Bachelorprojekt i Oversætterkonstruktion

Kitty

Maj 2019

Rapport fra gruppe 2:

Andreas Holm Jørgensen, andjo16.
Jeff Gyldenbrand Skov Jørgensen, jegyl16.
Mads Kempf, makem16.

Indhold

1 Indledning

Oversætteren beskrevet i denne rapport kan bruges til at oversætte programmer, der er skrevet i Kitty-sproget, der defineres senere i denne tekst, til Intel AT&T GAS assembler kode. Vores mål er at udbygge oversætteren til at kunne håndtere højniveau programmeringsopgaver som f.eks. at løse optimeringsproblemer. Der vil bl.a. blive refereret til *Knapsack* problemet et par gange.

Denne rapport vil gennemgå følgende emner:

- Defintion på kitty sproget med tilhørende grammatik, samt hvordan den skannes med `flex` og behandles i `bison`.
- Opbygning af det abstrakte syntakstræ samt udlugning af programmer med fundamental semantiske fejl.
- Opbygning af symboltabeller som holder styr på brugerens elementer, scopes mm.
- Typetjekning, hvor korrekt brug af programmets typer verificers ved at gennemløbe det abstrakte syntakstræ flere gange.
- Kodegenerering; forvandling af abstrakt syntakstræ til en hægtet listestruktur, der repræsenterer den assembler kode, der skal udskrives til sidst.
- Liveness-analyse og registerallokering, hvor så mange hukommelsestilgange som muligt forsøges fjernet, ved at lægge data i registre.
- Kighulsoptimering hvor unødig kode fjernes.
- Emitting af den endelige assemblerkode.

1.1 Afklaringer

Vi har bestemt, at alle binære operatorer er venstre associative. Vi har også defineret alle præcedens-regler for vores forskellige operatorer.

For `if-then-else`-handlinger har vi præciseret, hvordan parseren skal håndtere indlejrede `if-then`- og if-then-else-handlinger. Dette er gjort ved at lade `then` og `else` associere til højre, således at et `else` altid bindes til den seneste og dermed mest indlejrede `then`-del, der ikke selv har en `else`-del.

1.2 Begrænsninger

For korrekt at kunne håndtere konflikten imellem absolut værdi og den binære || operand har vi lavet den restriktion at || ikke kan optræde inde i et absolut værdi felt. At tage kardinaliteten af bolske udtryk giver generelt ikke meget mening, men det er ikke ellers forbudt at tage kardinaliteten af boolske udtryk i oversætteren.

1.3 Sprogudvidelser

I relation til det udkast til et basis-sprog, vi blev givet ved projektets begyndelse er der tilføjet `break` og `continue` som terminaler til sproget. Disse to kan hver for sig placeres som en handling i sig selv, men kan kun optræde i en *while*-løkke. `break` bryder ud af den nærmeste løkke. `continue` genstarter til næste iteration af den nærmeste løkke.

1.4 Implementationsstatus

Vi har implementeret en oversætter, der kan håndtere hele den grammatik vi beskriver. Vi har tilføjet registerallokering og liveness-analyse samt kighulsoptimering.

Grund-oversætteren virker godt; vi har skrevet i omegnen af 230 enheds-tests, som hver især tester en specifik del af kitty sproget. Derudover er oversætteren blevet testet med et par enkelte større programmer. Der er ingen større problemer med oversætteren, men eventuelle fejl og mangler vil blive beskrevet i det dertil indrettede kapitel 9.

Register allokering virker godt. Vi kan se en betydelig hastighedsforøgelse når registerallokeringen effektiviserer brugen af registre i stedet for hukommelse. Alle tests er kørt både med og uden register allokering og de virker i begge tilfælde.

Kighulsoptimeringen er implementeret med et meget begrænset antal mønstre, der senere kan udvides hvis dette ønskes. Implementationen ser ud til at virke, idet den kører vores tests fint. Vi oplever en lille forøgelse i hastighed når vi bruger kighulsoptimering sammenlignet med, hvis det udelades.

Vi har også implementeret køretids kontrol for derefereing af *null*-værdier, out-of-bounds, out-of-memory, negativ allokering og division med 0. Alle disse er testet ved at køre vores tests både med og uden runtime kontrol.

I parseren er der gjort forsøg på at implementere muligheden for at flere fejl kan findes i samme kørsel af oversætteren. Hvis parseren finder en fejl i en handling så reduceres denne handling til en fejl og der fortsættes med næste handling.

I de senere faser, vil oversætteren dog afslutte lige så snart en enkelt fejl findes. I forbindelse med typetjekning gemmes typefejl dog faktisk i det abstrakte syntakstræ, og der tjekkes i de fleste tilfælde for om en sådan fejl findes et givent sted i programmet. Det vil derfor være forholdsvist simpelt at ændre oversætteren til at kunne springe videre til næste udtryk eller handling når typefejl findes og evt. finde flere type-fejl i samme kørsel på den måde.

2 Parsning og abstrakte syntakstræer

2.1 Grammatikken

Sproget Kitty, som oversætteren skal kunne analysere og oversætte, er beskrevet i sin helhed via grammatikken herunder. Grammatikken har 17 non-terminaler angivet i $\langle\ \rangle$ og 48 terminaler markeret med fed font. Disse er bundet via 60 regler. Forskellige regler, der kan reduceres til den samme non-terminal, adskilles med '|'. En tom liste af terminaler og non-terminaler repræsenteres ved ϵ.

Startsymbolet i grammatikken er $\langle body \rangle$.

$\langle func \rangle$: $\langle head \rangle\ \langle body \rangle\ \langle tail \rangle$

$\langle head \rangle$: **func id (**$\langle par\_decl\_list \rangle$**)** : $\langle type \rangle$

$\langle body \rangle$: $\langle decl\_list \rangle \langle stmt\_list \rangle$

$\langle tail \rangle$: **end id**

$\langle type \rangle$: **id**
| **int**
| **bool**
| **array of** $\langle type \rangle$
| **record of** $\{\langle var\_decl\_list \rangle\}$

$\langle par\_decl\_list \rangle$: $\langle var\_decl\_list \rangle$
| ϵ

$\langle var\_decl\_list \rangle$: $\langle var\_type \rangle, \langle var\_decl\_list \rangle$
| $\langle var\_type \rangle$

$\langle var\_type \rangle$: **id :** $\langle type \rangle$

$\langle decl\_list \rangle$: $\langle decl \rangle \langle decl\_list \rangle$
| ϵ

$\langle decl \rangle$: **type id =** $\langle type \rangle$**;**
| $\langle func \rangle$
| **var** $\langle var\_decl\_list \rangle$**;**

$\langle stmt\_list \rangle$: $\langle stmt \rangle$
| $\langle stmt \rangle \langle stmt\_list \rangle$

$\langle stmt \rangle$: **return** $\langle exp \rangle$**;**
| **write** $\langle exp \rangle$**;**
| **allocate** $\langle var \rangle$**;**
| **allocate** $\langle var \rangle$ **of length** $\langle exp \rangle$**;**
| $\langle var \rangle$ **=** $\langle exp \rangle$**;**
| **if** $\langle exp \rangle$ **then** $\langle stmt \rangle$
| **if** $\langle exp \rangle$ **then** $\langle stmt \rangle$ **else** $\langle stmt \rangle$
| **while** $\langle exp \rangle$ **do** $\langle stmt \rangle$
| **continue;**
| **break;**

| $\{\langle stmt\_list \rangle\}$

$\langle var \rangle$: **id**
| $\langle var \rangle$**[**$\langle exp \rangle$**]**
| $\langle var \rangle$**. id**

$\langle exp \rangle$: $\langle exp \rangle$ **+** $\langle exp \rangle$
| $\langle exp \rangle$ **-** $\langle exp \rangle$
| $\langle exp \rangle \cdot \langle exp \rangle$
| $\langle exp \rangle$ **/** $\langle exp \rangle$
| $\langle exp \rangle$ **==** $\langle exp \rangle$
| $\langle exp \rangle \leq \langle exp \rangle$
| $\langle exp \rangle \geq \langle exp \rangle$
| $\langle exp \rangle < \langle exp \rangle$
| $\langle exp \rangle > \langle exp \rangle$
| $\langle exp \rangle$ && $\langle exp \rangle$
| $\langle exp \rangle$ || $\langle exp \rangle$
| $\langle term \rangle$

$\langle term \rangle$: $\langle var \rangle$
| **id (** $\langle act\_list \rangle$ **)**
| **(** $\langle exp \rangle$ **)**
| **!** $\langle term \rangle$
| | $\langle exp \rangle$ |
| **num**
| **true**
| **false**
| **null**

$\langle act\_list \rangle$: $\langle exp\_list \rangle$
| ϵ

$\langle exp\_list \rangle$: $\langle exp \rangle$
| $\langle exp \rangle, \langle exp\_list \rangle$

2.2 Brug af `flex` værktøjet

Skanningen af brugerens Kitty-program bliver gjort via værktøjet `flex` (fast lexical analyser generator), der genererer et `C`-program, som kan foretage leksikalsk analyse ud fra en specifikationsfil. Specifikationerne består af regulære udtryk og handlinger. Når skanneren finder et match på et regulært udtryk udfører den handlinger der passer til.

Vores implementation af specifikationen til `flex` ligger i `flex.l`-filen i `scan_parse` mappen. Vi benytter *start conditions* til at håndtere kommentarer. Hvis skanneren finder tegnene '(*', som indikerer start af flere-linje-kommentarer, så går den til `COMMENT` tilstanden, som sørger for at alt andet end tegnene '*)', som afslutter flere-linje-kommentarer, og '\n' bliver ignoreret. '\n' ignoreres ikke da vi gerne vil tælle linjetal. Når '*)' bliver læst vender scanneren tilbage til `INITIAL` tilstanden.

Vi kan håndtere indlejrede kommentarer ved hjælp af en tæller, der angiver antallet af indlejrede kommentarer. Derved er det kun tilladt at forlade kommentar-tilstanden når tæller værdien er 0. Almindelige en-linje-kommentarer håndteres på stort set samme måde som flere-linje kommentarer. Når '#' bliver læst af scanneren, går den til `ONELINECOMMENT` tilstanden som ignorerer alt andet end '\n' som får den til at vende tilbage til `INITIAL` tilstanden.

For at gøre det tydeligt, hvad der sendes fra `flex` til `bison` er det besluttet at `flex` kun kan returnere tokens defineret i `parser.y` og derfor ikke på noget tidspunkt direkte returnerer en karakter.

Som udgangspunkt er der et problem med den binære *eller* operator, $||$, og udtrykket for absolut værdi, $|expression|$, da begge disse bruger bar-symbolet. Et eksempel på et udtryk, som parseren har svært ved at genkende korrekt, er: $||a|+a|$, da $||$ i begyndelsen vil blive forstået som den logiske *eller*-operator. Løsningen på denne tvetydighed er implementeret ved at bemærke, at absolut værdi kun kan optræde som det første element i et udtryk og en *eller*-operator kun kan optræde efter og imellem udtryk. I skanneren er der lavet tre integers som holder styr på, hvilken situation vi er i. Disse beskriver fire mulige tilstande skanneren kan være i, når et $|$-symbol læses:

- Vi står i starten af et udtryk, hvor det kun kan være absolut værdi, der kan optræde.
- Vi står ikke i starten af et udtryk men er allerede inde i et absolut værdi felt: Hvis $|$ findes betyder det, at vi afslutter absolut værdi feltet.
- Vi står ikke i starten af et udtryk og er ikke inde i et absolut værdi felt: her kan der stå en *eller*-operator. Når den første $|$ af et *eller*-udtryk findes, sørges for at scanneren ved at den næste $|$, den finder hører til eller operatoreren.
- Den første del af en *eller*-operator er fundet og nu findes den anden del.

Med disse ændringer er det nu muligt korrekt at skanne og parse f.eks. $||a| + b|$. Det tilføjer dog den begrænsning at der ikke længere kan være en $||$-operator inde i et absolut værdi felt. Vi kan dog stadig have $\&\&$, og tage absolut værdi af bolske udtryk. Dette kan forbydes i typecheckeren men det har vi ikke gjort.

Der er defineret heltalskonstanter, der vedligeholder det nuværende linjenummer og karakterposition på en linje for at give bedre fejlmeddelelser. Disse værdier bruges af funktionen `yyerror` defineret i `parser.y` når skanneren eller parseren finder en syntaksfejl.

2.3 Brug af `bison` værktøjet

Til at udføre parsningen benyttes værktøjet `bison`, der hjælper med at genkende og afgøre om sekvenser af tokens fra den lexikalske analyse er tilladt i sproget. Parseren er implementeret i filen `parser.y`.

Alle reglerne i grammatikken fra afsnit 2.1 er angivet sidst i `parser.y`. Da hver non-terminal kan udvides via en af reglerne for den pågældende non-terminal, ønsker vi at kunne samle og gemme al relevant information, der er associeret med højresiden af den brugte regel. Dette gøres ved at bruge `bison`'s parserstak. For hver non-terminal, der er fundet og lagt på stakken, gemmes relevant information fra højresiden af den regel, der er brugt til at reducere med. Informationen gemmes i en C-struktur, som derefter peges til fra det indeks på parserstakken, hvor den dannede non-terminal er lagt, således at al den information der ønskes kan findes via en pointer på parser-stakken. Forskellige non-terminaler benytter forskellige strukturer for at gemme den relevante data. Derfor er parser-stakkens type en union med en C-variabel for hver af disse strukturer. Således kan `bison` vælge den relevante variabel med den korrekte type til at gemme associeret data, når en given regel bruges til at reducere til en given non-terminal.

Hver non-terminals associeret type på parser-stakken er angivet i forbindelse med definitionen af variablen, via nøgleordet `%type`. Det er også i forbindelse med denne defintion at non-terminalerne defineres.

Alle terminalsymboler er defineret via nøgleordet `%token`. De fleste terminalsymboler har ikke nogen associeret værdi, men de to symboler til at genkende et heltal og et id (det sidste for variabler, funktioner eller brugerdefineret typer) hhv. kaldet tINT og tID bruger variablerne uint og uid fra parserstakkens union med typerne `int` og `char*`. På den måde kan værdien af tal og navnet på id'er sendes fra skanneren til parseren via disse variabler.

Da det er tilladt i grammatikken at alternere mellem udtryk og operatorer giver det anledning til skift/reducér-konflikter i `bison`. Betragt følgende streng:

$$\langle expression \rangle \texttt{ tPlus } \langle expression \rangle \cdot \texttt{ tTIMES},$$

hvor de to udtryk og operatoren før '$\cdot$' er blevet skiftet på stakken, mens operatoren efter '$\cdot$' er det næste der læses på input. Her vil `bison` ikke kunne afgøre om operatoren skal skiftes på stakken eller om de øverste tre elementer på stakken skal reduceres først. Konflikterne er løst ved at angive hvor stærkt de enkelte operatorer binder. Det er valgt at den relative bindingstyrke er som angivet i listen herunder, hvor første element på listen binder stærkest og sidste element binder svagest. Det er desuden valgt at alle operatorer associerer til venstre, så udtrykket $1 + 2 + 3$ bliver evalueret som $((1 + 2) + 3)$.

1. !
2. $\times$, $\div$
3. $+$, $-$
4. $<$, $>$, $\leq$ $\geq$
5. $==$, $\neq$
6. &&

7. ||
8. =

Resultatet af at benytte `bison` til at analysere programmet, er at opbygge et Abstrakt Syntaks Træ (AST) der giver en fuld repræsentation af programmet, i et format der er lettere at arbejde med, end koden selv. Teknikken til at bygge træet beskrives kort her, men strukturen og opbygningen af træet beskrives dybere i afsnit 2.4.

Bison lader os lave en eller flere sideeffekter ved siden af selve reduktion, når en regel bruges. Dette benyttes til at lave en ny knude i AST'et med den relevante information (knuder der allerede findes på stakken og som bruges i reduktionen), som børn af denne knude via en konstruktør defineret i forbindelse med implementationen af AST'et. Der er defineret en konstruktør for næsten hver regel i grammatikken. Det betyder, at der for non-terminaler med flere regler, findes mere end en konstruktør til at konstruere og gemme dataene i denne non-terminals struktur. Dette er gjort sådan, da forskellige regler giver anledning til forskel i den data, der er til rådighed og som skal gemmes i strukturen for den pågældende non-terminal type.

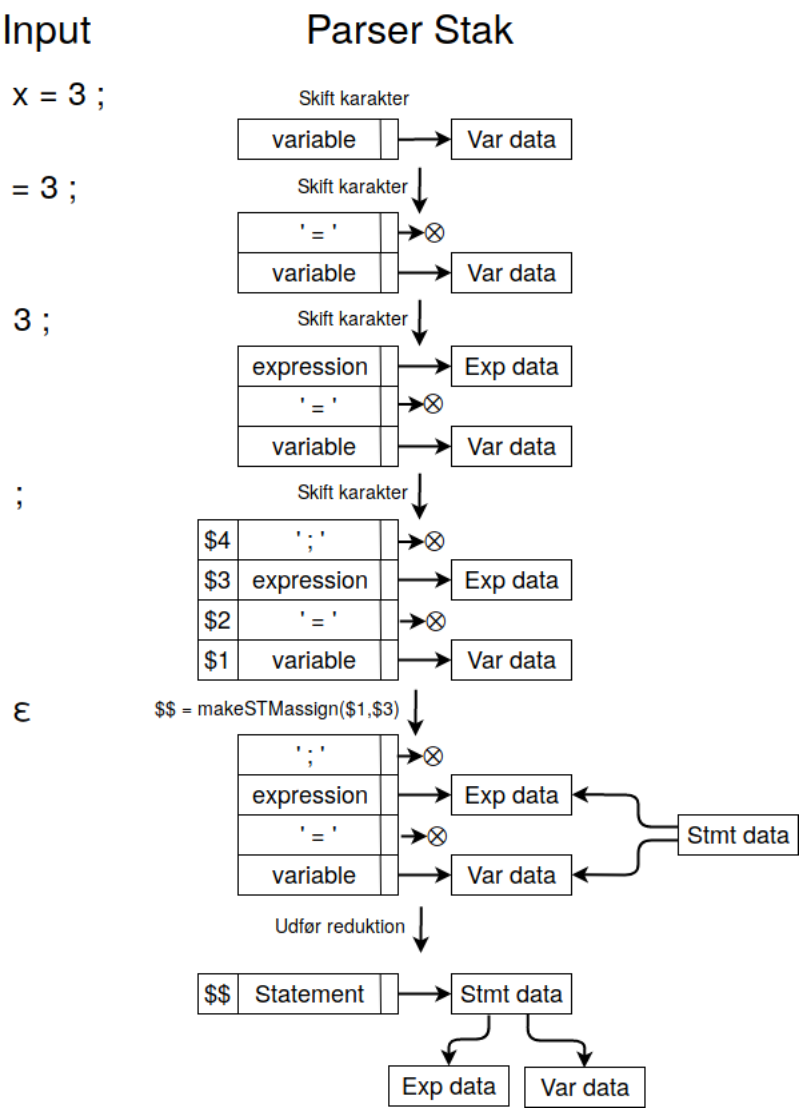
Eksempelvis har non-terminalen `stmt` der repræsenterer en handling (eng. statement) ni forskellige regler, hvilket medfører at der skal bruges ni forskellige konstruktører til at lave den data, der hører til en `stmt` non-terminal. Hvis vi ser på et udsagn for tildeling til en variable, så er der behov for at gemme to elementer i dette udsagn. Det ene er variablen, der skal have værdien, og det andet er det udtryk, der angiver den værdi, der skal lægges ind i variablen. Derfor tager handlings-konstruktøren `makeSTMassign` to argumenter, som begge to findes på parser-stakken indekseret relativt efter toppen af stakken og elementerne i reglen. Konstruktøren sørger for at gemme disse to elementer i den struktur, der repræsenterer en handling og returnerer så denne struktur, som efterfølgende lægges på toppen af parser-stakken i `bison`. Herfra kan handlings-strukturen findes og gemmes i en anden struktur, når `bison` finder at handlingen indgår i en anden regel og kan reduceres. Ideen beskrevet herover findes illustreret i figur 1.

Som ved operatorer, vil skift/reducer-konflikter også opstå ved `if-then-else`- handlinger. Betragt:

$$\textbf{if } \langle expression \rangle \textbf{ then } \langle stmt \rangle \cdot \textbf{else},$$

hvor det sidste **else** ligger i input, mens resten er på stakken. `bison` kan som udgangspunkt ikke afgøre, om der skal reduceres til en `if-then`-handling eller skiftes først, så der fås en `if-then-else`-handling. Dette er løst ved at lade terminalerne **tTHEN** og **tELSE** associere til højre, således at situationen beskrevet herover altid resulterer i en `if-then-else`-handling. Det betyder, at hvis programmøren ønsker at lægge en `if-then`-handling ind i `then`-delen af en `if-then-else`-handling, så skal indholdet i `then`-delen være omgivet af krøllede parenteser.

Udover de i grammatikken angivne regler (i afsnit 2.1), er der tilføjet en ekstra regel i `bison` (angivet først i afsnittet med regler), der oversætter start-symbolet `program` til symbolet `body`, der er den ydre ramme for hele programmet. Når denne regel bliver brugt, er hele programmet blevet parset og pointeren til den struktur der gemmer den overordnede program-krop flyttes fra parser-stakken til variablen `theexpression`.



Figur 1: Figuren viser, hvordan en handling findes og reduceres, således at et AST for handlingen bliver skabt. Hvert element på stakken er en terminal eller non-terminal, med associeret data, hvis relevant. I eksemplet er det kun non-terminalerne, der har associeret data (det er ikke illustreret hvordan denne data findes). Elementerne på stakken indekseres med '$' som vist og $$ angiver det indeks, hvor resultatet af reduktionen lægges (toppen).

2.4 Abstrakte syntakstræer

For hver non-terminal - der fremkommer på venstresiden af reglerne i grammatikken for Kitty-sproget beskrevet i afsnit 2.1 - er der lavet en struktur, som kan indeholde relevante elementer fra den respektive højreside af reglen. Hver gang en regel bruges til at reducere en mængde af terminaler og non-terminaler instantieres en ny struktur der indeholder relevant information fra højresiden af reglen. Instantieringen sker via en konstruktør, der tager en mængde af argumenter, svarende til den information fra højresiden af en regel, der skal gemmes, når højresiden reduceres væk. Hver af disse strukturer repræsenterer en knude i AST'et. Med andre ord så repræsenterer hver knude i træet en regel, hvor felterne i knuden indeholder terminalsymboler og/eller peger videre på andre knuder, der repræsenterer non-terminalsymboler. Non-terminalerne, der udgør venstresiderne af reglerne i grammatikken, vil blive betegnet som en knude i resten af afsnit 2. Eksempelvis bruges den første regel fra afsnit 2.1 til at danne en knude for funktioner, `function`, der som børn har referencerne `head`, `body` og `tail`. Disse børn refererer yderligere til deres respektive børn afhængig af hvad kitty-programmet gør.

Som udgangspunkt er der en konstruktør for hver regel. Dog gælder der ifølge grammatikken for `Kitty` at `par_decl_list` skal kunne tage imod `var_decl_list` og en tom streng. Det er valgt at lade `bison` håndtere denne forskel, ved blot at videregive værdien `NULL` som parameter til konstruktøren i stedet for at benytte en seperat konstruktør, da indeholdet i knuden ikke ændres af dette. Det samme gør sig gældene for `decl_list` og `act_list`. Således er det kun nødvendig med en enkelt konstruktør for hver af disse knuder.

2.4.1 AST data

Nogle knuder kan indeholde forskellige typer og mængder af data afhængig af hvilken regel og dermed konstruktør, der er brugt til at instantiere knuden. Dette er håndteret via `unions` i `C`. Eksempelvis kan `TYPE`-strukturen som ses i listing 1 have slagsene: `id`, `int`, `bool`, `array` og `record`. Feltet `kind` uden for denne union bruges til at indikere hvilken del af unionen, der aktiv for en given knude. Hver konstruktør for en given regel i grammatikken sætter `kind`-feltet forskelligt. For slagsene `id`, `array`, `record` er det nødvendigt at gemme information som navnet på id'et, listens indhold eller variablerne i recorden, som hver især kan gemmes i felterne i knudens union.

```
1  typedef struct TYPE {
2    int lineno;
3    Typekind kind;
4    SymbolTable *scope;
5    union{
6      char *id;
7      struct TYPE *arrayType;
8      struct VAR_DECL_LIST *vList;
9    } val;
10 } TYPE;
```

Listing 1: Indeholdet af en type-konstruktør

En anden struktur der er relevant at vise er `EXP`-strukturen, som ses i listing 2. Denne knude har et `kind`-felt, der kan bruge til at angive præcis hvilken operator den består af.

```
1  typedef struct EXP {
2    int lineno;
3    enum {termK, minusK, plusK, timesK, divK,\
4          leK, eqK, geK, greatK, lessK, neK, andK, orK} kind;
5    Typekind typekind;
6    TYPE *type;
7    union {
8      struct {struct EXP *left; struct EXP *right;} binOP;
9      struct TERM *term;
10   } val;
11 } EXP;
```

Listing 2: Indeholdet af en udtryks-kontruktør

En illustrativ men simplificeret repræsentation af sammenhængen mellem `EXP`-, `TERM`- og `STATEMENT`-knuderne, for udtaget af et Kitty-program vist i listing 3, kan ses i figur 2, hvor en `if_then_else` handling, har et udtryk og to handlinger som børn. Udtrykket indeholder selv to udtryk; et venstre og højre barn, bundet sammen af en binær operator. Begge udtryk har en `term` som barn, som henholdsvis indeholder et heltal `num` x og y.

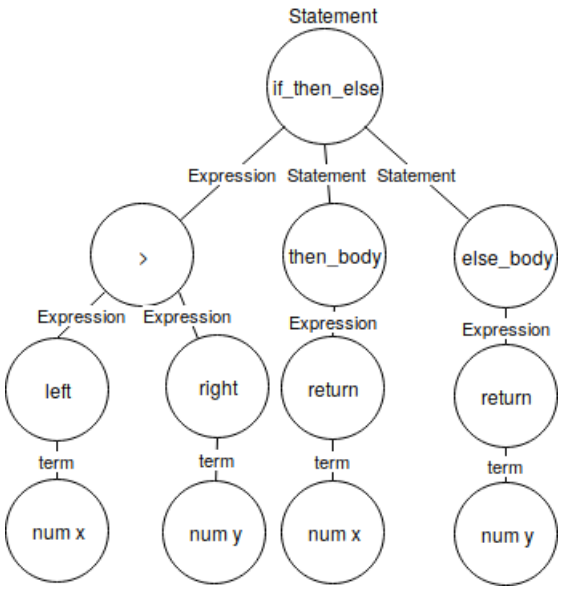
```
1  if(x>y)
2      then
3          return x;
4      else
5          return y;
```

Listing 3: Indeholdet af en udtryks-kontruktør

2.5 Udlugning

I udlugerfasen verificeres at alle funktionsdefinitioner har identiske `head` og `tail`-id. Samtidig verificeres der at alle mulige kørselsstier i funktionerne har retur-handlinger. Hvis ikke, udskrives der en fejlbesked. Derudover udskrives der en advarsel for utilgængelige handlinger, der kommer umiddelbart efter en retur-handling.

Først og fremmest kigger udlugeren ned gennem handlings-listen af `BODY`, hvis en retur-handling opdages melder udlugeren fejl tilbage med det samme, med et -1, og afslutter, da vi ikke ønsker at brugeren skal kunne afgøre returværdien for programmet. Dennne skal være 0, eller en veldefineret fejlkode. Ved fejl skrives en fejlbesked til `stderr` med linjenummer for hvor fejlen er opstået i brugerens kode og weederen returnere straks derefter. Hvis der derimod ikke findes en retur-handling i main-scopet og der dermed ikke er nogle fejl, vil udlugeren kigge ned gennem erklæringslisten af dette scope. Den første funktion, der findes, bliver kontrolleret for `HEAD`- og `TAIL`-id. Hvis et manglende match opdages vil udlugeren ligeledes melde fejl tilbage og afslutte. Hvis et match mellem id'er opdages, tjekkes først for om alle kørselsstier i funktionen når en retur-handling. Er det tilfældet tjekkes rekursivt for id'erne af funktioner erklæret

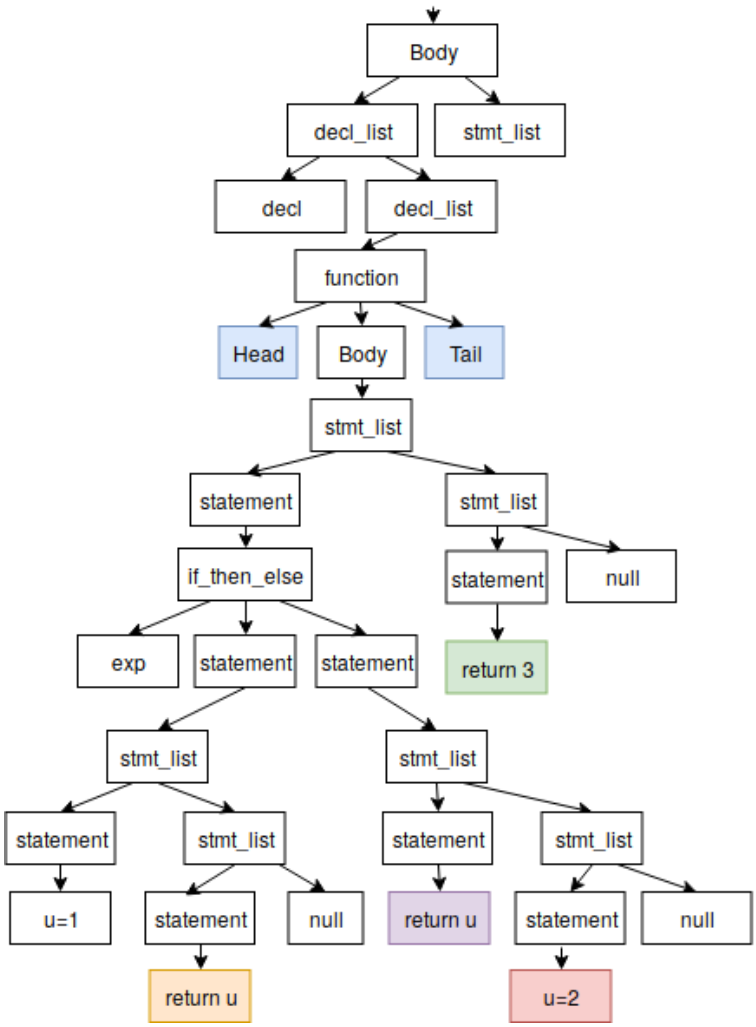


Figur 2: Eksempel på et abstrakt syntakstræ

inde i denne funktion. For at analysere om funktionen har de fornødne retur-handlinger gennemsøges handlingslisten i funktionens krop for returtyper. Hvis der opdages en `return`-handling, returneres 0 for at angive at funktionen returnerer korrekt. Hvis der derimod under gennemløbet opdages en `If-then-else`-handling, gennemløber udlugeren denne handlings, indbyggede handling(er) rekursivt. Dette er illustreret i figur 3, der angiver et lille kodeeksempel i `KITTY`-sproget og dets tilhørende træ. Udlugeren gennemløber `if-then-else`-handlingen og finder først en retur-handling i `then`-delen (markeret med gul), og ser, at der ikke findes yderligere handlinger i denne del. Her returneres derfor et 0 for succes. Herefter kigger den i `else`-delen, og finder igen en retur-handling (markeret med lilla). Men her er opdages at handlings-listen ikke er tom, derfor kan udlugeren melde tilbage, at vi har en utilgængelig handling (markeret med rød). Her returneres dog stadig et 0, da der blev fundet en retur-handling, og programmet derfor er lovligt. Hvis retur-handlingen i `then`-delen blev fjernet, ville weederen derimod melde et `-1` tilbage, da der i funktionen mangler en retur-handling, og programmet nu ville være ulovligt.

2.6 Afprøvning

I dette afsnit beskrives nogle få tests, som ikke kan gå igennem parseren. I de efterfølgende afsnit vil der blive beskrevet tests, som er skrevet med fokus på at teste




```
func test(x:bool, u:int) : bool
  if(x)
  then {
    u = 1;
    return u;
  }
  else {
    return u;
    u = 2;
  }
  return 3;
end test
```

Figur 3: Figuren viser et kodeeksempel i Kitty-sproget og dets tilhørende weeder-træ

anden funktionalitet, men som alle sammen går korrekt igennem parseren. Derudover er der medtaget tests der er relevante i forbindelse med de emner der er blevet diskuteret i kapitlet.

At tjekke om AST'et opbygges korrekt er blevet og kan gøres ved at køre oversætteren med flaget `-dBODY`. Hermed printes AST'et indlejret. Det vil sige at outputtet ligner det program, der bliver sendt ind, men der er tilføjet et par af parenteser til hver knude, så alt indholdet af træet, der er beskrevet ved den knude som rod, er indkapslet i et par parenteser. På den måde er der ikke tvivl om, hvad der hører sammen. Den printer der køres, medtager dog informationer fra type-tjekkeren, så der vil være noget ekstra indformation indsat i par af $<$ og $>$, sammenlignet med det originale program.

Testfilnavn	Beskrivelse	Forv.	Resultat
noStmtList.kit	Program uden handlingsliste.	Ikke tilladt	PASS
BoolAsName.kit	Nøgleordet `bool` er brugt som et id.	Ikke tilladt	PASS
ReturnDecl.kit	Returnerer en typeerklæring.	Ikke tilladt	PASS
CombinedType-DeclExp.kit	Angiv typen af en navngiven type til at være et udtryk.	Ikke tilladt	PASS
Unmatch-Comm.kit	Program med en uparret multi-linje kommentar markør.	Ikke tilladt	PASS
MatchComm-Inline.kit	Parret multi linje kommentarblok, med den ene markør i inline kommentar.	Ikke tilladt	PASS
CardOfOr.kit	`||`-operator i kardinalitetsblok. Dette er ikke tilladt.	Ikke tilladt	PASS
CardOfCard.kit	Kardinalitet med kardinalitet inde i. Konkret testes $\|\|a\| + b\|$.	Tilladt	PASS
CardNeg.kit	Kardinaliteten af en negering.	Tilladt	PASS
NegCard.kit	Negering af kardinalitet. Tilladt. Der meldes dog fejl i typetjekker da man ikke kan negere en integer.	Tilladt	PASS

Testfilnavn	Beskrivelse	Forv.	Resultat
nestedif.kit	Denne test viser, at hvis der laves en indlejret `if-then`-handling, i `then`-delen af en `if-then-else`-handling, så vil den inderste `if`-handling få `else`-delen fra den ydre `if`-handling.	Tilladt	PASS

Her vises 7 udvalgte test for udlugeren som viser, at der korrekt tjekkes for funktions id'er og retur-handlinger. Udlugning bliver gjort umiddelbart efter det valgte kitty program er blevet parset af `bison`. Alle programmer der bliver testet her går derfor igennem skanneren og parseren uden problemer. Vi forventer at udlugeren skriver en fejlmeddelelse til stderr, hvis den finder noget der ikke er tilladt, gerne med en lille brugbar besked til brugeren.

Testfilnavn	Beskrivelse	Forv.	Resultat
weedFuncID1.kit	Program der erklærer en funktion med matchene id'er.	Tilladt	PASS
weedFuncID2.kit	Program der erklærer en funktion med forskellige id'er.	Ikke Tilladt	PASS
weedFuncRet2.kit	Program der erklærer en funktion der mangler retur-handling.	Ikke tilladt	PASS
weedFuncRet7.kit	Program der erklærer en funktion med retur-handling i `then`- og `else`-delen af `if`-handling. I `then`-delen findes en handling efter et retur, så det tjekkes også om denne utilgængelige handling bliver fundet og meldt.	Tilladt	PASS
weedFuncRet8.kit	Program der erklærer en funktion med korrekt brug af retur-handlinger i indlejrede `if`-handlinger.	Tilladt	PASS
weedFuncRet9.kit	Baseret på `weedFuncRet8.kit`, men med en retur-handling fjernet, så ikke alle stier har en retur-handling.	Ikke tilladt	PASS

Testfilnavn	Beskrivelse	Forv.	Resultat
weedFunc-Ret11.kit	Program der erklærer en funktion med en indlejret `if`-handling, med manglende retur-handling, der dog findes udenfor.	Tilladt	PASS
weedRet.kit	Program med retur-handling i main-scopet.	Ikke tilladt	PASS

3 Symboltabeller

Symboltabellen har til formål at sørge for at det nemt kan slås op hvilke brugerdefinerede elementer, kaldet symboler, der findes og er tilgængelige på et givent tidspunkt i programmaet. Elementerne sorteres efter hvilket område de er definerede i og disse områder sættes sammen i en træsktruktur med referencer fra børn til forældre, dvs. man kan navigere nedefra og op i træet men ikke den anden vej. Hver knude i træet indeholder en tabel med `SYMBOL`'er som er illustreret i listing 4, der indeholder information omkring variabler, funktioner og typer.

Hver knude i træet refereres til, som et scope.

3.1 Scoperegler

- En ny erklæret funktion opretter et nyt scope inde i det scope, som funktionen blev erklæret i.
- Fra et givent scope kan alle symboler erklæret i dette scope og i scopes længere ude tilgås, i henhold til forklaringen i forrige afsnit.
- Der må ikke være to elementer i det samme scope med det samme navn.
- Hvis der er symboler i forskellige scopes, der har samme navn, så er det altid symbolet i det nærmeste scope, der findes, selvom dette symbol ikke har en kompatible slags (variabel, type eller funktion) eller type. Dette gælder også hvis et symbol i et fjernere scope ville passe ind i konteksten. Begrebet *slags* uddybes i kapitel 4.

3.2 Symboldata

Symbolstrukturen, vist i listing 4, indeholder først og fremmest et felt `kind`, som angiver, om symbolet er en variabel, en funktion eller en type. Hvis symbolet er en funktion, benyttes feltet `typeVal` til at indikerer retur-typen for funktionen. Hvis symbolet er en variabel eller navngivet type, benyttes feltet til at indikere hvilken type den den har. Feltet `typePtr` gemmer en pointer til typen, sådan at det er muligt at udlede yderligere information om typen. Eksempelvis, hvis en bruger har lavet en liste-type, bruges `typePtr` til at finde ud af hvilken type denne liste indeholder. Begrebet typer beskrives dybere i afsnit 4.1.

Når et nyt symbol oprettes gemmes navnet på symbolet i symboltabellen, sammen med en pointer til scopet, hvor symbolet er defineret i feltet `defScope`. I tilfælde af at symbolets `kind` er en funktion, gemmes en pointer til et nyt scope via feltet `scope`, der angiver indholdet af den funktion. Hvis `kind` er en record, gemmes i stedet en pointer i `content` til det scope, der angiver recordens felter. For variabler gemmes der ingen pointere til `content` og `defScope`.

```
1  typedef struct SYMBOL {
2    Symbolkind kind;
3    enum Typekind typeVal;
4    char *name;
5    struct SymbolTable* scope;
6    struct SymbolTable* content;
7    struct TYPE* typePtr;
8    char* typeId;
9    struct SymbolTable* defScope;
10   bool visited;
11   struct SYMBOL *next;
12
13   CODEGENUTIL *cgu;
14 } SYMBOL;
```

Listing 4: Indeholdet af en Symbol-konstruktør

I listing 5 vises strukturen for en symboltabel som repræsenterer et givent scope. Hvert scope opretholder en hashtabel hvori alle symbolerne for dette scope gemmes. Felterne `ParamHead` og `ParamTail` benyttes til at holde styr hvilke symboler der er funktions-parametre, til den funktion der *ejer* det pågældende scope og på rækkefølgen af disse. Alle parametre findes dog også i hash-tabellen så de kan findes her på lige fod med alle andre symboler. Derudover gemmer `next` en pointer som refererer til forældre scopet. Dette giver adgang til symboler erklæreret i højereliggende scopes.

```
1  typedef struct SymbolTable {
2      SYMBOL *table[HashSize];
3      ParamSymbol *ParamHead;
4      ParamSymbol *ParamTail;
5      struct SymbolTable *next;
6  } SymbolTable;
```

Listing 5: Indeholdet af en SymbolTable-konstruktør

3.3 Algoritme

Symboltabellens primære funktioner er at initialisere nye tabeller og dermed oprette nye scopes. Indsætte nye symboler og søge efter eksisterende symboler. Disse funktioner er beskrevet mere detaljeret herunder.

3.3.1 initSymbolTable og scopeSymbolTable

`InitSymbolTable` bruges til at oprette et nyt scope i en uafhængig tabel. Når et nyt scope oprettes med `scopeSymbolTable()` bruges `initSymbolTable()` funktionen til at oprette en ny tabel. Derudover sættes `next`-pointeren i det nye `SymbolTable` til argumentet `t` fra `scopeSymbolTable()` så den nye tabel bliver ”barn”af den forrige tabel.

3.4 Hash

`Hash()` funktionen der er brugt til at indsætte i og lave udtræk fra vores tabeller tager en pointer til et char-array som argument. Ascii værdien for første bogstav bliver binært skiftet en plads til venstre, herefter bliver ascii værdien for det næste bogstav lagt til. Summen bliver så skiftet til venstre. Dette gentages indtil sidste bogstav som kun bliver lagt til, uden det binære skift. Herefter returneres summen modulo størrelsen på vores `Symbol tabel`.

3.4.1 putSymbol og putParam

Et nyt `SYMBOL` indsættes i en tabel, via `putSymbol()`. Ved indsættelse af et symbol som allerede findes i symboltabellen, vil det allerede eksisterende symbol forblive i tabellen, det nye kasseret og `null` returneres, som indikator for at symbolet fandtes. Ved korrekt indsætning i tabellen returneres det netop indsatte symbol. Kollisioner løses ved brug af sammenkædning via `SYMBOL`-strukturens `next`-pointer.

`putParam` bruges til at vedhæfte parametre til funktioner ved at oprette et nyt scope. Dette er brugbart da det er vigtigt at kende rækkefølgen af funktioners parametre.

3.4.2 getSymbol og getRecordSymbol

For at finde et symbol benyttes `getSymbol()` som returnerer det symbol tættest på, som matcher med det søgte navn. Dvs, først søges der i det nuværende scope, hvis symbolet ikke findes, kigges der i forældre-scopet indtil roden nåes. Hvis symbolet findes returneres symbolet, ellers returneres et NULL.

`getRecordSymbol` fungerer på samme måde som `getSymbol()` bortset fra, at der udelukkende kigges i eget scope. Dvs. hvis ikke et symbol findes, så gennemløbes næste scope ikke. Grunden til dette er at vi kan risikere, at vi ikke ønsker at variabler uden for en record kan tilgås som var de inde i recorden.

3.4.3 dumpSymbolTable

Funktionen `DumpSymbolTable()` giver blot et output der indikerer indholdet i alle tabeller på stien fra den givne tabel (blad) til roden, med roden øverst.

3.5 Afprøvning

Alle test for symboltabellen er samlet i `main_test.c` i mappen `main`, og er derfor ikke en del af den automatiske test-suite. Systemet er ikke sat op til at kunne køre disse tests, så hvis de ønskes kørt, så skal funktionen `run_tests` køres i main i stedet for det der er i main-funktionen for oversætteren.

Test-funktion	Beskrivelse	Kommentar
test_Hash();	Tester hash-funktionen op imod kendte hash-værdier for en række strenge.	PASS
test_put-Symbol().kit	Tester for korrekt indsættelse af nye symboler.	PASS
test_get-Symbol().kit	Tester at symboler kan tilgåes.	PASS
test_doubleScope-SymbolTable().kit	Tester at vi kan indsætte symboler i forskellige tabeller, og disse kan tilgåes fra barnet af disse tabeller.	PASS

4 Typetjek

Efter et program er kommet igennem parseren og udlugeren i oversætteren uden, at der blev fundet nogle fejl, sendes det videre ind i typetjekkeren. Typetjekkeren har til ansvar at sikre at programmet, der skal oversættes, ikke har nogle semantiske fejl. Den primære funktionalitet af typetjekkeren er at sikre at typerne af de elementer, der bliver brugt, passer til konteksten.

4.1 Typer

Der findes tre slags elementer som brugeren har mulighed for at definere. Disse er variabler, funktioner og brugerdefinerede eller navngivne typer. Den sidste er umiddelbart til for at lade brugeren angive et simplere navn til en mere kompliceret type, som så kan bruges til at referere til typen i stedet.

Hver af disse tre elementer, kaldet symboler, kan have forskellige typer.

Sproget understøtter den simple heltalstype kaldet `integer`. Denne type lader brugeren modellere tal i programmet. Derudover findes boolske typer, som kan have de to værdier "`true`"og "`false`"for sandt og falsk.

Ud over disse simple typer understøtter programmet 2 sammensatte typer. Den ene er lister (eng. *array*), af de ellers i dette afsnit beskrevne typer. Det andet er en samling af variabler, der alle kan have typer som beskrevet her og uafhængigt af hinanden. En sådan samling kaldes fremover en record.

Sidst kan de tre symbolvarianter have navnet fra en navngiven type, som type. Den egentlige type af symbolet findes ved at udlede, hvad det pågældende navn står for.

Derudover findes også en uofficiel type i oversætteren, kaldet *null*. Dette er ikke en tilladt type for symbolerne at have, men symboler af en bestemt type kan være kompatible med `null`-typen.

4.2 Typeregler

Typerne i et program skrevet i Kitty-sproget skal overholde følgende regler for at oversætteren kan oversætte programmet.

- Integer og boolske typer er ikke kompatible med hinanden.
- Typerne på begge sider af de aritmetiske operatorer $+, -, *, /$ skal være integer. Den resulterende type er integer.
- Typerne på begge sider af sammenligningsoperatorerne $<, <=, >, >=, ==, != $ skal være integer. Den resulterende type er boolsk.
- Typerne på begge sider af de logiske operatorer $\&\&$ og $||$ skal være boolske. Den resulterende type er boolsk.

- Betingelser af `if`, `if-else` og `while` skal være af boolske typer.
- Kardinalitet af alle typer er tilladt og er selv af integer-type. Kardinaliteten af et tal, er dets numeriske værdi. For en boolsk type er det 1 ved `true` og 0 ved `false`. Kardinaliteten af en record er antallet af felter og for en liste er det den allokerede længde.
- Negering kan kun gøres af boolske typer.
- Navngivne simple typer er kompatible med hinanden og med de simple typer direkte.
- Det er strukturel ækvivalens mellem lister af samme dimension og sluttype. Dette gælder også selvom de to typer er erklæret vidt forskelligt. Foreksempel er typerne `A` og `B` ækvivalente og dermed kompatible i det følgende:
  `A = array of T1`,
  `B = T2`,
  `T1 = array of int`,
  `T2 = array of array of T3` og
  `T3 = int`.
- Der er ikke strukturel ækvivalens mellem records. Det vil sige at to anonyme records aldrig kan være kompatible med hinanden. Det gælder også, hvis de er erklæret med de samme felter og af samme typer.
- Enhver navngivet type er kompatibel med sig selv.
- `write` er kun kompatibel med integer- og boolske typer
- Operanden til en returhandling skal have en type der er kompatibel med funktionens erklærede returtype.
- Det er kun variabler af record- og liste-typer der kan tildeles konstanten `null`.
- Kun variabler af liste-type kan listeallokeres og -indekseres.
- Kun variabler af record-type kan (enkelt)-allokeres og benyttes i forbindelse med prik-notation.
- Det er ikke tilladt at erklærer symboler af ugyldige eller ikke-eksisterende typer. Dette gælder også, hvis det erklærede symbol ikke bliver brugt.
- Navngivne typer må ikke referere til hinanden i en løkke.
- Navngivne og liste-typer samt navngivne typer i records må gerne referer circulært til hinanden. Dvs. man kan skrive:
  `A = array of B`
  `B = array of A`
  samt
  `R = record of {x:S}`
  `S = record of {y:R}`

Som tidligere beskrevet, så virker symbol-tabellen sådan, at det altid er det nærmeste element for et givent scope, der med et givent navn, findes. Dette gælder også hvis det fundne symbol ikke er kompatibel med konteksten, selvom et sådan symbol måtte findes i et fjernere scope. Dette betyder bl.a., at hvis et felt i en record har en type, der deler navn med at andet felt i den record, så vil typetjekkeren tro at brugeren forsøger at angive en variabel som type, hvilket ikke er tilladt. Dette giver måske ikke så meget mening her, som det gør for indlejrede funktioner, men er gjort sådan for at oversætteren behandler symboler konsekvent på samme måde, hvilket gør det nemmere både at benytte og implementere oversætteren.

4.3 Algoritme

Typetjekkningen sker i 3 uafhængige faser, men hvor resultaterne fundet i en tidligere fase er nødvendige for de efterfølgende. Disse faser er symbolopsamlingsfasen, udtrykstypeudledningsfasen og typetjekningsfasen. Hver fase er beskrevet for sig i afsnittene herunder.

4.3.1 Symbolopsamling

I denne fase opsamles alle symboler defineret i programmet og gemmes i symboltabellen beskrevet i afsnit 3. For bedre overblik over sammenhængen mellem det abstrakte syntakstræ og symboltabellen, som beskrives her, og i de efterfølgende afsnit, viser figur 4, en smule simplificeret, denne sammenhæng for Kitty programmet i listing 6.

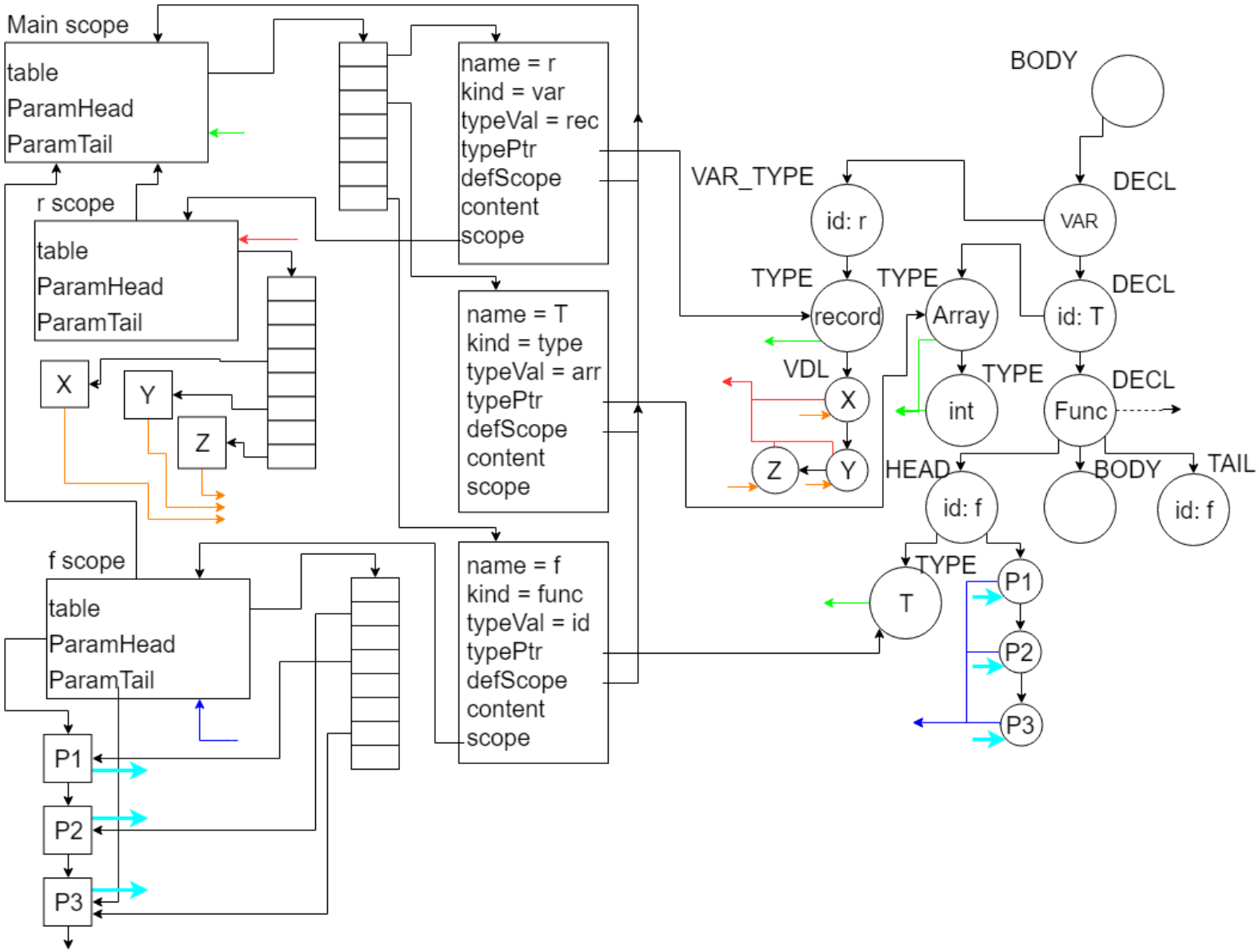
```
1  var r:record of(x:int, y:int, z:int)
2  type t = array of int;
3  func f(P1:int, P2:int, P3:int):T
4      ...
5  end f
6   ...
```

Listing 6: Kittyprogram der erklærer tre symboler. De to handlingslister er udeladt.

Opsamlingen sker ved at løbe igennem alle erklæringer i det abstrakte syntaks træ, startende ved roden. I den forbindelse oprettes der også en symboltabel, der repræsenterer main-scopets indhold. Hver gang en variabel findes (eks. `r` i figure 4), så lægges variablens navn sammen med relevant information for dens type ind i det relaterede scope. Dette gøres uden at undersøge symbolet nærmere i den antagelse, at der ikke er lavet fejl i symbol-erklæringerne. Således understøttes at symboler kan være erklæret ude af orden, så en variabel kan have en type, der først erklæres senere i erklæringslisten. I de senere faser tjekkes det, at typerne faktisk findes og passer sammen.

Opsamling af navngivne typer håndteres på fuldstændig samme måde, bortset fra at symbolet, der nu repræsenterer en type og ikke en variabel, sættes til at reflektere dette.

Hvis typen af symbolet er en record, som det er tilfældet for symbolet `r` i listing 6, så oprettes der et scope for indholdet af denne record, og det undersøgte symbol sættes til at pege på dette scope via `content` referencen (se figur 4 for symbolet `r`). Det nye



Figur 4: Sammenhængen mellem symboltabellen (firkantet) og det abstrakte syntakstræ (rundt). For programmet vist i listing 6. Figuren indeholder 3 scopes: main-scopet, funktionen `fs` scope og scopet for indeholdet af variablen `r`. Indeholdet af mainscopets symboler er skrevet ud. Symbolerne i de andre scopes er ikke skrevet ud. De orange og lyseblå pile ud af disse symboler repræsentere deres `typePtr`. Det øverste element i træet er main scopets krop. Erklæringslisten og dens erklæringer er for overskueligheden slået sammen til en liste af `DECL`. Det samme er tilfældet for variable-erklæringslister og deres variabler. Referencerne fra hver type til deres tilhørende scope er vist ved de grønne, røde og blå pile. For $P1$, $P2$ og $P3$ samt X, Y og Z er variabel-navnet og typen (som ikke er vist) slået sammen. Den stiplede pil i erklæringslisten illustrerer, at denne kan fortsætte her. Handlingslisten er helt udeladt fra figuren.

.

scope peger tilbage på det scope, recorden er defineret i, så der er adgang til typerne i dette eller højereliggende scopes. Efterfølgende løbes variabel-erklæringslisten, der beskriver indeholdet af denne records indhold, igennem og alle variablerne tilføjes til rekordens scope, som det er tilfældet for `x`, `y` og `z` i figur 4.

Hvis typen af et symbol er en liste (se `T` i figur 4), så udvides liste-typen rekursivt, til der ikke længere mødes lister. Hvis den fremkomne type viser sig at være en record, så oprettes et separat symbol for denne record-type og listens record-type erstattes med et id, der repræsenterer den record-type. Dette skyldes at indholdet af den anonyme record skal kunne gemmes i et scope, der kun skal være tilgængeligt fra denne liste-type. Ved at ændre typen af listens indhold til en *hemmelig* navngiven type, kan der genbruges den funktionalitet, der lader en liste have en navngiven type som indhold.

Hvis et symbol har en navngiven type så accepteres denne type blindt i opsamlingen og informationen medtages ind til videre.

Sidst kan erklæringslisten indeholde funktioner. Når en funktionserklæring findes, betyder det, at et nyt scope skal oprettes, der beskriver funktionens indhold. Derfor laves en ny symbol-tabel, der "scoper"tilbage på det scope, som funktionen er erklæret i. Et symbol for funktionen selv, med funktionens retur-type som type, lægges i det oprindelige scope. Dette symbol peger via `scope` til det nyligt oprettede scope, så funktionens krop kan findes ud fra dets navn. Efterfølgende løbes parameterlisten igennem. Alle parametrene behandles på nøjagtig samme måde som variabler, bortset fra at parameter-symbolerne også lægges ind i en liste i det nye scope. På den måde kan man via scopet altid finde navn og type af alle parametre i rækkefølge for funktionen, repræsenteret af det scope. Strukturen er illustreret på figur 4 for symbolet `f`.

En reference til funktionens krop (`BODY`), lægges ind i en liste. Dette gøres for i de senere faser af typetjekeren at kunne løbe igennem alle funktionskroppe uden at skulle bekymre sig om at analysere indlejrede funktioner med det samme, når de findes. Opsamlingen fortsætter dybdeførst direkte i kroppen for den erklærede funktion, og først når opsamlingen i dennes krop er ovre fortsættes der i det oprindelige scope, hvor funktionen blev erklæret.

Sidst skal det nævnes at alle type-strukturer, vist i listing 1, der bliver fundet i det abstrakte syntakstræ sættes til at pege tilbage på det scope de er defineret i. Dette er for at man ud fra en given type altid kan finde det scope typen er defineret i, og dermed også ved fra hvilket scope man skal lede efter navngivne typer. Dette er illustreret med de røde, grønne og blå pile på figur 4. Symboler har den samme reference, således at man fra et symbol altid kan finde det scope hvori det er defineret.

4.3.2 Udledning af udtrykstyper

I denne fase løbes der igennem alle handlings-lister for at opsamle og gemme typerne af alle udtryk. Typerne der gemmes i forbindelse med alle udtryk er den umiddelbart nærmeste type, der kan udledes. Foreksempel kan typen af 6 udeledes til at være integer, typen af x kan være en record-type og $a.g[l]$ kan vise sig at have en navngivet type X. I alle tilfælde vil det være den her angivne type, der gemmes sammen med udtrykket. Som udgangspunkt laves der ingen kontrol af type-kompatibilitet i denne fase og

der returneres kun fejl hvis det viser sig at der opstod et problem (f.eks et symbol, der ikke kunne findes).

En handling kan som udgangspunkt bestå af tre elementer: variabler, udtryk og "indlejrede"handlinger. Det sidstenævnte findes kun ved `if`- og løkke-handlinger og når disse mødes, løbes deres handlinger igennem for at undersøge evt. udtryk, der måtte findes i disse.

Når et udtryk findes, skal typen af dette analyseres og gemmes i forbindelse med udtrykket. Dette bliver gjort via et felt i `EXP`-strukturen, der angiver hvilken af de 5 typer, beskrevet i forbindelse med listing 2 i afsnit 4.1, som dette udtryk har. Sammen med dette gemmes en pointer til et `TYPE`-element, der giver yderligere information om typen.

Et udtryk kan bestå enten af to udtryk, der er bundet sammen af en operator eller et term. I første tilfælde undesøges rekursivt at begge del-udtryk har en type der er kompatibel med den gældende operator og afhængig af operatoren gemmes den simple type, som resultatet af operatoren, skal resultere i. Hvis der opstår en fejl undervejs i at finde typen af et udtryk, så gemmes dette som en fejltype og der returneres fejl tilbage. Oversætteren skulle gerne afbryde med en typefejl med det samme, hvis en fejl opstår, men det at gemme fejlen forbereder oversætteren til at kunne finde flere fejl inden den afbryder, da oversætteren så vil kunne genkende en fejl, fundet tidligere, i den videre analyse.

Hvis et udtryk er beskrevet af et term, så analyseres dette term for at finde typen. Hvis dette er en integer eller boolsk konstant, eller konstanten NULL så gemmes blot den type, der repræsenterer konstanten. Hvis termet viser sig at være en variabel, skal typen af denne variabel analyseres via funktionen `expTypeTravVar`. Hvis variablen indekserer en liste eller tilgår et felt i en record, så sikres det at variablen rent faktisk også er en liste eller en record. Dette gøres ved at lade funktionen kalde sig rekursivt på variabel-delen af `VARIABEL`-strukturen indtil en grundvariabel er fundet. Hvis typen viser sig at være en navngivet type, så vil det være nødvendigt at løbe i bund indtil en indbygget type findes. Denne type kan så returneres tilbage igennem de rekursive kald. Afhængig af om variablen blev brugt som en record eller en variabel, så tjekkes om den fundne type stemmer overens med dette og listens type eller typen af feltet i recorden findes og returneres tilbage.

Hvis variablen er brugt som en liste, skal typen af liste-indeksudtrykket også analyseres og gemmes. Derfor er det i denne fase også nødvendigt at undersøge alle variabler, der fremkommer direkte i en handling, i tilfælde af at de indeholder en liste-indeksering, for hvilket typen for udtrykket skal findes, når den skal bruges i næste fase.

Når et funktionskald udføres så løbes argumentlisten i funktionskaldet igennem for at undersøge og gemme typerne af alle de udtryk, der måtte være i den liste. Samtidig sikres at længden af argumentlisten passer med længden af paramterelisten for den pågældende funktion og at typen af hvert argument passer til parameteren. Såfremt ingen fejl blev fundet i denne liste, så gives udtrykket der introducerede funktionskaldet, den samme type som funktionens returtype.

Vi tillader at man kan tage kardinaliteteten af alle typer, så udtrykket inde i kardinalitets

operatoren undersøges for at at gemme dens type, men operatoren selv har typen integer i alle tilfælde.

4.3.3 Typetjek

I tredje fase af typetjekket laves den egentlige undersøgelse af kompatibilitet mellem typer. Igen bruges `BODY`-listen beskrevet i afsnit 4.3.1 til at undersøge hver funktionskrop for sig, som analyseres i sin helhed inden indlejrede funktioner analyseres. På den måde sikres at al relevant information allerede er fundet, hvis en indlejret funktion tilgår noget fra et højereliggende scope.

Det første, der sker i denne fase, er at funktionen `checkTypeTravDecls()` løber over erklæringslisten af hvert element i `BODY`-listen. Dette gøres for at sikre at typen af ethvert erklæret symbol findes, og hvis typen er en anonym record eller liste, så tjekkes det ligeledes, hvorvidt indholdet er af gyldige typer. På den måde sikres det, at der ikke kan erklæres noget af ugyldige typer, selv hvis det, der bliver erklæret, ikke bliver brugt. Desværre er det erfaret efter projektets deadline at denne fase har flere fejl.

Når en variabelerklæring findes behandles den forskelligt afhængig af dens type i funktionen `checkTypeTravVDecls`. Hvis dens type er en simpel liste-type, så findes typen af denne liste-type rekursivt indtil, der findes noget, der ikke er en listetype. Herefter undersøges typen på samme måde, som hvis det ikke i første omgang havde været en listetype. Hvis typen er en simpel integer- eller boolsk type, godkendes erklæringen, da typen er gyldig. Hvis det er en record-type, så undersøges elementerne i recorden rekursivt, og kun hvis der ikke blev fundet fejl i denne erklæringsliste, godkendes typen. Sidst kan typen være en navngiven type. I dette tilfælde undersøges blot om den angivne type findes i symboltabellen, og om det fundne symbol repræsentere en type, og i så fald godkendes erklæringen.

Erklæring af navngivne typer burde tjekkes på samme måde. Det er dog erfaret efter projektets deadline, at dette ikke er tilfældet. Når en navngiven type findes, undersøges det om dens type er en navngivet type, og der løbes i så fald i bund, til der findes en indbygget type. I denne forbindelse bliver det afgjort om den angivne type findes. Efterfølgende tjekkes det, om den er defineret som en record, og i så fald tjekkes om dens indhold har gyldige typer, ved at benytte den samme funktionalitet til at løbe over recordens felter som bruges til at verificerer typerne af variabler, som beskrevet herover. Det skal kort nævnes, at symboler har et felt kaldet `visited`, der benyttes i forbindelsen med denne fase. Koden, der benytter dette felt, gør dog ikke noget funktionelt.

Problemerne, der blev nævnt, opstår, hvis en navngivet type er defineret via en anonym liste-type. Her bliver det ikke tjekket om den type, listen indeholder, faktisk findes. Typetjekkeren vil opdage fejlen, hvis brugeren forsøger at definere og bruge en variabel af typen, men hvis brugeren kun definerer en variabel af typen uden at bruge den, så finder typetjekkeren ikke fejlen. Det medfører at der opstår fatale problemer i kodegenereringen, da kodegenereringen skal kende typen af alle variabler for at kunne lokalisere elementer, og der laves ikke tjek på om typen findes i den fase. Erfaringerne beskrevet her er også afspejlet i afprøvningsafsnittet 4.4.

Hvis det er en funktion, der erklæres, så er det ligeledes erfaret at typetjekkeren ikke

tjekker retur- og parameter-typer korrekt. Først tjekkes om retur-typen er en navngiven type og i så fald om den findes. Dog tjekkes det ikke, hvis returtypen er en anonym liste eller record, om den indeholder gyldige typer. Fejlen bliver dog fundet af typetjekkeren når, der defineres, hvad der skal returneres (hvis ikke det er `null`), eller når funktionen kaldes. Men hvis ingen af disse ting bliver gjort i programmet, så findes fejlen ikke. Efterfølgende løbes parametererklæringslisten igennem for at sikre at typen af alle parametre er gyldige. Igen viser det sig dog, at dette ikke bliver gjort korrekt. Som ved funktionens retur-type, tjekkes det korrekt om typen af hver parameter findes, hvis de er af navngivne typer, men det tjekkes ikke om de har gyldige anonyme liste- og record typer. Derfor overses det, hvis en parameter er en anonym record eller listetype, der indeholder (elementer af) typer, der ikke findes, hvis funktionen ikke bliver kaldt, eller parametrene ikke bliver brugt i funktionen. Men lige så snart en af disse to ting bliver gjort, så vil typetjekkeren finde fejlen i forbindelse med dette. Igen er erfaringerne afspejlet i afprøvningsafsnittet 4.4.

Efter erklæringslisten for en given `BODY` er blevet analyseret udføres typetjekning på handlingslisten. For hver handling er der forskellige invarianter, der skal tjekkes blandt typerne, som beskrevet i afsnit 4.2. I de fleste tilfælde kræver det en sammenligning af typerne mellem et symbol og et udtryk, to symboler eller to `TYPE`-elementer. Disse sammenligninger bliver udført af funktionerne `cmpTypeSymExp`, `cmpTypeTyExp`, `cmpTypeTyTy`, `cmpTypeSymTy` og `cmpTypeSymSym`. Disse funktioner returnere alle 0 ved sammenligning af kompatible elementer. De kalder hinanden rekursivt under bestemte omstændigheder afhængig af hvilken information, der findes om de elementer, der sammenlignes. Det er i forbindelse med alle disse sammenligninger at strukturen beskrevet af figur 4 er fundamental. De 5 nævnte funktioner, bruger hele tiden disse relationer til at finde frem til hvorvidt forskellige brugte typer er kompatible. Samtidig benyttes `recursiveSymbolRetrival` til at udlede den endelige type af en (kæde af) navngivne type(r). Det er også denne funktion, der sikrer at, der ikke er erklæret en løkke af navngivne typer, hvilket ikke tillades. Dette gøres simpelt ved at samle symbolerne op i en liste, og hver gang en ny type findes i en kæde af navngivne typer, sikres at symbolet for den pågældende type ikke allerede findes i listen, før den indsættes. Er det tilfældet, meldes der fejl.

Når en returhandling findes, laves der en sammenligning mellem udtrykket, der skal returneres og typen af den funktion, der bliver gennemløbet. Til at finde funktionens returtype benyttes funktionens id og forælderscopet til det scope, der bliver undersøgt til at slå funktionen op i symboltabellen. Returtypen er beskrevet som typen for det symbol der repræsentere funktionen.

For en ”write-handling undersøges blot om udtrykket har en integer- eller boolsk type.

Ved record og liste allokation sikres at den variabel, der skal allokeres, har en hhv. record eller liste-typen. Det betyder, at typen af variablen skal findes. Dog kan en variabel være beskrevet som både liste-indekseret eller et felt i en record, så det er nødvendigt rekursivt at løbe til grund-symbolet for variablen. Dette gøres på samme måde som i udtryks-type opsamlingsfasen beskrevet i afsnit 4.3.1. Dog undersøges det nu også at grundsymbolet for variablen faktisk beskriver en variabel. Hvis ikke, så meldes der fejl tilbage. For liste allokation sikres også at indeks-udtrykket er af

integer-type.

Ved en tildeling sikres at variablen på venstresiden og udtrykket på højresiden er kompatible. Dette betyder igen at typen af variablen skal findes.

Ved `if`- og løkkehandlinger sikres at betingelsesudtrykket er boolsk, før der løbes rekursivt igennem handlingslisterne for disse handlingers kroppe. Ved løkker tælles en loop-tæller op inden handlingslisten analyseres og tælles ned igen, når analysen er færdig. Denne tæller bruges til at holde styr på hvor mange lag af indlejrede løkker vi er i. Hvis en `break`- eller `continue`-handling findes, så sikres det blot, at vi er i mindst en indlejret løkke.

Rekursionen laves på samme måde for handlingslister i krølle-parenteser.

4.4 Afprøvning

Typetjekkeren er blevet testet med mere end 100 enheds-tests, til at teste enkelt-funktionaliter i oversætteren. Derudover er der også lavet et par enkelte større test. Denne fulde liste af udvalgte test, der er vurderet dækkende for typetjekkerens funktionalitet findes i bilagsafsnit C. Herunder er listet de tests som det viser sig at oversætteren ikke behandler korrekt, samt lignende test, der er relevante i forbindelse med de fejlafslørende test.

Testfilnavn	Beskrivelse	Forv.	Resultat
DeclArrType-NonExist.kit	Erklær en liste-type af en brugerdefineret type, der ikke findes og en variabel af denne type.	Ikke tilladt	Fejlen findes ikke og kernen smides i kodegeneringen.
DeclArrType-NonExistUse.kit	Baseret DeclArrTypeNonExist, men variablen bliver også brugt her.	Ikke tilladt	Fejlen findes, når variablen bruges.
DeclFuncParArr-NonExistType.kit	Erklær en funktion der tager som parameter et anonymt array indeholdende en type, der ikke findes	Ikke tilladt	Fejlen findes ikke af oversætteren.
DeclFunParArr-NonExistType-Use.kit	Baseret på DeclFuncParArrNonExist-Type.kit. Der laves at funktionskald til funktionen.	Ikke tilladt	Fejlen findes i forbindelse med typetjek af funktionskald.
DeclFuncParRec-NonExistType.kit	Erklær en funktion der tager som parameter en anonym record indeholdende en type der ikke findes.	Ikke Tilladt	Fejlen opdages ikke af oversætteren.

Testfilnavn	Beskrivelse	Forv.	Resultat
DeclFuncParRec-NonExistType-Use.kit	Baseret på DeclFuncParRec-NonExistType.kit. Parameteren bruges inde i funktionen.	Ikke tilladt	Fejlen findes når parameteren bruges.
DeclFuncRetArr-NonExistType.kit	Erklær en funktion, der returnerer en anonym liste indeholdende en type, der ikke findes.	Ikke tilladt	Fejlen bliver ikke opdaget
DeclFuncRetArr-NonExistType-Use.kit	Baseret på DeclFuncRetArrNonExist-Type.kit. Forsøg at returnere en liste af typen.	Ikke tilladt	Fejlen findes ved retur-handlingen inde i funktio-nen.
DeclFuncRetRec-NonExistType.kit	Erklær en funktion, der returnerer en anonym record indeholdende en type, der ikke findes.	Ikke tilladt	Fejlen finde sik-ke af typetjekke-ren.
DeclFuncRetRec-NonExistType-Use.kit	Baseret på DeclFuncRetRec-NonExistType.kit. Den returnerede record bliver brugt.	Ikke tilladt	Fejlen findes ved brug af recorden.
DeclFuncPar-NonExistType.kit	Erklærer en funktion, der tager som pa-rameter en variabel af en type, der ikke findes. Testen er medtaget for at frem-hæve, at hvis typen af en parameter ikke findes, så opdages det selv, hvis paremeteren ikke bruges. Det samme gælder ved returtypen.	Ikke tilladt	PASS
DeclFieldType-NameAsOther-Field.kit	Erklær et record-felt med en gyldig ty-pe, der dog deler navn med et andet felt i recorden. Dette medfører, at ty-pen bliver forstået som det felt, der er i recorden, som ikke er en type, men en variabel.	Ikke tilladt	PASS
ArrTypeEquiv.kit	Tjek korrekt brug af typeækvivalens mellem indlejrede lister indeholdene integers.	Tilladt	PASS
ArrTypeEquiv-Wrong1.kit	Tjek forkert brug af type ækvivalente lister af records.	Ikke tilladt	PASS

Testfilnavn	Beskrivelse	Forv.	Resultat
ArrTypeEquiv-Wrong2.kit	Tjek forkert brug af type ækvivalente lister af integers.	Ikke tilladt	PASS

5 Kodegenerering

5.1 Strategi

Til kodegenerering er der lavet 4 vigtige strukturer som benyttes til opbevaring af information samt navigering af den interne repræsentation. De fire strukturer er:

- **`INSTR`** - (Instruktion) Repræsenterer en linje assembler-kode. `INSTR` har et `enum`-felt, som holder styr på, hvilken slags instruktion, der er tale om, f.eks. `mov`, `add`, `jmp`, osv. `INSTR` har også en hægtet lister af `OPERAND`, som repræsenterer de operander, instruktionen skal bruge, i de fleste tilfælde skal der kun være 2 operander. Der findes én `INSTR` liste, som repræsenterer hele den interne repræsentation, og det er den, der manipuleres i denne og de efterfølgende faser.
- **`OPERAND`** - Repræsenterer operander for assembler instruktioner. `OPERAND` har et `enum`-felt som holder styr på hvilken slags operand, der bliver brugt, f.eks. `label`, `register`, `dereference`, osv. `OPERAND` har også en union af forskellige felter, som alt efter hvilken slags operand der bliver brugt enten indeholder en placering eller f.eks. et navn på en label.
- **`TEMPORARY`** - Er til midlertidige placeringer for ting, der i kodegenereringen ikke endnu har fået en endelig placering. `TEMPORARY` har nogle id-felter og en `next`-pointer, som bliver brugt i register allokeringen. Der er en `enum` som holder styr på hvilken slags `TEMPORARY` der bliver brugt, de forskellige slags inkluderer parameter, lokal, register og midlertidigt resultat. Alt efter hvilken slags `TEMPORARY`, der bliver brugt bruges en `union`, der gemmer en placering, enten som en addresse eller et register.
- **`CODEGENUTIL`** - En struktur der bliver koblet på `SYMBOL`'er fra typecheck-fasen, og som for funktioner gemmer label og parameter antal, og for variabler gemmer en tilknyttet `TEMPORARY`.

Vi har fastlagt os på denne repræsentation af en stackframe. Caller- og Calleé-prolog og -epilog er ens alle steder, er der altså ikke lavet nogen optimering på disse områder. Static link refererer til basepointeren i det scope (stack-frame) funktionen er erklæret (og potentielt indirekte blev kaldt fra). Vi adskiller variabler og temporaries. Variabler ligger altid på stakken og kommer først, mens temporaries kan flyttes til registre i senere faser.

lave adresser	
næste stackframe	...
nuværende stackframe	retur adresse static link parametre caller-saves ... temporaries variabler calleé-saves base-pointer
forrige stackframe	retur adresse static link parametre caller-saves ...
høje adresser	

Ideen med kodegenereringen er at omdanne det abstrakte syntaks-træ til en hægtet liste af instruktioner som nemt kan printes til assembler-kode i emit-fasen. Temporaries benyttes efter behov, og der er ikke sparet på dem. Lige så snart et resultat er fundet, lægges det i en temporary, så det er sikkert at informationen ikke ved en fejl bliver overskrevet eller på anden vis går tabt.

Repræsentationen af data i brugerens program gøres via 64-bits heltal. Det er naturligt at behandle integers i dette format. De boolske typer bliver repræsenteret som 1 for `true` og 0 for `false`.

I de følgende afsnit vil det blive beskrevet hvordan koden genereres. Dette vil indvoldvere brugen af labeler. Måde det sikres at en label er unik på er ved at tilføje værdien af en tæller i programmet bag navnet på labelen.

5.2 Kodeskabeloner

Her beskrives fremgangsmåden til at generere kode til faste strukturer, der fremkommer i et program skrevet i kitty-sproget.

- **`return`** - Først findes værdien eller placeringen for det, der skal returneres, og det flyttes ind i register `RAX`. Derefter laves et ubetinget hop til et label, der står for at rydde op efter den funktion, vi er i.
- **`while`** - Der laves et label, der står i starten af `while`-løkken. Derefter genereres kode til at udregne værdien af betingelsen for `while`-løkken. Resultatet af betingelsen sammenlignes med `true`, herefter laves der en `jne` instruktion som hopper til slutningen af while løkken, dvs. vi slutter `while`-løkken hvis betingelsen ikke er sand længere. Derefter genereres kode for de handlinger der står inde i `while`-løkken, til sidst hoppes tilbage til starten af `while`-løkken og en `endWhile`-label placeres efter denne `jmp` instruktion.

- **`if then else`** - Der er to labels til `if-then-else` skabelonen. Et til at hoppe til `else`-delen og et til at hoppe helt til slutningen af `if-then-else`. Først genereres kode for betingelsen for `if`. Herefter sammenlignes resultatet af betingelsen med `true`, og der laves en `jne`-instruktion, som hopper til `else`-delen hvis betingelsen ikke er sand. Herefter genereres kode for de handlinger, der står direkte efter `if`-delen. Efterfølgende placeres `else`-labelen, og der genereres kode for `else`-delen. Til sidst placeres `endIfLabel`-labelen. Det er næsten den samme fremgang, der benyttes til `if-then` skabelonen bare uden `else` delen.

- **`allocate`** - Vi har i starten af kodegenereringen afsat en statisk mængde hukommelse til allokering af records og arrays i løbet af et program. Denne hukommelse udgøre heapen, der tilgås via tre labels; `start-`, `free-` og `end-heaplabel`. Først findes typen og placeringen af den variabel, der skal allokeres. Herefter lægges en pointer til næste frie plads i heapen over i placeringen for variablen. Derefter udregnes størrelsen for det, der skal allokeres, og det der står ved `freeheaplabel` ændres, så referencen rykkes ned i følge den nyallokerede mængde. Dernæst laves runtimecheck for overtrædelse af den plads der er til rådighed til at allokere i. Koden for `allocate of length` genereres efter næsten samme fremgangsmåde. Her findes den mængde bytes, der skal allokeres som det heltal, vi får ud af det udtryk, der står efter terminalen `length` ganget med 8, da hvert element fylder 8 bytes.

- **`write`** - Finder placeringen eller værdien for det, der skal printes og placerer det i registeret `RSI`, derefter lægges et format, der er defineret i starte af filen, over i `RDI`, 0 over i `RAX` og `printf` kaldes.

- **`tildeling`** - Der genereres kode for det udtryk, der står på højre side af tildelingen, dette kan være andre variabler, matematiske udtryk mm. Derefter findes placeringen af den variabel, der står på venstresiden af tildelingen og resultatet placeres af udtrykket der.

- **`binære operatorer`** - Vi har operatorerne $+$, $-$, $*$, $/$, $<$, $<=$, $=$, $=>$, $>$, $!=$, $||$ og $\&\&$. I de fleste tilfælde starter vi med at generere kode til at finde resultatet af venstre udtryk først, og lægge det i en temporary. Derefter genereres kode for højre udtryk for derefter at udføre den passende behandling af de to resultater på baggrund af operanden. Til sidst placeres resultatet i en temporary. Undtagelser inkluderer $*$, $/$, $||$ og $\&\&$. Ved $*$ kræves det af assembler-syntaksen at anden operand er et register, derfor behandles det specielt. Ved $/$ laves et runtime check for division med 0 og resultatet ender i RAX, så det skal flyttes derfra og over i en temporary. I $||$ og $\&\&$ laves "lazy"evaluering. Dvs. ved $||$ laves et check efter generering af resultat fra venstre operand. Hvis denne er true, vides at hele udtrykket bliver true, og der hoppes videre uden evaluering af højre operand. Ved $\&\&$ gøres næsten det samme; det kontrolleres om venstre operand er false. Hvis dette er tilfældet vides at udtrykket bliver false og evaluering af højre operand springes over.

- **`liste indeksering`** - Her genereres først kode til at finde variablen, dvs. listen, der skal indekseres ind i, skal findes. Når det er gjort kontrolleres om listen er sat til `null` fra starten af. Hvis dette er tilfældet returneres en fejlkode til brugeren. Herefter genereres kode for det udtryk, der står som indeks og det kontrolleres om vi er ud over listens størrelse. Hvis indekset passer ind i listens størrelse indekserer vi ned i listen, her tager vi højde for at første element i listen er listens størrelse. Når det korrekte indeks er fundet kontrolleres om indekset er `null`. Hvis det er tilfældet returnerer vi en fejl-kode til brugeren. Hvis indekset ikke er null returnerer vi værdien på dette indeks.

- **`variabel-felt tilgang`** - Først genereres kode for variablen, dvs. den `record`, der holder den information vi leder efter. Det kontrolleres om variablen er sat til `null`. Hvis dette er tilfældet, returneres en fejlkode til brugeren. Ellers findes det felt der ledes efter i recorden. Hvert felt i en `record` har sit eget offset, der er gemt i symbolet for feltet. Når offsettet er fundet regnes den endelige adresse ud. Det kontrolleres om feltet er sat til `null`. Hvis dette er tilfældet returneres en fejlkode til brugeren. Hvis feltet er sat korrekt lægges adressen til dette over i en temporary og returnerer denne.

- **`funktions kald`** - Det er stackframen opbygges. Ved et funktions kald laves vi først caller-saves. Det består af 8 `push` instruktioner af registrerne `rcx, rdx, rsi, rdi, r8, r9, r10, r11`. Herefter gennemgås funktionens parameterliste. Der afsættes plads til parametrene ved at trække en repræsenterende størrelse for antallet af parametre fra stackpointeren. Herefter gennemgås parametrene fra venstre mod højre og de lægges på stakken, så første parameter ligger tættest på stackpointeren. Efter det genereres kode, der finder static link for funktionen, når det korrekte static link er fundet pusher vi det på. Så laves det egentlige funktions kald. Når der returneres fra funktionskaldet fjernes static link og parametre ved at lægge til stackpointeren og derefter poppes alle registrerne fra caller-saves af stakken igen.

- **`static-link`** - Vi har valgt at static link indeholder adressen til basepointeren fra den stackframe funktionen er blevet (statisk) defineret i. Når static link skal findes, bruges `SymbolTable`'en til at tælle, hvor mange scopes vi skal ud for at finde, der hvor funktionenen eller variablen, der tilgås, er erklæret. Hvis vi skal 0 scopes ud kan bruges den nuværende basepointer. Ved mere end 0 scopes, findes static link under basepointeren og denne derefereres. Alt efter hvor mange gange vi skal ud, lægges 16 til adressen vi er nået til (for at finde static link fra base pointeren) og denne derefereres. På den måde navigeres der ned igennem tidligere static links, og der findes frem til der, hvor en funktion eller variabel er erklæret. Hvis dette bruges til funktionskald, pushes værdien på stakken, og hvis det er til at finde variabler ude af scope, lægges værdien i `rdi`, som så bruges til at angive variabler og parameters scope.

- **`break and continue`** - når vi kommer til et *while* loop sørger vi for at passere loop labeler med ned i de næste par interne fuktions-kald så hvis man støder på et break eller et continue kan man hoppe til det tilhørende label.

5.3 Algoritme

Fra typecheck fasen har findes en liste af funktions-kroppe (liste af `BODY` strukturer). Første element er altid yderste scope i det program, der bliver behandlet. Listen bruger vi til at repræsentere hele programmet, da alting er omsluttet af en `BODY` i det AST'et. I kodegenereringen løbes denne liste igennem fra start til slut for at generere en intern repræsentation, som skal have samme effekt, som det program der bliver beskrevet af disse funktions-kroppe.

Det første der gøres i kodegenereringsfasen er at sætte nogle labels op, som skal bruges i tilfælde af fejl og i forbindelse med adgang og allokeation af heapen. Efterfølgende gennemløbes alle funktions-kroppene og der sørges for at de alle har labels, og at vi ved hvor mange parametre de har. Når opsætningen er færdig løbes funktiones-krops-listen igennem fra start til slut, og der genereres kode for handlings-listerne i disse funktions-kroppe. Når vi kommer til en ny funktion, tilføjes først det tilhørende label til den interne repræsentation. Halen af den interne repræsentation gemmes, da placeringen skal bruges senere. Derefter laves et end-label til funktionen og hvis det viser sig at være main-funktionen, der skal genereres, gemmes stack- og base-pointer i to data labels. Vi skal bruge disse, hvis der sker fejl i løbet af programmet. Det er også her, heap-labelerne initialiseres. Når det er gjort, gennemgås funktionens handlings-liste og skabelonerne der er beskrevet ovenfor benyttes.

Når handlings-listen er gennemgået ved vi hvor meget plads der skal bruges på stakken til de variabler og temporaries, der bliver brugt i funktionen. Den tidligere gemte position bruges den interne repræsentation bruges nu til at indsætte en instruktion, der trækker antallet af variabler og temporaries fra stack-pointeren for at gøre plads til disse. Afslutningen af funktionen generes nu. Hvis vi er i main funktionen lægges 0 i `rax` for at symbolisere, at der ikke er sket fejl under kørslen af programmet. Derefter indsættes et label, der repræsenterer slutningen af funktionen. Efterføglende kommer funktions-epilogen, der rykker stack-pointeren ned over vores variabler og temporaries og kalder funktions epilog til sidst. Herefter begynder laves kode for den næste funktion i funktions-krop-listen. Hvis der ikke er flere er kodegenereringen ovre.

5.4 Køretidskontrol

Vores runtime-kontrol introducerer et nyt label i bunden af assembler-koden samt to nye labels, der gemmer stack og basepointer fra mainscope. Dette bliver brugt, når der findes en Køretids-fejl ved at den korrekte fejlkode lægges i `rax`, til at nulstiller stack pointer og base pointer til sådan som de stod i main scopet og derefter hopper til slutningen af main funktionen. På den måde lukkes programmet korrekt ned, med den relevante fejlkode i `%rax`. Out of bounds har kode 2, division med 0 har kode 3, negativ liste-allokering har kode 4, null pointer dereferering har kode 5 og out of memory har kode 6.

5.5 Afprøvning

For at afprøve at kode genereres korrekt, er der lavet en del små tests, der afprøver enkelte funktionaliteter af kitty-sproget og kontrollerer, om vi får det rigtige output. Derudover er der også testet nogle større programmer. Oversætteren er bl.a. testet på `O_Knapsack.src`, ved at køre programmet, og kontrollerer at outputtet, er det, der forventes. Denne fase er selvfølgelig kun testet på programmer, der går igennem typepetjekkeren uden problemer. Vi forventer at scan, parse, typecheck, weeder, og emit fungerer uden problemer.

Ud over at afprøve alle testfilerne ved at sikre at deres output er korrekt, er der for nogle få tests også lavet analyse af den interne repræsentation. Analysen af den interne repræsentation er lavet vha. en printer til at udskrive denne. Hvis resultatet af dette print ønskes set, når et givent Kitty-program oversættes, kan oversætterne køres med `-dASM`-flaget.

Testfilnavn	Beskrivelse	Forv.	Res.
O_Knapsack.src	Et komplekst program, der benytter de fleste funktionaliteter fra Kitty sproget og kræver korrekt allokering og static link mm.	foventer kørsel uden fejl og retur 50 369 600	PASS
LeftDerivation.kit	Program der kontrollerer, om vi navigerer venstre først når udtryk udledes kitty sproget.	kørsel uden fejl output 1 2 3 6	PASS
IfBoolExpr.kit	Program der indeholder `if-then`-handlinger. Testtet er brugt til at verificere at oversætteren kan lave disse strukturer.	kørsel uden fejl output 23 2	PASS
AllocNamed-Rec.kit	Alloker en navngivet record	kørsel uden fejl	PASS
AllocNamed-Arr.kit	Alloker en navngivet record	kørsel uden fejl	PASS
AccessField.kit	Tilgår felt i record	kører uden fejl	PASS

Testfilnavn	Beskrivelse	Forv.	Res.
LinkNestVar.kit	Test af static link. funktion defineret i yderste scope definere en rekursiv funktion der kalder sig selv et par gange og til sidst kalder en funktion ude fra main scope som printer en global værdi.	kørsel uden fejl, output 420 4200 42000 420000 4200000 1337	PASS

6 Register allokering

Som beskrevet i kapitel 5, så er alle midlertidige resultater lagt i såkaldte *midlertidige*, som er en midlertidig placering af disse inden vi finder deres endelige lokation.

Registerallokeringsfasen sørger for at lægge så mange af disse midlertidige i registre, som muligt, således at deres indhold kan tilgås signifikant hurtigere end, hvis de blot fik en placering i hukommelsen. Der er dog en begrænset mængde af registre, og derfor kan det i nogle tilfælde stadig være nødvendigt at placere noget data i hukommelsen. Vi definerer data til at være *levende* i et punkt i programmet, hvis denne data skal bruges senere i programmet, inden den overskrives. Hvis der i nuværende punkt skal skrives ny information til dataene før den bruges, så siges dataen at være død i øjeblikket. Det er op til register-allokatøren at bestemme hvor al levende data skal gemmes og findes.

Registerallokationen er selv opbygget af flere faser. Først analyseres den midlertidige programrepræsentation for at finde den data, der bliver læst og skrevet. Efterfølgende laves en livenessanalyse for at bestemme, hvor længe data er levende. På baggrund af informationen fundet i livenessanalysen, opbygges derefter en konfliktgraf, der beskriver hvilket data, der er levende på samme tid, og derfor ikke kan have den samme fysiske placering i maskinen, programmet kører på. Sidst laves selve registerallokeringen ved at udføre graffarvning på denne konfliktgraf.

Hver fase beskrives for sig herunder.

6.1 Livenessanalyse

Den første del af registerallokationen består i at analysere, hvornår forskelligt data er levende.

Analysens resultater gemmes i en liste af `LivenessInstructionArray`, med et element for hver instruktion i programmet. Hvert element er bygget op som det ses i listing 7. Efterhånden som information opsamles i forbindelse med livenessanalysen, vil den blive gemt i denne liste.

```
1  typedef struct LivenessInstructionArray{
2    TempList *in;
3    TempList *out;
4    TempList *use;
5    TempList *def;
6    int succ[3];
7    int isMove;
8  } LivenessInstructionArray;
```

Listing 7: Struktur indeholdende opsamlet liveness-information.

Hver gang en instruktion er blevet tilføjet til den midlertidige repræsentation, så gives denne instruktion et id, der repræsenterer, instruktionens placering i den midlertidige repræsentation, og der holdes styr på hvor mange instruktioner, der nu er i den midlertidige repræsentation.

Analysen starter med at gennemløbe hver instruktion i programmet for at finde og gemme information om hvilken data, der bliver brugt og hvilken data, der bliver defineret.

Helt konkret gøres dette ved at gemme hvilke midlertidige der bliver brugt og defineret. Dataene gemmes under det indeks i livenessanalyse-listen der repræsenterer den instruktions id.

For hver instruktion undersøges typen af denne instruktion, og på baggrund af denne viden kan det afgøres hvor mange operander, den har, og hvilke, der bliver læst og skrevet. Hvis disse operander viser sig at være midlertidige, så lægges disse ind i den releveante `tempList` i strukturen, der ses i listing 7, afhængig af om operanden bliver brugt eller defineret. For eksempel vil en `mov`-instruktion bruge den første operand og definere den anden. For `mov`-instruktioner gemmes ligeledes at det er en sådan, i feltet `isMove`, ved at sætte denne til værdien 1. Denne information skal bruges senere i analysen.

`tempList` er struktur, der implementerer mængder, via sorterede hægtede lister. Disse er beskrevet i afsnit 6.1.1.

Hvis en given instruktion viser sig at bruge indholdet af en midlertidig, så vil det betyde, at denne midlertidig er levende ved indgangen for denne instruktion. Derfor gemmes dette også umiddelbart med det samme i listen `in`.

For at kunne finde ud af hvilken data, der er levende ved en given instruktion, skal det vides hvilken data, der er levende i de efterfølgende instruktioner, som forklaret i afsnit 6.1.2. Derfor gemmes også information om hvilke instruktioner, der kan følge umiddelbart efter den nuværende instruktion. Som udgangspunkt vil det kun være den umiddelbart efterfølgende instruktion i det statisk skrevne program, der vil følge dynamisk efter denne instruktion i det dynamisk kørende program. Men hvis det er er en `jump`-instruktion, så kan den efterfølgende instruktion være et helt andet sted i det statiske program. Instruktionen, der skal hoppes til, findes ved at slå instruktionen op via det label, som `jump`-instruktionen angiver at hoppe til, i en hash-tabel (der benytter samme inplementation som symboltabellen). Denne tabel indeholder alle label-instruktioner. Id'et for den fundne label-instruktion gemmes i liveness analyse strukturen for den instruktion, der er et hop. Hvis hoppet er ubetinget, så sørges for at det kun er den fundne instruktion, der angives som efterfølger (eng. *successor*). Hvis det derimod er et betinget hop, så sørges der for, at det både er den fundne og den fysisk efterfølgende instruktion, der er angivet som efterfølger.

6.1.1 Mængdeimplementation for liveness-analysen

Som tidligere nævnt så er `in`-, `ud`-, `use`- og `def`-mængderne, brugt i liveness-analysen, implementeret via sorterede hægtede lister af midlertidige. Sorteringen er udelukkende for at kunne udføre en mere effektiv forening og difference mellem mængderne, så det er arbitrært valgt, at denne sortering laves på de midlertidiges unikke id'er. Selve indsætningen af et element i en mængde udføres ved at løbe mængden igennem enten til slutningen er nået, eller til der findes et element i mængden, der har et større id, end det nuværende element, hvilket betyder at elementet skal indsættes før dette element i ordningen.

Som det bliver beskrevet i afsnit 6.1.2, så er det i forbindelse med livenessanalysen nødvendigt at kunne udføre foreningen og difference. Forening udføres ved at tilføje

alle elementer i den ene liste til den anden liste. Dette gøres lidt naivt ved at undersøge om de enkelte elementer i den ene liste findes i den anden og hvis ikke, så indsættes de en ad gangen i den anden liste. Måden det er blevet implementeret på giver en asymptotisk kvadratisk køretid på længden af listerne, og det er efter projektets deadline indset at sorteringen af listerne kunne være udnyttet til at opnå en lineær køretid.

Differensen er implementeret en smule mere effektivt, og der opnås lineær køretid på størrelsen af de to mængder, idet de to mængder der skal subtraheres løbes igennem parallelt. Der oprettes en ny liste, der kun får tilføjet de elementer fra minuenden, der ikke er i subtrahenden.

6.1.2 Algoritmen

Implementationen af algoritmen, der udfører liveness analysen på baggrund af den opsamlede information beskrevet i afsnit 6.1, ses i listing 8. Implementationen er stærkt baseret på en lignende implementation beskrevet i [1].

Algoritmen benytter sig af følgende to udtryk, der beskriver sammenhængen mellem ind- og ud-mængderne for hver instruktion, der igen, angiver de midlertige, der er levende ved henholdsvist indgangen og udgangen til og fra hver instruktion.

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \qquad \forall \text{instruktion } n \qquad (1)$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] \qquad \forall \text{instruktion } n \qquad (2)$$

(1) beskriver hvilken data, der er levende ved indgangen til instruktion n på baggrund af den data, der bliver brugt i denne instruktion, som nævnt tidligere og den data, der bliver brugt senere, og derfor er ud-levende. Et data-element, som er ud-levende, men som også bliver defineret ved instruktion n, vil ikke have behov for at være ind-levende ved denne instruktion, da indeholdet i dette data element ikke er relevant, hvis det alligevel bliver defineret på ny, medmindre dataene skal bruges i denne instruktion. Dette forklarer differencen.

(2) beskriver den data, der er ud-levende for instruktion n som foreningen af den data, der er ind-levende i alle efterfølgende instruktioner. Det kunne udelades at vedligeholde ud-mængderne for hver instruktion, da al relevant information kunne vedligeholdes via ind-mængderne, men for nemheds skyld er begge mængde-typer alligevel valgt vedligeholdt i implementationen beskrevet her.

Algoritmen, beskrevet i listing 8, løber gentagende gange over livenessanalyse elementerne og opdaterer ind- og ud-mængderne i henhold til 1 og 2, indtil disse mængder ikke længere ændres. For at minimere antallet af gennemløb, løbes listen igennem bagfra, for at få samlet mest muligt information op hurtigt. Dette skyldes at liveness-informationen for en given instruktion n er baseret på informationen fra de efterfølgende instruktioner, som i de fleste tilfælde er placeret efterfølgende i den statiske kode. Dog grundet løkker og muligheden for at hoppe i koden generelt, kan det ske

at liveness-informationen for en instruktion afhænger af informationer fra instruktioner tidligere i den statiske kode, og dette er grunden til at algoritmen kan være nødsaget til at løkke over alle instruktioner flere gange. Derfor findes `while`-løkken på linje 3, der kører indtil, der ikke længere laves ændringer i ind- og ud-mængderne, ifølge betingelsen på linje 17. For-løkken på linje 5 løber over alle instruktionerne bagfra. For hver instruktion findes først differensen i (1), som forenes med den allerede eksisterende ind-mængde for denne instruktion. Dette kan gøres på den måde, da (1) og (2) ikke på noget tidspunkt kan fjerne elementer fra mængderne, selv hvis vi ikke havde den fuldendte information om en instruktionen fra starten. For-løkken på linje 11 sørger for at ud-mængden for instruktionen, kommer til at indeholde indeholdet af alle indmængderne for de efterfølgende instruktioner.

```
1  int livenessAnalysis(){
2    int isChanged, res;
3    while(true){
4      isChanged = 0;
5      for(int n=intermediateInstrCount - 1; n>=0; n--){
6        TempList *diff = listDiff(lia[n].out, lia[n].def);
7        res = listUnion(diff, lia[n].in);
8        freeList(diff);
9        if(res == -1){ return -1; }
10       isChanged += res;
11       for(int j=0; j<3; j++){
12         if(lia[n].succ[j] != -1){
13           res = listUnion(lia[lia[n].succ[j]].in, lia[n].out);
14           if(res == -1){ return -1; }
15             isChanged += res;
16         }}}
17     if(!isChanged){
18       break;
19   }}
20   return 0;
21 }
```

Listing 8: Implementation af livenessanalysen

6.2 Opbygning af konfliktgraf

Efter at al information om levende data for hver instruktion er fundet og gemt i de respektive ind- og ud-mængder opbygges en konfliktgraf, der beskriver den data, der er levende på samme tid, og derfor ikke kan få den samme fysiske lokation.

Konfliktgrafen består af en knude for hver midlertidig, og det er disse knuder, der manipuleres i de kommende faser af registerallokationen, inden den fundne allokation overføres til den midlertidige, som hver knude repræsenterer. En knudes indhold ses i listing 9. Alle knuder er samlet i en liste, og hver knude har et id, der svarer til knudens indeks i denne liste, således at en knude kan findes i konstant tid ud fra dens id.

```
1  typedef struct GraphNode {
2    int id; //used as an internal identifier
3    registers reg;
4    BITMAP *neighbors; //this one should stay here
```

```
5     int isMarked;
6     int inDegree;
7     int outDegree;
8     TEMPORARY *temp;
9   } GraphNode;
```

Listing 9: Indholdet af en grafknude.

Når grafen først er oprettet, så kan antallet af knuder ikke ændres. Dette er ikke nødvendigt, da der ved registerallokeringsfasen vides nøjagtig, hvor mange midlertidige der er i spil for et givent program, og der vil ikke efterfølgende skulle ændres på dette.

Fordelen ved at fiksere grafens størrelse er både, at der ikke skal reallokeres en ny liste af knuder, når grafen ændrer størrelse. Samtidig bliver det også nemmere at håndtere kanter, der er implementereret via bitmaps, da størrelsen af disse maps så heller ikke skal ændres fra tid til anden.

Implementationen af bitmappene er kort beskrevet i afsnit 6.2.1. Bitmappet i en knude angiver om der findes en orienteret kant fra knuden til en given nabo. Dette identificeres ved at sætte den bit, der er ved indekset i bit-mappet for id'et af den knude der er nabo til den pågældende knude.

Som det ses af listing 9, har hver knude en reference til en `TEMPORARY`. Dette er for senere at kunne overføre resultaterne fra analysen på grafen til de midlertidige.

6.2.1 Implementation graph via bitmaps

En kant fra en given knude til en nabo, indsættes eller fjernes ved hhv. at sætte eller nulstille (`reset'te`) bitten ved indekset for id'et af nabo-knuden i bitmappet for knuden, der har kanten ud fra sig. Fordelen ved at benytte bit-maps er at disse operationer så kan udføres i konstant tid. Prisen er at det bliver dyrere at finde listen af naboer til en knude, da det vil være nødvendigt at løbe alle bittene i gennem for at finde de bits, der er sat. Dette bliver derfor lineært i antallet af knuder. Hver knude i grafen har et felt, der angiver hhv. ind- og ud-graden for knuden, således at det hurtigt kan afgøres hvilken knude der har størst og mindst grad. Dette skal bruges senere.

Selve bitmappet er implementeret som en struktur, der har en liste af integers, der udgør bittene og en størrelse, der angiver hvor mange bits, der er i dette bitmap. For at finde antallet af integers (4-bytes størrelser), der skal bruges for at repræsentere bitmappet, divideres antallet af bits med 32 bits pr. integer, og hvis der er en rest, skal der bruges en ekstra integer.

At sætte en bit i bitmappet gøres ved først at finde den integer som indekset findes i, i listen af integers i bitmappet, ved division med 32 bits. Efterfølgende findes offsettet ind i denne integer ved modulo-beregning og tallet 1, shiftes lige så mange gange til venstre, som offsettet angiver, før det bitvist `or`'ed med tallet og resultatet lægges ind i den samme integer i mappet, for at skrive et 1-tal på denne plads. At nulstille en bit gøres på samme måde, men efter at have skiftet 1-tallet til venstre, så negeres resultatet inden det `and`es med integeren i bitmappet og resultatet lægges i denne integer.

6.2.2 Algoritme

På baggrund af en liste af alle de midlertidig, der ønskes at udføres livenessanalyse på, opbygges en graf, der til at starte med blot indeholder en knude for hver midlertidig. Herefter løbes informationen gemt i liveness-instruktionslisten igennem for hver instruktion, som det ses af listing 10. For hver instruktion, løbes der over alle midlertidige, der bliver defineret i denne instruktion på linje 3. Hver af disse midlertidige sammenlignes med de midlertidige der er ud-levende i den pågældende instruktion, på linje 5. Som udgangspunkt laves der en kant mellem de to knuder, der repræsentere disse midlertidige på linje $7-12$, medmindre det er en `mov`-instruktion og den definerede midlertidig er den samme som den midlertidig, der er levende ud. Dette er for at repræsentere at den nyligt definerede midlertidig ikke kan have den samme lokation som alle de midlertidige, der er ud-levende, medmindre det er kilden af en `mov`-instruktion, da kilden og destinationen af en `mov`-instruktion begge vil indeholde den samme værdi, og derfor kan være placeret samme sted.

```
1  for(int n=0; n<intermediateInstrCount; n++){
2      defNode = lia[n].def->head;
3      while(defNode != NULL){
4        liveNode = lia[n].out->head;
5        while(liveNode != NULL){
6          if((!lia[n].isMove) || defNode->temp != liveNode->temp){
7            error = IGinsertNeighbor(defNode->temp->graphNodeId,
8              liveNode->temp->graphNodeId);
9            if(error == -1){ return -1; }
10           error = IGinsertNeighbor(liveNode->temp->graphNodeId,
11             defNode->temp->graphNodeId);
12           if(error == -1){ return -1; }
13         }
14         liveNode = liveNode->next;
15       }
16       defNode = defNode->next;
17     }
18   }
```

Listing 10: Indsætning af interferenskanter i interferensgrafen på baggrund af den information, der er opsamlet i livenessanalysen

6.3 Graffarvning

Som nævnt i begyndelsen, er den sidste fase af registerallokeringen at udføre graffarvning af konfliktgrafen, opbygget som beskrevet i afsnit 6.2, således at ingen forbundne knuder har den samme farve.

6.3.1 Algoritmen

Implementationen af graffarvningen ses i listing 11 nedenfor, der benytter farvning ved simplficering. Algoritmen er baseret på et eksempel beskrevet i [1]. Implementationen benytter sig af 8 farver svarende til regstre `R8` til `R15`. Hvis der viser sig at være for

meget levende data på samme tid, til at det alt sammen kan være i et register, så bliver der udvalgt noget data, der skal *spill*'es, og dermed placeres på program-stakken.

Algoritmen starter med gentagende gange at fjerne knuder fra grafen og lægge dem over på en stak (linje 3-15, listing 11). Denne stakken er implementeret som en hægtet liste af integers, hvor nye elementer skubbes på og tages af toppen. Det vil sige, at når en graf-knude lægges på stakken, så er det blot knudens id, der lægges på stakken. Implementationen vil ikke blive yderligere beskrevet. Fjernelse af knuder fra grafen gentages indtil, der ikke længere er flere knuder i grafen, hvorefter grafen genopbygges i baglæns rækkefølge ved at poppe knuder fra stakken samtidig med at de farves (linje 16-30). Når knuderne fjernes fra grafen, fjernes alle kanter fra de tilbageværende knuder i grafen til den pågældende knude, men ikke i den modsatte retning, på linje 10. På den måde ved en knude, der bliver lagt på grafen nøjagtig hvilke af dens naboer, der er farvet, når knuden poppes af stakken og selv skal farves, da den stadig er forbundet til disse knuder. Den skal dog ikke *bekymre* sig om knuder der allerede er på stakken, og derfor først farves efter denne knude.

Algoritmen benytter en heuristik, der som udgangspunkt forsøger at fjerne knuder med færrest konflikter og kun knuder, der har færre konflikter med (tilbageværende) knuder i grafen, end der er farver til rådighed. På den måde ved vi at denne knude altid vil kunne farves med en af farverne, selv hvis de knuder, der er i konflikt, alle har en forskellig farve. Hvis der på et tidspunkt ikke findes en knude med færre konflikter, end der er er farver, så kan man overveje om det skal være den knude med mindst grad, der alligevel skal fjernes fra grafen, eller om der skal vælges en anden. Egentligt var tanken at det skulle være den knude med størst grad der blev fjernet, da vi, når vi potentielt skal lægge en midlertidig på stakken, lige så godt kan vælge den, der løser flest konflikter. Denne implementation er dog gået tabt i den afleverede oversætter. I stedet vælges knuden med den mindste grad. Fordelen ved dette er, at der så faktisk er større sandsynlighed for, at denne midlertidige ikke alligevel skal lægges på stakken, når farvningen er gennemført, da et færre antal naboer minimere risikoen for at alle farver er dækket ind af disse naboer. Samtidig kan graden af knude med høj grad skyldes, at dens repræsenterende midlertidig lever længe. Dette kan igen betyde, at den bliver tilgået mange gange. Det vil derfor være uheldigt at lægge en sådan midlertidig på stakken. Men da en knude med høj grad har mange konflikter, skal den højstsandsynligt lægges på stakken alligevel.

Når der ikke er flere knuder i grafen, så poppes hver knude af stakken igen, mens de farves. Der undersøges for hver mulig farve, om den er til fælles med nogle af naboerne for den knude, og hvis en farve er det, så springes der videre til næste farve (linje 23-24). Når der findes en farve, der ikke er tilfælles med nogle naboer, så farves, knuden med denne farve og der kan poppes en ny knude fra stakken på linje 26 og 29 i listing 11.

Hvis der ikke er nogle ledige farver, så bliver knuden markeret som spilled på linje 28, og den midlertidig, den repræsenterer, vil blive placeret på stakken.

Når der ikke er flere knuder tilbage på graf-stakken, så er farvningen færdig, og farvningen kan direkte overføres fra hver knude i grafen til den midlertidig den repræsenterer. Det vil sige at for midlertidige, der ikke er spilled, vil den blive omdannet til en

register-placering, der angiver det register, som den korresponderende graf-knude er blevet farvet med. Hvis graf-knuden og dermed den midlertidig, knuden repræsenterer, er blevet spilled, så får den midlertidige lov til at beholde den program-stak-position, den var tildelt i første omgang.

```
1   int colorCount = R15–R8+1;
2   Stack *graphStack = stackCreate();
3   int lowestID = IGlowestOutDegree();
4   while(lowestID != -1){
5     GraphNode lnode = graphNodes[lowestID];
6     if(lnode.outDegree >= colorCount){lnode.reg = SPILL;}
7     int *neighbors = IGgetNeighbors(lowestID);
8     for(int i = 1; i <= neighbors[0]; i++){
9       if(neighbors[i] == lowestID){continue;}
10      IGremoveNeighbor(neighbors[i],lowestID);
11    }
12    stackPush(graphStack, lowestID);
13    graphNodes[lowestID].isMarked = 1;
14    lowestID = IGlowestOutDegree();
15  }
16  int i = stackPop(graphStack);
17  while(i != -1){
18    int *neighbors = IGgetNeighbors(i);
19    int colorFound = 1;
20    for(registers reg = (registers)R8;
21      reg <= (registers) R15; reg = (registers) (reg+1)){
22      colorFound = 1;
23      for(int j = 1; j<=neighbors[0]; j++){
24        if(graphNodes[neighbors[j]].reg == reg){colorFound = 0; break;}
25      }
26      if(colorFound){graphNodes[i].reg = reg; break;}
27    }
28    if(!colorFound){graphNodes[i].reg = SPILL;}
29    i = stackPop(graphStack);
30  }
31  return 0;
32 }
```

Listing 11: Implementation af graffarvning for endelige register allokering

6.4 Afprøvning

Alle de test, der er beskrevet i afsnit 5.5 og alle test derudover der findes i det medfølgende testmiljø er alle blevet kørt med liveness implementeret, og der er ikke set nogle ændringer i hvilke test der fejler.

Derudover er der for 3 af testene, som tager længere tid at køre, lavet nogle målinger på den tidsmæssige speedup, når programmet oversættes hhv. med og uden livenessanalysen. Alt output til terminalen er undertrykt så print ikke har haft indflydelse på tidstagningen. Tidsmålingerne er blevet foretaget med den indbyggede `time`-systemkommando i linux. Alle tider vist i tabellen er et gennemsnit af 10 målinger og enheden er i sekunder.

Testfilnavn	Beskrivelse	Kørsel u. liveness (s)	Kørsel m. liveness (s)
O_Knapsack.src	Implementation knapsacks problemet	53.35	28.98
fibbo.kit	Finder det 47 fibbonacci tal.	102.19	56.73
primeFactor.kit	Primatalsfactorisere tallene fra 1 til 2000000	10.34	7.87

For de længere test ser vi en hastighedsforøgelse på omkring en faktor 1.8, mens den for `primeFactor` er på omkring 1.3.

Hermed ses det tydeligt at registerallokeringen har haft indflydelse på de oversatte programmers køretid.

7 Kighulsoptimering

Kighulsoptimeringen har til formål at reducere mængden af ubrugelig assemblerkode genereret af kodegeneratoren i oversætteren. Når oversætteren genererer den midlertidige repræsentation af den endelige assemblerkode, der skal implementere brugerens program, gøres det, som beskrevet i afsnit 5, via en masse små skabeloner, der sættes sammen. Dette kan skabe nogle uhensigtsmæssige sekvenser af kode, der kan udelades eller optimeres.

Som det vil blive beskrevet i afsnit 7.1 så er peephole-implementeringen lavet ved gentagene gange at løbe over koden i den interne repræsentation, og for hver linje af kode, afgøre om der her findes et mønster, der kan optimeres på. I så fald udføres optimeringen på baggrund af et implementeret mønster. Disse mønstre er hver især implementeret som en funktion, der først tjekker om mønstret findes ved den nuværende instruktion og i så fald laver den optimerende ændring af mønsteret på den interne repræsentation.

7.1 Algoritmen

Algoritmen, der implementerer det generiske gennemløb af koden, ses som pseudokode i listing 12. Det er under selve kørslen af mønsteret, at det tjekkes om det passer på den nuværende instruktion og programmet optimeres hvis det er muligt. Algoritmen er inspireret fra [2].

```
1    Indtil der ikke sker nogen ændring
2    gentag:
3        For hver instruktion i programmet
4            For hvert mønster
5                Kør mønster på denne instruktion
```

Listing 12: Pseudokode der beskriver implementationen af kighulsoptimeringen i oversætteren

7.1.1 Mønstre

Der er i alt implementeret 7 mønstre, der udgør den kighulsoptimering, der er implementeret i oversætteren. Disse er listet herunder angivet med navnet på den funktion, der beskriver mønstret i oversætteren. Som det ses af beskrivelserne herunder så benytter flere af mønstrene resultater fra registerallokeringsfasen, og kighulsoptimeringen kan derfor ikke umiddelbart køres uden den fase.

- wastedMovq: Her optimeres mønsteret:

```
mov %rbx t
mov t %rbx
```

`t` angiver en midlertidig. Den anden linje kan altid fjernes i dette tilfælde, da den blot genskriver `%rbx` med den information der allerede findes der. Eftersom den anden instruktion fjernes kunne man fristes til at tro, at den første også kan fjernes. Det er dog kun tilfældet hvis `t` ikke bliver brugt senere. Derfor fjernes den første instruktion kun hvis `t` ikke er ud-levende i den anden instruktion. I så fald vil `t` aldrig blive brugt og den første `mov` instruktion kan fjernes.

- wastedMovqSeq: Her omdannes en sekvens af to `mov`-instruktioner til en enkelt. Dvs. sekvensen

```
mov a b
mov b c
```

  omdannes til

```
mov a c
```

  hvor `a, b, c` kan være både midlertidige og registre. Optimeringen kan dog kun udføres, hvis `b` ikke er ud-levende ved den anden instruktion. Optimeringen udføres selvom de to `b`'er ikke er det samme element (eks midlertidig), men blot er tildelt det samme register.

- moveToSelf: Denne fjerner instruktioner på formen

```
mov %r %r
```

  hvor `%r` kan være ethvert register. Det kan ligeledes være (forskellige) midlertidige tildelt til samme register. Mønsteret opstår på baggrund af kørsler af *wastedMovSeq*, hvis `a=c`.

- addZero: Fjerner instruktioner på formen

```
add 0 a
```

  hvor `a` kan være hvad som helst.

- addOne: Erstatter instruktioner på formen

```
add 1 a
```

  med

```
inc a
```

  hvor `a` kan være hvad som helst.

- incPattern/decPattern: Erstatter mønstre på formen

```
mov 1 %rbx
add/sub %rbx t
```

med

```
inc/dec t
```

hvor `t` er en midlertidig.

7.1.2 Termineringsfunktion

For at sikre at kighulsoptimiseren ikke kører uendeligt, kan man på baggrund af de 7 mønstre beskrevet i afsnit 7.1.1 lave en termineringsfunktion, der lader ethvert lovligt program blive mappet til antallet af linjer, programmet fylder plus antallet af `add`-instruktioner i programmet. Det er let at se, at optimeringen kun vil give faldende funktionsværdi, når der udføres optimering, da alle mønstre fjerner mindst en instruktion uden at tilføje noget bortset fra *addZero*, der dog erstatter en `add`-instruktion med en `inc` instruction, der derfor også leder til en faldende funktionsværdi, da antallet af `add` instruktioner reduceres. Denne mapping sikre at optimeringen vil stoppe på et tidspunkt, da der ikke vil være nogen programmer, der giver en negativ værdi ifølge mappingen, og mappingen reducere værdien ved hver optimering.

7.2 Afprøvning

Som ved afprøvning af registerallokeringen er alle testprogrammer fra afsnittet 5.5 inklusiv yderligere test angivet i det udleverede testmiljø brugt til at teste kighulsoptimering. Det kan af denne test afgøres at den implementerede kighulsoptimering ikke har introduceret nogle nye problemer i oversætteren.

Derudover er der målt kørselstid for 3 længerevarende programmer. Resultaterne af disse ses i tabellen herunder. Alle resultater er fundet på baggrund af gennemsnittet af 10 kørsler. Tider er vist i sekunder. Alle kørsler er lavet med registerallokering, da kighulsoptimeringen benytter resultater fundet i livenessanalyse, der derfor ikke kan udelades.

Testfilnavn	Beskrivelse	Kørsel u. optim. (s)	Kørsel m. optim. (s)
O_Knapsack.src	Implementation knapsacks problemet	29.71	23.27
fibbo.kit	Finder det 47 fibbonacci tal.	56.91	52.38
primeFactor.kit	Primatalsfactoriserer tallene fra 1 til 2000000	7.89	6.84

Igen ses det, at der findes en hastighedsforøgelse, når optimeringen medtages, som forventet. Vi ser dog også, at det er en noget mere begrænset forøgelse sammenlignet med det tilsvarende resultater set for registerallokering. Dette skyldes selvfølgelig, at det at fjerne en instruktion eller ændrer den til en mere effektiv udgave ikke skaber en nær så stor tidsmæssig besparelse, som hvis man kan undgå at tilgå hukommelsen, som er det registerallokeringen forsøger. Det kan dog sagtens tænkes at kighulsoptimeringen kan fjerne tilgange til hukommelsen hvis data kun lægges på stakken som en midlertidig placering inden det straks flyttes tilbage til et register.

8 Emit

Efter den interne output-kode repræsentation er blevet genereret og er blevet efterbehandlet ved livenessanalysen og kighulsoptimeringen, er den sidste fase af oversætteren nået: Emit af den endelige assemblerkode.

Denne fase løber ganske simpelt over den hægtede liste, der beskriver den interne repræsentation og for hver instruktion, printes Intel x86 AT&T med GAS syntaks assembler koden ud. Dette gøres ved for hver instruktion at printe assembler navnet for den instruktionstype, der skal til at skabes og efterfølgende løbe operandlisten for denne instruktion igennem for at printe instruktionens operander.

Hvis operanden er en konstant eller et register, så kan dette printes direkte til det endelige output, når den findes i den interne repræsentation. Er en operand derimod en midlertidig, der stadig er placeret på stakken, så skal adressen for denne midlertidig findes og derefereres før operandens indhold kan tilgåes.

Derfor undersøges operandlisten for hver instruktion før instruktionen selv printes af funktionen `checkOffsetOperand`. Hvis det viser sig at en af operanderne skal findes på stakken, og det er en midlertidig eller en lokal variabel, så sørger denne funktion for at flytte operandens offset, angivet i midlertidig'en, over i registeret `%rdx`, med negativt fortegn, da midlertidige og lokale variabler lægger over base-pointeren for den pågældende stak-ramme. Når operand-listen gennemløbes, så findes stak-operandens indhold ved at dereferere på følgende måde: `(%rbp, %rdx, 8)`. På den måde angives at vi vil indeksere 8 gange det angivne offset i `%rdx` over `%rbp`. Hvis det er en lokal variabel eller en parameter der skal tilgås, så bruges `%rdi` istedet for `%rbp`, da denne indeholder base-pointeren for det scope, som dataet findes i, hvilket kan være forskelligt fra det lokale scope, som beskrevet i afsnit 5. Hvis det er en parameter, der skal tilgåes, så findes offsettet for denne parameter på samme måde, men i stedet for at angive det negativt, så bliver det angivet positivt, da parametre i modsætning til lokale variabler og midlertidige ligger dybere i stakken end basepointeren.

Det skal ligeledes nævnes, at der er en invariant fra kodegenereringen at ingen instruktion på noget tidspunkt kan tilgå mere end en stak-operand. Derfor vil det altid højst være et offset, der er nødvendigt at finde og gemme i `%rdx`, for hver instruktion.

I forbindelse med liste-indeksering eller tilgang til felter i records kan det være nødvendigt at skulle dereferere en midlertidig. Hvis den midlertidige er placeret i et register, så flyttes indholdet blot over i registret `%rcx`. Er det derimod en midlertidig på stakken, der skal derefereres, så sørger `checkOffsetOPerand()` for at lægge det korrekte offset i `%rdx` som beskrevet og efterfølgende at flytte indholdet fra denne stak-adresse over i `%rcx`. Når instruktionen med den derefererede operand printes, så sørger emitteren blot for at printe `(%rcx)` for at tilgå den derefererede data af operanden.

8.1 Eksempelkode

Den fulde udgave af den kode der er generet i de følgende eksempler kan findes i afsnit D i bilagene.

8.2 Factorial

Herunder ses et kitty-program, der bruger en generisk fakultets-funktion til at beregne 5!.

```
1 func factorial(n: int): int
2     if (n == 0) || (n == 1) then
3         return 1;
4     else
5         return n * factorial(n-1);
6 end factorial
7
8 write factorial(5);
```

Listing 13: Kitty program til at beregne 5!

Assemblerkoden som oversætteren skaber for at implementere programmet i assembler ses herunder. Programmets funktionalitet i relation til listing 13 beskrives her i grove træk. Programmet starter ved `main:` labelen på linje 17. På linje 18-25 laves alle callee-save operationerne inden selve programmet bliver startet op. På linje 26-30 laves opsætning af heapen og main-scopets stak og base-pointer gemmes til brug for korrekt progamafbrydelse ved fejl. Denne del er beskrevet nærmere i de senere eksempler. På linje 31 og nedefter laves koden til den ene `write`-handling, som findes i mainscopet på linje 8 af listing 13. På de efterfølgende linjer udføres caller-save prologen, der skal forberede kaldet til `factorial`-functionen. På linje 41 skabes der plads til en parameter og på linjerne 42-45 lægges værdien 5 ind som parameter. Selve kaldet laves på linje 48, og på de efterfølgende linjer laves caller-save epilogen, der rydder op efter funktionsskaldet. Efter funktionen har returneret, så lægges svaret i `%rax`, så på linje 58 lægges dette resultat over i en midlertidig, der viser sig at være placeret i `%r8`. Linjerne 60-63 laver printet af resultatet via formateringen angivet på linje 2. Det ses af dette, at oversætteren kun kan håndtere at printe heltal. På linje 65 flyttes 0 over i `%rax` for at indikere at programmet er afsluttet med succes, og på linjerne 66-76, laves callee epilogen, der rydder op efter main-scopets kørsel og returnere.

Funktionen `factorial` begynder på linje 79 og fortsætter til linje 175. Denne funktion starter ligeledes med callee-prologen. På linje 87 gøres der plads til 13 lokale variabler og midlertidige. På linje 95 laves sammenligningen, der afgør resultatet af venstresiden af disjunktionen af betingelsen på linje 2 i listing 13. Hvis den er falsk så flyttes der 0 over i `%r9` på linje 97, og ellers er der allerede et 1-tal i `%r9` fra linje 89. Det tjekkes på linje 100 om venstresiden er sand. I så fald er det nemlig ikke nødvendigt at tjekke højresiden. og der springes til `lazy11` labelen på linje 115. Ellers tjekkes højresiden af disjunktionen på linje 103-114 (helt specifikt på linje 109). På linje 116, tjekkes om resultatet af disjunktionen er sand (dvs. 1), og i så fald kan værdien 1 sættes til at returneres på linje 119 og der springes til slutningen af funktionen, jf. linje 3 af listing 13. Hvis betingelsen ikke kunne opfyldes så springes der i stedet til linje 122 hvor et rekursivt kald skal laves. Først læses parameteren ind i `%r10` på linje 124-126. Bemærk hvordan basepointeren først lægges over i `%rdi`, da det er det lokale scope, hvorefter offsettet 3 for parameteren lægges i `%rdx` inden der derefereres på line 126. Efterfølgende gøres der klar til et funktionskald på linje 128-138. På linje 139 gøres der plads til argumentet, der er defineret som funktionens egen parameter,

fundet på linje 140-142, substraheret 1, hvilket gøres på linje 144, før den lægges ind på sin ”parameter-plads”på linje 147. På linje 150 laves det rekursive kald, og resultatet af dette bliver på linje 161 ganget med parameteren i `%r10`, og resultatet lægges i `%rax`, før der gøres klar til at returneres på linje 166-174.

```
1
...     Data section
16  .globl main
17  main:
...     Callee Prolog
25
...     Setup heap
30
32                              #line: 8 write statement
33      movq %rbp, %r9
...     Caller Prolog
41      subq $8, %rsp
42      movq %rsp, %r8
43      movq $5, %rbx
44      movq %r8, %rcx
45      movq %rbx, (%rcx)
46      movq %r9, %rbx
47      push %rbx
48      call factorial7
49      addq $16, %rsp          #remove static link and parameters
...     Caller Epilog
58      movq %rax, %r8
59      push %rdi
60      movq %r8, %rsi
61      movq $format, %rdi
62      movq $0, %rax
63      call printf
64      pop %rdi
65      movq $0, %rax
66  mainend:
67      addq $16, %rsp
...     Callee Epilog
75      ret
...
79  factorial7:
...     Callee Prolog
87      subq $104, %rsp
88                              #line: 5 if then else statement
89      movq $1, %r9
90      movq %rbp, %rdi
91      movq $3, %rdx
92      movq (%rdi,%rdx,8), %r8
93      movq %rbp, %rdi       #resetting basepointer
94      movq $0, %rbx
95      cmp %rbx, %r8
96      je cmp12
97      movq $0, %r9
98  cmp12:
99      movq %r9, %rbx
100     cmp $1, %rbx
101     movq %rbx, %r8
102     je lazy11
```

```
103     movq $1, %r10
104     movq %rbp, %rdi
105     movq $3, %rdx
106     movq (%rdi,%rdx,8), %r9
107     movq %rbp, %rdi      #resetting basepointer
108     movq $1, %rbx
109     cmp %rbx, %r9
110     je cmp13
111     movq $0, %r10
112 cmp13:
113     movq %r10, %rbx
114     or %r8, %rbx
115 lazy11:
116     cmp $1, %rbx
117     jne else10
118                          #line: 3 return statement
119     movq $1, %rax
120     jmp factorial7end
121     jmp endif10
122 else10:
123                          #line: 5 return statement
124     movq %rbp, %rdi
125     movq $3, %rdx
126     movq (%rdi,%rdx,8), %r10
127     movq %rbp, %rdi          #resetting basepointer
128     movq %rbp, %rbx
129     addq $16, %rbx
130     movq (%rbx), %r11
...     Caller Prolog
139     subq $8, %rsp
140     movq %rbp, %rdi
141     movq $3, %rdx
142     movq (%rdi,%rdx,8), %r8
143     movq %rbp, %rdi          #resetting basepointer
144     decq %r8
145     movq %rsp, %r9
146     movq %r9, %rcx
147     movq %r8, (%rcx)
148     movq %r11, %rbx
149     push %rbx
150     call factorial7
151     addq $16, %rsp           #remove static link and parameters
...     Caller Epilog
160     movq %rax, %rbx
161     imulq %r10, %rbx
162     movq %rbx, %rax
163     jmp factorial7end
164 endif10:
165 factorial7end:
166     addq $104, %rsp
...     Callee Epilog
174     ret
...
178 errorCleanup6:
179     movq mainSPoint4, %rsp
180     movq mainBPoint5, %rbp
181     jmp mainend
```

8.3 Liste-allokation og -brug

I dette afsnit er beskrevet et eksempel på brug af records inde i en liste. Et simpel kitty-program, der benytter denne funktionalitet ses i listing 14.

```
1  var x: array of record of {a:int, b:bool};
2  allocate x of length 30;
3  allocate x[3];
4  x[3].a = 7;
5  write x[3].a;
```

Listing 14: Kitty program, der benytter liste og record allokation

Den genererede assembler kode, der implementerer dette program ses i listing 15. Koden er oversat uden at generere kode for køretidsfejl, for at undgå at koden ville blive for overvældende at indsætte her.

```
1    format:
2    .string "%d\n"
3    .data
4    .align 8
5    heap1:
6    .space 1048576
7    freeHeap2:
8    .space 8
9    endHeap3:
10   .space 8
11   mainSPoint4:
12   .space 8
13   mainBPoint5:
14   .space 8
15   .text
16   .globl main
17   main:
18
...      Callee Prolog
24
25       subq $104, %rsp
26       movq %rsp, mainSPoint4
27       movq %rbp, mainBPoint5
28       movq $heap1, freeHeap2
29       movq $heap1, endHeap3
30       addq $1048576, endHeap3
31                          #line: 3 allocate of length statement
32       movq %rbp, %rdi
33       movq freeHeap2, %r8
34       movq $-6, %rdx
35       movq %r8, (%rdi,%rdx,8)
36       movq %rbp, %rdi  #resetting basepointer
37       movq $30, %r9
38       movq %r9, %rbx
39       incq %rbx           #making room for arraySize
40       imulq $8, %rbx
41       addq %rbx, freeHeap2
42       movq %r8, %rcx
43       movq %r9, (%rcx)
44                          #line: 4 allocate statement
```

```
45      movq %rbp, %rdi
46      movq $-6, %rdx
47      movq (%rdi,%rdx,8), %r8
48      movq %rbp, %rdi  #resetting basepointer
49      movq $3, %rbx
50      incq %rbx           #moving past array-size-value
51      imulq $8, %rbx
52      addq %rbx, %r8
53      movq freeHeap2, %rbx
54      movq %r8, %rcx
55      movq %rbx, (%rcx)
56      addq $16, freeHeap2
57                          #line: 5 assign statement
58      movq $7, %r9
59      movq %rbp, %rdi
60      movq $-6, %rdx
61      movq (%rdi,%rdx,8), %r8
62      movq %rbp, %rdi     #resetting basepointer
63      movq $3, %rbx
64      incq %rbx           #moving past array-size-value
65      imulq $8, %rbx
66      addq %rbx, %r8
67      movq %r8, %rcx
68      movq (%rcx), %r8
69      movq %r8, %rcx
70      movq %r9, (%rcx)
71                          #line: 6 write statement
72      movq %rbp, %rdi
73      movq $-6, %rdx
74      movq (%rdi,%rdx,8), %r8
75      movq %rbp, %rdi     #resetting basepointer
76      movq $3, %rbx
77      incq %rbx           #moving past array-size-value
78      imulq $8, %rbx
79      addq %rbx, %r8
80      movq %r8, %rcx
81      movq (%rcx), %r8
82      push %rdi
83      movq %r8, %rcx
84      movq (%rcx), %rsi
85      movq $format, %rdi
86      movq $0, %rax
87      call printf
88      pop %rdi
89      movq $0, %rax
90  mainend:
91      addq $104, %rsp
92
...     Callee Epilog
98
99      ret
...
103 errorCleanup6:
104     movq mainSPoint4, %rsp
105     movq mainBPoint5, %rbp
```

106 **jmp** mainend

Listing 15: Genereret assemblerprogram for det kittyprogram, der ses i listing 14.

Som beskrevet i afsnit 8.2, starter programmet med noget opsætning, hvilket vil blive uddybet en smule nu. På linje 5 genereres en label, der peger på starten af heapen med plads til i alt 1MB. På linje 7 og 9 genereres to labels, begge med plads til en hukommelsesaddresse. Meningen med disse er at de addresser, disse labels peger på skal indeholde hhv. den næste frie heap-addresse og addressen for slutningen af heapen. Disse værdier sættes som noget af det første i `main` funktionen, på linjen 28 og 29 i listing 15. På linje 31 til 43 allokeres listen x, svarende til linje 2 i listing 14. Indeholdet af `freeHeap2` labelen flyttes til `%r8` og videre til den lokale variabel `x`, der viser sig at have offset -6 på stakken, på linje 33 til 35. På den måde vil `x` og `%r8` nu indeholde en pointer til den frie plads. På linje 37 lægges liste-størrelsen over i `%r9`. Linje 39 og 40 tæller antallet af placeringer, der skal allokeres en gang op, således at størrelsen af listen kan lægges ind på den første plads og der ganges med 8, da hvert indeks i listen fylder 8 bytes. Resultatet lægges nu til `freeHeap2` labelen, således at den nye allokation er gemt. På linje 42-43 derefereres værdien i `%r8` der er addressen for den nyligt allokerede liste, og værdien, der angiver længden af listen, lægges der. På den måde vil første position af listen indeholde dets længde.

På linje 3 af listing 14 bliver den record, der er på indeks 3 af listen allokeret. Dette sker på linje 44-56 i listing 15 herunder. På linje 45-47 findes først indholdet af variablen `x`, der er en pointer til listen. På linje 49 til 52 findes så den egentlige addresse for indeks 3 af listen, ved først at lægge en til, da vi ønsker at springe første offset af listen over, der angiver listens størrelse, og efterfølgende gange med 8. Den fundne værdi lægges til start-addressen for listen på linje 52. På linje 53-55 gemmes pointeren til den næste frie plads på denne adresse. Linje 56 sørger for at den nødvendige mængde af plads bliver allokeret. Størrelsen af recorden kendes ved oversættertid til at være 2 felter, og derfor vides at dette er 16 bytes.

Linjerne 58 til 71 gennemfører den tildeling der er på linje 4 af listing 14. Først findes resultatet af højresiden, der blot er et 7-tal og gemmes i `%r9` på linje 58. På linje 60 til 68 findes indholdet af det 3. indeks af listen, da dette indeks skulle allokeres. Adressen fundet her derefereres, da det vides, at det er en record, der findes på denne placering. Det gøres på linje 68. Vi skal nu have skrevet værdien i `%r9` til det første felt i recorden, hvilket svarer til den adresse, som det tredje indeks i listen peger på. Det er derfor hvad der bliver gjort på linje 69-70.

`write`-handlingen fungerer på stort set samme måde og vil ikke blive yderligere beskrevet her. På linje 89 og frem til 99 findes callee-epilogen for main funktionen.

8.4 Fejlafslutning

De to programeksempler beskrevet i afsnit 8.2 og 8.3 har begge en funktion kaldet `errorCleanup6` angivet til sidst. Denne funktion bliver kaldt, hvis der i løbet af programmet blive fundet en køretidsfejl. I dette tilfælde vil en relevant retur-værdi blive lagt i `%rax` og der vil blive hoppet til `errorCleanup6`. Denne sørger for at

gendanne stak- og basepointeren inden der springes direkte til afslutningen på main-funktionen, der så returnerer, med den angivne værdi i `%rax`.

Herunder ses et meget lille kitty program, der allokere en liste.

```
1  var arr:array of int;
2  allocate arr of length 8;
3  arr[4] = 11;
```

Listing 16: Listeallokering i Kitty

Dette program er oversat til programmet i listing 17. I modsætning til programmet beskrevet i afsnit 8.3, så er køretidsfejl-tjek ikke fjernet her i denne oversætning. Til gengæld er der udeladt noget af den opsætning der allerede er vist i de tidligere eksempler. Det er indikeret ved en #*** ***#-sekvens.

Noget af det første der bliver gjort i main-funktionen er at stak- og base-pointeren bliver gemt under lablerne `mainSpoint4` og `mainBPoint5` på linje 26 og 27 i listing 17

På linje 31 til 56 laves allokeringen af listen fra linje 2 i listing 16 ligesom beskrevet i afsnit 8.3. Dog er der tilføjet ekstra kode til køretidsfejl tjekkene.

På linje 39-40 tjekkes det om allokationsstørrelsen er større end 0. Hvis ikke den er det, så lægges værdien 4 ind i `%rax`, der indikerer dette og der hoppes til `errorCleanup6` på linje 42. På linje 49-50 tjekkes det at en allokation ikke går ud over slutningen af den totale heap størrelse. Dette gøres ved at sammenligne om den nye `freeHeap2`-adresse ligger efter `endHeap3` adressen. Hvis ikke der er allokeret ud over den totale heap plads, så lægges listens størrelse ind på første indeks på linje 54-55 som beskrevet tidligere.

Når et liste-element skal tilgås så tjekkes det på linje 62 først at listen ikke er en `NULL`-pointer. Efterfølgende tjekkes det på linje 69-72 at det angivne indeks ikke er negativt eller går ud over listens størrelse. Efterfølgende findes adressen for det angivne indeks, hvis ikke der blev fundet nogle fejl indtil nu, og når denne er fundet, så tjekkes om denne adresse er `NULL` på linje 80-81 (det kan diskuteres om det tjek er relevant). Hvis ingen fejl blev fundet indtil nu, så kan tildelingen udføres nøjagtig som forklaret afsnit 8.3. Dette gøres på linje 85-86.

```
1
... Data sektion
15  .text
16  .globl main
17  main:
...     Callee prolog
25      subq $56, %rsp
26      movq %rsp, mainSPoint4
27      movq %rbp, mainBPoint5
...     Heap pointers setup
31                      #line: 2 allocate of length statement
32      movq %rbp, %rdi
33      movq freeHeap2, %r8
34      movq $-6, %rdx
35      movq %r8, (%rdi,%rdx,8)
36      movq %rbp, %rdi     #resetting basepointer
37      movq $8, %r9
```

```
38      movq %r9, %rbx
39      cmp $0, %rbx
40      jg allocPos8
41      movq $4, %rax       #negative allocation size
42      jmp errorCleanup6
43  allocPos8:
44      movq %r9, %rbx
45      incq %rbx           #making room for arraySize
46      imulq $8, %rbx
47      addq %rbx, freeHeap2
48      movq freeHeap2, %rbx
49      cmp endHeap3, %rbx
50      jl allocSucc9
51      movq $6, %rax       #outofMemory
52      jmp errorCleanup6
53  allocSucc9:
54      movq %r8, %rcx
55      movq %r9, (%rcx)
56                          #line: 3 assign statement
57      movq $11, %r9
58      movq %rbp, %rdi
59      movq $-6, %rdx
60      movq (%rdi,%rdx,8), %r8
61      movq %rbp, %rdi     #resetting basepointer
62      cmp $0, %r8
63      jne nonNullDeref12  #not NULL
64      movq $5, %rax
65      jmp errorCleanup6
66  nonNullDeref12:
67      movq $4, %rbx
68      movq %r8, %rcx
69      cmp (%rcx), %rbx
70      jge indeksError11   #indexOutOfBounds
71      cmp $0, %rbx
72      jge indeksAllowed10 #not indexOutOfBounds
73  indeksError11:
74      movq $2, %rax       #IndexOutOfBounds
75      jmp errorCleanup6
76  indeksAllowed10:
77      incq %rbx           #moving past array-size-value
78      imulq $8, %rbx
79      addq %rbx, %r8
80      cmp $0, %r8
81      jne nonNullDeref13  #not NULL
82      movq $5, %rax
83      jmp errorCleanup6
84  nonNullDeref13:
85      movq %r8, %rcx
86      movq %r9, (%rcx)
87      movq $0, %rax
88  mainend:
89      addq $56, %rsp
...     Callee Epilog
97      ret
...
101 errorCleanup6:
102     movq mainSPoint4, %rsp
```

```
103     movq mainBPoint5, %rbp
104     jmp mainend
```

Listing 17: Genereret assemblerprogram for det kittyprogram, der ses i listing 16.

8.5 Afprøvning

Denne fase er testet meget i sammenhæng med kodegenereringsfasen. Når vi har haft noget i kodegenereringen, vi vidste virkede som det skulle, har vi forsøgt at få emit-fasen til at printe det ud og derefter oversætte den assembler-fil, der er blevet skrevet for at se, om koden var korrekt oversat. Vi har manuelt sammenlignet meget af det med det vi regner med der skulle komme ud og vi har kontrolleret output fra det oversatte kode for at se at det opførte sig som forventet.

I tessuiten findes mere end 100 enhedes-tests, der alle går igennem hele oversætteren og alle disse opføre sig som forventet.

9 Fejl og mangler

- **Absolut værdi særheder** - vi kan tage absolut værdi af booleans. Grundet måden vi håndterer forskellen imellem absolut værdi og ELLER operatoren kan der sagtens være sandhedsudtryk inde i absolut værdi paranteser. Det er dog ikke muligt at have en ELLER operator inde i absolut værdi paranteser. Det er meningen at det skal være forbudt at have andet end lister og heltals-udregninger inden i absolut værdi paranteser, dette kan gøres i typecheck fasen

- **Lister med ikke-eksisterende typer** - Når en liste-type defineres indeholdende en navngiven type, der ikke findes, så bliver dette ikke tjekket og afsløret når typen erklæres. Først når en variabel erklæres af den type og efterfølgende bruges, så opdages type-fejlen. På den måde kan programmer altså skrives med ugyldige erklæringer selvom dette ikke er ønsket. Hvis en variabel, der ikke bruges, erklæres af array-typen, så vil fejlen ikke blive opdaget, men kodegeneratoren vil forsøge at samle information om variablens type, hvilket resultere i at kernen smides.

- **Anonyme parameter og retur-typer** - Når en funktion erklæres med en anonym record- eller liste-type som typen af en parameter eller retur-type, så undersøges indeholdet af disse ikke for type-korrekthed, og de kan bl.a. indeholde typer, der ikke findes, hvis disse typer ikke bliver brugt. Dvs. så snart funktionen bliver kaldt eller parametrene bruges så vil fejlen findes i den forbindelse.

10 Konklusion

Det er lykkedes at implementere en oversætter, der kan oversætte et nyt defineret sprog til assembler kode. Implementationen benytter de faser, der er typisk for en oversætter, herunder skanning, parsning, udlugning, typetjek, kodegenering og optimering inden den endeligt oversatte kode skrives ud.

I forbindelse med optimeringen er det lykkedes at implementere både registerallokering, der effektiv udnytter systemets registre, og kighulsoptimering, der optimerer på den i kodegenereringsfasen genererede kode.

Vores oversætter virker i rigtig mange tilfælde, vi har efter aflevering af kilde-koden fundet et par fejl. De udvidelser vi har forsøgt at implementere virker tilfredsstillende.

Dato: ————

Underskrifter:

Andreas Holm Jørgensen, andjo16. ————————

Jeff Gyldenbrand Skov Jørgensen, jegyl16. ————————

Mads Kempf, makem16. ————————

A Kildetekster

I rækkefølge efter faser.

[1] Appel, Andrew W., *Modern compiler implementatione*, Cambridge University Press, 1998.

[2] Larsen, Kim Skak, *Supplementary notes for DM546,* , Februar 2019.

B Brug af TestSuite og make til Oversætter

Vi har lavet vores egen test-suite som et bash-script. Test-suiten ligger i mappen *compiler/test/* og køres ved kommando `./testSuite.sh`. Den seneste version af test-suiten er afleveret sammen med denne rapport og indeholder en stor del af de tests oversætteren er blevet afprøvet med. Dette inkludere nogle tests, der vil vise nogle af de fejl og mangler, der er kommenteret på i rapporten. Test-suiten printer kun ting i terminalen hvis der er tests der ikke outputter det vi forventer.

Skulle det ønskes at oversætte vores oversætter fra ny af kan dette gøres med den make-fil der ligger i *compiler/* mappen (forbindelse med afleveringen af oversætteren). Den eksekverbare fil vil blive lagt i *compiler/build/* og hedder *compiler*. Vi har tilføjet fem flag som kan bruges ved kørsel af vores oversætter.

- **dBODY** - benytter en debugmetode i vores compiler som printer det abstrakte syntakstræ.
- **dASM** - benytter en debugmetode i vores compiler som printer vores interne struktur..
- **noLIVENESS** - fjerner liveness-analyse og registerallokering (må ikke fjernes hvis peephole stadig bliver kørt).
- **noPEEPHOLE** - fjerner peephole optimering.
- **noRUNCHECK** - fjerner runtime checks.

Vores oversætter tager input fra stdin, skriver assembler kode til stdout og alle fejlmeddelelser bliver skrevet til stderr.

C Udvalgte afprøvninger til typetjekning

Et repræsentativt udvalg af test fra det medfølgende testmiljø findes i tabellen her. Hvis der ikke står noget i *resultat*-kolonnen, så er det fordi at testen kører igennem oversætteren som forventet.

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
DeclVarNon-Exist.kit	Erklærer en variabel af ikke-eksiterende type.	Ikke tilladt	
DeclTypeOut-OfOrder.kit	Erklærer en navngivet simpel type baseret på en anden navngivet type, der er erklæret senere i programmet.	Tilladt	
DeclCircType.kit	Erklærer en cirkulær type. Erklærer variabler for flere af kædens led.	Ikke tilladt	
DeclCircArr.kit	Erklæring af cirkulære liste typer. Dvs. at indholdet af liste er en type, der selv er en list, der indeholder den første liste som type.	Tilladt	
DeclCircRec.kit	Erklæring af cirkulære liste typer. Dvs. en record indeholder en variabel af en type der en en record, som selv indeholder en variabel af type, der er den første record.	Tilladt	
DeclRedef.kit	Generklæring af typer i samme scope.	Ikke tilladt	
DeclRedef-Scopes.kit	Gendefinering typer og variabler i forskellige scopes.	Tilladt	
DeclVarType-FromVar.kit	Erklæring af en variabel af en type der er en variabel.	Ikke tilladt	
DeclFieldType-NameAsOther-Field.kit	Erklær et record felt med en gyldig type, der dog deler navn med et andet felt i recorden. Dette medfører at typen bliver forstået som det felt der i recorden, som ikke er en type, men en variabel.	Ikke tilladt	
DeclVarType-NameAsVar.kit	Erklær en variabel af en type som findes i et ydre scope som deler navn med en variabel i det lokale scope. Det er ikke tilladt da typen for variablen bliver evalueret til en variabel.	Ikke tilladt	

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
DeclTypeType-NameAsVar.kit	Erklær en type som har en type fra et andet scope, der deler navn med en variabel i det lokale scope. Typen bliver her forstået som en variabel, der ikke kan bruges som type.	Ikke tilladt	
DeclTypeType-NameAsFunc.kit	Erklær en type som har en type fra et andet scope, der deler navn med en funktion i det lokale scope. Typen bliver forstået, som den lokale funktion, der ikke kan bruges som type.	Ikke tilladt	
AccessFuncNot-Var.kit	Tilgå en funktion i et andet scope, der deler navn med en variabel i det lokale scope. Dette bliver opfattet som hvis man kalder variablen, hvilket ikke er tilladt.	Ikke tilladt	
AccessVarNot-Type.kit	Tilgå en variabel i et andet scope, der deler navn med en type i det lokale scope. Dette er ikke tilladt, da oversætteren tror det er typen der tilgås.	Ikke tilladt	
AllocFunc.kit	Alloker funktion.	Ikke tilladt	
AllocType.kit	Alloker type.	Ikke tilladt	
AllocInt.kit	Alloker integer variabel.	Ikke tilladt	
AllocRecAs-Arr.kit	Alloker anonym record som en liste, med en længde.	Ikke tilladt	
AllocArrAs-Rec.kit	Alloker navngivet liste som record.	Tilladt	
AllocAnonRec-InArr.kit	Alloker anonym record i liste.	Tilladt	
AllocNamed-ArrInRec.kit	Alloker navngivet liste i record.	Tilladt	

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
AllocNestArr.kit	Erklær og alloker indlejrede lister i tre niveauer.	Tilladt	
AllocNestRec.kit	Erklær og alloker indlejrede records i tre niveauer.	Tilladt	
AssiArrRec-Elem.kit	Tildeling mellem elementer i lister (record typer).	Tilladt	
AssiArrElem-SimpleWrong.kit	Tildeling forkert mellem elementer i lister (simple typer)	Tilladt	
AssiArrs.kit	Tildeling mellem lister af simple typer. Listerne er erklæret en smule forskelligt, så denne tests tester ligeledes typeækvivalens mellem lister.	Tilladt	
AssiArrsRec.kit	Tildeling mellem lister med navngivne og kompatible recordtyper.	Tilladt	
AssiNull-Simple.kit	Tildeling af `null` til simpel type.	Ikke tilladt	
AssiNullAnon-Rec.kit	Tildeling af `null` til anonym record variabel.	Tilladt	
AssiNullNamed-Arr.kit	Tildeler værdien null til en navngivet liste variabel.	Tilladt	
AccessNonExists-Field.kit	Tilgang til record element, der ikke findes	Ikke tilladt	
AccessVarAs-Field.kit	Tilgå element uden for record som var det inde i recorden.	Ikke tilladt	
AccessField-AsNestField.kit	Tilgå record element som var det i en indlejret record	Ikke tilladt	
ArrTypeEquiv.kit	Tjek korrekt brug af typeækvivalens mellem indlejrede lister indeholdene integers.	Tilladt	

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
ArrTypeEquiv-Wrong1.kit	Tjek forkert brug af type ækvivalente lister af records.	Ikke tilladt	
ArrTypeEquiv-Wrong2.kit	Tjek forkert brug af type ækvivalente lister af integers.	Ikke tilladt	
ArrTypeEquiv-Wrong3.kit	Tjek forkert brug af type ækvivalente lister (simple slut typer).	Ikke tilladt	
PlusConst.kit	Addition af to integer konstanter.	Tilladt	
DivVarSimple.kit	Division af to integer variabler.	Tilladt	
MinusBool.kit	Subtraktion af integer variabel fra boolsk variabel	Ikke tilladt	
lessBool.kit	Tjek af sammenligning mellem integer variabel boolsk variabel.	Ikke tilladt	
IfTrue.kit	`if`-handling med betingelse der er konstanten `true`.	Tilladt	
IfBoolExp.kit	`if`-betingelse er større boolsk udtryk.	Tilladt	
IfIntVar.kit	Integer variabel som betingelse i `if`-handling. Betingelsen skal være boolsk.	Ikke tilladt	
WhileBreak-Out.kit	Tjek af `break`-betingelse erklæret udenfor while-betingelse.	Ikke tilladt	
WhileCont-Nest.kit	Tjek af `continue`-handling erklæret i indlejret while-løkke.	Tilladt	
FuncSimple-Types.kit	Funktioner der returnerer simple typer.	Tilladt	

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
FuncAnonRec1.kit	Funktion der tager anonyme records. Det kan ikke lade sig gøre at kalde den funktion, da der ikke findes nogen variabel, der kan have samme type som den anonym record parameter. Det samme gælder ved retur-typer.	Ikke tilladt	
FuncNamed-Rec.kit	Funktion der tager og returnerer navngivne records.	Tilladt	
FuncNamed-Arr.kit	Funktion der tager og returnere arrays.	Tilladt	
FuncAnonArr.kit	Funktion der tager og returnerer lister.	Tilladt	
FuncCallNon-LegalParam1.kit	Test af funktionskald med for mange parametre.	Ikke tilladt	
FuncCallNon-LegalReturn5.kit	Test hvor retur-type ikke passer med udtryk i retur-handling.	Ikke tilladt	
FuncCallAs-Param.kit	Funktionskald i parameterliste for andet funktionskald.	Tilladt	
FuncCallAs-ParamWrong.kit	Funktionskald i parameterliste for andet funktionskald, hvor returtype af kaldt funktion ikke passer med typen af parameren.	Ikke tilladt	
FuncRedef.kit	Test med gendefinition af funktion i eget scope.	Ikke tilladt	
FuncRedef-Scope.kit	Test hvor funktion redefineres i et andet scope.	Tilladt	
LinkNestVar.kit	Tilgang til variabel i højereliggender scope.	Ikke tilladt	
LinkNestPar.kit	Tilgang til parameter i højereliggende scope.	Tilladt	

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
LinkNestVar-Wrong.kit	Forsøg på adgang til parameter der ikke findes (i pågældende eller højereliggende scopes).	Ikke tilladt	
LinkNestPar-Wrong.kit	Forsøg på adgang til variabel der ikke findes (i pågældende eller højereliggende scopes).	Ikke tilladt	
DeclVar-NonExist.kit	Erklær en variabel af ikke-eksiterende type	Ikke tilladt	
DeclType-NonExist.kit	Erklær en type af ikke-eksiterende type.	Ikke tilladt	
DeclRecType-NonExist.kit	Erklær en record-type med et felt af en ikke-eksiterende type	Ikke tilladt	
CardExp.kit	Udtryk der involdvere Kardinalitet af forskellige typer.	Tilladt	
NegBoolExp.kit	Neger et boolsk udtryk	Tilladt	
NegIntExp.kit	Neger et taludtryk	Ikke tilladt	
NegRecordVar.kit	Neger en record variabel	Ikke tilladt	
DeclArrType-NonExist.kit	Erklær en liste-type af en brugerdefineret type, der ikke findes og en variabel af denne type.	Ikke tilladt	Smider kernen i kodegeneringen
DeclArrType-NonExistUse.kit	Baseeret på DeclArrTypeNonExist, men variablen bliver også brugt her.	Ikke tilladt	Fejlen findes når variablen bruges
DeclFuncParArr-NonExistType.kit	Erklær en funktion der tager som parameter et anonymt array indeholdende en type, der ikke findes	Ikke tilladt	Fejlen findes ikke og oversætteren

Tabel 8: Udvalgte tests for typetjekker

Testfilnavn	Beskrivelse	Forv	Resultat
DeclFunParArr-NonExistType-Use.kit	Baseret på DeclFuncParArrNonExist-Type.kit. Der laves at funktionskald til funktionen.	Ikke tilladt	Fejlen findes i forbindelse med typetjek af funktions-kald
DeclFuncParRec-NonExistType.kit	Erklær en funktion der tager som parameter en anonym record indeholdende en type der ikke findes	Ikke Tilladt	Fejlen opdages ikke af oversætteren.
DeclFuncParRec-NonExistType-Use.kit	Baseret på DeclFuncParRec-NonExistType.kit. Parameteren bruges inde i funktionen.	Ikke tilladt	Fejlen findes når parameteren bruges.
DeclFuncRetArr-NonExistType.kit	Erklær en funktion der returnerer et anonymt array indeholdende en type, der ikke findes.	Ikke tilladt	Fejlen bliver ikke opdaget
DeclFuncRetArr-NonExistType-Use.kit	Baseret på DeclFuncRetArrNonExist-Type.kit. Forsøg at returner et array af typen.	Ikke tilladt	Fejlen findes ved returnhandlingen inde i funktionen.
DeclFuncRetRec-NonExistType.kit	Erklær en funktion der returnerer en anonym record indeholdende en type, der ikke findes.	Ikke tilladt	
DeclFuncRetRec-NonExistType-Use.kit	Baseret på DeclFuncRetRec-NonExistType.kit. Den returnerede record bliver brugt	Ikke tilladt	Fejlen findes ved brug af recorden.
DeclFuncPar-NonExistType.kit	Erklær en funktion der tager som parameter en variabel af en type, der ikke findes.	Ikke tilladt	Fejlen findes
DeclFuncRet-NonExistType.kit	Erklær en funktion der returnerer en variabel af en type der ikke findes.	Ikke tilladt	

D uafkortet kode til emit

D.1 factorial.s

```
1  format:
2  .string "%d\n"
3  .data
4  .align 8
5  heap1:
6  .space 1048576
7  freeHeap2:
8  .space 8
9  endHeap3:
10 .space 8
11 mainSPoint4:
12 .space 8
13 mainBPoint5:
14 .space 8
15 .text
16 .globl main
17 main:
18  push %rbp
19  movq %rsp, %rbp
20  push %rbx
21  push %r12
22  push %r13
23  push %r14
24  push %r15
25  subq $16, %rsp
26  movq %rsp, mainSPoint4
27  movq %rbp, mainBPoint5
28  movq $heap1, freeHeap2
29  movq $heap1, endHeap3
30  addq $1048576, endHeap3
31 #line: 8 write statement
32  movq %rbp, %r9
33  push %rcx
34  push %rdx
35  push %rsi
36  push %rdi
37  push %r8
38  push %r9
39  push %r10
40  push %r11
41  subq $8, %rsp
42  movq %rsp, %r8
43  movq $5, %rbx
44  movq %r8, %rcx
45  movq %rbx, (%rcx)
46  movq %r9, %rbx
47  push %rbx
48  call factorial7
```

```
49   addq $16, %rsp#remove static link and parameters
50   pop %r11
51   pop %r10
52   pop %r9
53   pop %r8
54   pop %rdi
55   pop %rsi
56   pop %rdx
57   pop %rcx
58   movq %rax, %r8
59   push %rdi
60   movq %r8, %rsi
61   movq $format, %rdi
62   movq $0, %rax
63   call printf
64   pop %rdi
65   movq $0, %rax
66  mainend:
67   addq $16, %rsp
68   pop %r15
69   pop %r14
70   pop %r13
71   pop %r12
72   pop %rbx
73   movq %rbp, %rsp
74   pop %rbp
75   ret
76
77
78
79  factorial7:
80   push %rbp
81   movq %rsp, %rbp
82   push %rbx
83   push %r12
84   push %r13
85   push %r14
86   push %r15
87   subq $104, %rsp
88  #line: 5 if then else statement
89   movq $1, %r9
90   movq %rbp, %rdi
91   movq $3, %rdx
92   movq (%rdi,%rdx,8), %r8
93   movq %rbp, %rdi  #resetting basepointer
94   movq $0, %rbx
95   cmp %rbx, %r8
96   je cmp12
97   movq $0, %r9
98  cmp12:
99   movq %r9, %rbx
100  cmp $1, %rbx
101  movq %rbx, %r8
102  je lazy11
103  movq $1, %r10
104  movq %rbp, %rdi
105  movq $3, %rdx
```

```
106   movq (%rdi,%rdx,8), %r9
107   movq %rbp, %rdi  #resetting basepointer
108   movq $1, %rbx
109   cmp %rbx, %r9
110   je cmp13
111   movq $0, %r10
112  cmp13:
113   movq %r10, %rbx
114   or %r8, %rbx
115  lazy11:
116   cmp $1, %rbx
117   jne else10
118  #line: 3 return statement
119   movq $1, %rax
120   jmp factorial7end
121   jmp endif10
122  else10:
123  #line: 5 return statement
124   movq %rbp, %rdi
125   movq $3, %rdx
126   movq (%rdi,%rdx,8), %r10
127   movq %rbp, %rdi  #resetting basepointer
128   movq %rbp, %rbx
129   addq $16, %rbx
130   movq (%rbx), %r11
131   push %rcx
132   push %rdx
133   push %rsi
134   push %rdi
135   push %r8
136   push %r9
137   push %r10
138   push %r11
139   subq $8, %rsp
140   movq %rbp, %rdi
141   movq $3, %rdx
142   movq (%rdi,%rdx,8), %r8
143   movq %rbp, %rdi  #resetting basepointer
144   decq %r8
145   movq %rsp, %r9
146   movq %r9, %rcx
147   movq %r8, (%rcx)
148   movq %r11, %rbx
149   push %rbx
150   call factorial7
151   addq $16, %rsp#remove static link and parameters
152   pop %r11
153   pop %r10
154   pop %r9
155   pop %r8
156   pop %rdi
157   pop %rsi
158   pop %rdx
159   pop %rcx
160   movq %rax, %rbx
161   imulq %r10, %rbx
162   movq %rbx, %rax
```

```
163   jmp factorial7end
164  endif10:
165  factorial7end:
166   addq $104, %rsp
167   pop %r15
168   pop %r14
169   pop %r13
170   pop %r12
171   pop %rbx
172   movq %rbp, %rsp
173   pop %rbp
174   ret
175
176
177
178  errorCleanup6:
179   movq mainSPoint4, %rsp
180   movq mainBPoint5, %rbp
181   jmp mainend
```

D.2 allocArr.s

```
1   format:
2   .string "%d\n"
3   .data
4   .align 8
5   heap1:
6   .space 1048576
7   freeHeap2:
8   .space 8
9   endHeap3:
10  .space 8
11  mainSPoint4:
12  .space 8
13  mainBPoint5:
14  .space 8
15  .text
16  .globl main
17  main:
18   push %rbp
19   movq %rsp, %rbp
20   push %rbx
21   push %r12
22   push %r13
23   push %r14
24   push %r15
25   subq $56, %rsp
26   movq %rsp, mainSPoint4
27   movq %rbp, mainBPoint5
28   movq $heap1, freeHeap2
29   movq $heap1, endHeap3
30   addq $1048576, endHeap3
31  #line: 2 allocate of length statement
32   movq %rbp, %rdi
33   movq freeHeap2, %r8
34   movq $-6, %rdx
35   movq %r8, (%rdi,%rdx,8)
```

```
36   movq %rbp, %rdi  #resetting basepointer
37   movq $8, %r9
38   movq %r9, %rbx
39   cmp $0, %rbx
40   jg allocPos8
41   movq $4, %rax#negative allocation size
42   jmp errorCleanup6
43  allocPos8:
44   movq %r9, %rbx
45   incq %rbx#making room for arraySize
46   imulq $8, %rbx
47   addq %rbx, freeHeap2
48   movq freeHeap2, %rbx
49   cmp endHeap3, %rbx
50   jl allocSucc9
51   movq $6, %rax#outofMemory
52   jmp errorCleanup6
53  allocSucc9:
54   movq %r8, %rcx
55   movq %r9, (%rcx)
56  #line: 3 assign statement
57   movq $11, %r9
58   movq %rbp, %rdi
59   movq $-6, %rdx
60   movq (%rdi,%rdx,8), %r8
61   movq %rbp, %rdi  #resetting basepointer
62   cmp $0, %r8
63   jne nonNullDeref12#not NULL
64   movq $5, %rax
65   jmp errorCleanup6
66  nonNullDeref12:
67   movq $4, %rbx
68   movq %r8, %rcx
69   cmp (%rcx), %rbx
70   jge indeksError11#indexOutOfBounds
71   cmp $0, %rbx
72   jge indeksAllowed10#not indexOutOfBounds
73  indeksError11:
74   movq $2, %rax#IndexOutOfBounds
75   jmp errorCleanup6
76  indeksAllowed10:
77   incq %rbx#moving past array-size-value
78   imulq $8, %rbx
79   addq %rbx, %r8
80   cmp $0, %r8
81   jne nonNullDeref13#not NULL
82   movq $5, %rax
83   jmp errorCleanup6
84  nonNullDeref13:
85   movq %r8, %rcx
86   movq %r9, (%rcx)
87   movq $0, %rax
88  mainend:
89   addq $56, %rsp
90   pop %r15
91   pop %r14
92   pop %r13
```

```
93    pop %r12
94    pop %rbx
95    movq %rbp, %rsp
96    pop %rbp
97    ret
98
99
100
101   errorCleanup6:
102    movq mainSPoint4, %rsp
103    movq mainBPoint5, %rbp
104    jmp mainend
```

D.3 arrRecNoCheck.s

```
1   format:
2   .string "%d\n"
3   .data
4   .align 8
5   heap1:
6   .space 1048576
7   freeHeap2:
8   .space 8
9   endHeap3:
10  .space 8
11  mainSPoint4:
12  .space 8
13  mainBPoint5:
14  .space 8
15  .text
16  .globl main
17  main:
18   push %rbp
19   movq %rsp, %rbp
20   push %rbx
21   push %r12
22   push %r13
23   push %r14
24   push %r15
25   subq $104, %rsp
26   movq %rsp, mainSPoint4
27   movq %rbp, mainBPoint5
28   movq $heap1, freeHeap2
29   movq $heap1, endHeap3
30   addq $1048576, endHeap3
31  #line: 3 allocate of length statement
32   movq %rbp, %rdi
33   movq freeHeap2, %r8
34   movq $-6, %rdx
35   movq %r8, (%rdi,%rdx,8)
36   movq %rbp, %rdi  #resetting basepointer
37   movq $30, %r9
38   movq %r9, %rbx
39   incq %rbx#making room for arraySize
40   imulq $8, %rbx
41   addq %rbx, freeHeap2
42   movq %r8, %rcx
```

```
43   movq %r9, (%rcx)
44  #line: 4 allocate statement
45   movq %rbp, %rdi
46   movq $-6, %rdx
47   movq (%rdi,%rdx,8), %r8
48   movq %rbp, %rdi  #resetting basepointer
49   movq $3, %rbx
50   incq %rbx#moving past array-size-value
51   imulq $8, %rbx
52   addq %rbx, %r8
53   movq freeHeap2, %rbx
54   movq %r8, %rcx
55   movq %rbx, (%rcx)
56   addq $16, freeHeap2
57  #line: 5 assign statement
58   movq $7, %r9
59   movq %rbp, %rdi
60   movq $-6, %rdx
61   movq (%rdi,%rdx,8), %r8
62   movq %rbp, %rdi  #resetting basepointer
63   movq $3, %rbx
64   incq %rbx#moving past array-size-value
65   imulq $8, %rbx
66   addq %rbx, %r8
67   movq %r8, %rcx
68   movq (%rcx), %r8
69   movq %r8, %rcx
70   movq %r9, (%rcx)
71  #line: 6 write statement
72   movq %rbp, %rdi
73   movq $-6, %rdx
74   movq (%rdi,%rdx,8), %r8
75   movq %rbp, %rdi  #resetting basepointer
76   movq $3, %rbx
77   incq %rbx#moving past array-size-value
78   imulq $8, %rbx
79   addq %rbx, %r8
80   movq %r8, %rcx
81   movq (%rcx), %r8
82   push %rdi
83   movq %r8, %rcx
84   movq (%rcx), %rsi
85   movq $format, %rdi
86   movq $0, %rax
87   call printf
88   pop %rdi
89   movq $0, %rax
90  mainend:
91   addq $104, %rsp
92   pop %r15
93   pop %r14
94   pop %r13
95   pop %r12
96   pop %rbx
97   movq %rbp, %rsp
98   pop %rbp
99   ret
```

```
100
101
102
103  errorCleanup6:
104    movq  mainSPoint4, %rsp
105    movq  mainBPoint5, %rbp
106    jmp  mainend
```