til mere, da jeg ikke kunne nå at lave mere end de allerede valgte udvidelser.

Udvidelse med run-time sikkerhed

Der blev i projektformuleringen stillet krav om, at der skulle laves mindst et element af run-time sikkerhed.

Jeg har lavet følgende udvidelser:

- check af array index for under- og overløb
- check for division med nul
- check for positivt antal elementer ved array allokering
- check for out-of-memory på heap

Jeg fravalgte også, at nå den sidst foreslåede med, at checke for manglende initiering og null-pointere, fordi det først krævede en form for præcisering, og jeg havde ikke mere tid til denne del.

Udvidelse med avanceret element

Der blev i projektformuleringen stillet krav om, at der skulle laves mindst et element af denne type.

Peep-hole optimering valgt

Jeg har valgt at lave peep-hole optimering.

Når jeg har valg at fokusere på peep-hole er det dels fordi jeg synes det ville være interessant at se hvor meget der kunne optimeres, hvis jeg ikke gjorde det store ud af at gøre det på forhånd.

En fordel rent projektmæssigt er, at det var mere sikkert, idet jeg har genereret kode, der direkte kunne bruges til emit, og som ikke har binding til det abstrakte syntakstræ. Jeg har derfor kunnet sikre mig, at jeg kunne udvide denne del efter, at jeg var sikker på min compiler virkede, som den skulle.

De andre foreslåede udvidelser kunne ikke på samme måde udvikles incrementelt.

Andre overvejelser om heap og garbage collection

Jeg har også gjort nogle overvejelser om organisering af heapen med henblik på garbage collection.
Overvejelserne omkring organisering af heap og garbage collection er ikke de tre metoder, vi har fået beskrevet, men en anden løsning, som jeg vil komme ind på her.

Da jeg ikke kunne nå mere, valgte jeg ikke at påbegynde justeringer i brugen af min heap, men det er ikke så store justeringer der i første runde skulle til for senere at kunne udbygge med garbage collection.

Jeg kunne også have forestillet mig at den sidste del ville jeg kunne skrive direkte i assembler som en rutine der så kunne linkes ind.

Foreslag til heap strukturen med henblik på garbage collection

Som jeg organiserer heapen nu, tildeles en memoryblok og jeg gemmer lige forud størrelsen på den allokerede blok. Jeg kunne her også gemme en counter som man gør i den simple løsning til garbage collection, men herudover også en adresse på en nedlægningsrutine (destruktor) der svarer til den pågældende struktur.
Denne løsning kunne sikre at der ikke blev blokke, som blev ”hægtet” fra uden referencer, idet ”destructor” metoden kunne kalde alle referencer på datablokken for sletning, før datablokken selv blev nedlagt.

Tilbage står stadig problemet med fragmentering, men det kan løses ved, at der i stedet for at gemme en reference direkte til data på heapen oprettes en reference til en pointer placeret i en særlig blok til administration af heapen. Tilgang til data på heapen må så ske gennem denne indirekte tilgang. Hvis der flyttes rundt med data på heapen, skal der nu kun rettes i referencen i denne blok.

Da alle referencer fylder det samme kan en nedlagt reference let genbruges. I forbindelse med garbage collection, kan man let finde ikke brugte data, da det så ikke vil være en reference til den i denne særlige reference-blok.

Denne løsning sammen med ”destruktor” løsningen kan give en let garbage administration.

Den eneste ulempe er, at der nu må et ekstra opslag til for, at få adgang til data.

1.4 Implementationsstatus

Alle de beskrevne elementer af mit tony-sprog er blevet fuld ud implementeret, afprøvet og virker, hvilket fremgår af min testdokumentation.

2 Parsing og abstrakte syntakstræer

Parsing og opbygning sker gennem brugen af 3 værktøjer i samspil.

- Flex, som bruges til at opsamle tokens og overgiver disse til Bison.
- Bison, som parser i forhold til grammatikken og aktiver C-funktionerne.
- Egne C-strukturer og funktioner, der opbygger noderne i det abstrakte syntakstræ.

2.1 Grammatikken

Bison grammatikken

Grammatikken med mine udvidelser fremgår af nedenstående uddrag fra Bison inputtet. Der findes desuden en oversigt i bilag A, der fremstår som den oprindelige version uden C-feltnavne.

Jeg har medtaget beskrivelserne, der beskriver de grammatiske præcedens- og associationsregler, der ikke er udtrykt i selve grammatikken under afsnit 2.3 (brug af Bison værktøjet), hvor jeg også gør rede for de ændringer, der er i forhold til den oprindelige grammatik med udvidelser.

```
ntProgram : ntBody

ntFunction: ntHead ntBody ntTail

ntHead    : tFUNC tIDENTIFIER tBEGPARANTES ntPar_decl_list tENDPARANTES
            tCOLON ntType

ntTail    : tEND tIDENTIFIER

ntType    : tIDENTIFIER
          | tINT
          | tBOOL
          | tARRAY tOF ntType
          | tRECORD tOF tBEGTUBORG ntVar_decl_list tENDTUBORG

ntPar_decl_list: ntVar_decl_list
          | /* empty */

ntVar_decl_list: ntVar_decl_list tKOMMA ntVar_type
          | ntVar_type

ntVar_type: tIDENTIFIER tCOLON ntType

ntBody    : ntDecl_list ntStatement_list

ntDecl_list: ntDecl_list ntDeclaration tSEMICOLON
          | /* empty */

ntDeclaration: tTYPE tIDENTIFIER tASSIGN ntType
          | ntFunction
          | tVAR ntVar_decl_list

ntStatement_list: ntStatement_list ntStatement
          | ntStatement

ntStatement: tRETURN ntExpression tSEMICOLON
          | tWRITE ntExpression tSEMICOLON
          | tNEW ntVariable tOF tLENGTH ntExpression tSEMICOLON
          | tNEW ntVariable tSEMICOLON
          | ntVariable tASSIGN ntExpression tSEMICOLON
          | ntVariable tADDTO ntExpression tSEMICOLON
```

```
          | ntVariable tSUBFROM ntExpression tSEMICOLON
          | tIF ntExpression tTHEN ntStatement tELSE ntStatement
          | tIF ntExpression tTHEN ntStatement
          | tWHILE ntExpression tDO ntStatement
          | tBEGTUBORG ntStatement_list tENDTUBORG

          | tFOR ntVariable tASSIGN ntExpression
            tTO ntExpression tDO ntStatement

          | tFOR ntVariable tASSIGN ntExpression
            tDOWNTO ntExpression tDO ntStatement

ntVariable: tIDENTIFIER
          | ntVariable tBEGINDEX ntExpression tENDINDEX
          | ntVariable tPUNKTUM tIDENTIFIER

ntExpression: ntTerm
          | ntExpression tMUL ntExpression
          | ntExpression tDIV ntExpression
          | ntExpression tADD ntExpression
          | ntExpression tSUB ntExpression
          | ntExpression tLT ntExpression
          | ntExpression tGT ntExpression
          | ntExpression tLTEQ ntExpression
          | ntExpression tGTEQ ntExpression
          | ntExpression tEQ ntExpression
          | ntExpression tNEQ ntExpression
          | ntExpression tAND ntExpression
          | ntExpression tOR ntExpression

ntTerm    : ntVariable
          | tSUB ntVariable
          | tIDENTIFIER tBEGPARANTES ntAct_list tENDPARANTES
          | tBEGPARANTES ntExpression tENDPARANTES
          | tNOT ntTerm
          | tNUMERIC ntExpression tNUMERIC
          | tSUB tINTCONST
          | tINTCONST
          | tTRUE
          | tFALSE
          | tNULL

ntAct_list: ntExp_list
          | /* empty */

ntExp_list: ntExpression
          | ntExp_list tKOMMA ntExpression
```

2.2 Brug af Flex værktøjet

Flex bruges til at opsamle tokens til Bison. De forskellige tokens kan beskrives som regulære udtryk, og disse kan genkendes af en endelig automat. Flex er bygget netop med henblik på at kunne fungere som en endelig automat, der kan sende besked, hver gang en del-accept-tilstand er nået. Da man ofte i sprog indlægger elementer, der ikke skal medsendes til parseren som fx kommentarer, er Flex blevet udbygget så der kan indlægges forskellige tilstands skift, og på en den måde lettere beskrive noget, der er lidt mere kompliceret.

På trods af denne facilitet, er det dog stadig som udgangspunkt en endelig automat. Der kan med tilstandsmuligheden nu let defineres at Flex skal kunne genkende et udtryk som (* en kommentar *), idet det blot kræver at der defineres en ekstra tilstand.

Kravet til vores tony var dog, at der skulle accepteres nestede kommentarer, hvilket vil kræve en ekstra tilstand for hvert niveau.

For at definere dette med ovenstående mulighed, måtte vi altså definere hvor mange niveauer der skulle være, men så kunne det også lade sig gøre at løse det med en endelig automat. I praksis ville vi sikkert sagtens kunne acceptere, at der f.eks. kun kunne være fx 10 niveauer.

Da man i Flex imidlertid også har mulighed for at indlægge C-programkode, er vi ikke mere begrænset til en endelig automat. Vi har ikke blot en push-down automat (der ellers kunne løse opgaven) men en turing-maskine. Opgaven løses derfor meget enkelt med en simpel tæller (int giver i sig selv 4 mia. tilstande).

Tokens identificeres af Flex, der generer den c-kode der buges til opsamling af disse. For at koden som Bison genererer fungerer, skal tokens der bruges i Bison også defineres i Bison. Bison tildeler hver token-variabel en entydig værdi. I Flex kan der refereres til disse ved, at inkludere en header-fil, som Bison genererer.

Da de tokens Bison bruger identificeres med et entydigt nummer (enum) må Flex returnere et sådan.

Man kan betragte char som et tal, og kunne derfor også returnere et tegn f.eks. '='.
Fordelen ved dette ville være, at grammatikken i Bison kunne skrives med direkte brug af disse specialtegn og lettere læses, men da ikke alle tokens er tegn, vil nogle skulle defineres i Bison.

Jeg har ikke undersøgt, om disse tegn evt. kunne konflikte med den nummerering af tokens Bison anvender, da jeg på forhånd har valgt at lave en token definition i Bison for alle tokens Bison skal kunne genkende.

Der er også den fordel ved, at lade Flex returnere et Bison-defineret-token frem for et tegn, idet det er lettere at ændre et symbol. Det skal så kun ske i Flex og ikke også i grammatikken i Bison.

Lige nu bruges = for assign og koden i Flex ser sådan ud:

```
"="   return tASSIGN;
```

Hvis vi vil ændre det til := kræver det blot at ovenstående ændres til:

```
":="  return tASSIGN;
```

Det er også forholdsvis let at rette I TONY, så der f.eks. ikke skelnes mellem små og store bogstaver på "reserverede ord", ved f.eks. at ændre "if" til [iI][fF].

For identifier vil det blot kræve, at yytext konverteres til upper- eller lower-case før der returneres.

Da TONY skal kunne indeholde nestede kommentar, har jeg valgt at der arbejdes i 2 Flex-tilstande, hvor alt kode i kommentar-tilstanden ignoreres.
Linienummer tælles stadig op i kommentar-tilstanden.

Jeg har valgt at tage højde for både windows/dos og unix/linux linieskift.

Her er et uddrag af tony.lex der bruges i Flex.

Uddraget omhandler fjernelse af kommentarerne.
Variablen comment-level er fra start sat til 0.

```
%x COMMENT
%%
::::::::::::::::::::::::::

<COMMENT>"\r\n"   {++linienr;} /* Windows */
<COMMENT>"\n"     {++linienr;} /* linux */
<COMMENT>[ \t]+   {}/* ignorer */
<COMMENT>"(*"     {
                    ++commentLevel;
                  }
<COMMENT>"*)"     {
                    --commentLevel;
                    if (commentLevel==0)
                      BEGIN(INITIAL);
                  }
<COMMENT><<EOF>>  {
                    yyerror("Fejl comment ikke slutet");
                  }
<COMMENT>.        {}/* ignorer  andet inde i kommentar*/
```

Det samlede input til Flex findes i bilag A.

2.3 Brug af Bison værktøjet

Bison bruges til at genkende grammatikken og kan, via action delen på en genkendt nontermial også bruges til, at opbygge et abstrakt syntakstræ, som danner grundlaget for de efterfølgende faser. Bison funger som en LR(1) parser i samspil med Flex, således at grammatikken ikke skal beskrives ned på tegn-niveau, men kun til de tokens der kan genkendes af Flex. Som jeg har været inde på tidligere under afklaringer, er det nødvendigt, at grammatikken på den ene eller anden måde fremstår entydigt. Bison rummer mulighed for at beskrive præcedens- og associationsregler, så man slipper for at beskrive en entydig grammatik manuelt. Hermed er det muligt at få en entydig grammatik, med en enkelt undtagelse (if-else), som jeg kommer ind på senere i dette afsnit.

Bemærkninger til grammatikken

Jeg har valgt at tilføje et ekstra startsymbol:

```
<Program>         : <Body>
```

Dette er mest af kosmetiske grunde. I Bison angives startsymbolet, så det behøver ikke være den førstnævnte nonterminal, men jeg får samtidig et startsted, der lettere senere kan ændres, hvis der f.eks. skulle startes med en funktion.

Jeg har valgt at lukke new- og if- konstruktionerne i <Statement>:

```
<Statement>       …….
                  | new <Variable> of length <Expression> ;
                  | new <Variable> ;
                  …….
                  | if <Expression> then <Statement> else <Statement>
                  | if <Expression> then <Statement>
                  …….
```

Der er flere begrundelser for dette.

Den ene er, at jeg har prøvet at fastholde en selvstændig struktur for hver nonterminal og en funktion for hver regel. Jeg har med ovenstående reduceret begge dele.

Den anden er, at det giver samlet en enkelt regel mindre for hvert af udtrykkene ved at lukke dem, og kun en regel mere for hver under <statement>, som stadig er let at overskue, samtidig at det bliver lettere at se hver af de 2 muligheder for new og if.

Jeg har valgt indsætte en regel for hver operator i <Statement>:

```
<Expression>      ……..
                | <Expression> * <Expression>
                | <Expression> / <Expression>
                | <Expression> + <Expression>
                | <Expression> - <Expression>
                | <Expression> < <Expression>
                | <Expression> > <Expression>
                | <Expression> <= <Expression>
                | <Expression> >= <Expression>
                | <Expression> == <Expression>
                | <Expression> != <Expression>
                | <Expression> && <Expression>
                | <Expression> || <Expression>
```

Jeg har valgt ikke at lave en ny nonterminal, da jeg som tidligere nævnt gerne ville fastholde en selvstændig struktur for hver nonterminal, og jeg hermed ville få en ekstra (jeg har dog her kun én funktion for dem i programmet).

Ud over udvidelserne i tony har jeg valg ikke at ændre yderligere i grammatikken. Jeg har således også fasthold <Act_list>, der ellers også let kunne lukkes som en ekstra regel under <Term>, samt <Par_decl_list>, der kunne lukkes som to ekstra regler under <Head>.

Præcisering af præcedens- og associationsregner med Bison

Der er som tidligere nævnt muligt at beskrive præcedens- og associationsregler i Bison, således at det ikke er nødvendigt at specificere i grammatikken manuelt.

Jeg havde defineret følgende præcedensregler:

```
(1) !
(2) * /
(3) + -
(4) < > <= =>
(5) == !=
(6) &&
(7) ||
```

Jeg ville desuden have at der ikke skulle være alm. associering, men binding fra venstre mod højre.

Dette har jeg i Bison beskrevet på følgende måde:

```
%left tOR
%left tAND
%left tEQ tNEQ              /* ønskes indbyrdes venste-mest binding */
%left tLT tGT tLTEQ tGTEQ   /* ønskes indbyrdes venste-mest binding */
%left tADD tSUB             /* ønskes indbyrdes venste-mest binding */
%left tMUL tDIV             /* ønskes indbyrdes venste-mest binding */
/* %left tNOT unødvendig her da grammatik løser det */
```

Problemet med if-else uden match

Som beskrevet i afsnit 1.1, var grammatikken heller entydig med hensyn til if-else konstruktioner, hvor der ikke var else-match på alle if.
Bison fanger denne problemstilling, hvilket ses at følgende output fra Bison.

```
State 118 conflicts: 1 shift/reduce

state 118

   30 ntStatement: tIF ntExpression tTHEN ntStatement . tELSE ntStatement
   31            | tIF ntExpression tTHEN ntStatement .

    tELSE  shift, and go to state 132

    tELSE     [reduce using rule 31 (ntStatement)]
    $default  reduce using rule 31 (ntStatement)
```

Ovenstående viser, at der er en konflikt i reglerne, når der kunne genkendes en if på stakken og en else på input og bare en if på stakken uden at se på input.

Det, der vises til venstre for punktummet, er indhold på stakken, og det der vises til højre er input.

Konsekvensen af at vælge regel 30 ville være, at der for en if-konstruktion med else først shiftes, når else er genkendt som input. Hermed ville else blive budet til den første if, hvilket ikke var den måde den skulle virke på ifølge min præcision i afsnit 1.1.

Det vi imidlertid kan se er, at Bison vælger at reducere med regel 31, hvor det er muligt (uanset else eller ej). Det betyder, at der efter if nu vil skulle genkendes et statement, der kunne være en if efterfulgt af en else.

Konsekvensen er, at else vil blive bundet til den sidste else, idet if-else nu må reduceres til et statement før første if kan reduceres.

Dette er netop den ønskede virkning, og jeg har derfor ikke behøvet at gøre andet end at acceptere Bisons valg. Det er blevet bekræftet ved test, at virkningen også er som beskrevet, at else bindes på sidste (nærmeste) if.

Ud over ovenstående, som sammen med reglerne skal få Bison til at genkende grammatikken korrekt, skal de nonterminaler, der bruges såvel i selve Bison, som de tokens, der udveksles mellem Flex og Bison, defineres.

Først og fremmest har Bison brug for at der findes en variabel til den værdi, Bison genkender som input, da returværdierne er typedefineret. Bison arbejder med lookahead på en, så de kan placeres i en union, for at spare plads.

Disse definitioner ser sådan ud (uddrag):

```
%union {
  int intconst;
  char *stringconst;
  struct sFunction *function;
  struct sHead *head;
  struct sTail *tail;
  struct sType *type;
……
```

For at Bison kan få kædet disse variable sammen med de tokens / nonterminaler / terminaler, den genkender som input, må man definere, hvilken variable der skal bruges i forbindelse med de enkelte nonterminaler. Det sker gennem følgende definitioner (uddrag):

```
%type <function> ntFunction;
%type <head> ntHead;
%type <tail> ntTail;
%type <type> ntType;
%type <par_decl_list> ntPar_decl_list;
%type <var_decl_list> ntVar_decl_list;
%type <var_type> ntVar_type;
```

Endelig skal de tokens, der udveksles mellem Flex og Bison, defineres sammen med evt. tilknyttede data. Dette sker gennem følgende definitioner (uddrag)

```
%token tANDET;
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%token  tASSIGN
```

2.4 Abstrakte syntakstræer

Strukturer til AST

Det abstrakte syntakstræ opbevares i en række strukturer, der er opbygget efter en ensartet mønster.

Jeg har fastholdt en selvstændig struktur for hver nonterminal, hvilket giver en meget ligetil løsning.

Der er i hver struktur en understruktur, som matcher lige præcis de data, der skal gemmes for den enkelte regel, idet der er en enkelt undtagelse med expression med 2 operanter, hvor der er så mange ensartede, at jeg nøjes med én enkelt og i stedet overfører operant-typen som en ekstra parameter.

Der kunne ”spares” nogle få interne strukturer, idet nogle af reglerne giver samme data, men jeg har valgt at fastholde en struktur for hver bortset fra de nævnte expressions, idet den slaviske måde giver større sikkerhed i navngivningen og hermed giver større sikkerhed mod fejl i kodningen.

Strukturerne er defineret i tonyAST.h og ser sådan ud:

```
typedef struct sProgram {
  int linienr;
  int maxScopeLevel;
  struct SymbolTable *symbolTable;
  union {
    struct { struct sBody *body; } eProgram;
  } val;
}sProgram;

typedef struct sFunction {
  int linienr;
  struct SymbolTable *symbolTable;
  union {
    struct { struct sHead *head; struct sBody *body;
             struct sTail *tail; } eFunction;
  } val;
}sFunction;
```

```
typedef struct sHead {
  int linienr;
  int labelnr;  /* for unique id */
  struct asElm *labelEntry;
  struct asElm *labelExit;
  union {
    struct { char *id; struct sPar_decl_list *par_decl_list;
         struct sType *type; int id_linienr; struct SYMBOL *symbol;} eHead;
    } val;
}sHead;

typedef struct sTail {
  int linienr;
  union {
    struct { char *id; } eTail;
  } val;
}sTail;

typedef struct sType {
  int linienr;
  struct sType *slutType;   /* sættes på i forb. med type-check */
  int dataLength;
  enum {kType_Id, kType_Int, kType_Bool, kType_Array,
        kType_Record, kType_Null} kind;
        /* sidste ikke i AST men en expression-type */
  union {
    struct { char *id; struct sType *type; int id_linienr; } eId;
    struct { struct sType *type; int elmLength; } eArray;
    struct { struct sVar_decl_list *var_decl_list;
             struct SymbolTable *symbolTable; int elmLength; } eRecord;
  } val;
}sType;

typedef struct sPar_decl_list {
  int linienr;
  int dataLength;
  enum {kPar_decl_list_List, kPar_decl_list_Empty} kind;
  union {
    struct { struct sVar_decl_list *var_decl_list; } eList;
  } val;
}sPar_decl_list;

typedef struct sVar_decl_list {
  int linienr;
  enum {kVar_decl_list_List, kVar_decl_list_Var} kind;
  union {
    struct { struct sVar_decl_list *var_decl_list;
             struct sVar_type *var_type; } eList;
    struct { struct sVar_type *var_type; } eVar;
  } val;
}sVar_decl_list;

typedef struct sVar_type {
  int linienr;
  int dataOffset;
  int scopeLevel;
  int labelnr;  /* for unique id */
  union {
    struct { char *id; struct sType *type; int id_linienr;
             struct SYMBOL *symbol;} eVar;
  } val;
}sVar_type;
```

```
typedef struct sBody {
  int linienr;
  int dataLength;
  union {
    struct { struct sDecl_list *decl_list;
             struct sStatement_list *statement_list; } eBody;
  } val;
}sBody;

typedef struct sDecl_list {
  int linienr;
  enum {kDecl_list_List, kDecl_list_Empty} kind;
  union {
    struct { struct sDecl_list *decl_list;
             struct sDeclaration *declaration; } eList;
  } val;
}sDecl_list;

typedef struct sDeclaration {
  int linienr;
  enum {kDeclaration_Type, kDeclaration_Function, kDeclaration_Var} kind;
  union {
    struct { char *id; struct sType *type; int id_linienr; bool aktiv;
             struct SYMBOL *symbol; } eType;
    struct { struct sFunction *function; } eFunction;
    struct { struct sVar_decl_list *var_decl_list; } eVar;
  } val;
}sDeclaration;

typedef struct sStatement_list {
  int linienr;
  enum {kStatement_list_List, kStatement_list_Statement} kind;
  union {
    struct { struct sStatement_list *statement_list;
             struct sStatement *statement; } eList;
    struct { struct sStatement *statement; } eStatement;
  } val;
}sStatement_list;

typedef struct sStatement {
  int linienr;
  bool aktiv;   /* kan sættes false i weed-fasen */
  enum {kStatement_Return, kStatement_Write, kStatement_NewLength,
        kStatement_New, kStatement_Assign, kStatement_IfElse,
        kStatement_If, kStatement_While, kStatement_Compound,
        kStatement_ForTo, kStatement_ForDownto} kind;
  union {
    struct { struct sExpression *expression; } eReturn;
    struct { struct sExpression *expression; } eWrite;
    struct { struct sVariable *variable;
             struct sExpression *expression; } eNewLength;
    struct { struct sVariable *variable; } eNew;
    struct { struct sVariable *variable;
             struct sExpression *expression; } eAssign;
    struct { struct sExpression *expression;
             struct sStatement *ifStatement;
             struct sStatement *elseStatement; } eIfElse;
    struct { struct sExpression *expression;
             struct sStatement *ifStatement; } eIf;
    struct { struct sExpression *expression;
             struct sStatement *statement; } eWhile;
    struct { struct sStatement_list *statement_list; } eCompound;
```

```
    struct { struct sVariable *variable;
             struct sExpression *expressionInit;
             struct sExpression *expressionTo;
             struct sStatement *statement; } eForTo;
  } val;
}sStatement;


typedef struct sVariable {
  int linienr;
  struct sType *type;       /* opsamlet ved typecheck */
  enum {kVariable_Id, kVariable_Indexed, kVariable_Struct} kind;
  union {
    struct { char *id; int id_linienr; struct sVar_type *var_type; } eId;
    struct { struct sVariable *variable;
             struct sExpression *expression; } eIndexed;
    struct { struct sVariable *variable; char *id; int id_linienr;
             struct sVar_type *var_type;} eStruct;
  } val;
}sVariable;

typedef struct sExpression {
  int linienr;
  struct sType *type; /* opsamlet ved typecheck */
  enum {kExp_Eq, kExp_Neq, kExp_Lt, kExp_Gt, kExp_Lteq, kExp_Gteq,
        kExp_Add, kExp_Sub, kExp_Mul, kExp_Div, kExp_Or, kExp_And,
        kExp_Term} kind;
  union {
    struct { struct sExpression *expressionLeft;
             struct sExpression *expressionRight; } eLeftRight;
             /* fælles for binaere - dyadiske */
    struct { struct sTerm *term; } eTerm;
  } val;
}sExpression;

typedef struct sTerm {
  int linienr;
  struct sType *type; /* opsamlet ved typecheck */
  enum {kTerm_Variable, kTerm_Call, kTerm_Parantes, kTerm_Not,
        kTerm_Numeric, kTerm_Num, kTerm_True, kTerm_False,
        kTerm_Null} kind;
  union {
    struct { struct sVariable *variable; } eVariable;
    struct { char *id; int id_linienr; struct sAct_list *act_list;
             struct sHead *head; } eCall;
    struct { struct sExpression *expression; } eParantes;
    struct { struct sTerm *term; } eNot;
    struct { struct sExpression *expression; } eNumeric;
    struct { int num; } eNum;
  } val;
}sTerm;

typedef struct sAct_list {
  int linienr;
  enum {kAct_list_List, kAct_list_Empty} kind;
  union {
    struct { struct sExp_list *exp_list; } eList;
  } val;
}sAct_list;
```

```
typedef struct sExp_list {
  int linienr;
  enum {kExp_list_Expression, kExp_list_List} kind;
  union {
    struct { struct sExpression *expression;
             struct sVar_type *var_type; } eExpression;
    struct { struct sExp_list *exp_list; struct sExpression *expression;
             struct sVar_type *var_type; } eList;
  } val;
}sExp_list;
```

Funktioner til opbygning af AST

Der findes en funktion for hver reduktionsregel med undtagelse af:

- syntaktisk sukker, hvor der netop ikke er lavet nye funktioner, men de eksisterende er brugt.
- dyadiske expression, hvor jeg kun har en funktion.

Funktionerne er declareret i tonyAST_Make.h og implementeret i tonyAST_Make.c, der begge findes i bilag A.

2.5 Afsukring

Som det fremgik i afsnit 1.3, havde jeg valgt at udvide med mulighed for minus foran konstanter og variable samt med += og -= i forbindelse med assignment.

Den første udvidelse med minus foran konstanter kunne jeg have valgt at gøre i scanneren, ved at genkende minus konstant og så returnere token med negativt fortegn.

Denne løsning ville have været lige så enkel som at gøre det i parseren, men jeg skulle så have krævet et mellemrum for at scanneren stadig skulle kunne adskille minus som operator.

Jeg har i stedet valgt at gøre det i parseren, hvor der er en ny regel, der bare sætter minus foran konstanten, som så er negativ.

Dette kommer til at se sådan ud i Bison:

```
ntTerm    :
          | tSUB tINTCONST {
            $$ = makeTerm_Num(-$2); }
```

Den anden udvidelse med minus foran en variabel er en smule mere kompliceret og skal løses af parseren. Jeg har her en regel, der skal reducere til en term fra et minus og en variabel og opbygger derfor en parentes-term, med et expression 0 – x, hvor x er variablen.

Jeg undgår herved nye elementer i grammatikken og nye strukturer og funktioner i mit AST. Udtrykket -x bliver altså til (0-x) i AST.

Dette kommer til at se sådan ud i Bison:

```
ntTerm    :
          | tSUB ntVariable {
            $$ = makeTerm_Parantes (
                  makeExpression_Dyadisk (
                  makeExpression_Term(makeTerm_Num(0)),
                  kExp_Sub,
                  makeExpression_Term(makeTerm_Variable($2))))
               }
```

De sidste udvidelser med += og -= i forbindelse med assign har jeg ligeledes løst med to ekstra regler. Her er det en anelse lettere, da der skal reduceres til et statement fra en plus/minus, et assign og et expression. Jeg løser det derfor ved at indlægge et assign-statement med et nyt expression, hvor variablen indgår som 1. operant (som term), og det gamle expression som anden operant. Endelig anvendes fortegnet som operator.

Det ser således ud i Bison:

```
ntStatement:
          | ntVariable tADDTO ntExpression tSEMICOLON {
            $$ = makeStatement_Assign($1,
                 makeExpression_Dyadisk(
                 makeExpression_Term(
                 makeTerm_Variable($1)),kExp_Add,$3)); }
```

2.6 Udlugning

Kravet var her at weed fasen som minimum skulle sikre at navn i funktionshoved og end skulle være identisk, samt at der var sikkerhed for et return statement i en funktion.
Return checket skulle være statisk, således at hvis der f.eks. var en if med et return, skulle det også sikres, at der var en return, hvis if-sætningen ikke blev udført.

Jeg har valgt ikke at checke for return i ”main” programmet (body), idet jeg her regner med at både tillade return af en int og ingen return.

Det skal således være muligt at lave en return, hvilket jeg har tænkt mig skal svare til exit(x) med x som den angivne returværdi. Hvis der ikke er nogen return i ”main” programmet har jeg tænkt mig det skal svare til exit(0).

Jeg har implementeret ovenstående fuldt ud og har desuden lavet nogle udvidelser.

Deaktivering af inaktive sætninger

Jeg har valgt at udvide yderligere ved at markere alle sætninger, der ikke kan nås på grund at et return-statement, som inaktive.

Jeg har valgt ikke at fjerne dem, men i stedet at medtage dem til typecheck.

Det er min erfaring, at hvor der er inaktive sætninger, skyldes det ofte, at der midlertidig er indlagt en return under en test. Det er derfor ofte en fordel, at få de resterende sætninger korrekte, så man bare kan slette return-sætningen, når testen er færdig, uden at der så kommer fejl.

Jeg har dokumenteret, at denne funktionalitet virker i bilagene.

Selvreferende typer

Ud over ovenstående har jeg valgt at checke, om typer refererer direkte til sig selv.

Jeg har implementeret det således, at disse typer deaktiveres, og man vil således få typefejl ved at referere til dem.

Jeg har dokumenteret, at denne funktionalitet virker i bilagene.

Andre ikke valgte muligheder

Jeg har ikke i denne fase fanget cykliske referencer, men da jeg kun tillader, at fremadrettet brug at typer er tilladt, hvis disse er af grundtyperne int, bool, array eller record, fanger jeg disse i syntaks-checket, hvor det også er lettere, idet jeg der kan bruge symboltabellen.

Skulle denne type fejl fanges i weed fasen, ville det blive en del mere kompliceret.

Ud over de valgte implementeringer kunne jeg også have valgt at reducere expressions med konstanter. Jeg har ikke valgt dette, dels da jeg ikke kan nå det hele og dels fordi der her var nogle interessante muligheder for at se på peep-hole optimeringen, hvor jeg får lavet de fleste reduktioner (undtagen for division).

Det ville have været muligt at checke for division med konstanten 0, men det har jeg fravalgt på grund af tiden. Det ville ellers være en god idet at gøre det i weed-fasen, som det er nu vil det først blive fanget på runtime-tidspunktet. Det kunne også i en vis udstrækning være fanget i forbindelse med min peep-hole optimering, men det vil dog være ikke et passende tidspunkt at fange fejl, idet de bør fanges inden ressource-fasen.

2.7 Afprøvning

Afprøvning med kommentarer som løses af Flex

I bilag B findes testdokumentation for at behandlingen af kommentarer virker, som den skal. Jeg har ikke medtaget dem her, men vil nøjes med kort at beskrive de enkelte test.

test_com1.tony

Denne test viser, at korrekt match af start og slutkommentarer accepteres, og at kode i mellem bliver ignoreret.
Test dokumentationen findes i bilag B.

test_com2.tony

Denne test viser, at manglende slutkommentar fanges.
Test dokumentationen findes i bilag B.

test_com3.tony

Denne test viser, at slut kommentar uden startkommentar fanges.
Test dokumentationen findes i bilag B.

test_com4.tony

Denne test viser, at nestede kommentar behandles korrekt.
Test dokumentationen findes i bilag B.

Afprøvning af opbygning af AST

For at teste, at grammatik kan genkendes, behøves definitionerne i Flex og Bison, men der behøves ikke action på Bison reglerne. Jeg lavede i starten en sådan test på dele af grammatikken, men jeg har ikke gemt disse test som ligeså meget var foreløbige eksperimenter.

Tilbage står så at teste brugen af Flex, Bison, AST strukturerne og de tilhørende metoder til opbygning af disse.

For at teste dette lavede jeg først en XML-pretty printer, der kunne udskrive det abstrakte syntakstræ i XML-format. Formatet giver gode muligheder for at få et grafisk overblik, idet jeg her kunne anvende et lille program xmlnotepad fra Micro Soft. Programmet er et windowsprogram og jeg ved ikke om der findes noget tilsvarende til en linux platform. Xml-

formatet kan også vises i f.eks Mirco Soft explorer, hvor der vises indrykninger, men nu skulle dokumentationen også på papir og her var udskrift af XML-formatet ikke så læsevenligt, så jeg lavede derfor også en tekstudgave. Disse 2 udskriftsrutiner er senere blevet udbygget med påsatte typer, så de viser nu ikke mere kun det abstrakte syntakstræ, som er blevet opbygget.

Mine oprindelige tony-test-programmer er stadig de samme som jeg anvendte i starten.

Jeg har ikke lavet isoleret test for opbygning af AST for min for-konstruktion, men bruger samme test som til type-check.

test_term1.tony

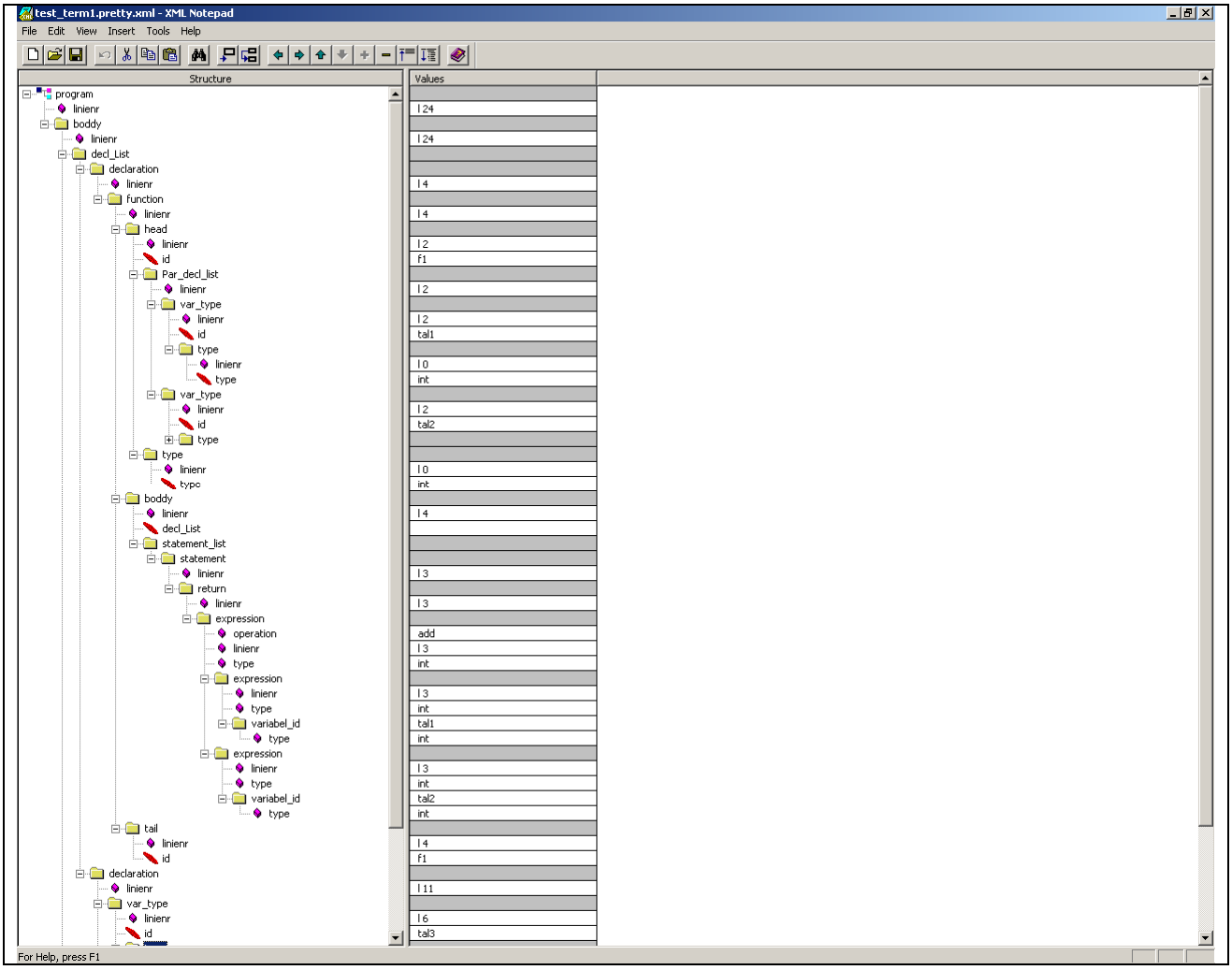
Denne test skal vise at grammatikken genkendes, og at der opbygges et korrekt AST.

Programmet er udformet, så det også kunne bruges til typecheck, da der i denne fase ikke stilles krav om variable findes m.m.

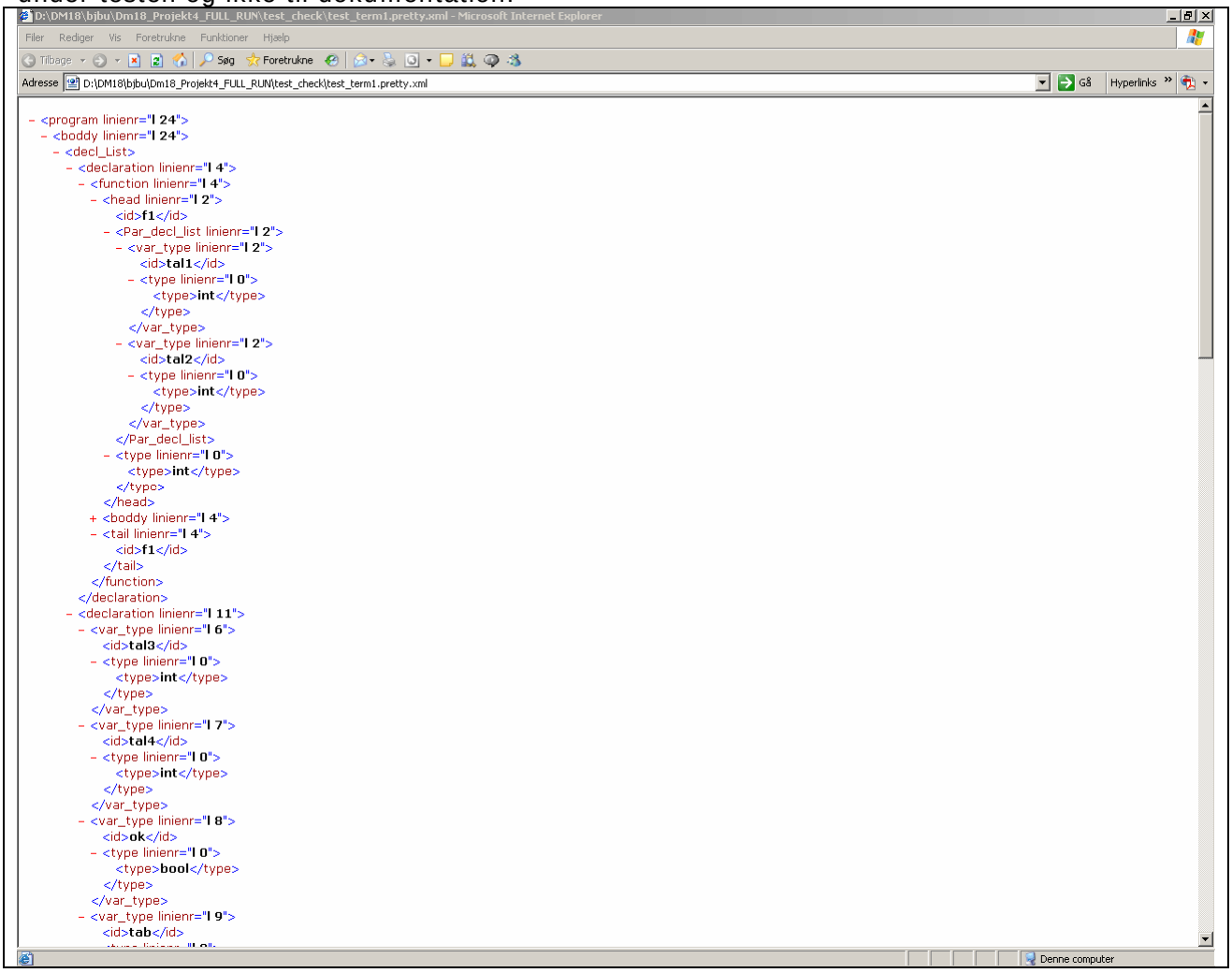
Mine test var på dette tidspunkt nok lidt større end nødvendigt for at vise bestemte dele, men jeg har ikke synes, der skulle laves om på dette, efter jeg havde sikret mig opbygningen var rigtigt.

```
001: (*Test af TONY term's *)
002: func f1 (tal1: int, tal2: int): int
003:   return tal1 + tal2;
004: end f1;
005: var
006:   tal3:  int,
007:   tal4:  int,
008:   ok:    bool,
009:   tab: array of int,
010:   rec: record of {t1: int}
011:   ;
012:
013: tal3 = 1;     (* num *)
014: tal4 = tal3;    (* variable *)
015: tal4 = (tal3+1);(* (expression) *)
016:
017: ok = true;      (* true *)
018: ok = false;     (* false *)
019: ok = !ok;     (* ! *)
020: tab = null;     (* null *)
021: tal3 = |tab|;   (* | expression | *)
022:
023: tal3 = f1(1,2);    (* call *)
```

Outputtet fra min XML pretty printer vist med xmlnotepad giver denne oversigt, og har været rigtig god til at få stræstrukturer frem.



I Micro Soft Internet Explore ser det sådan ud og viser stadig træstrukturen rimeligt, men den kan ikke gemmes sådan med indrykninger, kun ved skærm-dump. Den kan derfor kun bruges under testen og ikke til dokumentation.



Udskriften fra min tekst-pretty-printer er bedre som dokumentation, men giver ikke samme indtryk af AST. Her ser ovenstående sådan ud.

tonyprettyTxt.txt

```
:
func fd#f1 (
      vd#tal1: t#int,
      vd#tal2: t#int      ):t#int

   return t_int#( v_int#tal1 + v_int#tal2 );

end fd#f1;

var
   vd#tal3: t#int,
   vd#tal4: t#int,
   vd#ok: t#bool,
   vd#tab: t#array of t#int,
   vd#rec: t#record of {
      vd#t1: t#int
   };

v_int#tal3 = c#1;
```

```
v_int#tal4 = v_int#tal3;
v_int#tal4 =  (t_int#( v_int#tal3 + c#1 )) ;
v_bool#ok = c#true;
v_bool#ok = c#false;
v_bool#ok =  !(v_bool#ok) ;
v_array#tab = c#null;
v_int#tal3 =  |v_array#tab| ;
v_int#tal3 = f_int#f1 (c#1, c#2);
```

Testen dokumenter at det opbyggede abstrakte syntakstræ svarer nøje til det der gerne skulle udledes af grammatikken, og at det hermed virker korrekt på de medtagede elementer.

Jeg vil ikke inddrage flere eksempler, men vil her nøjes med en oversigt over de test jeg har lavet specielt med henblik på check af opbygningen af AST ud fra grammatikken.
En fuldstændig dokumentation forefindes i bilag B

test_decl1.tony

Denne test viser, at type declarationer behandles korrekt.
Test dokumentationen findes i bilag B.

test_decl2.tony

Denne test viser, at var declarationer behandles korrekt.
Test dokumentationen findes i bilag B.

test_decl3.tony

Denne test viser, at funktions declarationer behandles korrekt.
Test dokumentationen findes i bilag B.

test_decl4.tony

Denne test viser, at syntaksfejl fanges, her at der ikke kan declareres efter et statement.
Test dokumentationen findes i bilag B.

test_stm1.tony

Denne test viser, at expression behandles korrekt.
Test dokumentationen findes i bilag B.

test_stm2.tony

Denne test viser, at if-statements, med og uden else og compound behandles korrekt.
Test dokumentationen findes i bilag B.

test_stm3.tony

Denne test viser, at statements excl. if og for behandles korrekt.
Test dokumentationen findes i bilag B.

test_precedens1.tony

Denne test viser, at en række præcedens-regler behandles korrekt.
Test dokumentationen findes i bilag C.

test_precedens2.tony

Denne test viser, at en række præcedens-regler behandles korrekt.
Test dokumentationen findes i bilag C.

test_precedens3.tony

Denne test viser, at en række præcedens-regler behandles korrekt.
Test dokumentationen findes i bilag C.

test_precedens4.tony

Denne test viser, at en række præcedens-regler behandles korrekt.
Test dokumentationen findes i bilag C.

test_typefor1.tony

Denne test viser, at for-statements behandles korrekt.
Test dokumentationen findes i bilag B.

Afprøvning af weed fasen

De syntaktiske sukker-udvidelser er kommet ind, efter jeg var påbegyndt kodegenereringen. Jeg har derfor valgt at medtage test af afsukringen sammen med testen af den færdigt genererede programkode. Jeg har her lavet følgende test program.

test_code_Suker.tony

```
001: (* programmør: Bjørk Busch - bjbu@tietgen.dk  - 2005.05.15
002:
003: Aftester grammatisk "sukker"-udvidelser
004: *)
005: var x:int;
006:
007: x = -2;          write x;  (* -2 *)
008: x = 4; x += 3;     write x;  (* 7 *)
009: x = 4; x -= 3;     write x;  (* 1 *)
010: x = 7; x = -x;     write x;  (* -7 *)
011: x = 3; x = 5 - -x;  write x;  (* 8 *)
012: x = 3; x = 8 + -x;  write x;  (* 5 *)
013: x = 7; x += x;     write x;  (* 14 *)
014: x = 7; x -= x;     write x;  (* 0 *)
```

En udskrift fra min pretty-tekst-printer giver følgende:

```
var
    vd#x: t#int;

v_int#x = c#-2;
write v_int#x;
v_int#x = c#4;
v_int#x = t_int#( v_int#x + c#3 );
write v_int#x;
v_int#x = c#4;
v_int#x = t_int#( v_int#x - c#3 );
write v_int#x;
v_int#x = c#7;
v_int#x =  (t_int#( c#0 - v_int#x )) ;
write v_int#x;
v_int#x = c#3;
v_int#x = t_int#( c#5 -  (t_int#( c#0 - v_int#x ))  );
write v_int#x;
v_int#x = c#3;
v_int#x = t_int#( c#8 +  (t_int#( c#0 - v_int#x ))  );
write v_int#x;
v_int#x = c#7;
v_int#x = t_int#( v_int#x + v_int#x );
write v_int#x;
v_int#x = c#7;
v_int#x = t_int#( v_int#x - v_int#x );
write v_int#x;
```

For symbolforklaring henvises til afsnit 4 om typecheck.

Her viser udtrykket, der kommer ud fra linie 007

```
v_int#x = c#-2;
```

at afsukringen af – konstant virker.

Fra linie 14 ses, hvad udtykket med += giver

```
v_int#x = t_int#( v_int#x - v_int#x );
```

Dette dokumenterer, at også += bliver tolket korrekt.
Tilsvarende kan ses i udskriften om de øvrige regler.

Ovenstående udskrift er ikke med i bilagene, men her kan findes en kørselsafvikling, hvor der generes assembler. Assembler udskriften vil her kunne dokumentere det samme som ovenstående, men er noget mere omfattende som illustration lige her.

Afprøvning af weed fasen

Som beskrevet under afsnit 2.6 har jeg gennemført de krav, der var forudsat som minimum. Der er fyldestgørende dokumentation for at denne del virker i bilag B, og jeg har valgt at udelade dem her og kun referere til dem og så nøjes med at uddybe 2 af de tre ekstra udvidelser, jeg har lavet.

test_weed1.tony

Denne test viser, at kontrollen af om id i funktions hoved og end er ens virker som det skal. Test dokumentationen findes i bilag B.

test_weed2.tony

Denne test viser, at kontrol af, at der er sikkerhed for return virker.

Testen omfattede en lang række eksempler af såvel nestede if-sætninger med og uden compound-statements, som while konstruktioner.

For while-konstruktionen gælder, at der ikke generelt kan garanteres, at den stopper (stop-problemet). Derfor vil en return i en while ikke være tilstrækkelig.

For min for-konstruktion forholder det sig anderledes på grund af den måde, de skal virke på. Der vil altid være garanti for, at de stopper. En return vil i en for-konstruktion derfor give garanti på samme måde som uden for for-konstruktionen.

Jeg har ikke justeret i min test efter denne udvidelse og kan principielt derfor ikke dokumentere ved test, at det virker for udvidelsen, men da behandlingen i programmet er identisk skulle jeg formelt kunne argumentere for, at den gør.

Test dokumentationen findes i bilag B.

test_weed3.tony

Jeg har som tidligere beskrevet udvidet med at fange statements efter et garanteret return og så deaktivere disse.

Dette virker og dokumenteres med følgende test.

```
001: (* programmør: Bjørk Busch - bjbu@tietgen.dk  - 2005.03.24
002:
003: Aftester om statements efter garanteres return fanges
004: *)
005: func f6 (): int     (* normal med statement efter return *)
006: var tal: int;
007: tal = 1;
008: return 1;
009: tal = 2;
010: tal = 3;
011: end f6;
012:
013: var t: int;
014: (* krav om mindst eet statement *)
015: t = 1;
016:
017: (* ingen krav om return i "main" rutine *)
```

Følgende er et uddrag af meddelelserne fra min compiler.

```
TONY Parser afsluttet normal

TONY weeding fase start
9: Advarsel statement vil aldrig blive udfoert p.g.a. tidligere return
10: Advarsel statement vil aldrig blive udfoert p.g.a. tidligere return
TONY weeding fase slut

Antal alvorligFejl: 0
Antal fejl: 0

Antal advarsler: 2
```

Det fremgår her, at jeg har fanget de 2 statements efter return.

Fra min tekst-pretty-printer er hentet følgende uddrag:

```
return c#1;
inakt#  v_int#tal = c#2;
inakt#  v_int#tal = c#3;
```

Det fremgår her at de sidste 2 statements er markeret inaktive.

Den fulde testdokumentation af ovenstående findes i bilag B.

test_weed4.tony

Denne test viser, at kontrol af typer, der refererer direkte til sig selv (type x = x;), fanges og kontrollen virker, som den skal.
Test dokumentationen findes i bilag B.

3 Symboltabeller

Jeg har grundlæggende foretaget 2 valg.

For det første har jeg valgt at have en symboltabel, der omfatter såvel type declarationer, variable og funktioner, idet jeg har valgt, at der ikke kan være sammenfald mellem navne på disse inden for samme scope. Med en fælles tabel er det let at checke, hvor det med en opdeling i 2 eller 3 tabeller ville være lidt mere omstændigt.
Jeg kunne have valgt at fastholde 2 eller 3 environment i dele af koden og så "skubbet" metoder ind, der bagved kunne styre hvor mange tabeller der skulle bruges. Herved kunne det blive lettere senere at lave tilpasninger, hvis jeg ville opdele i flere. Den løsning, jeg har valgt, er imidlertid meget enkel og jeg benytter mig af køretids-stakken til styre det rigtige environment, når jeg traverserer ned gennem det AST.

For det andet har jeg valgt at gemme så meget information i det AST som muligt. Symboltabellens primære rolle er at kunne referere tilbage til det AST-element, som et symbol repræsenterer.

Symboltabellerne gemmes i det AST element, hvor de oprettes, hvilket vil sige i sProgram, sFunction og sType under record-element.

3.1 Scoperegler

Der vil blive oprettet en symboltabel for "main"-programmet, og for hver funktion vil der blive oprettet en ny med reference til den ydre, og hermed etableres en scope-reference. Jeg har endvidere valgt at oprette en symboltabel for hver record, så symboler heri delvis kan behandles på samme måde som øvrige symboler.

Symboltabellen for en record vil ikke have reference til andre symboltabeller og kan således betragtes som et isoleret scope. Hvis jeg skulle udvide med arv på record, ville en nedarvet kunne have en symbol tabel med reference til symboltabellen for den record, der blev nedarvet fra. Dette ville også give mulighed for overload af felter !!!

Jeg har valgt, at det skal være muligt at kalde funktioner i samme scope, der først declareres senere, således at der kan laves recursion mellem flere funktioner. Da jeg imidlertid også gerne vil kunne afskære, at der kun kan refereres til typer og variable der declareret senere i tony-koden, har jeg valgt, at declarationer nummereres fortløbende og gemmer dette nummer med symbolet. Dette giver senere mulighed for at checke om declarationen er før eller efter.

Muligheden for at kræve, at en variabel skal være erklæret forud, giver mulighed for at hindre en funktion kan få adgang til en variabel ved at declarere den efter funktionen. Det svarer til muligheden i "gode gamle pascal".
Jeg har valg, at implementere det således, at der kommer en advarsel, hvis declarationen er efter brugen, men at der også kan sættes et compiler-direktiv "-restriktiv" hvor det vil udløse en fejl.

3.2 Symboldata

Her følger definitionerne der bruges i forbindelse med mine symboltabeller.

```
typedef struct XREF  /* til reference af hvor symbol anvendes */
{
  int linienr;
  struct XREF *next;
}XREF;
```

```
typedef struct SYMBOL {
  int linienr;
  char *name;
  enum {symType, symHead, symVar} kind;  /* reference til */
  void *astNode;              /* stype, sHead, sVar_Type */
  int symbolBuildDeclare_bloknr;      /* kan bruges til at se om def. før eller
efter */
  int antalXref;
  struct XREF *xref;
  struct SYMBOL *next;
} SYMBOL;

typedef struct SymbolTable {
  int level;
  int symboltabelnr;
  SYMBOL *table[HashSize];
  struct SymbolTable *next;
} SymbolTable;
```

3.3 Algoritme

Mine symboltabeller er opbygget som hash-tabeller, der er linket til overliggende scope. Jeg har samtidig valgt at lave en liste med referencer til symbolet, således at for hver gang der refereres til symbolet registres det linienr, hvorfra referencen kommer. Der kan f.eks. i forbindelse med et assign-statement forekomme flere referencer fra samme linie, men hver af disse registreres.

Jeg har lavet dette for at checke mine tests, men det kan samtidig være et nyttigt hjælpe-værktøj i forbindelse med compileren. Mine registreringer kunne også være nyttige i en live-ness-analyse, hvis jeg stadig havde holdt koblingen over til mit AST fra min intermediate-kode.

3.4 Afprøvning

Jeg har ændret meget i mine symboltabeller siden de blev testet isoleret, så jeg kan kun her dokumentere en afprøvning i forbindelse med den samlede compiler.

Jeg har her et enkelt eksempel fra testen frem til type-check.

test_typecall1.tony

```
001: (* programmør: Bjørk Busch - bjbu@tietgen.dk  - 2005.03.24
002:
003:    Aftester om type-check af parametre og ved funktionskald
004: *)
005: type tab = array of int;
006: type rec = record of { t: tab, i: int };
007: func f1 (t: tab, r: rec, i: int): int
008:    return 1;
009: end f1;
010:
011: var mt: tab, mr: rec, mi: int;
012:
013: mi = f1 (mt, mr, mi);  (* ok *)
014: mi = f1 (mt, mi);      (* fejl i antal *)
015: mi = f1 (mr, mr, mi);  (* fejl i typer *)
```

tonysymbol_xref.txt

```
SymbolTable for MAIN:<0>  tabelnr:1 scope-level:0 - start
mi<11>   variabel af typen: int   ref: 5   <13, 13, 14, 15, 15>
mr<11>   variabel af typen: rec<6> => record<6>   ref: 3  <13, 15, 15>
mt<11>   variabel af typen: tab<5> => array of int   ref: 1  <13>
tab   <5>     type = array of int   ref: 3   <6, 7, 11>
rec   <6>     type = record<6>   ref: 2   <7, 11>
f1 <7>    function af typen: int   ref: 3   <13, 14, 15>
-----------------------------------------
SymbolTable for rec:<6>  tabelnr:2 scope-level:0 - start
i   <6>    variabel af typen: int
t   <6>    variabel af typen: tab<5> => array of int
-----------------------------------------
SymbolTable for f1:<7>  tabelnr:3 scope-level:1 - start
i   <7>    variabel af typen: int
r   <7>    variabel af typen: rec<6> => record<6>
t   <7>    variabel af typen: tab<5> => array of int
```

4 Typecheck

Denne fase har til formål at opsamle alle symbolske referencer, der her udgøres af type-declarationer, variabler og funktioner for at kunne løse referencer til disse.

I denne fase skal der ud over opsamlingen verificeres, om de symbolske referencer til typer, variabler og funktioner bruges korrekt. Dette indebærer også, at der fastlægges og verificeres typer på alle expressions og sub-expressions, og at disse kan indgå i de enkelte statements.

Grundlaget for denne fase er strukturen og funktionerne for symboltabellerne samt det op-byggede AST.

Jeg har valgt at opdele type-check fasen i to delfaser, hvor jeg i første delfase opbygger sym-boltabeller og anden delfase laver type-check.

4.1 Typer

Der findes følgende faste typer i sproget.

- int
- bool
- array
- record

Der kan desuden defineres nye typer, men alene baseret på ovenstående.

4.2 Typeregler

Under afsnittet 1.1 om afklaring kom jeg ind på, at programmeringssprogets syntaks ikke kan beskrives alene ved brug af den kontekstfrie grammatik. Det er også nødvendigt at beskrive typereglerne.

I dette afsnit beskriver jeg disse regler, som jeg har valgt dem.

Jeg har valgt at følge de mest oplagte reger, som vil ligge mest tæt på sprog som C og java m.fl.

Det gælder generelt, at typen null både kan betragtes som typen record, array og null.

Herudover har jeg fra starten valgt følgende tolkninger.

Declarationer	Beskrivelse / krav
type heltal = int;	giver en ny unique type
var x: heltal, y: heltal, z: int;	x og y er af samme type, men z er af en anden type, men de er alle tre af hvad jeg betegner som sluttype Jeg definerer her sluttype som den type, der kan henføres til og ikke selv er en type declaration.
type tabtype = array of int;	
var xt: tabtype, yt: tabtype;	xt og yt er af samme type
var zt: array of int, wt: array of int;	zt og wt er af forskellig type og også af forskellig type fra xt og yt (jvf. ovenstående). De er også af forskellig sluttype.
type rectype = record of ….	
var xr: rectype, yr: rectype;	xr og yr er af samme type
var zr: record of …. , wr: record of ….	zt og wt er af forskellig type og også af forskellig type fra xr og yr (jvf. ovenstående). De er også af forskellig sluttype.

Term'er	Beskrivelse / krav
f (x, y… z)	kræver at for hver af argumenterne (x, y… z) gælder at typen er identisk med typen på funktionens tilhørende parameter. x skal altså have samme type som første parameter i f. Antallet af parametre skal selvfølgelig også stemme overens. Bemærk at returtypen tolkes som angivet i det expression, den indgår i.
!x	kræver at x er af typen bool – tolkes som normal negation på bool
|x|	kræver at x er af typen int eller array – tolkes for int som absolut værdi, for array som antal elementer, der er allokeret til array'et

Expression	Beskrivelse / krav
x*y	Kræver at både x og y er af typen int – tolkes som normal heltals multiplikation
x/y	Kræver at både x og y er af typen int – tolkes som normal heltals division
x+y	Kræver at både x og y er af typen int – tolkes som normal heltals addition
x–y	Kræver at både x og y er af typen int – tolkes som normal heltals subtraktion
x<y, x>y, x<=y og x>=y	Kræver at både x og y er af typen int – tolkes som sammenligninger af heltal
x==y og x!=y	kræver at x og y er af samme type – tolkes som normal sammenligning for bool og int, for record og array tolkes det som en sammenligning af adresser, hvor de er ens, hvis adressen er ens. Det gælder at typen null både kan betragtes som typen record, array og null.
x&&y og x||y	Kræver at både x og y er af typen bool – tolkes som normal and og or på bool. Det kunne også her præciseres om der blev tolket med short-circuit evaluation. Dette har jeg overvejet at gøre, men ikke realiseret.

Statement	Beskrivelse / krav
return x;	Kræver sluttypen på x er af samme type som funktionen, hvor return-sætningen indgår i. Jeg giver her mulighed for simpel type-konvertering, så hvis funktionen er at typen int, kan følgende accepteres. type heltal = int; var x: heltal; return x; I forbindelse med "main" programmet kræves, at x er af sluttypen int.
write x;	Kræver at sluttypen på x er af typen int, se return – udskriver værdien af x (bemærk krav er med compilerflag –restriktiv)
new x of length y;	kræver at x er af sluttypen array og y af typen int – opretter et array med y elementer og tildeler x startadressen på dette array.
new x;	kræver at x er af sluttypen record – opretter en record og tildeler x startadressen på denne record.

Statement fortsat	Beskrivelse / krav
x=y;	kræver at x er af samme type som y – hvis x er af sluttypen array eller record kan y være null-typen. Hvis x og y har sluttypen bool eller int, sættes x til samme værdi som y. Hvis x og y er af sluttypen array eller record, sættes x til samme adresse som y.
if x …. og while x ….	Kræver at x er af typen bool
for x=y to z do for x=y downto z do	Kræver at x er af typen int samt at y og z er expression af typen int

Indexing og strukturer	Beskrivelse / krav
x[y]	kræver at x er af sluttypen array og y er af typen int.
x.y	Kræver at x er sluttypen record og y er element i denne. Det skal specielt her bemærkes at strukturer kan være i flere niveauer og indeholde arrays og array kan have record elementer og dermed også strukturer.

Nestede arrays og records	Beskrivelse / krav
array of x	Kræver at x er af sluttypen int, bool, array eller record. For typen array vil x være en referencetype (adresse). For typen record, har jeg fra starten også valgt at det var en referencetype, men har senere ændret det så recorden er et element på heapen, idet den første stadig er mulig med compilerdiriktivet –noRecordArray For referencetype og data skal oprettes med brug af henholdsvis new og new of length, ellers oprettes elementer sammen med array med new of length. Hvis x er en type af array følges næste beskrivelse.
array of array .. array of x	kræver at x er af sluttypen int, bool, record eller en typedeclaration, der i sidste niveau ikke må være et array. For det sidste niveau, der altså kun kan være int, bool eller record følges ovenstående beskrivelse
record of {…. v:x ….}	Kræver at x er af sluttypen int, bool, array eller record. Der kan således laves nestede records direkte, hvor hver af disse også kan indeholde array. record of { r: record of { ….}, a: array of .. } r og a vil være en referencetype og data skal oprettes med brug af henholdsvis new og new of length.

Jeg valgte fra starten yderligere, at det ikke skulle være muligt at referere til typer der var declareret senere i koden, men alene til typer der var declareret forud. Det gælder også i forbindelse med de senere scope-regler. Begrundelsen var jeg ville have nogle meget stramme tolkninger, for at vise jeg kunne typechecke disse.

Jeg har senere åbnet for en mindre stram tolkning, idet jeg inspireret af gcc har et compilerdirektiv (-restriktiv) der fastholder den stamme tolkning og under dette tillader følgende. Hvor kravene er blevet beskrevet som samme type, kræves blot samme sluttype.
Eksempel:

```
type heltal = int;
var x: heltal, y: int;
```

Her vil x og y begge være af sluttypen int og kan f.eks. assignes og sammenlignes, hvad ikke er tilladt i den stramme udgave.

Jeg tillader envidere at udskrive alle typer og ikke kun typen int når compiler-direktiv (-restriktiv) ikke er sat.
For bool realiseres de som en int med 0 for false og 1 for true og kan behandles som en int, ligesom array og record er adresser, der også kan behandles som en int i forbindelse med write.
Denne tolkning i forbindelse med write har også været nyttig i min test, da jeg kunne slippe for at bruge debug-værktøj.

I min sidste fase, hvor jeg skulle teste, har jeg lavet en ekstra debug tolkning, der kan aktiveres med et flag. ”write null;”, der ikke giver den store mening at skrive i et normalt program, vil her give et dump af registre og heap. Jeg kunne på denne måde slippe nemt ved at se, om adresser og heap blev brugt rigtigt uden specielle debug værktøjer.

4.3 Algoritme

Typer og strukturer til abstrakt syntaks træ (AST)

Jeg har valgt ikke at lave et sæt nye strukturer til at definere typer, men i stedet at udvide de allerede eksisterende i det AST.

Declarationen af sType indeholder stor set det nødvendige, idet der dog bliver behov for kunne repræsentere term-null som en type. Denne kan ikke rigtig placeres inden for de eksisterende, så her oprettes en ny.

For at gøre det enkelt at sammenligne typer har jeg valgt at justere oprettelsen af typer, således at der for ”basis-typerne” int, bool og null ikke oprettes et nyt element hver gang, men i stedet anvendes en såkaldt singleton(løst med global variabel), så der kun findes én af hver af disse. Hermed kan man lave en enkel sammenligning at typer ved blot at sammenligne deres adresser.

Da typer kan være declarationer til nye typer i flere omgange, har jeg valgt at gemme en reference til den type der kan føres tilbage til (slutType) for den enkelte type. Jeg har senere lavet en funktion, der kan gøre dette fordi leg ville undgå flere test for null-referencer, så der er nu lidt overlap. Jeg kunne fjerne slutType referencen og i stedet bruge min funktion eller omvendt, men jeg har ikke nået at rette denne del igennem.

Type definitionen ser nu sådan ud (fra tonyAST.h):

```
typedef struct sType {
  int linienr;
  struct sType *slutType;   /* sættes på i forb. med type-check */
  int dataLength;
  enum {kType_Id, kType_Int, kType_Bool, kType_Array,
        kType_Record, kType_Null} kind;
        /* sidste ikke i AST men en expression-type */
  union {
    struct { char *id; struct sType *type; int id_linienr; } eId;
    struct { struct sType *type; int elmLength; } eArray;
    struct { struct sVar_decl_list *var_decl_list;
             struct SymbolTable *symbolTable; int elmLength; } eRecord;
  } val;
```

Oprettelsen af en af de primitive typer (int) ser sådan ud (fra tonyAST_Make.c):

```
/*-------------------------------------------------------------
Strukturer der skal sikre singleton på primitive typer
-------------------------------------------------------------*/
sType *primitiv_Type_Int = NULL;    /* for type def på expression */

sType *makeType_Int ()
{
  sType *s;
  if (primitiv_Type_Int == NULL) {
    s = NEW(sType);
    s->linienr = 0;
    s->kind = kType_Int;
    s->slutType = s;  /* er sluttype */
    primitiv_Type_Int = s;
  }
  else
    s = primitiv_Type_Int;
  return s;
}
```

Det AST er som tidligere nævnt også blevet udbygget med referencer til symboltabel, referencer declarationer og typer på termer og expressions.

Strukturen for sVariable ser nu sådan ud (fra tonyAST.h):

```
typedef struct sVariable {
  int linienr;
  struct sType *type;       /* opsamlet ved typecheck */
  enum {kVariable_Id, kVariable_Indexed, kVariable_Struct} kind;
  union {
    struct { char *id; int id_linienr; struct sVar_type *var_type; } eId;
    struct { struct sVariable *variable;
             struct sExpression *expression; } eIndexed;
    struct { struct sVariable *variable; char *id; int id_linienr;
             struct sVar_type *var_type;} eStruct;
  } val;
}sVariable;
```

I starten ses, at der er blevet udvidet med type-reference.
Under eId og eStruct ses, at der yderligere er blevet udvidet med en reference til den tilknyttede variabel.

Som en lille kuriøsitet ses i øvrigt, at jeg har tilføjet et ekstra id_linienr. Det har jeg gjort alle de steder, hvor der opsamles en id. Det skyldes, at linienr henføres til der, hvor hele nonterminalen er fanget, men hvis elementerne er placeret over flere linier, svarer det ikke til linienr, hvor det pågældende id optræder. Jeg har derfor ændret i Flex, så jeg opsamler et separat linienr, hvor en id bliver genkendt.

I sType definitionen, der blev vist på foregående side, ses under eRecord en udbygning med reference til den tilknyttede symboltabel. En tilsvarende findes i sProgram og sFunction.

Opbygningen af symboltabeller ud fra AST

Første del, af min type-check fase, består i at få opbygget symboltabellerne.

Dette sker i en selvstændig del-fase, hvor det AST bliver gennemløbet. Årsagen til dette er, at jeg ønsker, der skal kunne ske referencer til funktioner i samme scope fremadrettet.
Det betyder, at jeg først er nødt til at opsamle alle disse, før jeg checker referencer til funktioner.

Jeg overvejede om det også skulle gælde for type-declarationer og variable, men fandt at det nok for disse ville være mere nyttigt, at give mulighed for at udelukke tilgang ved at placere declarationer efter funktionerne, der ikke skulle have tilgang.

Jeg kunne have valgt at vente med at indsætte type-declarationer og variable i symboltabeller indtil gennemløbet, hvor de skulle checkes. Det ville gøre det nemt at sikre mod fremadrettet referencer, da så ikke var i symboltabellen før der måtte refereres til dem. Jeg ville imidlertid gerne kunne give en advarsel/fejlmeddelelse, der fortalte, at der ikke kunne refereres til dem, fordi de var declareret senere og ikke, at de ikke var declareret.
Endelig ville jeg gerne have mulighed for evt. at ændre så det blev tilladt at referere frem (det blev senere default og den stramme løsning kun med –restriktiv direktivet)

Jeg valgte derfor, at opbygge alle symboler i samme omgang og løse problemet med fremadrettede referencer på en anden måde.

Problemet med fremadrettet reference har jeg løst ved at indføre en fortløbende nummerering af declarationer og tilføje dette nummer til symbolet.

Ved den senere kontrol nummerer jeg igen declarationerne. Jeg kan så afgøre ud fra dette om et symbol er declareret før eller efter det bruges.
Dette er en yderst simpel men effektiv måde at løse problemet på.

Jeg har overvejet ligeledes at lave et funktions-level nummer, som gemme sammen med symbolet for senere let at kunne afgøre, i hvilket funktions-scope en variabel placeret for at klare referencer i assembler-koden, hvor data må tilgås forskelligt afhængig af scope.

For at holde styr på, hvilken symboltabel symboler skal placeres i, har jeg også valgt en simpel løsning. I stedet for at lave en selvstændigt environment kontrol med funktioner og en stak til at etablere en ny symboltabel og reetablere senere, har jeg valgt at lave en variabel til den aktuelle symboltabel og så udnytte køretidsstakken ved gennemløb af det AST.

Når jeg opretter en nyt scope og en ny symboltabel, sker det altid i node-funktion, hvor jeg også kan slutte med at reetablere.

Her ses de variable der danner environment for opbygningen af symboltabellerne (fra tonySymbols_Build.c):

```
/*-------------------------------------------------------------
Fælles data for dette modul
-------------------------------------------------------------*/
int symbolBuildDeclare_bloknr = 0;  /* variabel der bruges til kontrol af om
decleration er forud for brug */
struct SymbolTable *buildSymbolTable;
```

Her ses et eksempel på ”opsamlingen” af et nyt element til symboltabellen (fra tonySymbols_Build.c):

```
/*-------------------------------------------------------------*/
void symbolsBuildVar_type (sVar_type *s)
{
  s->val.eVar.symbol = putSymbol(buildSymbolTable, s->val.eVar.id,
                          symVar, s, symbolBuildDeclare_bloknr,
                          s->val.eVar.id_linienr);
  symbolsBuildType(s->val.eVar.type);
}
```

Her ses, hvordan der registreres en ny declarations-blok (fra tonySymbols_Build.c):

```
/*-------------------------------------------------------------*/
void symbolsBuildDeclaration (sDeclaration *s)
{
  ++symbolBuildDeclare_bloknr;
  switch(s->kind)
  {
    case kDeclaration_Type:     symbolsBuildDeclaration_Type    (s);break;
    case kDeclaration_Function: symbolsBuildDeclaration_Function(s);break;
    case kDeclaration_Var:      symbolsBuildDeclaration_Var     (s);break;
  }
}
```

Her ses et eksempel på etablering af et nyt scope og reetableringen af det gamle (fra tonySymbols_Build.c):

```
void symbolsBuildFunction (sFunction *s)
{
  /* gem adresse på aktuel symboltabel */
  SymbolTable *saveSymbolTable = buildSymbolTable;
  /* opret ny symboltabel - nyt level */
  buildSymbolTable = initSymbolTable(buildSymbolTable);
  /* gem symboltabel i AST */
  s->symbolTable = buildSymbolTable;
  symbolsBuildHead(s->val.eFunction.head);
    symbolsBuildBody(s->val.eFunction.body);
  /* reetabler adresse på aktuel symboltabel */
  buildSymbolTable = saveSymbolTable;
}
```

Type-check ud fra symboltabeller og AST

Anden del af min type-check fase består i:

- at verificere om de symbolske referencer til typer, variabler og funktioner bruges korrekt
- at bestemme og verificere typer på alle expressions og subexpressions
- at bestemme om typer passer med kravene for de enkelte statement-typer.

Denne del klares ved et enkelt gennemløb af det AST.

Jeg har ligesom under opbygningen af symboltabellerne valgt den simple løsning med en variabel til den aktuelle symboltabel og så brugt køretidsstakken til at etablere og reetabler det korrekte environment.

Jeg har ligeledes en tællevariabel, der holder styr på hvilken declarations-blok jeg er nået til, så jeg kan afgøre, om symboler er declareret før eller efter en reference til disse.

Jeg har endvidere valgt en enkel løsning på typecheck fra return statement, idet jeg også her har en en variabel, der indeholder den aktuelle funktionstype og ligesom symboltabellerne etableres og reetableres fra udvalgte node-funktioner, idet jeg også her udnytter køretidsstakken.

I forbindelse med at jeg finder en et symbol ved opslag i symboltabellerne, fastsættes type og reference til den node i det AST som symbolet repræsenterer. Der bliver således gem en række informationer i det AST, som senere også kan bruges i forbindelse med kodegenereringen.

Jeg har haft overvejelser om også at holde styr på hvilket level en funktion er på, og ligeså med de tilhørende variable, for at kunne håndtere referencer til et ydre scope fra funktioner. Jeg har dog valgt ikke at implementere dette på nuværende tidspunkt.

Her ses hvordan der laves et opslag i den aktuelle symboltabel for en type, checkes at symbolet er en type og at typen er erklæret forud for anvendelsen, hvis compiler-diriktiv –restriktiv er sat (fra tonySymbols_Check.c):

```
void symbolsCheckType_Id (sType *s)
{
  bool walvorligFejl = false;
  sDeclaration *decl;
  SYMBOL *symbol = getSymbol(checkSymbolTable, s->val.eId.id, s->linienr);
  s->val.eId.type = NULL;
  if (symbol == NULL) {
    fprintf(msgFile, "%d: Fejl: type %s er ikke defineret\n",
        s->val.eId.id_linienr,s->val.eId.id);
    ++alvorligFejl;
    walvorligFejl = true;
  }
  else if (symbol->kind != symType) {
    if (symbol->kind == symVar) {
      fprintf(msgFile,
        "%d: Fejl: <%s> er en variabel og kan ikke bruges som type\n",
        s->val.eId.id_linienr,s->val.eId.id);
      ++alvorligFejl;
      walvorligFejl = true;
    }
    else if (symbol->kind == symHead) {
      fprintf(msgFile,
        "%d: Fejl: <%s> er en funktion og kan ikke bruges som type\n",
        s->val.eId.id_linienr,s->val.eId.id);
      ++alvorligFejl;
      walvorligFejl = true;
    }
    else {
      fprintf(msgFile,
        "%d: Fejl: <%s> ikke en type!!!!!\n",
        s->val.eId.id_linienr,s->val.eId.id);
      ++alvorligFejl;
      walvorligFejl = true;
    }
  }
```

```
  else if (symbolCheckDeclare_bloknr < symbol->symbolBuildDeclare_bloknr) {
    decl = symbol->astNode;
    if (restriktivCheck)
    {
      fprintf(msgFile,
        "%d: Fejl: typen <%s> er ikke tilgaengelig - erklaeret senere<%d>\n",
        s->val.eId.id_linienr,s->val.eId.id, decl->linienr);
      ++alvorligFejl;
      walvorligFejl = true;
    }
    else {
        fprintf(msgFile,
"%d: Advarsel: reference til typen <%s> som foerst erklaeres senere<%d>\n",
          s->val.eId.id_linienr,s->val.eId.id, decl->linienr);
      ++advarsel;
    }
  }
  if (walvorligFejl==false)
  {
    decl = symbol->astNode;
    s->val.eId.type = decl->val.eType.type;/*registrer ref. til sType i AST*/
    s->slutType = s->val.eId.type->slutType;
    if (s->slutType == NULL) {    /* fanger cykliske */
      fprintf(msgFile,
        "%d: Fejl: type <%s> reference er ikke gyldig\n",
        s->val.eId.id_linienr,s->val.eId.id);
      ++alvorligFejl;
    }
  }
}
```

Her ses, hvordan der laves typecheck på en term – jeg har her valgt den mindst komplicerede for illustrationen (fra tonySymbols_Check.c):

Teknikken er en sædvanlig post-order løsning, hvor der checkes på tilbageturen, efter termen er blevet typebestemt, før der returneres fra denne term. I det konkrete tilfælde sættes termen her til en fast type, der er den eneste gyldige og vil således ikke medføre flere typefejl op igennem det AST.

```
void symbolsCheckTerm_Not (sTerm *s)
{
  s->type = makeType_Bool();
  symbolsCheckTerm(s->val.eNot.term);
  if (s->val.eNot.term->type == NULL) {
    fprintf(msgFile,
      "%d: Fejl: negation ikke mulig - type på udtryk mangler\n",s->linienr);
    ++alvorligFejl;
  }
  else if (s->val.eNot.term->type->kind != kType_Bool) {
    fprintf(msgFile,
"%d: Fejl: negation ikke mulig med typen <%s> men kun med typen <bool>\n",
        s->linienr, type_Kind_Txt(s->val.eNot.term->type));
    ++alvorligFejl;
  }
}
```

Den mest komplicerede kontrol er vel nok kontrollen af en struktur (record variabel), idet der her kan være flere niveauer, hvor det foregående kan være såvel en simpel variabel, en indekseret variabel eller selv være en record struktur.

Jeg har valgt at forsøge at give så meget information tilbage om evt. fejl som muligt, hvilket selvfølgelig giver øget kompleksitet.

Nedenstående viser hvordan jeg har løst dette:

```
void symbolsCheckVariable_Struct (sVariable *s)
{
  bool walvorligFejl = false;
  SYMBOL *symbol;

  symbolsCheckVariable(s->val.eStruct.variable);

  s->type = NULL;
  if (s->val.eStruct.variable->type == NULL) {
    fprintf(msgFile,
      "%d: Fejl: struktur-niveau ikke mulig her\n",s->linienr);
    ++alvorligFejl;
    walvorligFejl = true;
  }
  else if (s->val.eStruct.variable->type->slutType == NULL) {
    fprintf(msgFile,
      "%d: Fejl: struktur-niveau ikke mulig her\n",s->linienr);
    ++alvorligFejl;
    walvorligFejl = true;
  }
  else if (s->val.eStruct.variable->type->slutType->kind != kType_Record) {
    fprintf(msgFile,
      "%d: Fejl: strutur-niveau ikke mulig paa typen <%s> men kun \
på typen <record>\n",
      s->linienr, type_Kind_Txt(s->val.eStruct.variable->type->slutType));
    ++alvorligFejl;
    walvorligFejl = true;
  }
  else {
    symbol = getSymbol(
        s->val.eStruct.variable->type->slutType->val.eRecord.symbolTable,
        s->val.eStruct.id, s->linienr);
    if (symbol == NULL) {
      fprintf(msgFile,
        "%d: Fejl: variabel <%s> er ikke defineret\n",
        s->val.eStruct.id_linienr,s->val.eStruct.id);
      ++alvorligFejl;
      walvorligFejl = true;
    }
    else if (symbol->kind != symVar) {
      if (symbol->kind == symType) {
        fprintf(msgFile,
          "%d: Fejl: <%s> en type og kan ikke bruges som variabel\n",
          s->val.eStruct.id_linienr,s->val.eStruct.id);
        ++alvorligFejl;
        walvorligFejl = true;
      }
      else if (symbol->kind == symHead) {
        fprintf(msgFile,
          "%d: Fejl: <%s> en funktion og kan ikke bruges som type\n",
          s->val.eStruct.id_linienr,s->val.eStruct.id);
        ++alvorligFejl;
        walvorligFejl = true;
      }
      else {
        fprintf(msgFile,
          "%d: Fejl: <%s> ikke en variabel!!!!!\n",
          s->val.eStruct.id_linienr,s->val.eStruct.id);
        ++alvorligFejl;
        walvorligFejl = true;
      }
```

```
    }
    else if (symbolCheckDeclare_bloknr < symbol->symbolBuildDeclare_bloknr){
      if (restriktivCheck) {
        fprintf(msgFile,
        "%d: Fejl: variablen <%s> er ikke tilgaengelig - erklaret senere\n",
          s->val.eStruct.id_linienr,s->val.eStruct.id);
        ++alvorligFejl;
        walvorligFejl = true;
      }
      else {
        fprintf(msgFile,
          "%d: Fejl: variablen <%s> er erklaret senere en brug\n",
          s->val.eStruct.id_linienr,s->val.eStruct.id);
        ++advarsel;
      }
    }
  }
  if (!walvorligFejl) {
    /* registrer ref. til sVar_Type i AST */
    s->val.eId.var_type = symbol->astNode;
    s->type = s->val.eId.var_type->val.eVar.type;
    if (s->type->slutType == NULL) {
      fprintf(msgFile,
        "%d: Fejl: type <%s> kan ikke foeres tilbage til gyldig type\n",
        s->val.eStruct.id_linienr,s->val.eStruct.id);
      ++alvorligFejl;
      walvorligFejl = true;
    }
  }
}
```

4.4 Afprøvning

Afprøvningen er sket med henholdsvis min xml-pretty-printer, som jeg selv har brugt en del sammen med xmlnotepad, og min tekst- pretty-printer, som jeg anvender som dokumentation.
Jeg har i disse test fastholdt min restriktive brug af typer (-restriktiv flag sat), som jeg oprindelig brugte. Jeg har haft afprøvet uden flaget, og set at jeg så godkendte de der skulle godkendes og som blev afvist med flaget sat. Jeg har dog ikke medtaget disse test her, på grund af omfanget.

I udskriften anvendes følgende notation for typer:

- c# for konstanter num-, true-, false- og null-term
- v_x# for variabel id hvor x er type
- t_x# for type id hvor x er type
- vd# for variabel declaration
- td# for type id declaration
- fd# for funktions declaration
- t# for grund type i declaration

Nedenstående viser et uddrag fra en sådan udskrift (uddrag fra test_typeexp1.tony):

```
var
   vd#tal: t#int,
   vd#ok: t#bool;

v_int#tal = t_int#( c#1 + t_int#( c#2 * c#3 ) );
v_int#tal = t_int#( c#1 + t_int#( c#2 / c#3 ) );
v_int#tal = t_int#( c#1 + t_int#( t_int#( t_int#( c#2 * c#3 ) / c#4 ) * c#5 ) );
```

Jeg har lavet række test der skal vise om symbol fasen virker som den skal.

Jeg har under vejs haft testet opbygningen af symboltabeller selvstændigt, men det giver ikke mere mening da den delfase nu indgår i resten af denne fase.

Jeg vil i det efterfølgende trække nogle af disse test frem.

Test af record strukturer

Jeg har valgt denne frem fordi det er en af de lidt større test, der både kan illustrere hvordan symboltabellerne bruges til records og der påsættes typer på variable.

Det viser også hvordan jeg markerer typefejl.

test_struct1.tony

```
001: (* programmør: Bjørk Busch - bjbu@tietgen.dk  - 2005.03.24
002:
003: Test af TONY record structurer
004: *)
005: type recdef =  record of {niv2: record of {niv3: record of {t: int, b:bool}}};
006: var
007: tal:   int,
008: ok:      bool,
009: rec:  recdef,
010: rec2: recdef,
011: rec3: record of {niv2: record of {niv3: record of {t: int, b:bool}}}
012: ;
013: tal = rec.niv2.niv3.t;  (* ok *)
014: rec.niv2.niv3.t = tal;  (* ok *)
015: ok  = rec.niv2.niv3.b; (* ok *)
016: rec.niv2.niv3.b = ok;  (* ok *)
017: rec2 = rec;          (* ok *)
018: rec = rec2;          (* ok *)
019: rec2.niv2 = rec.niv2;  (* ok *)
020: rec.niv2 = rec2.niv2;  (* ok *)
021: rec.niv2.niv3 = rec2.niv2.niv3;(* ok *)
022: rec.niv2.niv3.t = rec2.niv2.niv3.t;(* ok *)
023: tal = rec.niv2;          (* fejl *)
024: tal = rec.niv2.t;     (* fejl *)
025: ok  = rec.niv2.niv3.t;  (* fejl *)
026: rec3 = rec2;         (* fejl *)
```

Console output

```
Options:
-msg sat til test_typestruct1.consol.txt
-S er sat
-restriktiv er sat
-prettyTxt er sat
-xref er sat

TONY Parser afsluttet normal

TONY weeding fase start
TONY weeding fase slut
TONY opbygning af symboltabel start
TONY opbygning af symboltabel slut
```

```
TONY type-check start

23: Fejl: type i assignment <int> = <record> : der tilades kun ens typer
24: Fejl: variabel <t> er ikke defineret
24: Fejl: assign ikke mulig da expression type ikke def
25: Fejl: type i assignment <bool> = <int> : der tilades kun ens typer
26: Fejl: type i assignment <record> = <recdef> : der tilades kun ens typer
TONY type-check slut

TONY SYMBOL XREF udskrives til tonysymbol_xref.txt
TONY abstract syntax tree udskrives i TXT-format til <tonyprettyTxt.txt>

Antal alvorligFejl: 5

Antal fejl: 0

Antal advarsler: 0
```

output test_struct1.pretty.txt

```
type td#recdef :t#record of {
   vd#niv2: t#record of {
      vd#niv3: t#record of {
         vd#t: t#int,
         vd#b: t#bool
      }
   }
};

var
   vd#tal: t#int,
   vd#ok: t#bool,
   vd#rec: t#recdef,
   vd#rec2: t#recdef,
   vd#rec3: t#record of {
      vd#niv2: t#record of {
         vd#niv3: t#record of {
            vd#t: t#int,
            vd#b: t#bool
         }
      }
   };

v_int#tal = v_recdef#rec.v_record#niv2.v_record#niv3.v_int#t;
v_recdef#rec.v_record#niv2.v_record#niv3.v_int#t = v_int#tal;
v_bool#ok = v_recdef#rec.v_record#niv2.v_record#niv3.v_bool#b;
v_recdef#rec.v_record#niv2.v_record#niv3.v_bool#b = v_bool#ok;
v_recdef#rec2 = v_recdef#rec;
v_recdef#rec = v_recdef#rec2;
v_recdef#rec2.v_record#niv2 = v_recdef#rec.v_record#niv2;
v_recdef#rec.v_record#niv2 = v_recdef#rec2.v_record#niv2;
v_recdef#rec.v_record#niv2.v_record#niv3 = v_recdef#rec2.v_record#niv2.v_record#niv3;
v_recdef#rec.v_record#niv2.v_record#niv3.v_int#t =
v_recdef#rec2.v_record#niv2.v_record#niv3.v_int#t;
v_int#tal = v_recdef#rec.v_record#niv2;
v_int#tal = v_recdef#rec.v_record#niv2.v_???#t;
v_bool#ok = v_recdef#rec.v_record#niv2.v_record#niv3.v_int#t;
v_record#rec3 = v_recdef#rec2;
```

output test_struct1.xref.txt

```
SymbolTable for MAIN:<0>  tabelnr:1 scope-level:0 - start
ok <8>  variabel af typen: bool  ref: 3  <15, 16, 25>
recdef<5>  type = record<5>  ref: 2  <9, 10>
rec<9>  variabel af typen: recdef<5> => record<5>  ref: 13  <13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25>
tal <7>  variabel af typen: int  ref: 4  <13, 14, 23, 24>
rec2  <10>  variabel af typen: recdef<5> => record<5>  ref: 7 <17, 18, 19, 20, 21, 22, 26>
rec3  <11>  variabel af typen: record<11>  ref: 1  <26>
-----------------------------------------
SymbolTable for recdef:<5>  tabelnr:2 scope-level:0 - start
niv2  <5>  variabel af typen: record<5>  ref: 15  <13, 14, 15, 16, 19, 19, 20, 20, 21, 21, 22, 22,
23, 24, 25>
-----------------------------------------
SymbolTable for niv2:<5>  tabelnr:3 scope-level:0 - start
niv3  <5>  variabel af typen: record<5>  ref: 9 <13, 14, 15, 16, 21, 21, 22, 22, 25>
-----------------------------------------
SymbolTable for niv3:<5>  tabelnr:4 scope-level:0 - start
b  <5>  variabel af typen: bool  ref: 2 <15, 16>
t  <5>  variabel af typen: int  ref: 5  <13, 14, 22, 22, 25>
-----------------------------------------
SymbolTable for rec3:<11>  tabelnr:5 scope-level:0 - start
niv2  <11>  variabel af typen: record<11>
-----------------------------------------
SymbolTable for niv2:<11>  tabelnr:6 scope-level:0 - start
niv3  <11>  variabel af typen: record<11>
-----------------------------------------
SymbolTable for niv3:<11>  tabelnr:7 scope-level:0 - start
b  <11>  variabel af typen: bool
t  <11>  variabel af typen: int
```

I tony programmet kan vi se at der er en type declaration i linie 5 af ”recdef” på en record struktur i 3 niveauer.

```
type recdef =  record of {niv2: record of {niv3: record of {t: int, b:bool}}};
```

I tekst-pretty printeren kom den til at se sådan ud:

```
type td#recdef :t#record of {
    vd#niv2: t#record of {
        vd#niv3: t#record of {
            vd#t: t#int,
            vd#b: t#bool
        }
    }
};
```

Her kan vi se at der er blevet på sat typer som der skal og at der henholdsvis kommer en typedeclaration og nogle variable ud af dette.

I ”main” symboltabellen kan vi se at recdef optræder med følgende linie:

```
recdef  <5>     type = record<5>   ref: 2   <9, 10>
```

Vi kan her se at symbolnavnet er defineret i linie 5, at der er en record type også defineret i linie 5, at der er 2 referencer til symbol-navnet - en i linie 9 og en i linie 10, hvilet passer med det vi kan se i tony-programmet.

Der kommer endvidere 3 nye symboltabeller ud af declarationen, en for hver record declaration, hvilket der også skulle.

```
SymbolTable for recdef:<5>  tabelnr:2 scope-level:0 - start
SymbolTable for niv2:<5>  tabelnr:3 scope-level:0 - start
SymbolTable for niv3:<5>  tabelnr:4 scope-level:0 - start
```

I linie 9 erklæres variablen ”rec”.

```
rec:    recdef,
```

I tekst-pretty printeren kom den til at se sådan ud:

```
vd#rec: t#recdef,
```

Det ses at typen er gendkendt “t#recdef”.

I ”main” symboltabellen er der nu også kommet en reference til den nye declaration, her ser de sådan ud:

```
rec  <9>  variabel af typen: recdef<5> => record<5>   ref: 13  <13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25>
```

Vi kan se at ”rec” er erklæret i linie 9, får typen recdef som er erklæret i linie 9 og kan føres tilbage til typen record.
Vi kan desuden se der er 13 referencer til variablen og i hvilke linier der sker fra.

I linie 13 bruges variablen og de underliggende niveauer.

```
tal = rec.niv2.niv3.t;
```

I tekst-pretty printeren kom den til at se sådan ud:

```
v_int#tal = v_recdef#rec.v_record#niv2.v_record#niv3.v_int#t;
```

Vi ser her at der er blevet identifcere typer på hver niveau, idet det dog kun fremgår at der er record og ikke hvilken. Dette skyldes at det ellers ville blive ulæseligt. Vi kan dog konstatere at typerne er blevet identificeret ned til sidste niveau som var en int.

Vi også se i symboltabellerne at de er blevet genkendt, da der er kommet referencer på rec og de underliggende niveauer.

```
SymbolTable for MAIN:<0>  tabelnr:1 scope-level:0 - start
rec  <9>  variabel af typen: recdef<5> => record<5>   ref: 13  <13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25>
---------------------------------------------------
SymbolTable for recdef:<5>  tabelnr:2 scope-level:0 - start
niv2 <5>  variabel af typen: record<5>   ref: 15  <13, 14, 15, 16, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 25>
---------------------------------------------------
SymbolTable for niv2:<5>  tabelnr:3 scope-level:0 - start
niv3 <5>  variabel af typen: record<5>   ref: 9  <13, 14, 15, 16, 21, 21, 22, 22, 25>
---------------------------------------------------
SymbolTable for niv3:<5>  tabelnr:4 scope-level:0 - start
t  <5>  variabel af typen: int   ref: 5 <13, 14, 22, 22, 25>
```

I linie 23 forsøges på at bruge rec.niv2 (record) sammen med tal (int) i en assignment.

```
tal = rec.niv2;
```

Dette giver anledning til følgende fejlmeddelelse da assignment kræver at det er samme type.

```
23: Fejl: type i assignment <int> = <record> : der tilades kun ens typer
```

I linie 24 referetes til en variabel der ikke er på det angive niveau.

```
tal = rec.niv2.t;
```

Dette giver anledning til følgende fejlmeddelelser.

```
24: Fejl: variabel <t> er ikke defineret
24: Fejl: assign ikke mulig da expression type ikke def
```

Den første viser at variablen ikke var defineret.
Den anden fås som følge af den første, da det ikke har været muligt at få en type på variablen og hermed kan der ikke foretages en assignment der kræver ens typer.

I tekst-pretty printeren kom den til at se sådan ud:

```
v_int#tal = v_recdef#rec.v_record#niv2.v_???#t;
```

Det ses at ”t” ikke er blevet typebestemt v_???#.

Jeg har valgt ikke at afbryde ved første fejl, men i stedet så fange ”følgefejl”. Der er også nogle expression og term’er, der vil give en fast type retur f.eks. |x| selv om der er foregående fejl, og hvor der så ikke nødvendigvis vil blive følgefejl. Det er således enklere bare at fortsætte og så fange så mange fejl som muligt, en teknik som de fleste oversættere også benytter sig af.

Test af return

Jeg har valgt at return kan foretage simple type-konverteringe, selv når mit compilerdirektiv ”–restriktiv” er sat.

Der er alene tale om konverteringer der følger at flere niveauer at type-declarationer.
F.eks. vil en variabel baseret på typen heltal declareret som ”type heltal = int” her kunne returneres som int, ligesom en int ville kunne returneres som heltal. Jeg betegner dette som samme grundtype.

Jeg tillader ikke 2 record eller array, der er ens opbygget bruges med ”forkert” type.

Jeg har lavet 3 test at return statement.

test_typereturn1.tony

Dette er en simpel aftestning der skal vise om return fra en funktion med afvigende type bliver afvist og med samme type godkendt.
Testen viser at der sker en afvisning af en afvigende type, og at det godkendes hvis der er tale om samme type.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typereturn2.tony

Her testes at return med en simple typer, der kan føres tilbage til samme ”grundtype” som funktionen godkendes.
Testen viser at det godkendes, hvis der er tale om samme type ”grundtype”.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typereturn3.tony

Her testes at return med en ikke simpel type godkendes, hvis den er baseret på samme type-declaration som funktionens returværdi.
Testen viser at det godkendes, hvis der er tale om værdier baseret på samme type og at det afvises hvis det ikke er selv om typerne er ens opbygget.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

Test af funktionskald

Jeg har lavet 2 test at funktionskald.

test_typecall.tony

Denne test skal vise om kontrol af antal argumenter og deres typer passer med funktions declarationer.
Testen viser der både "fanges" et forkert antal argumenter og hvis typerne ikke stemmer overens. Det godkendes hvis antal og typer stemmer overens.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typecall2.tony

Her testes at retur-typerne fra funktionerne (funktions-type) er blevet korrekt bestemt, så de kan bruges som expression. Testen sker ved brug af asignment, der så også her bliver delvis aftester.
Testen viser at funktionerne typebestemmes korrekt og at de kan bruges i assignment hvis typerne svarer til variabel-typen.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

Test af statement excl. return

Jeg har lavet 1 test for hvert statement type.

test_typeassign1.tony

Her testes typecheck ved assign statements. Det chekkes af assign kun tillades hvis variabel og expression er af samme type, idet det for typer der kan henføres til array og record også tillades at tildele disse null-typen.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typewrite1.tony

Her testes at der i write statements kun accepteres typer der kan henføres til int-typen.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typenew1.tony

Her testes at new uden length of kun accepter typer, der kan henføres til record-typer, og med length of kun acceptere typer, der kan henføres til array-typen. Det testes envidere at der i length og kun accepteres typen int.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeif1.tony

Her testes at der kun acceptetes expression af typen bool i if statements.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typewhile1.tony

Her testes at der kun acceptetes expression af typen bool i while statements.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeexp1.tony

Her testes aritmetiske expression samt brugen af numeric operatoren på int.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeexp2.tony

Her testes boolske expression.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeexp3.tony

Her testes boolske expression og sammenlignings expression.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeexp4.tony

Her testes expression som i foregående men udvidet med negation.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typearray1.tony

Her testes array af en og flere dimmensioner, array af record og array af record med array.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typearray2.tony

Her testes brugen af numeric operatoren i forbindelse med array af diveres dimensioner
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typescope1.tony

Her testes scope-reglerne, herunder at funktioner kan tilgås i samme scope selv om de er defineret efter, hvorimod typer og variable kun kan tilgås, hvis de er defineret forud.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typeminus.tony

Her testes minus foran konstanter og variable.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

test_typefor1.tony

Her testes typecheck for for-to og for-downto konstruktionen.
Testen dokumenter, at dette virker, som det skal. Der henvises til bilaget for dokumentation.

5 Ressourceberegning

5.1 Ressourcer

Jeg foretager får kodefasen en beregning af alle størrelser af data, herunder beregnes specielt record-størrelser, der jo kan være af forskellig længde. Endvidere foretages en tilsvarende beregning for parametre.

Jeg nummererer endvidere alle typer og variable, så der kan dannes et entydigt navn i forbindelse med kodegenereringen.

Beregningerne foretages i en selvstændig fase.

5.2 Algoritme

Beregningerne foretages ved en recursiv nedstigning gennem AST elementerne, hvor der påsættes længder på tilbageturen.

I et enkelt tilfælde kan der ikke ved denne nedstigning findes en længde, hvilket er når en type defineres som en record, der refererer til samme type (f.eks. en linket liste), men her sættes længden så bare til 4, da det så vil være en adresse.
På denne enkle måde får jeg løst at skulle checke for cykliske referencer.

5.3 Afprøvning

Jeg har ikke foretaget en selvstændig afprøvning af denne fase, men den er indgået sammen med min code-fase.

6 Kodegenerering

Min kodegenerering baseres på det AST, hvor der er blevet påsat elementstørrelser og placering, der direkte kan bruges til offset-fastlæggelse.

6.1 Strategi

Jeg har valgt at min assemblerkode skal afspejle det oprindelige program så meget som muligt. Jeg havde derfor på forhånd besluttet at jeg ville fastholde symbolske navne, hvor det var muligt. Dette skulle for lokale data ske ved brug af .equ direktivet, som jeg ville allokere på stakken, da jeg på forhånd havde fravalgt at arbejde med temporære variable (tilvalgt peep-hole optimering).

Jeg valgte fra starten først at kodegenerere direkte, så jeg kunne afprøve om skabelonerne virkede. Jeg anvendte herefter dette som grundlag for min intermediate kode. Da peep-hole optimering udføres efter kodegenereringen er denne mellemform nødvendigt, men jeg kunne til gengæld godt lave en abstrakt assembler, der stor set var færdig. Der er blevet tale om en ren liste, dog med referencer til labels i forbindelse med jumps. Jeg har således også fravalgt at lade min kode indeholde træstrukturer for adressering og expression. Jeg mister ved dette ganske vist nogle af de muligheder der kunne være efterfølgende for at ”se” disse elementer i forbindelse med optimering, men det skulle stadig være muligt at lave livenes-analyse og arbejde med temporære variable på trods af den strategi.

Jeg kunne i forbindelse med min ressourcefase have dannet variable, således at jeg kunne vente med at tage stilling til placering og adresse, ved først at indlægge dem i assemblerkoden til sidst, det har jeg imidlertid ikke gjort. Jeg har kun registreret navn og et entydigt nummer. Der bliver således ingen forbindelse mellem den definerende equ og brugen ud over navnet. Jeg overvejede, om jeg skulle definere og oprette et variabel element eller jeg skulle oprette en symboltabel til assemblerkoden, men fravalgte dette da jeg kunne klare mig med symbolet. Det betyder også at mine adresser fastlægges allerede i kodegenereringen.

Vedrørende adresser har jeg valgt følgende strategi for variable.
Disse placeres på stakken, så de ligger med den først definerede på laveste adresse. De ligger således ikke omvendt på stakken, som de normalt vil komme til, hvis lægges ud efterhånden som man møder den ned gennem træet. Jeg har defineret et par variable, der bestemmer retningen, således at den kan ændres, hvis man hellere vil have det modsat.

For lokale variable sker adresseringen med det symbolske offset baseret på ebp registeret. Hvis der skal adresseres til et andet scope, vil det ske ved at esi bliver brugt som frame-pointer. Når der placeres parametre på stakken opbygges disse i den rækkefølge de forekommer. Her allokeres først plads på stakken til hele parameterblokken og edi bruges som frame-pointer, idet det er justeret så variable har samme placering i forhold til dette, så de senere vil få i forhold til den kaldtes frame-pointer opsat med ebp.

Denne løsning bevirker, at jeg kan bruge samme symbolske referencer som bruges i den kaldte rutine. Det bliver derfor lettere at læse koden.

6.2 Kodeskabeloner

Strukturer til den abstrakte assembler

Den abstrakte assembler opbygges med følgen strukturer fra tonyAbstractAsm.h

```
typedef struct asAdr {
  union {
    struct { int reg; } eReg;
    struct { struct asElm *offset; int base;
        int faktor; int disp; } eMem;   /* explicit angivelser */
    struct { int offset; int base;
        int faktor; int disp; } eMemEx; /* explicit angivelser */
    struct { int imm; } eImm;          /* Immidiate (value)   */
    struct { struct asElm *label;
        } eLabel;   /* memory label med implicit segment ... */
  } val;
  enum {aaReg, aaMem, aaMemEx, aaImm, aaLabelValue, aaLabelAdr} kind;
}asAdr;

typedef struct asElm {
  bool slettet;
  struct asElm *next;
  int linienr;
  char *comment;
  enum {akComment, akDef, akOffset, akLabel,  /* spec. */
  akPush, akPop, akMul, akDiv, akInc, akDec, akNeg, /* 1 op */
  akMove, akLoadAdr, akAdd, akSub, akAnd, akOr, akXor, akCmp, /*2 op*/
  akJump, akCall, akReturn, akCltd} kind;
  union {
      struct { char *def; } eDef;
      struct { char *name; int nr; int offset; } eOffset;
      struct { char *name; int nr; char *suffix; } eLabel;
      struct { struct asAdr *adr; } eInstr1;
      struct { struct asAdr *fAdr; struct asAdr *tAdr; } eInstr2;
      struct { struct asElm *label;
         enum {akJmp, akJe, akJne, akJl, akJg, akJle, akJge,
         akJnl, akJng} jump;
      } eJump;
      struct { struct asElm *label; } eCall;
  } val;
}asElm;

enum {nreg, eax, ebx, ecx, edx, esi, edi, ebp, esp} reg;
```

Funktioner til den abstrakte assembler

Den abstrakte assembler opbygges med følgende funktioner

```
asAdr *asMakeAdrReg (int reg);
asAdr *asMakeAdrMem (asElm *offset, int base, int faktor, int disp);
asAdr *asMakeAdrMemEx (int offset, int base, int faktor, int disp);
asAdr *asMakeAdrImm (int imm);
asAdr *asMakeAdrLabelValue (asElm *label);
asAdr *asMakeAdrLabelAdr (asElm *label);

asElm *addAsmElm (asElm *elm);

asElm *asMakeComment (int linienr, char *comment);
asElm *asMakeDef (char *def, int linienr, char *comment);
asElm *asMakeOffset (char *name, int nr, int offset,
        int linienr, char *comment);
```

```
asElm *asMakeLabel (char *name, int nr, int linienr, char *comment);
asElm *asMakeLabel2 (char *name, int nr, char *suffix,
        int linienr, char *comment);
asElm *asMakeInstr1 (int kind, asAdr *adr, int linienr, char *comment);
asElm *asMakeInstr2 (int kind, asAdr *fAdr, asAdr *tAdr,
        int linienr, char *comment);
asElm *asMakeJump (int jump, asElm *label, int linienr, char *comment);
asElm *asMakeCall (asElm *label, int linienr, char *comment);
asElm *asMakeReturn (int linienr, char *comment);
asElm *asMakeCltd (int linienr, char *comment);
```

Skabeloner for gennering af den abstrakte assembler

Jeg vil her beskrive nogle af de skabeloner jeg har brugt, idet der vil være en skabelon for stort set hver element i det AST.

Skabelon for en if-else konstruktion

Precondition: Boosk if-expression er leveret i eax – 0=falsk og 1=sand

```
      cmpl   $1,%eax
      jne    lab_else
      statements for if-delen
      jmp    lab_endif
lab_else:
      statements for else-delen
lab_endif:
```

Skabelon for en if konstruktion uden else

Precondition: Boosk if-expression er leveret i eax – 0=falsk og 1=sand

```
      cmpl   $1,%eax
      jne    lab_endif
      statements for if-delen
lab_endif:
```

Skabelon for en while konstruktion

Precondition: Boosk while-expression er leveret i eax – 0=falsk og 1=sand

```
lab_while:
      cmpl   $1,%eax
      jne    lab_endwhile
      statements for while-delen
      jmp    lab_while
lab_endwhile:
```

Skabelon for en for-to konstruktion

```
      Init-expression beregnes og leveres i eax
      store expression i init-variabel, med adresse som for assign
      To-expression beregnes og leveres i ebx

      pushl    %ebx          #gem stopværdi
lab_fortest:
      cmpl   0(%esp),%eax
      jg     lab_forend
      pushl    %eax          #gem tæller til næste loop
      statements for for-delen
      popl   %eax          #retabler tæller
      incl   %eax          #step +1
      store tæller variabel, med adresse som assign
      jmp    lab_fortest
lab_forend:
```

Skabelon for en new of length konstruktion

Precondition: antal-elementer-expression er leveret i eax

<elm-length> vil afgøres af typen (konkret er alle grundtyper 4), hvis det er en array- eller record-type vil typen normalt være en adresse. Hvis det er e record og denne allokeres direkte på stakken (-noRecordArray er IKKE sat), vil længden af recorden blive brugt.

```
      orl      %eax,%eax          #test for >0
      jg     lab_newlength
      movl   <linienr>,%eax       #gem tony-linienr til fejludskrift
      jmp    lab_nonpositivelements  #fejl rutine som afslutter
lab_newlength:
      pushl    %eax               #gem antal elementer
      movl   <elm-length>,%ebx    #hent elementlængde
      imul   %ebx                #size beregnes
      addl   $4,%eax             #der sættes plads af til antal elm. variabel
      movl   lab_myheapfreeadr,%esi    #hent fri-adresse
      movl   %eax,0(%esi)        #gem allokeringssize på heap
      addl   %esi,%eax           #ny fri-adresse
      addl   $4,%eax             #  incl. felt til allokeringssize
      movl   %eax, lab_myheapfreeadr # og gemmes
      cmpl   $lab_aftermyheapspace,%eax#test om græse er overskredet
      jng    lab_newlength2      #ok
      movl   <linienr>,%eax      #gem tony-linienr til fejludskrift
      jmp    lab_outofmemory     #fejl rutine som afslutter
lab_newlength2:
      leal   8(%esi),%eax        #sæt adresse til elm index 0
                                 #antal elm er i -4(%eax)
                                 #allokeringssize i -8(%eax)
      popl   %ebx,-4(%eax)       #gem antal elm.

      store %eax i array variabel på samme måde som ved assign
```

Skabelon for dyadiske expression

I forbindelse med dyadiske expression anvender jeg en skabelon, der skal sikre at 1. operanen kommer i eax og 2. opranten i ebx.

Det sker gennem følgende overordnet skabelon.

```
  Hvis ikke 1. operant så:
      pushl    %eax          #save 1. operant
  Slut_hvis:
      beregn venstre expression som 1. operant og returner i %eax
      beregn venstre expression som 2. operant og returner i %ebx
      foretag aktuelle dyadiske beregning på %eax og %ebx
  Hvis ikke 1. operant så:
      movl   %eax,%ebx     #flyt til 2. operant
      popl   %eax          #reetabler 1. operant
```

6.3 Algoritme

Først oprettes alle globale labels og herefter sker opbygningen af assemblerkoden ved at først at opbygge globale data i datadelen.

Denne del udgøres dels af en række formater til udskrifter. Den ene er format til at udskrive tal til write statementet. De øvrige er til de forskellige runtimefejl der kan forekomme.

Herudover oprettes en scope-tabel, med et element for hver niveau, hvor en funktion ved starten kan gemme sin frame-pointer, så de næste nivauer kan tilgå dens data.

Herefter sættes plads af til heap sammnen med nødvendige kontrolelementer.

Når dette er gjort har jeg valgt at starte med ”main” rutinen, hvor der først udføres en prolog med opsætning af frame-pointer.
Herefter løbes alle type- og variabel-declarationer igennem og der opbygges symbolske navne ved brug af equ direktivet.

Når dette er gjort løbes alle statements igennem og for hver tager en funktion sig af opbygningen som det skete i de tidligere faser.
Herefter udføres epilogen, der afslutter ”main” og så følger rutiner til behandling af runtime fejl.

Endelig løbes funktionsdeclarationerne igennem og for hver af dem gælder at der først laves en prolog, så defineres data ved at løbe type og variabel declarationer igennem. Herefter følger funktionens statements og en epilog, hvorefter denne funktions funktioner løbes igennem.

Resultatet er, at der foretages en serialisering af programmet, der nu ligger i en liste.

6.4 Afprøvning

Jeg startede som tidligere nævnt med at lave en rutine, der kodegenerede direkte. Jeg afprøvede den først på nogle få programmer, jeg selv havde fremstillet, men sikrede mig så at jeg kunne få det rigtige resultat på alle de programmer, der var lagt ud på imada. Efter jeg havde sikret mig dette grundlag var i orden, lavede jeg en ny rutine, der kunne opbygge den abstrakte assembler i liste-form. I stedet for at lave en speciel pretty-printer til at verificere denne abstrakte assembler, valgte jeg at lave min emit funktion, så jeg kunne gennerer færdig assembler. Denne er blevet udviklet parallelt med funktionerne til den abstrakte assembler.

Afprøvningen er dels sket ved manuelt at kontrollere, at den assembler der blev genereret svarede til den forventede, og dels ved at oversætte den generede assemblerkode og køre programmet. Jeg har derfor ingen selvstændig test for denne fase, idet testen er foretaget løbende sammen med emit-fasen (inden peep-hole optimering blev tilføjet).

Jeg har haft testet min direkte generede kode med de samme eksempler, men efter de sidste rettelser vedrørende allokering af array of record, som ikke ajourført denne kode, anser jeg det som uinteressant af vise test fra denne som dokumentation af kodegennerings fasen. Jeg vil derimod tillade mig, at henvise til testene beskrevet sammen med min emit-fase i afsnit 8.2, hvor det dokumenteres af såvel kodegennereringens- som emit-fasen virker som de skal.

7 Faser før emit

Jeg har som nævnt i afsnit 1.3 under udvidelser valgt at fokusere på peep-hole optimering, men også gjort mig tanker om garbage-collection.

Den sidstnævnte kunne imidlertid også delvis placeres, som en udvidelse der blev linket ind, således at analysen her kun ville omhandle frigivelsen af en allokering på heapen.

Selve allokeringen og mekanismer til frigivelse af data på heapen, vil jeg vurdere høre hjemme under ressourcer, og jeg har derfor medtaget dette under afsnit 5.

7.1 Analyser

Peep-hole analyser

Jeg har foretaget defineret en række peep-hole mønstre.
Nedenstående uddrag fra programkoden viser disse mønstre sammen med den funktion der laver analysen. Alle disse mønstre er implementeret.

```
void peepHoleOptimizeAddSub2() /* add/sub 2. operant */
  /* mønster:   (tilsvarede for subl) (regx != regy)
       movl adr,regx
       addl regx,regy
     hvis adr = 1 eller - 1 reduceres til:
       incl regy  eller decl regy
     ellers reduceres til:
       addl adr,regy
  */

void peepHoleOptimizeAddConst1() /* add const 1. operant */
  /* mønster:   (regx != regy && adr != regx)
       movl const,regx
       movl adr,regy
       addl regy,regx
     hvis const = 1 eller - 1 reduceres til:
       movl adr,regx
       incl regx  eller decl regx
     ellers reduceres til:
       movl adr,regx
       addl const,regx
  */

void peepHoleOptimizeAddSubConstTwize()
  /* mønster:
       addl const1,adr2   eller subl const1,adr2
       addl const2,adr2   eller subl const1,adr2
     reduceres til:
       addl const3,adr2
  */
  /* mønster:
       movl const1,adr2
       addl const2,adr2   eller subl const1,adr2
     reduceres til:
       movl const3,adr2
  */
void peepHoleOptimizeCompare()
  /* mønster:
       movl adr,regx
       cmpl regx,regy
     reduceres til:
       cmpl adr,regy
```

```
  */
void peepHoleOptimizeStoreLoad()
{
  /* mønster:
       movl reg,adr
       movl adr,reg
     reduceres til:
       movl reg,adr
  */
void peepHoleOptimizeMoveTwize()
  /* mønster: (tilsvarende for leal)
    (hvis adr1 aaMem eller aaMemEx må base og disp ikke være adr2)
       movl adr1,adr2
         * antal push adrx eller pop adry
       movl adr1,adr2
     reduceres til:
       movl adr1,adr2
         * antal push adrx eller pop adry

      push adrx eller pop adry skal gælde
       adry != adr1 og adry != adr2
       hvis adr1 aaMem eller aaMemEx må base og disp ikke være adry
  */
void peepHoleOptimizeLoadLoadImm()
  /* mønster:
       movl imm,regx
       movl regx,regy
     reduceres til:
       movl imm,regy
  */
void peepHoleOptimizeMulConst2() /* 2. operant const */
  /*
     mønster: (regx != eax)
       movl 0,regx
       imul regx
     reduceres til:
       xorl eax
     mønster: (regx != eax)
       movl 1,regx
       imul regx
     reduceres til:
       ingenting
     mønster: (regx != eax)
       movl -1,regx
       imul regx
     reduceres til:
       negl eax
     mønster: (regx != eax)
       movl 2,regx
       imul regx
     reduceres til:
       addl eax,eax
     mønster: (regx != eax)
       movl -2,regx
       imul regx
     reduceres til: (regx != eax)
       addl eax,eax
       negl eax
  */
void peepHoleOptimizeMulConst1() /* 1. operant constant */
  /* mønster: (regx != eax og adr != eax)
       movl 0,eax
       movl adr,regx
```

```
      imul regx
    reduceres til:
      xor eax,eax
    mønster: (regx != eax og adr != eax)
      movl 1,eax
      movl adr,regx
      imul regx
    reduceres til:
      movl adr,eax
    mønster: (regx != eax og adr != eax)
      movl -1,eax
      movl adr,regx
      imul regx
    reduceres til:
      movl adr,eax
      negl eax
    mønster: (regx != eax og adr != eax)
      movl 2,eax
      movl adr,regx
      imul regx
    reduceres til:
      movl adr,eax
      movl eax,eax
    mønster: (regx != eax og adr != eax)
      movl -2,eax
      movl adr,regx
      imul regx
    reduceres til:
      movl adr,eax
      movl eax,eax
      negl eax
  */
void peepHoleOptimizeJumpTwize()
  /* mønster: (lab1 != lab2)
      jxx1 lab1
      lab1:
      jxx2 lab2
    reduceres til:
      jxx1 lab2
      lab1:
      jxx2 lab2
  */
void peepHoleOptimizeIncDecFollow()
  /* mønster:
      incl adr
      incl adr
      incl adr
    reduceres til:
      addl 3,adr
   mønster:
      decl adr
      decl adr
      decl adr
    reduceres til:
      addl -3,adr
  */
void peepHoleOptimizeTwizeCmpOrAnd()
  /* mønster: op2 (cmpl, orl, andl, xorl)
      op2 adr1,adr2
      op2 adr1,adr2
    reduceres til:
      op2 adr1,adr2
  */
```

Det skal bemærkes, at jeg her kan fjerne cmpl, da den ikke kan forekomme to gange efter hinanden, da den altid vil blive efterfulgt af en jump.
Det har jeg bare ikke fået ændret i koden og derfor også ladet det blive stående her.

```
void peepHoleOptimizeContra()
  /* mønster:
       negl adr
       negl adr
     eller
       incl adr
       decl adr
     eller
       decl adr
       incl adr
     eller
       pushl adr
       popl  adr
     reduceres til:
       ingenting   (NB: flag bruges ikke)
  */
```

7.2 Algoritmer

Peep-hole optimering

Da min abstrakte assembler ligger i en liste uden referencer til det AST, kan disse jeg direkte bruge denne del af compileren løsrevet i en anden sammenhæng.

For at argumentere for at optimeringen afslutter, defineres en termineringsfunktion, der skal tælle ned mod en endelig værdi (0).

Min terminerings funktion er defineret ved følgende:
T(”code”) = antal instruktioner + antal multiplikationer + antal jumps i kæde.

Herefter skal der argumenteres for, at T(”code”) bliver mindre for hver gennemløb.
Hvis man ser på mønstrene er der to, som ikke direkte vil give færre instruktioner.

Det ene mønster er ved multiplikation med konstanten -2, hvor 3 instruktioner bliver til 3 instruktioner, men her vil der blive en multiplikation mindre, så terminerings-funktionen holder.

```
     mønster: (regx != eax og adr != eax)
       movl -2,eax
       movl adr,regx
       imul regx
     reduceres til:
       movl adr,eax
       movl eax,eax
       negl eax
```

Det andet mønster er ved en følge af to jump, hvor der er det samme antal instruktioner, men her vil der blive en jump mindre ”i følge”, og dermed holder terminerings-funktionen.

```
  /* mønster: (lab1 != lab2)
       jxx1 lab1
       lab1:
       jxx2 lab2
     reduceres til:
       jxx1 lab2
       lab1:
       jxx2 lab2
  */
```

Jeg har valgt ikke at slette instruktioner, men blot markere dem, ligesom jeg ikke ændre i de eksisterende, men markere dem slettet og indsætter nye. Dette er primært af hensyn til at kunne dokumentere forløbet.

For alle de øvrige vil der ske en reduktion af antallet af instruktioner uden at de to andre variable ændres og dermed holder terminerings-funktionen for alle.

Mine algoritmer arbejder med en focus-instruktion, der tages udgangspunkt fra.

Overordnet prøves at optimere ud fra hvert mønster ud fra focus-instruktionen i et loop, der kører så længe der blev foretaget en optimering.

For at optimere optimerings-processen flytter en optimering-funktion en instruktion frem, hvis den sletter en instruktion, så ikke alle optimeringsmønstre skal kaldes forgæves.

```
  while (optimer) {
    ++antLoops;
    optimer = false;
    focusInstr = asmListeHead;
    elm = focusNextInstr();
    while (elm != asmListeTail)
    {
      peepHoleOptimizeContra();
      peepHoleOptimizeAddConst1();
      peepHoleOptimizeAddSub2();
      peepHoleOptimizeCompare();
      peepHoleOptimizeAddSubConstTwize();
      peepHoleOptimizeStoreLoad();
      peepHoleOptimizeMoveTwize();
      peepHoleOptimizeTwizeCmpOrAnd();
      peepHoleOptimizeLoadLoadImm();
      peepHoleOptimizeMulConst2();
      peepHoleOptimizeMulConst1();
      peepHoleOptimizeJumpTwize();
      peepHoleOptimizeIncDecFollow();
      elm = focusNextInstr();
    }
  }
```

Her er et eksempel på en af de mindre funktioner, der ofte vil få effekt.

```
void peepHoleOptimizeAddSubConstTwize()
{
  asElm *elm2;
  int kind;
  int const3;
  if (focusInstr->slettet) return;
  /* mønster:
       addl const1,adr2   eller subl const1,adr2
       addl const2,adr2   eller subl const1,adr2
     reduceres til:
       addl const3,adr2
  */
  /* mønster:
       movl const1,adr2
       addl const2,adr2   eller subl const1,adr2
     reduceres til:
       movl const3,adr2
  */
  if (focusInstr->kind != akAdd && focusInstr->kind != akSub &&
      focusInstr->kind != akMove) return;
  if (focusInstr->val.eInstr2.fAdr->kind != aaImm) return;
```

```
  elm2 = nextInstr(focusInstr);
  if (elm2->kind != akAdd && elm2->kind != akSub) return;
  if (elm2->val.eInstr2.fAdr->kind != aaImm) return;
  if (!equalsAdr(focusInstr->val.eInstr2.tAdr,elm2->val.eInstr2.tAdr))
    return;
  if (focusInstr->kind == akSub)
      const3 = -focusInstr->val.eInstr2.fAdr->val.eImm.imm;
  else
      const3 = focusInstr->val.eInstr2.fAdr->val.eImm.imm;
  if (elm2->kind == akSub)
      const3 -= elm2->val.eInstr2.fAdr->val.eImm.imm;
  else
      const3 += elm2->val.eInstr2.fAdr->val.eImm.imm;

  if (focusInstr->kind == akMove) kind = akMove;
  else                            kind = akAdd;
  insertAsmElm(asMakeInstr2(kind,
              asMakeAdrImm (const3),
              elm2->val.eInstr2.tAdr,
              optimizenr,"optimized twize add/sub const"));
  deleteInstr (focusInstr);
  deleteInstr (elm2);
  focusNextInstr(); /* flyt til nyindsatte*/
  optimer = true;
  if (optimizeLog)
    fprintf(msgFile, "Add/sub const twize optimize: %d\r\n",optimizenr--);
}
```

7.3 Afprøvning

Peep-hole optimering

Min abstrakte assembler er uden referencer til det AST og de øvrige data fra før code-fasen. Det betyder at jeg alene udnytter den abstrakte assembler i peep-hole optimeringen. Jeg havde således også fra starten gjort det muligt, at isolere optimeringen, ved at danne noget test-abstrakt-assembler og optimere på dette. Jeg valgte imidlertid at tage udgangspunkt i noget, det var genereret ud fra tony-programkode, så jeg også kunne sandsynliggøre at mønstrene forekom i praksis. Jeg har også flittigt gjort brug af netop at finde mønstrene i den genererede kode.

Da jeg havde lavet min emit-fase før jeg tog fat på peep-hole optimeringen, har jeg kunnet anvende de tony-programmer, der blev lavet til at teste denne del.

Der er kun 3 mønstre, jeg ikke fandt brugt i den kode og dem der indgår i funktionerne:

- `void peepHoleOptimizeIncDecFollow()`
- `void peepHoleOptimizeTwizeCmpOrAnd()`
- `void peepHoleOptimizeLoadLoadImm()`

Jeg har ikke lavet særlige test, der kunne fremprovokere dem og hermed teste dem og kan derfor ikke formelt dokumentere, at de forekommer og virker.

For de øvrige gælder:

`void peepHoleOptimizeAddSub2() /* add/sub 2. operant */`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeAddConst1() /* add const 1. operant */`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeAddSubConstTwize()`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeCompare()`
Forekommer og dokumenteret virker med test_code_exp2.tony i Bilag C.

`void peepHoleOptimizeStoreLoad()`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeMoveTwize()`
Forekommer og dokumenteret virker med test_code_exp2.tony i Bilag C.

`void peepHoleOptimizeMulConst2() /* 2. operant const */`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeMulConst1() /* 1. operant constant */`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

`void peepHoleOptimizeJumpTwize()`
Forekommer og dokumenteret virker med test_code_if.tony i Bilag C.

`void peepHoleOptimizeIncDecFollow()`
Forekommer og dokumenteret virker med test_code_exp1.tony i Bilag C.

Eksempel på peep-hole-optimeringen (uddrag)

test_code_exp1.tony

```
001: (* programmør: Bjørk Busch - bjbu@tietgen.dk  - 2005.05.12
002:
003: Aftester expression med int, herunder nogle optimizing mønstre
004: *)
005: var  x: int;
006: write 10;       (* 10 *)
007: write -9;       (* -9 *)
008: write 5+3;      (*  8 *)
009: write 4-11;     (* -7 *)
010: write 3--3;     (*  6 *)
011: write -10+5;    (* -5 *)
012: write 5*4;      (* 20 *)
013: write 20/3;     (*  6 *)
014: write 3+4*5;    (* 23 *)
015: write 3+4*-5;   (*-17 *)
016: write 3+4*5/3; (*  9 *)
017: write (3+4)*5; (* 35 *)
018: write 3+4*5+6; (* 29 *)
019: write 1+1+1+1; (*  4 *)
020: write 1*2*3*1; (*  6 *)
021: write -1*2*4*1;(* -8 *)
022: write 1*-2*5*1;(*-10 *)
023: write 3*3*0*3; (*  0 *)
024: write 3*3*3;    (* 27 *)
025:
026: x = 10;
027: write x+5;       (* 15 *)
028: write 5+x;       (* 15 *)
029: write x+5+x;      (* 25 *)
030: write 5+x+5;      (* 20 *)
031: write 5+x+5+5-5+5; (* 25 *)
032: write x+1+4-3;    (* 12 *)
```

```
033: write x*1*-1*-1*2*2;  (* 40 *)
```

Console output (uddrag)

```
Options:
-optimizeLog er sat

……

TONY peep-hole optimimering start

add/sub const 1. op optimize: -1
Add/sub const twize optimize: -2
add/sub optimize: -3
add/sub optimize: -4
add/sub const 1. op optimize: -5
Add/sub const twize optimize: -6
add/sub const 1. op optimize: -7
Add/sub const twize optimize: -8
add/sub optimize: -9
add/sub const 1. op optimize: -10
add/sub optimize: -11
add/sub optimize: -12
Mul const 1.op optimize: -13
Mul const 2.op optimize: -14
Mul const 1.op optimize: -15
Mul const 2.op optimize: -16
……
TONY peep-hole optimimering slut:
    5 gennemloeb og reduceret med:
    59 instruktioner
    14 multiplikationer
    0 jumps
```

Jeg har i min abstrakte assembler overført linienr fra det AST, som relateres til tony-programmet. Jeg har ved optimeringen brugt min indbyggede linienr facilitet og så blot givet negative numre, så kunne jeg spore hvilke optimeringer der hørte sammen i min gennerede assemblerkode. Jeg har dog ved rapportskrivningen konstateret, at jeg ved sidste ændring er kommet til at udelade det i udskriften, men det ville ellers stå i starten af kommentaren. Dette er sket i forbindelse med indlæggelse at compiler-direktivet –noComment, der gør det muligt at udlade alle kommentarer i den generede assemblerkode.

Optimeret assemblerkode

En betragtning af nedenstående assemblerkode fra tony-programmets linie 8, viser en optimering:

```
1. #8:write
2. ###  movl  $5,%eax  #num
3. ###  movl  $3,%ebx  #num
4. ###  addl  %ebx,%eax  #+
5. ###  movl  $3,%eax  #optimized add/sub const 1. op
6. ###  addl  $5,%eax  #optimized add/sub const 1. op
7.   movl  $8,%eax  #optimized twize add/sub const
```

Da der er foretaget flere optimeringer, er den lidt vanskelig at tolke direkte, så her er en forklaring på, hvad der er sket.

De oprindelige instruktioner var:

```
2.  movl $5,%eax #num
3.  movl $3,%ebx #num
4.  addl %ebx,%eax #+
```

Disse er blevet reduceret til:

```
3.  movl $3,%ebx #num
6.  addl $5,%eax #optimized add/sub const 1. op
```

Dette er så igen blevet reduceret til:

```
7.  movl $8,%eax #optimized twize add/sub const
```

Hvis mit nummer stadig havde været skrevet med ud, ville det kunne ses I forhold til udskriften på consolen, hvilke regel, der havde ændret linien.

8 Emit

Denne fase skal omsætte den abstrakte assembler (intermediate koden) til intel assembler.

8.1 Eksempelkode

Da jeg allerede i min kode-fase producerer stort set færdig assemblerkode, er denne fase yderst enkelt, idet jeg blot skal løbe min liste igennem og omsætte til tekstformat. Funktionerne er opdelt, så der løses en specifik opgave i hver f.eks. at udskrive en adresse.

printAsmFileProgram

Hovedfunktionen, der løber listen igennem, ser således ud:

```
void printAsmFileProgram ()
{
  int i = 0;
  asElm *e = asmListeHead;
  openprintAsmFile();
  fprintf(msgFile,
          "TONY assembler udskrives til <%s> start\n",
          printAsmFile_FILENAME);

  fprintf(printAsmFile,"# Oversat fra TONY\n# Compiler skrevet af %s\n\n",
      "Bjørk Boye Busch - bjbu@tietgen.dk");

  e = asmListeHead;
  while (e != NULL) {
    printAsmElm (e);
    e = e->next;
    ++i;
  }
  fprintf(msgFile,
          "TONY assembler udskrives til <%s> slut\n",
          printAsmFile_FILENAME);
  fclose(printAsmFile);
}
```

printAsmElm

Funktionen, der behandler de enkete liste-elementer ser således (uddrag):

```
void printAsmElm (asElm *e)
{
  if (!printComment && e->slettet) return;
  if (e->slettet)
    fprintf(printAsmFile,"###");
  switch (e->kind)
  {
    case akComment: if (printComment) printAsmComment(e);  break;
    case akDef:     printAsmDef(e);                 break;
    case akOffset:  printAsmOffset(e);              break;
    case akLabel:   printAsmLabel(e);               break;

    case akPush:    printAsmInstr1(e);              break;
    case akMove:    printAsmInstr2(e);              break;
    case akJump:    printAsmJump(e);                break;
    case akCall:    printAsmCall(e);                break;
    case akReturn:  printAsmReturn(e);              break;
    case akCltd:    printAsmCltd(e);                break;
```

```
    default:
            fprintf(msgFile, "Fejl: Assembler element mangler def.\r\n");
  }
}
```

printAsmInstr2

Funktionen, der behandler normale instruktioner med 2 operanter ser således:

```
void printAsmInstr2 (asElm *e)
{
  fprintf(printAsmFile,"  %s  ",getInstrName(e));
  printAdr (e->val.eInstr2.fAdr);
  fprintf(printAsmFile,",");
  printAdr (e->val.eInstr2.tAdr);
  printAsmComment (e);
}
```

8.2 Afprøvning

En omfattende afprøvning kan findes i mit bilag C, hvor jeg har testet de fleste elementer af med 23 egne testprogrammer, herunder linkede lister og komplicerede datastrukturer med array i flere dimensioner. Desuden dokumenteres også en test af de programmer, der var lagt på IMADA, i bilag D.

Alle disse test viser, at compileren virker som den skal.

Der er en enkelt ting som ikke er testet og derfor heller ikke viser en fejl. Det er, at jeg ikke har taget højde for at statisk binding af data, når der kaldes til en indre funktion til en ydre og den indre efterfølgende referere til statiske data.

Dette skal løses ved at referencer til andre scope (frames) kopieres over på stakken i prologen og lokal-tilgang herefter sker via den kopierede reference i stedet for den en global.

Eksempel på emit ud fra alle tidligere faser

Jeg har valgt at medtage nedenstående eksempel, som er blevet anbefalet og stammer fra eksemplerne lagt ud fra imada. Dette findes også i bilag D.

Jeg vil desuden kort omtale de test, jeg selv har udarbejdet.

test_code_sdu_Factorial.tony

```
001: func factorial(n: int): int
002:    if (n == 0) || (n == 1) then
003:       return 1;
004:    else
005:       return n * factorial(n-1);
006: end factorial;
007:
008: write factorial(5);
```

consol output fra kørsel af oversat program

```
120
```

test_code_sdu_Factorial.s

Assemblerkoden er gennereret med kommentarer (kan slås fra med –noComment)

```
# Oversat fra TONY
# Compiler skrevet af Bjørk Boye Busch - bjbu@tietgen.dk

.data

_int_format:
.ascii "%d\n\0"
_zerodivide_format:
.ascii  "runtime fejl: nul-division i linie %d\n\0"
_outofmemory_format:
.ascii  "runtime fejl: out of memory i linie %d - heapsize(10000000)\n\0"
_nonpositivelements_format:
.ascii  "runtime fejl: array <= 0 elementer i linie %d\n\0"
_indexunderflow_format:
.ascii  "runtime fejl: index underloeb i linie %d\n\0"
_indexoverflow_format:
.ascii  "runtime fejl: index overloeb i linie %d\n\0"
.align  4
_scopeadr_0:  #frame-adr. til scopelevel
.long -1  #adr. til lokale data paa scope-level 0
_scopeadr_1:  #frame-adr. til scopelevel
.long -1  #adr. til lokale data paa scope-level 1
_myheapfreeadr:
.long  _myheapspace  #adresse på ledig heap-hukommelse
_myheapspace:
  .space  10000000  #heap-space
_aftermyheapspace:
.long  0  #4 byte ekstra da længde gemmes før test

.text

.globl _main  ##entry for windows
.globl main  ##entry for linux

#Data adresser defineret i dette scope


_main:  #entry for windows
main:  #entry for linux
  pushl  %ebp  #save ebp
  movl  %esp,%ebp  #set frame ptr
  movl  %ebp,_scopeadr_0  #gem frame ptr til andre scopes
  subl  $0,%esp  #reserver memory for lokale data
#main insructions start
#8:write
  pushl  %edi  #8:save edi
  subl  $4,%esp  #8:reserver memory for parameterblok
  movl  %esp,%edi  #8:etabler base-reference til parametre
  subl  $8,%edi  #8:juster som ebp efter kald
  movl  $5,%eax  #8:num
  movl  %eax,n_2(%edi)  #8:gem parameter paa stak
  call  factorial_1  #8:aktiver funktion
  addl  $4,%esp  #8:frigiv parameterblok
  popl  %edi  #8:restore edi
  pushl  %eax
```

```
  pushl  $_int_format
  call  printf
  addl  $8,%esp  #Fjern parametre igen

#main insructions slut
  xorl  %eax,%eax  #afslut normal med returværdi 0
main_end:
  movl  %ebp,%esp  #restore stak ptr
  popl  %ebp  #restore caler frame-ptr
  ret
#--------------------------
_indexunderflow:
  pushl  %eax
  pushl  $_indexunderflow_format
  call  printf
  addl  $8,%esp  #Ryd op paa stak
  movl  $2,%eax  #Returvaerdi 2
  jmp  _exit

_indexoverflow:
  pushl  %eax
  pushl  $_indexoverflow_format
  call  printf
  addl  $8,%esp  #Ryd op paa stak
  movl  $2,%eax  #Returvaerdi 2
  jmp  _exit

_zerodivide:
  pushl  %eax
  pushl  $_zerodivide_format
  call  printf
  addl  $8,%esp  #Ryd op paa stak
  movl  $2,%eax  #Returvaerdi 2
  jmp  _exit

_nonpositivelements:
  pushl  %eax
  pushl  $_nonpositivelements_format
  call  printf
  addl  $8,%esp  #Ryd op paa stak
  movl  $2,%eax  #Returvaerdi 2
  jmp  _exit

_outofmemory:
  pushl  %eax
  pushl  $_outofmemory_format
  call  printf
  addl  $8,%esp  #Ryd op paa stak
  movl  $2,%eax  #Returvaerdi 2
  jmp  _exit

_exit:
  movl  _scopeadr_0,%ebp  #restore frame ptr til main scope
  movl  %ebp,%esp  #restore main start stak ptr
  popl  %ebp
  ret  #return fra main
#--------------------------

#1:#function factorial_1
```

```
#Parameter adresser i dette scope
.equ  n_2,8  #1:int lokal adr. på ebp

#Data adresser defineret i dette scope

factorial_1:
  pushl  %ebp  #save ebp
  movl  %esp,%ebp  #set frame ptr
  movl  %ebp,_scopeadr_1  #gem frame ptr til andre scopes
  subl  $0,%esp  #reserver memory for lokale data
  pushl  %ebx
  pushl  %esi
  pushl  %edi
#funktion instructions start
#5:if-then-else
  movl  n_2(%ebp),%eax  #2:Load lokal variabel
###  movl  $0,%ebx  #num
###  cmpl  %ebx,%eax  #sammenlignings exp
  cmpl  $0,%eax  #optimized cmp
  je  _equation_2_true
  xorl  %eax,%eax  #2:exp false
  jmp  _equation_2_end
_equation_2_true:
  movl  $1,%eax  #2:exp true
_equation_2_end:
  pushl  %eax  #2:gem 1. operant eax
  movl  n_2(%ebp),%eax  #2:Load lokal variabel
###  movl  $1,%ebx  #num
###  cmpl  %ebx,%eax  #sammenlignings exp
  cmpl  $1,%eax  #optimized cmp
  je  _equation_3_true
  xorl  %eax,%eax  #2:exp false
  jmp  _equation_3_end
_equation_3_true:
  movl  $1,%eax  #2:exp true
_equation_3_end:
  movl  %eax,%ebx  #2:aflever i 2. operant ebx
  popl  %eax  #2:reetabeler 1. operant eax
  orl  %ebx,%eax  #2:||
  orl  %eax,%eax  #test for false
  je  _if_1_else  #false -> hop

#3:return
  movl  $1,%eax  #3:num
  jmp  factorial_1_end  #return med vaerdi i eax

  jmp  _if_1_end  #slut if
_if_1_else:

#5:return
  movl  n_2(%ebp),%eax  #5:Load lokal variabel
  pushl  %eax  #5:gem 1. operant eax
  pushl  %edi  #5:save edi
  subl  $4,%esp  #5:reserver memory for parameterblok
  movl  %esp,%edi  #5:etabler base-reference til parametre
  subl  $8,%edi  #5:juster som ebp efter kald
  movl  n_2(%ebp),%eax  #5:Load lokal variabel
###  movl  $1,%ebx  #num
```

```
### subl %ebx,%eax #-
 decl %eax #optimized add/sub
 movl %eax,n_2(%edi) #5:gem parameter paa stak
 call factorial_1 #5:aktiver funktion
 addl $4,%esp #5:frigiv parameterblok
 popl %edi #5:restore edi
 movl %eax,%ebx #5:aflever i 2. operant ebx
 popl %eax #5:reetabeler 1. operant eax
 imull %ebx #5:*
 jmp factorial_1_end #return med vaerdi i eax

_if_1_end:

#funktion instructions slut
factorial_1_end:
 popl %edi
 popl %esi
 popl %ebx
 movl %ebp,%esp #restore stak ptr
 popl %ebp #restore caler frame-ptr
 ret
#---------------------------
#end of program
```

Eksemplet dokumenterer, at jeg kan generere assemblerkode, som ved afvikling giver det rigtige resultat.

test_code_exp1.tony

Denne test viser, at expressions med int-typen bliver behandlet korrekt.

test_code_exp2.tony

Denne test viser, at expressions med bool-typen bliver behandlet korrekt.

test_code_assign1.tony

Denne test viser, at assignment med int-typen bliver behandlet korrekt.

test_code_suker.tony

Denne test viser, at udvidelserne med syntaktisk sukker bliver behandlet korrekt.

test_code_if.tony

Denne test viser, at varianterne af if-konstruktion og expressions af bool-typen bliver behandlet korrekt.

test_code_while1.tony

Denne test viser, at varianterne af while-konstruktion og expressions af bool-typen bliver behandlet korrekt.

test_code_while2.tony

Denne test viser, at varianterne af while-konstruktion og expressions af bool-typen bliver behandlet korrekt, idet der her bliver andre optimeringsmønstre.

test_code_for1.tony

Denne test viser, at varianterne af for-konstruktion bliver behandlet korrekt.

test_code_array1.tony
Denne test viser, at simpelt array af int bliver behandlet korrekt.

test_code_array2.tony
Denne test viser, at array af referencer til int-array (2-dim) bliver behandlet korrekt.

test_code_array2Global.tony
Denne test viser, at brug af non-lokal array af referencer til int-array (2-dim) bliver behandlet korrekt.

test_code_recordListe.tony
Denne test viser, at brug af referencer i record bliver behandlet korrekt, samt at ”kaskade strukturer” uden array bliver behandler korrekt. Eksemplet er en linket liste.

test_code_arrayRecord1a.tony
Denne test viser, at record af array af record direkte på heap bliver behandler korrekt.
Record bliver her allokeret som array-element uden new.
Dette eksemplet skal køres UDEN compiler direktiv -noRecordArray sat.

test_code_arrayRecord1b.tony
Dette eksempel svarer til ovenstående, men MED compiler direktiv -noRecordArray sat.
Array af record vil derfor være referencer til record og disse skal allokeres hver især med new. Eksemplet dokumentere at denne allokeringsmåde også virker som den skal.

test_code_recordArray1a.tony
Denne test viser, at array af record direkte på heap bliver behandler korrekt.
Record bliver her allokeret som array-element uden new.
Dette eksempel skal køres UDEN compiler direktiv -noRecordArray sat.

test_code_recordArray1b.tony
Dette eksempel svarer til ovenstående, men MED compiler direktiv -noRecordArray sat.
Array af record vil derfor være referencer til record og disse skal allokeres hver især med new. Eksemplet dokumentere at denne allokeringsmåde også virker som den skal.

test_code_recordArray2a.tony
Denne test viser liste med array af record direkte på heap bliver behandler korrekt.
Record bliver her allokeret som array-element uden new.
Dette eksemplet skal køres UDEN compiler direktiv -noRecordArray sat.

test_code_recordArrayCyklisk.tony
Denne test er et sofistikeret eksempel på strukturer med flere niveauer, hvor nogle er array bliver behandler korrekt.
Dette eksemplet skal køres UDEN compiler direktiv -noRecordArray sat.

test_code_call.tony
Denne test viser ar recursion med reference til non-lokale data bliver behandler korrekt. Eksemplet får ikke vist problemet, som der ikke er taget højde for, med brug af recursion til scope, hvor der også bruges non-lokale data. Det afslører således ikke at der her vil være en fejl.

test_code_indexHigh.tony
Denne test viser, at runtimefejl med for højt index til array bliver behandlet korrekt.

test_code_indexLow.tony
Denne test viser, at runtimefejl med for lavt index til array bliver behandlet korrekt.

test_code_heapend.tony
Denne test viser, at runtimefejl med for out of memory på heap bliver behandlet korrekt.

test_code_zerodiv.tony
Denne test viser, at runtimefejl med nul-division bliver behandlet korrekt.

9 Konklusion

Mit projekt er som det gerne skulle fremgå af rapporten og de medfølgende bilag lykkedes. Jeg har opfyldt de krav der er blevet stillet, hvilket gerne skulle fremgår af rapporten. De udvidelser jeg havde planlagt er også blevet gennemført og ligeledes dokumenteret. Jeg har fået fremstillet en tony-compiler, der er testet grundigt igennem og virker med enkelte undtagelser (se nedenstående) efter hensigten og skulle derfor have opfyldt målet.

Kendte fejl / uhensigtsmæssigheder

Jeg har i rapporten nævnt at, at jeg ikke har fået taget højde for recursion til funktioner i et ydre scope og samtidig direkte tilgang til denne funktions lokale data (statisk link), ved tilbagevenden til den indre funktion. Jeg har ikke testet for det, men indset det og anvist hvad en løsning vil kræve i rapporten afsnit 8.2 (scope-frame-pointerene skal i en funktions prolog kopieres til dennes activation-record på stakken).

Jeg har efter denne rapport næsten var færdigskrevet checket op mod de test, der er blevet brugt til den udvidede test på imada.

Fejl fundet i forhold til ekstra-programmerne på imada

ErrFuncParam1.tony

Programmet afslører, at der er fejl ved kald af funktion med en parameter uden angivelse af argumenter.

Uddrag fra funktionen symbolsCheckAct_list fra tonySymbols_Check.c

```
  if (wPar_decl_list->kind == kPar_decl_list_Empty &&
      wAct_list->kind == kAct_list_Empty) {
    return;
  }
  /* check af parametre antal */
  wVar_decl_list = wPar_decl_list->val.eList.var_decl_list;
  while ( wVar_decl_list->kind == kVar_decl_list_List ) {
    ++antalVar;
    wVar_decl_list = wVar_decl_list->val.eList.var_decl_list;
  }
  wExp_list = wAct_list->val.eList.exp_list;
  while ( wExp_list->kind == kExp_list_List) {
    ++antalExp;
    wExp_list = wExp_list->val.eList.exp_list;
  }
```

Som det fremgår, har jeg ikke her taget højde for, at kind på enten parameter- eller action-listen er af den tomme type, hvorfor compileren vil fejle, hvis der er en liste på den ene og ikke den anden.

Ovenstående skulle have set sådan ud:

```
  if (wPar_decl_list->kind == kPar_decl_list_Empty &&
      wAct_list->kind == kAct_list_Empty) {
    return;
  }
  if (wPar_decl_list->kind != kPar_decl_list_Empty {
     /* check af parametre antal */
     wVar_decl_list = wPar_decl_list->val.eList.var_decl_list;
     while ( wVar_decl_list->kind == kVar_decl_list_List ) {
       ++antalVar;
       wVar_decl_list = wVar_decl_list->val.eList.var_decl_list;
     }
  }
```

```
  if (wPar_decl_list->kind == kPar_decl_list_Empty &&
      wAct_list->kind == kAct_list_Empty) {
    wExp_list = wAct_list->val.eList.exp_list;
    while ( wExp_list->kind == kExp_list_List) {
      ++antalExp;
      wExp_list = wExp_list->val.eList.exp_list;
    }
  }
```

RedefinesReturnType.tony

Programmet afslører, at jeg anvender den forkerte symboltabel i forbindelse med funktions-hovedet.
Funktion symbolsCheckFunction fra tonySymbols_Check.c ser sådan ud:

```
void symbolsCheckFunction (sFunction *s)
{
  /* gem adresse på aktuel symboltabel */
  SymbolTable *saveSymbolTable = checkSymbolTable;
    symbolsCheckBody(s->val.eFunction.body);
  /* reetabler adresse på aktuel symboltabel */
  checkSymbolTable = saveSymbolTable;
}
```

Her skal head checkes før der skiftes scope på symboltabellen:

```
  symbolsCheckHead(s->val.eFunction.head);
  /* opsæt adresse til symboltabel */
  checkSymbolTable = s->symbolTable;
```

Uhensigtsmæssigheder i forhold til ekstra-programmerne på imada

ErrNullPointer.tony

Programmet giver runtime-fejl, men dette er forventet, idet jeg ikke har valgt at checke for null-pointer og manglende initiering.

TypesRedefine.tony

Programmet giver en type-check-fejl, men dette er forventet, idet jeg har fastlagt i mit type-check at:

```
type a = int;
type b = int;
```

defineres som forskellige typer og assignment mellem disse ikke accepteres.

Jeg kunne have valgt at slægge på typekravet ved at bruge min slutType til check, men det er ligeså rimeligt at fasthold typekravet og kræve explictit typecast og så udvide med en sådan sproglig konstruktion.

MultiPasses.tony

Programmet giver en type-check-fejl, men dette er forventet, idet jeg har fastlagt i mit type-check at:

```
type a = b;
type b = x;
```

kun er gyldigt hvis x ikke er en selvdefineret type, og hermed kun accepteres hvis x er int, bool, array of.. eller record of..

Hertil kommer at der også i dette eksempel er tale om forskellige typer jævnfør ovenstående program.

Dato : 2005-05-16 Bjørk Busch