Kompilering av blokkorienterte språk

INF2100-dokumentasjon for spesielt interesserte

Dag Langmyhr ⟨dag@ifi.uio.no⟩ 27. november 2015

Sammendrag

Blokkorinterte språk som Pascal krever litt ekstra omtanke når man skal generere kode for dem. I INF2100 trenger man ikke å vite så mye om dette siden det er det vist en oppskrift for hva man skal gjøre, men det er alltid noen som gjerne vil vite nøyaktige hva som skjer. Dette skrivet er for dem.

1 Bakgrunn

Når man skal kompilere et blokkorientert språk som Pascal, der man kan deklarere prosedyrer inni prosedyrer inni prosedyrer så dypt man vil, gir dette et par utfordringer under kompileringen:

- Variabler i en blokk må opprettes på stakken når den tilhørende programmet/funksjonen/prosedyren kalles og fjernes når den er ferdig.
- Det må være mulig å aksessere variabler ikke bare i den lokale blokken men også alle variabler i globale blokker som er synlige.

1.1 Kontekstvektor

Én løsning på problemet er å opprette en såkalt **kontekstvektor**, dvs en tabell over hvor alle de synlige globale blokkene befinner seg på stakken. På den måten får man enkelt tilgang til dem alle.

Denne løsninger er valgt i INF2100-prosjektet siden prosessoren vår x86 har to instruksjoner som gjør dette usedvanlig enkelt: enter og leave.

1.2 Et eksempel

Som eksempel skal vi bruke programmet vist i figur 1; det inneholder en funksjon inni en prosedyre inni hovedprogrammet. Den koden som referansekompilatoren genererer, er vist i figur 5 på side 4.

```
program Blokker;
var V1A: Integer;  V1B: Integer;

    procedure P1 (A1A : Integer;  A1B: Integer);
    var V2: Integer;

       function F2 (A2: Integer): Integer;
       var V3 : Integer;
       begin
          V3 := A2+1;  F2 := V3
       end; { F2 }

    begin
       V2 := F2(A1A);
       V1A := V2*A1B
    end; { P1 }

begin
   P1(-3, 7);
   Write('V1A er ', V1A, EoL)
end.
```

Figur 1: En enkelt testprogram

2 Start av hovedprogrammet

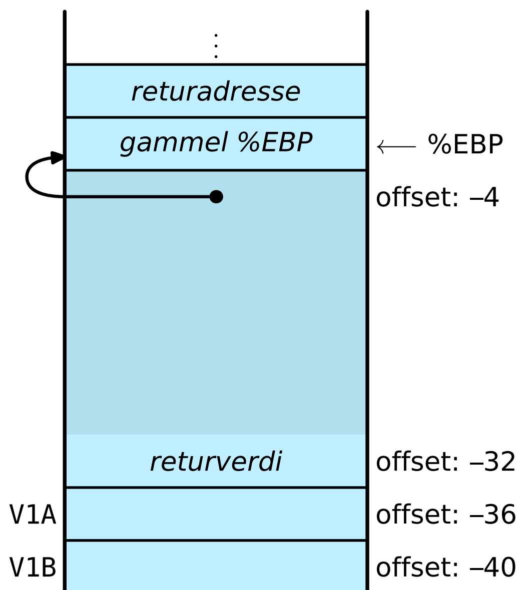
I Pascal er det enklest å behandle hovedprogrammet på samme måte som funksjoner og prosedyrer. Følgende skjer da:

1. Hovedprogrammet kalles og legger returadressen (dvs adressen til instruksjonen etter call-instruksjonen) på stakken.
2. Den første instruksjonen i hovedprogrammet er **enter** som gjør flere ting:
   (a) Innholdet i %EBP-registeret gjemmes unna på stakken.
   (b) Det settes av plass til kontekstvektoren (28 byte),[1] returverdien (4 byte)[2] og 2 lokale variabler ($2\times4 = 8$ byte); tilsammen 40 byte.
   (c) Kontekstvektoren fra blokken utenfor kopieres inn, men siden hovedprogrammet er på blokknivå 1, er det ingen ytre blokk.
   (d) Kontekstvektoren utvides med en peker til denne blokken.

Vi får da situasjonen vist i figur 2 på neste side. Kontekstvektoren er markert med litt mørkere farge.

[1] Siden vi setter av 28 byte til kontekstvektoren, kan vi ikke ha indre blokker dypere enn 7 nivåer, men det er nok for alle praktiske formål. (Vi kunne ha valgt å sette av et antall byte avhengig av blokknivået, men det ville gitt mer komplisert kode, så i INF2100 har jeg valgt å sette av et fast antall.)

[2] Selv om vi bare trenger å lagre en returverdi i funksjoner, setter vi av plassen også i hovedprogrammet og i prosedyrer; det blir enklere kode da.



Figur 2: Stakken når hovedprogrammet starter

3 Start av en prosedyre

Når hovedprogrammet kaller prosedyren P1, skjer akkurat det samme, bortsett fra at parametrene legges på stakken før kallet skjer.

Prosedyren er på blokknivå 2, så kontekstvektor fra blokken utenfor (hovedprogrammet på blokknivå 1) kopieres inn i vår nye kontekstvektor før den utvides med en peker til den lokale blokken (vår egen).

Etter enter-instruksjonen ser stakken ut som vist i figur 3.

4 Start av en indre funksjon

Prosedyren P1 kaller funksjonen F2, og igjen skjer det samme. Figur 4 på neste side viser situasjonen etter at enter-instruksjonen i F2 er ferdig.

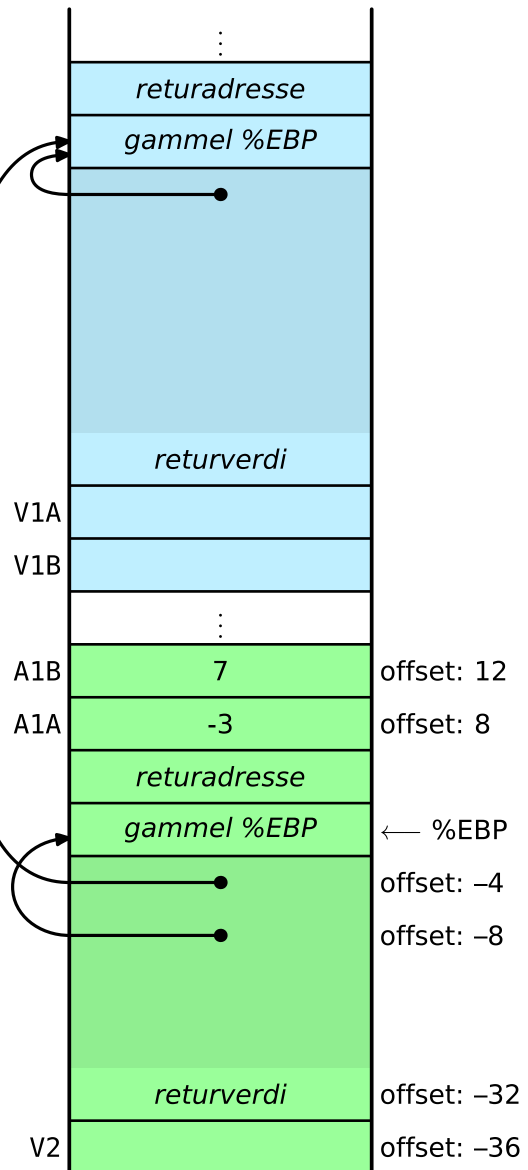
Vi ser nå at vi kan få tak i alle synlige variabler ved å gå via kontekstvektoren. For eksempel får vi tak i den lokale V3 som ligger på blokknivå 3 ved først å gjøre følgende:

1. Slå opp på element nr 3 i kontekstvektoren (og dette har offset 4×3 = 12).
2. Nå har vi adressen til riktig blokk, og der finner vi variabelen V3 med offset –36.

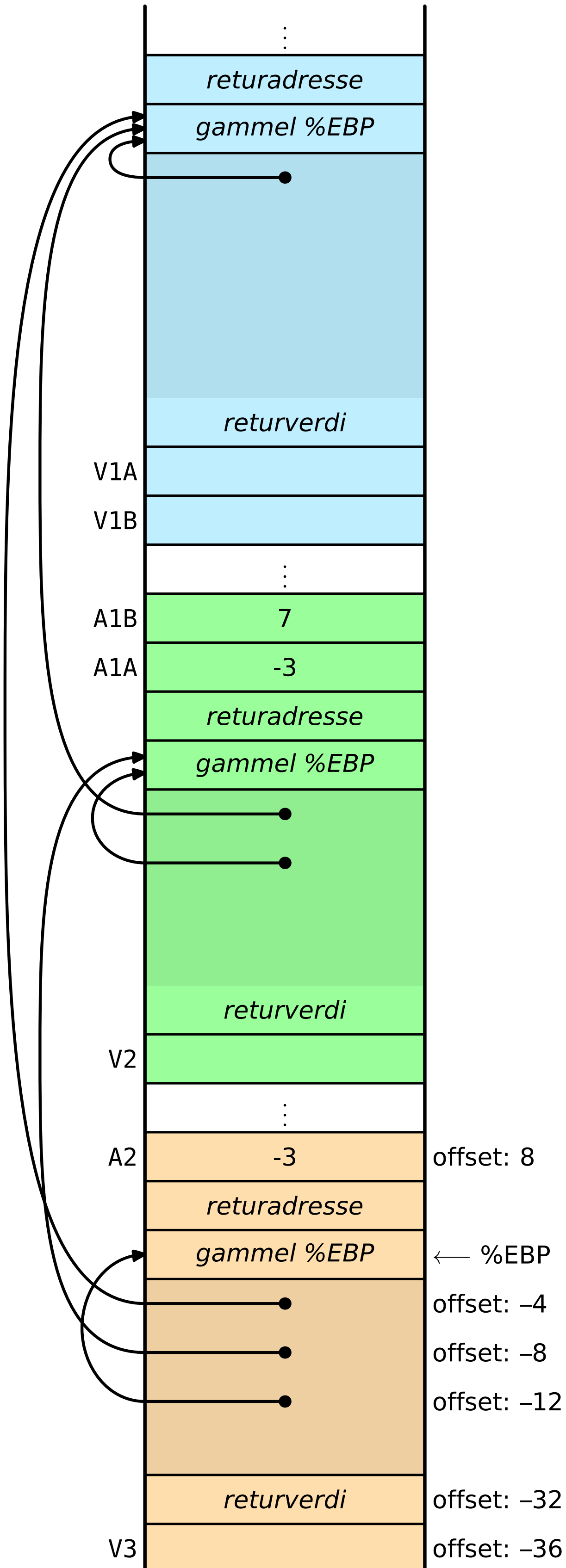
```
        movl    -12(%ebp),%edx
        movl    -36(%edx),%eax          #   v3
```

På samme måte kan vi få tak i den globale variabelen V1B som ligger på blokknivå 1 og har offset –40:

```
        movl    -4(%ebp),%edx
        movl    -40(%edx),%eax          #   v1b
```



Figur 3: Stakken når prosedyren har startet



Figur 4: Stakken når den indre funksjonen har startet

```
# Code file created by Pascal2100 compiler 2015-11-27 15:08:39
        .extern write_char
        .extern write_int
        .extern write_string
        .globl  _main
        .globl  main
_main:
main:   call    prog$blokker_1          # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
func$f2_3:
        enter   $36,$3                  # Start of f2
        movl    -12(%ebp),%edx
        movl    8(%edx),%eax            #   a2
        pushl   %eax
        movl    $1,%eax                 #   1
        movl    %eax,%ecx
        popl    %eax
        addl    %ecx,%eax               #   +
        movl    -12(%ebp),%edx
        movl    %eax,-36(%edx)          # v3 :=
        movl    -12(%ebp),%edx
        movl    -36(%edx),%eax          #   v3
        movl    %eax,-32(%ebp)          # f2 :=
        movl    -32(%ebp),%eax          # Fetch return value
        leave                           # End of f2
        ret
proc$p1_2:
        enter   $36,$2                  # Start of p1
        movl    -8(%ebp),%edx
        movl    8(%edx),%eax            #   a1a
        pushl   %eax                    # Push param #1
        call    func$f2_3
        addl    $4,%esp                 # Pop parameters
        movl    -8(%ebp),%edx
        movl    %eax,-36(%edx)          # v2 :=
        movl    -8(%ebp),%edx
        movl    -36(%edx),%eax          #   v2
        pushl   %eax
        movl    -8(%ebp),%edx
        movl    12(%edx),%eax           #   a1b
        movl    %eax,%ecx
        popl    %eax
        imull   %ecx,%eax               #   *
        movl    -4(%ebp),%edx
        movl    %eax,-36(%edx)          # v1a :=
        leave                           # End of p1
        ret
prog$blokker_1:
        enter   $40,$1                  # Start of blokker
        movl    $7,%eax                 #   7
        pushl   %eax                    # Push param #2.
        movl    $3,%eax                 #   3
        negl    %eax                    #   - (prefix)
        pushl   %eax                    # Push param #1.
        call    proc$p1_2
        addl    $8,%esp                 # Pop parameters.
        .data
.L0004: .asciz   "V1A er "
        .align  2
        .text
        leal    .L0004,%eax             # Addr("V1A er ")
        pushl   %eax                    # Push param #1.
        call    write_string
        addl    $4,%esp                 # Pop parameter.
        movl    -4(%ebp),%edx
        movl    -36(%edx),%eax          #   v1a
        pushl   %eax                    # Push param #2.
        call    write_int
        addl    $4,%esp                 # Pop parameter.
        movl    $10,%eax                #   char 10
        pushl   %eax                    # Push param #3.
        call    write_char
        addl    $4,%esp                 # Pop parameter.
        leave                           # End of blokker
        ret
```

Figur 5: Assemblerkoden generert for programmet i figur 1 på side 1