# Oversættere E09, K-opgave: Udvidelser af Janus-oversætter

Anders Bjerg Pedersen, 070183-2477
Eksamensnummer: 46, Tillægsopgave: 1. Stak

29. januar 2010

## Indhold

# Indledning

Vi skal i denne rapport gennemgå implementation og test af K-opgaven stillet i kurset Oversættere i vinteren 2009-10. Opgaven drejer sig om udvidelse af en eksisterende oversætter for sproget Janus. Specifikt skal vi udvide oversætteren, så det er tilladt at bruge samme variable til input og output, samt implementere en stak, som brugeren kan anvende til midlertidigt at gemme værdier af variable under programkørselen. Der er stillet forskellige krav til de nye kombinerede input-/outputvariable, ligesom stakken skal understøtte forskellige standardoperationer og forespørgsler foruden at leve op til bestemte krav, der skal sikre reversibiliteten af operationerne.

Kode og rapport tager udgangspunkt i den udleverede vejledende løsning til G-opgaven i samme kursus.

## 0.1 Generelt indledende om tests

I både den fælles og individuelle opgave er tests udført ved at køre samtlige (relevante) udleverede testprogrammer, både fra G-opgaven og K-opgaven. For at lette testningen er de forskellige testprogrammer samlet i i tre shell-scripts: test af programmer fra G-opgaven (`testGcases.sh`), test af programmer til den fælles opgave (`testIO.sh`) samt test af programmer til stak-opgaven (`testStack.sh`). I hvert script oversættes først det relevante testprogram, og derefter køres det (med eventuelt input, og output sammenlignes med de udleverede .out-filer). Output af programmer, som *skal* fejle, er selve fejlen, output af alt andet er tomt, hvis programmet opfører sig som forventet. For hvert testprogram udskrives først programmets navn og forventet output i parentes. Disse tre filer er vedlagt sammen med resten af kildeteksten og testprogrammerne. Der skal køres `source compile`, inden scriptsene afvikles. Bemærk desuden, at procedurerne `push` og `pop` i testfilen `stack.jan` er omdøbt til hhv. `pushf` og `popf`, da de ellers (ganske korrekt) vil give parsefejl under oversættelse.

Alle tests er afviklet på DIKUs systemer, samt på MacOS X 10.6.

# 1 Fælles opgave: Input-/output-variable

Vi skal i denne opgave implementere en udvidelse til Janus, som muliggør, at inputvariable også kan forekomme som outputvariable. Disse skal i så fald ikke nulstilles fra start og heller ikke kontrolleres for nul til slut. Her er der ingenting at lave i hverken lexer eller parser, derfor springes disse to afsnit over.

## 1.1 Typecheckeren

Typecheckeren er det væsentligste element i denne opgave. Her skal det tjekkes, at eventuelle variable, der både er input- og outputvariable, er af samme type og eventuelt dimension. De lokale variable skal stadig være entydigt deklareret, uden overlap med input- eller outputvariable. Fra parseren får typecheckeren tre `Def`-lister med variable: `ins`, `outs` og `locals`. I den udleverede implementation tager funktionen `checkDefs` sig af alle tre lister. Først tjekkes inputvariablene og lægges i en tom symboltabel. Denne symboltabel sendes med videre til næste kald af `checkDefs`, der nu kører på outputvariablene, og til sidst gentages processen for de lokale variable.

Som udgangspunkt behøver vi ikke i typecheckeren at tage os anderledes af input- eller lokale variable. Her gælder de samme regler som før. Med hensyn til outputvariablene tilføjer vi nu funktionen `checkOutDefs`, der opfører sig meget lig den oprindelige `checkDefs`. Hvis en outputvariabel ikke findes i symboltabellen (`vtable`), opretter vi den som normalt. Hvis den allerede findes, tjekker vi, om den allerede er erklæret som inputvariabel. Hvis dette er tilfældet, tjekkes overensstemmelse mellem type og eventuelt størrelse. Hvis den ikke allerede er erklæret som inputvariabel, er der tale om en dobbeltdeklaration, og en fejl meddeles. Vi henviser til `Type.sml` for koden for `checkOutDefs`.

For at kunne tjekke, om en variabel er erklæret som inputvariabel, har vi brug for et alternativ til `lookup`-funktionen, som kan slå op i listerne `ins`, `outs` og `locals`, samt returnere variablens type og størrelse (ved tabeller). Symboltabellen og den udleverede `lookup`-funktion kan ikke skelne de forskellige variabeltyper (input-, output- eller lokal). Den nye funktion er gengivet nedenfor. Vi får senere hen brug for en version af denne funktion igen i kodegeneratoren (se afsnit 1.2).

```
1 fun lookupDef x []
2         = NONE
3     | lookupDef x (Janus.IntVarDef (y,pos)::table)
4         = if x=y then SOME Integer else lookupDef x table
5     | lookupDef x (Janus.ArrayVarDef (y,size,pos)::table)
6         = if x=y then SOME (Array size) else lookupDef x table
```

Man kunne i løsningen af denne opgave have valgt at oprette en særskilt liste indeholdende de variable, der var deklareret som både input- og outputvariable. Dette ville lette udvidelsen af kodegeneratoren, som ville kunne skrives mere kompakt, men ville også øge pladskravet en smule og sandsynligvis ikke gøre meget for køretiden. Den implementerede løsning øger ikke pladskravet samt gør brug af eksisterende datastrukturer, og de forholdsvist få `lookup`-kald vil ikke give anledning til en mærkbar forøgelse af køretiden. Desuden er den case-baserede implementation nem at udvide til flere datatyper senere.

### 1.1.1 Test af typecheckeren

Typecheckeren er testet på samtlige udleverede testcases, samt alle de oprindelige fra G-opgaven. Opførselen er som forventet, og alle tests bliver udført med korrekt resultat og korrekt type og positionering af fejl.[1] Se også generelt om tests i afsnit 0.1.

De udleverede testcases afspejler meget godt de faldgruber, man som Janus-programmør kan falde i ved brug af kombinerede input-/outputvariable. Der testes for navneoverlap mellem lokale og blandede variable (`io-error04`), for matchende navn og størrelse på input-/outputvariable (`io-error01-03`), samt for anvendelser af input-/output-variable som er hhv. heltal og tabeller (`io-identity` og `io-reverse`). Da disse nye input-/outputvariable ikke er underlagt de samme krav som almindelige variable, og at det derfor i forhold til typecheckeren er ret ligegyldigt, hvad der sker med dem i selve indholdet af programmet, er der ikke belæg for yderligere test end de udleverede. De almindelige typecheck af normale egenskaber for variable er der taget hånd om i den udleverede typechecker, og disse fungerer stadig på de oprindelige testcases.

[1] Bemærk, at `error01.jan` nu som forventet går igennem uden fejl.

### 1.2 Kodegeneratoren

I kodegeneratoren findes de mekanismer, der opfylder kravene om nulstilling og check af variabelværdier ved opstart og afslutning af et program. Det skal sikres, at almindelige inputvariable er nul ved afslutning, almindelige outputvariable samt lokale variable er nul ved opstart, og at kombinerede input- og outputvariable friholdes for disse regler. Dette kræver modifikation af funktionerne `makeZero` og `checkZero`.

Som før nævnt får vi her igen brug for `lookupDef`-funktionen, som derfor er kopieret over.[2] `makeZero` og `checkZero` ændres nu, så de først tjekker, om den variabel, der arbejdes med, er en kombineret input- og outputvariabel. Hvis dette er tilfældet, springes den over, og der fortsættes til den næste. For at kunne gøre dette, tilføjer vi listen `ins` som ekstra parameter til funktionen `makeZero` og `outs` som ekstra parameter til `checkZero`. Et eksempel kan ses i `checkZero` for heltalsvariable herunder. Koden for tabeller er udvidet på præcis samme måde.

```
1 | checkZero (Janus.IntVarDef (x,_)::defs) outs = (* MODIFIED *)
2   (case lookupDef x outs of
3      NONE   =>  (* check normal in or local variable *)
4        [Mips.BNE (x,"0","_NonZero_")] @ checkZero defs outs
5    | SOME _ =>  (* skip if also an out variable *)
6        checkZero defs outs
7   )
```

Tilføjelserne i `makeZero` er af samme karakter og kan ses i `Compiler.sml`. Udover ovennævnte ændringer er der ikke lavet yderligere i kodegeneratoren.

### 1.3 Test af kodegeneratoren

Kodegeneratoren er, ligesom typecheckeren, testet på samtlige udleverede IO-testcases, samt de oprindelige fra G-opgaven. Alle tests forløber som forventet og med forventet resultat. Igen er de relevante testcases kørt fra de tre shell-scripts (se afsnit 0.1).

Der er ikke umiddelbart behov for yderligere tests af kodegeneratoren for den fælles opgave, da ændringerne her er meget overskuelige og kun har noget med check af variable før og efter kode at gøre. De to udleverede testprogrammer `io-identity` og `io-reverse` går igennem uden problemer og med korrekt output. Derfor kan vi i disse tilfælde antage, at kodegeneratoren ikke nulstiller de pågældende variable, ligesom den heller ikke fejler, fordi de er forskellige fra nul ved programmets afslutning.

## 2 Opgave 1: Stak

Vi skal i denne opgave implementere en stak i Janus, som brugeren kan anvende til midlertidigt at gemme værdier af variable (både heltalsvariable og værdier i tabeller). Stakken skal understøtte standardoperationerne `push x` (lægger værdien af `x` øverst på stakken), `pop x` (fjerner øverste element fra stakken og gemmer værdien i `x`), `swap` (bytter om på de to øverste elementer på stakken) samt `empty` (om stakken er tom eller ej). Der skal undervejs udføres en del tjek af forskellige egenskaber, og passende fejlmeddelelser skal gives til brugeren. Stakken til være stor nok til, at samtlige testprogrammer kan

[2] Bemærk, at der i udgaven anvendt i kodegeneratoren ikke returneres specifikke datatyper, da vi kan antage, at typecheckeren har fanget eventuelle uoverensstemmelser allerede. Ellers ville vi også have været nødt til at deklarere de to Janus-typer `Integer` og `Array of int` i kodegeneratoren.

afvikles uden problemer. Vi gemmer de overordnede overvejelser om lagring af stakken til afsnit 2.4.

## 2.1 Lexeren

Det eneste, der umiddelbart skal laves i lexeren, er at gøre den bekendt med de nye nøgleord `pop`, `push`, `swap` og `empty`. Disse tilføjes til den eksisterende liste af nøgleord:

```
fun keyword (s, pos) =
    case s of
        ...
        | "push"   => Parser.PUSH pos  (* ADDED for stack *)
        | "pop"    => Parser.POP pos   (* ADDED for stack *)
        | "swap"   => Parser.SWAP pos  (* ADDED for stack *)
        | "empty"  => Parser.EMPTY pos  (* ADDED for stack *)
```

Da lexeren i sig selv ikke er særlig nem at teste uden først at have implementeret parseren, udskyder vi dette til afsnit 2.2.1.

## 2.2 Parseren

I `Parser.grm` skal tilføjes de manglende fire produktioner: `push`, `pop`, `swap` og `empty`. Disse skal tilføjes både som tokendefinitioner samt som deciderede produktioner, så lexeren kan kalde dem. Der er ikke behov for at ændre grammatikken eller tilføje nye præcedenserklæringer eller associativitetsregler; de nye produktioner er selvstændige og kan for så vidt ikke sættes sammen med andre udtryk eller sætninger (pånær index i tabeller). En kommando ala f.eks. `(pop x) - 3` (pop `x` fra stakken og træk bagefter 3 fra) er ikke defineret i opgaven og vil derfor heller ikke blive tilladt i nærværende implementation. Vi laver følgende tilføjelser i `Parser.grm`:

```
%token <(int*int)> PUSH POP SWAP EMPTY
   ...
Stat :
   ...
    | PUSH Lval   { Janus.Push ($2, $1) }
    | POP Lval    { Janus.Pop ($2, $1) }
    | SWAP        { Janus.Swap $1 }
Cond :
   ...
    | EMPTY       { Janus.Empty $1 }
```

Produktionerne i parseren gør brug af abstrakte datatyper fra filen `Janus.sml`, hvor abstrakte datatyper for de nye produktioner skal deklareres:

```
datatype Cond
   ...
    | Empty of pos   (* ADDED for stack *)

datatype Stat
   ...
    | Push of Lval * pos   (* ADDED for stack *)
    | Pop of Lval * pos    (* ADDED for stack *)
    | Swap of pos   (* ADDED for stack *)
```

### 2.2.1 Test af lexer og parser

Indledningsvist skal det siges, at lexer og parser genereres uden fejl med `source compile`. Dette indikerer blandt andet, at der ingen præcedenskonflikter eller ikke-entydighed er i grammatikken. Lexer og parser er testet på de udleverede filer (se hvordan i afsnit 0.1). På de udleverede testprogrammer, der ikke skal fejle, får vi ingen fejl fra lexer og parser. Da der ikke er udleveret direkte tests af *fejl* i syntaks, gengives herunder nogle forskellige mindre tests af eventuelle fejlsituationer i lexer og parser.

**Tegn i lexeren**

De eneste tilføjelser i lexeren er de fire nye tokens. Korrekt syntaks af disse er testet i de udleverede testprogrammer, og hvis man skulle teste lexeren grundigt, skulle man følge de forskellige tokens rundt i programmet, hvilket er langt mere, end der er plads til i denne rapport. Lexerens primære opgave er at genkende nøgleord, variable og værdier, og dette gør den tilfredsstillende for de tilføjede elementer i de udleverede testprogrammer.

**Parsefejl i nye stak-operationer**

Her bør det testes, at der kun kan komme enten heltalssvariable eller tabelelementer efter kommandoerne `push` og `pop`, at `swap` ingen argumenter må have, samt at `push`, `pop` og `swap` ikke kan indlejres eller kombineres med andre udtryk eller sætninger. Vi illustrerer med følgende små eksempelprogrammer:

(1)
```
a -> b;
push fib
```

(2)
```
a -> b;
swap a
```

(3)
```
a -> b;
push (a-=3);
```

(4)
```
a -> b;
pop push a;
```

Det første scenarie vil rent faktisk ikke blive fanget i parseren, da lexeren opfatter alt andet end nøgleord og tal som variabelnavne. Derfor vil (1) gå igennem parseren og først blive fanget i typecheckeren (se afsnit 2.3.1), da der ikke er deklareret en variabel ved navn `fib`. Med hensyn til `swap`, så giver (2) ganske korrekt en parsefejl, da `a` eller noget som helst andet ikke kan efterfølge `swap`. Det samme er testet for `swap` efterfulgt af tabelelementer og procedurenavne. Til sidst vil (3) give parsefejl på venstreparentesen, da kun variable kan efterfølge `push`. Det samme er testet for `pop`. Indlejring af disse to stakoperationer giver ikke syntaktisk mening, men f.eks. vil (4) parse-fejle på `push`.

Der er ikke testet for yderligere fejl i parser og lexer, da dette ville ende i uvæsentlige peditesser. Selvom lexer og parser ikke er testet på samtlige mulige inddata, er der overvejende sandsynlighed for, at de fungerer som forventet.

## 2.3 Typecheckeren

For typecheckerens vedkommende er der stillet specifikke krav til, at der ved `push`- og `pop`-operationer på tabelelementer ikke må forekomme elementer fra tabellen selv i indexet. Dette er for at sikre invertibiliteten i sproget. Herudover er der ikke meget at tilføje: `empty` og `swap` skal der ikke ske noget med i typecheckeren Derfor tilføjes `| Janus.Empty pos => ()` i `checkCond`, og `| Janus.Swap pos => ()` tilføjes i `checkStat` i `Type.sml`.

Arbejdet ligger i `checkStat` for `Janus.Push` og `Janus.Pop`. Her skal det tjekkes, at variablen er defineret, og hvis der er tale om en tabel, skal det tjekkes, at tabellen ikke selv indgår i udtrykket i dens index. Dette har vi allerede tilgængeligt i `checkExp`, hvor vi sender en avoid-variabel med. Hvis vi skal pushe eller poppe en heltalsvariabel, er der ingen problemer. Hvis der er tale om et tabelelement, tjekker vi med `checkExp e vtable x`, at tabellen ikke selv indgår i indexet `e`. Hvis vi bliver bedt om at pushe fra eller poppe til noget, som ikke er defineret som variabel (som i f.eks. `a -> b; push foo`), bør dette også meddeles.[3] Vi illustrerer tilføjelserne med koden for `push` i `checkStat`:

```
| Janus.Push (lv,pos) =>  (* ADDED *)
    (case lv of
       Janus.IntVar (x,p) =>
        (case lookup x vtable of
            SOME Integer => () (* num requires no typecheck *)
          | _ => raise Error ("Push␣from␣undefined␣variable",p))
     | Janus.ArrayIndex (x,e,p) =>
         (case lookup x vtable of
            SOME (Array _) => checkExp e vtable x (* avoid x in array index e *)
          | _ => raise Error ("Push␣from␣undefined␣variable",p))
    )
```

En lignende implementation er tilføjet for `Janus.Pop` og kan ses i `Type.sml`.

### 2.3.1 Test af typecheckeren

Vores primære fokus for test af typecheckeren er de to invertibilitetskrav, ligesom vi også tjekker for udefinerede variable. Her vil vi igen komme med små eksempelprogrammer, som dog ikke er vedlagt som kode.

**Invertibilitetskrav**

Disse er testet i de udleverede testfiler `stack-error05` og `stack-error06`. De har det forventede resultat.

**Udefinerede variable**

Følgende fire programmer giver allesammen korrekt fejlmeddelelse:

```
a -> b;          a -> b;          a -> b;          a -> b;
push fib         push fib[0]      push a;          push a;
                                  pop fib          pop fib[0]
```

Da typecheckeren ikke laver noget for hverken `empty` eller `swap`, konkluderer dette hermed tests af typecheckeren. Det er af samme årsager som i lexer og parser ikke fundet nødvendigt at gå i peditesser med flere småtests. Alle udleverede testprogrammer samt de fire ovenstående giver det forventede resultat.

## 2.4 Kodegeneratoren

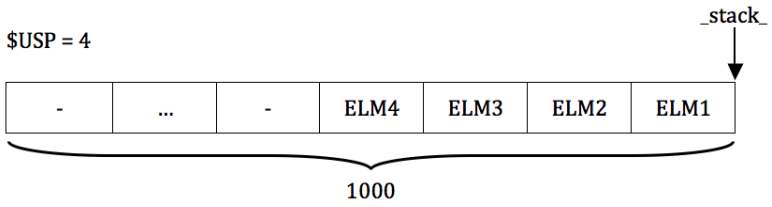
Langt det meste af arbejdet med stak-opgaven ligger i kodegeneratoren. Her skal selve "arbejdet" på stakken udføres, ligesom det er her, der skal afsættes plads og tjekkes, om

[3] Generelt mangler dette specifikke check i den udleverede kode. Dette bør implementeres, da man ellers ikke vil kunne rapportere sigende fejlmeddelelser eller tjekke for stavefejl i variabelnavne. Den udleverede implementation rapporterer i disse tilfælde blot sammenblanding mellem heltals- og tabelvariable, hvilket jo ikke er korrekt.

stakken er tom ved afslutning af programmet. Typecheckeren har taget sig af de fleste tjek mht. `push` og `pop`, men kodegeneratoren skal desuden tage sig af tjekket af, om der er mindst to elementer på stakken, når `swap` kaldes.

I denne implementation er en stak valgt til at være et statisk stykke plads afsat i DATA-området, der er tilpas stort til, at samtlige testprogrammer kan afvikles uden problemer, samt at den mest almindelige brug af en stak muliggøres. Der er til stakken afsat plads til 1000 elementer, men dette kan meget nemt ændres. Koden vil derfor komme til at ligne koden for tabeller til forveksling, men med én væsentlig undtagelse: stakpegeren. Den umiddelbare forskel på en tabel og et array er, at man i tabellen kan lave vilkårlig tilgang til et hvilket som helst element. Dette er ikke tilladt med stakken, hvor der kun tillades adgang til øverste element.

Vi har altså brug for en form for stakpeger, der angiver, hvor i dataområdet vi er "nået til". Denne *kunne* ligge som første element i den afsatte plads i dataområdet, men dette ville kræve unødigt mange opslag i hukommelsen. I stedet lægges stakpegeren i register 28, som er ubrugt at registerallokatoren. Vi kalder for nemheds skyld dette register for "*User Stack Pointer*" (USP). Her ligger ikke en decideret peger, men antallet af elementer på stakken. Hermed kan vi ved hjælp af en enkelt MIPS-kommando se, om stakken er tom (mere herom senere), vi sparer tilgangen til hukommelsen både ved læsning og opdatering af stakpegeren, og vi kan nemt shifte stakpegeren mod venstre med 2 for at få det korrekte offset til det øverste element i dataområdet. Vi bemærker her, at da Janus kun indeholder to slags variable (heltal og arrays), kan der kun forekomme elementer af typen **num** på stakken. Disse er hver af størrelse 4 bytes. Baseadressen for stakken kalder vi `_stack_`, og vi tildeler pladsen med `Mips.SPACE` ligesom for tabeller. I Figur 1 er skitseret opbygningen af en stak med fire elementer. Vi har anvendt følgende kode




Figur 1: Skitse af stak-implementation. ELM4 er nyeste element.

for at oprette stakken i starten af kodegeneratoren (øverst, samt i funktionen `compile`):

```
1 (* ADDED dedicated register for User Stack "Pointer" *)
2   val USP = "28"
3 ...
4 (* ADDED: initialise stack *)
5 @ [Mips.DATA "",      (* move to DATA area *)
6    Mips.LABEL ("_stack_"), (* label of stack *)
7    Mips.SPACE (makeConst (4*1000)), (* allocate space for 1000 elements *)
8    Mips.TEXT "",   (* back to code area *)
9    Mips.ADD (USP,"0","0")]      (* set USP to zero *)
```

Vi vil nu gennemgå implementation af krav til og funktionalitet af stakken.

### Push et element

Når et element `x` pushes på stakken, skal værdien af `x` lægges som nyt øverste

element på stakken i lageret, og USP skal inkrementeres med 1. Herefter skal `x` sættes til værdien nul. Der skal ikke foretages yderligere tjek, udover hvad der allerede er foregået i typecheckeren. Vi kan altså antage, at `push` har fået korrekt og eksisterende input. Følgende kode implementerer `push` for heltal:

```
| Janus.Push (Janus.IntVar (x,p),pos) =>  (* ADDED for stack *)
    [Mips.LA ("2","_stack_"), (* load address of stack *)
     Mips.ADDI (USP,USP,"1"), (* increment USP by one *)
     Mips.SLL ("3",USP,"2"), (* shift USP left by 2 *)
     Mips.ADD ("2","2","3"), (* add stack base address and offset *)
     Mips.MOVE ("3",x), (* store value of x in $3 *)
     Mips.SW ("3","2","0"), (* store value of x on stack *)
     Mips.MOVE (x,"0") ] (* set x to zero *)
```

Implementationen for `push` af tabelelementer ligner til forveksling. Her skal tabellens index dog først beregnes, hvorefter værdien kan hentes fra lageret. Vi henviser derfor til `Compiler.sml` for detaljer.

### Pop et element

Når et element poppes fra stakken og ned i variablen `x`, skal det inden verificeres, at `x` er nul, og at stakken ikke er tom. Herefter gemmes det øverste element fra stakken i `x`, og USP dekrementeres med 1. De to verifikationer laves først, hvorefter selve operationen udføres. Kodens elementer er næsten de samme som for `push`, derfor går vi ikke i detaljer med den her og henviser i stedet igen til `Compiler.sml`.

### Swap de to øverste elementer

Swap verificerer først, at der er mindst to elementer på stakken. Dette gøres igen vha. USP-registeret. Hvis dette er tilfældet, skal de to øverste elementer på stakken byttes om. Dette gøres ved ved at indlæse de to øverste elementer i hver deres register og gemme dem tilbage på stakken i omvendt rækkefølge. Det er her ikke nødvendigt at flytte stakpegeren, da der ikke ændres på antallet af elementer på stakken. Vi ved, at der altid kun er tale om de to øverste elementer på stakken, og derfor kan vi lige så godt styre offsettet uden stakpegeren.

```
| Janus.Swap (line,col) =>  (* ADDED for stack *)
    [Mips.LA ("2","_stack_"), (* load address of stack *)
     Mips.SLTI ("4",USP,"2"), (* set $4 = 1 if USP < 2 *)
     Mips.LI ("5",makeConst line),
     Mips.BNE ("4","0","_SwapError_"), (* error if USP < 2 *)
     Mips.SLL ("4",USP,"2"), (* shift USP left by 2 *)
     Mips.ADD ("5","2","4"), (* add stack base address and offset *)
     Mips.LW ("6","5","0"), (* load value at top of stack *)
     Mips.ADDI ("5","5","-4"), (* move to next stack element *)
     Mips.LW ("7","5","0"), (* load second value from top of stack *)
     Mips.SW ("6","5","0"), (* store first element value on second place *)
     Mips.ADDI ("5","5","4"), (* move to top stack element *)
     Mips.SW ("7","5","0") ] (* store second element value on top of stack *)
```

### Forespørg, om stakken er tom

Denne forespørgsel kan laves ved hjælp af kun to liniers kode i funktionen `checkCond`, da vi altid har antallet af elementer liggende i USP-registeret:

```
| Janus.Empty pos =>  (* ADDED for stack *)
    [Mips.BEQ (USP,"0",tlab), (* branch to tlab if USP = 0 *)
     Mips.J flab ] (* jump to flab if USP != 0 *)
```

#### Reversibilitet af stakoperationer

`push` og `pop` er hinandens inverse, mens `swap` inverterer til sig selv. Den udleverede kode gør det nemt at implementere disse produktioner i funktionen `Reverse`:

```
1 | Reverse (Janus.Push (lv,pos)) =  (* ADDED reverse for push *)
2     Janus.Pop (lv,pos)
3 | Reverse (Janus.Pop (lv,pos)) =  (* ADDED reverse for pop *)
4     Janus.Push (lv,pos)
5 | Reverse (Janus.Swap pos) =  (* ADDED reverse for swap *)
6     Janus.Swap pos
```

#### Tjek, at stakken er tom til sidst

Med brugen af USP gøres dette meget nemt ved at tilføje en enkelt MIPS-kommando i funktionen `compile`. For overskuelighedens skyld er der oprettet en ekstra `val`:

```
1 val checkEmptyStack = [Mips.BNE (USP,"0","_NotEmptyStack_")]
```

Denne `val` tilføjes herefter til den overordnede kodevariabel i linie 57.

#### Fejlmeddelelser

Der er oprettet fejlmeddelelser til følgende situationer: der poppes til et element, som ikke er nul; der poppes fra en tom stak; der swappes med færre end to elementer på stakken; stakken er ikke-tom ved programmets afslutning. Disse fejlmeddelelser er lavet på præcis samme måde som de eksisterende run-time fejlmeddelelser i bunden af funktionen `compile`. Vi vil derfor ikke gå nærmere ind på disse.

### 2.4.1 Test af kodegeneratoren

I det følgende vil vi beskrive, hvordan de forskellige krav og funktionalitet i kodegeneratoren er testet. For en beskrivelse af tests generelt henviser vi til afsnit 0.1. Overordnet har vi selvfølgelig ikke mulighed for at teste alle de uendeligt mange muligheder for inddata, og derfor vil konklusionerne principielt kun være antagelser baseret på testfilerne.

#### Push

Denne funktionalitet er testet i `stack-empty`, `stack-fib`, `stack-reverse`, `stack-swap` og `stack-uncall`. Alle disse testfiler går igennem med forventet korrekt resultat. Der er ikke de store muligheder for fejl i `push` udover dem, som typecheckeren allerede har taget sig af. Indlejring er ikke en mulighed, og dette tager typecheckeren sig desuden af.

#### Pop

`pop` er testet i filerne `stack-empty`, `stack-fib`, `stack-reverse`, `stack-swap`, `stack-uncall` (som korrekt ikke rapporterer fejl), samt `stack-error01` og `stack-error02`, som begge giver de korrekte fejlmeddelelser. Igen finder vi det ikke nødvendigt at teste yderligere, da langt de fleste fejl vil forekomme i typecheckfasen.

#### Swap

Funktionaliteten med at ombytte de to øverste elementer på stakken er testet i filerne `stack-swap` og `stack-uncall` (som korrekt ouputter de forventede resultater uden fejl), samt i `stack-error04`, hvor der gives korrekt fejlmeddelelse til brugeren. Igen, da `swap` ingen argumenter tager, er der ikke umiddelbart behov for

flere tests, da funktionaliteten virker, der kan ikke indlejres, og tjekket for korrekt mindste antal elementer giver den rigtige fejlmeddelelse.

#### Empty

Dette testes i filerne `stack-empty` og `stack-uncall` (som begge giver korrekt output uden fejl). I disse programmer kaldes `empty` både, når stakken er tom, og når den ikke er det. I og med at programmerne kommer frem til de rigtige resultater, at `empty` ikke kan indlejres, samt at koden er så slående simpel, må vi formode, at funktionaliteten opfører sig som forventet i det generelle tilfælde.

#### Reversibilitet af stakoperationer

Disse er testet i filen `stack-uncall`. Her testes dog ikke reversibilitet af `pop`, men en hurtig kørsel af programmet

```
a -> a;
uncall run;
call run
procedure run
pop a
```

giver samme forventede opførsel som `identity.jan`. Det samme er testet, hvor `a` er en tabel.

#### Tjek, at stakken er tom til sidst

Dette testes implicit i ethvert af de udleverede testprogrammer til stakken, som ikke skal fejle. Fejlen fremprovokeres ligeledes korrekt i programmet `stack-error03`.

#### Fejlmeddelelser

De for kodegeneratoren relevante fejlprogrammer er `stack-error01-04`. Disse fire programmer giver de korrekte fejlmeddelelser, mens kørsel af samtlige andre testprogrammer for stakken implicit tester, at fejlene korrekt *ikke* rapporteres.

## Konklusion

Vi har i denne opgave analyseret, implementeret og testet to udvidelser af en oversætter for sproget Janus: mulighed for at bruge inputvariable også som outputvariable samt en stak, som understøtter operationerne `push`, `pop`, `swap` og `empty` og ligeledes overholder visse reversibilitetskrav og andre betingelser. Implementationen er gennemført, så der løbende er taget hensyn til kompakthed og køretidseffektivitet, og alle tests giver de forventede korrekte resultater. Med forbehold for test af samtlige mulige inddata kan vi konkludere, at opgaven er løst for alle de stillede kravs vedkommende.

Med hensyn til en senere udvidelse af oversætteren med eksempelvis flere datatyper eller flere stakoperationer er implementationen så vidt muligt lavet med SMLs `case`-konstruktion, der letter tilføjelse af sådanne. Desuden er der under kørsel og test af programmer ikke observeret nogen forhøjelse af tidsforbrug under oversættelse eller afvikling af programmer.