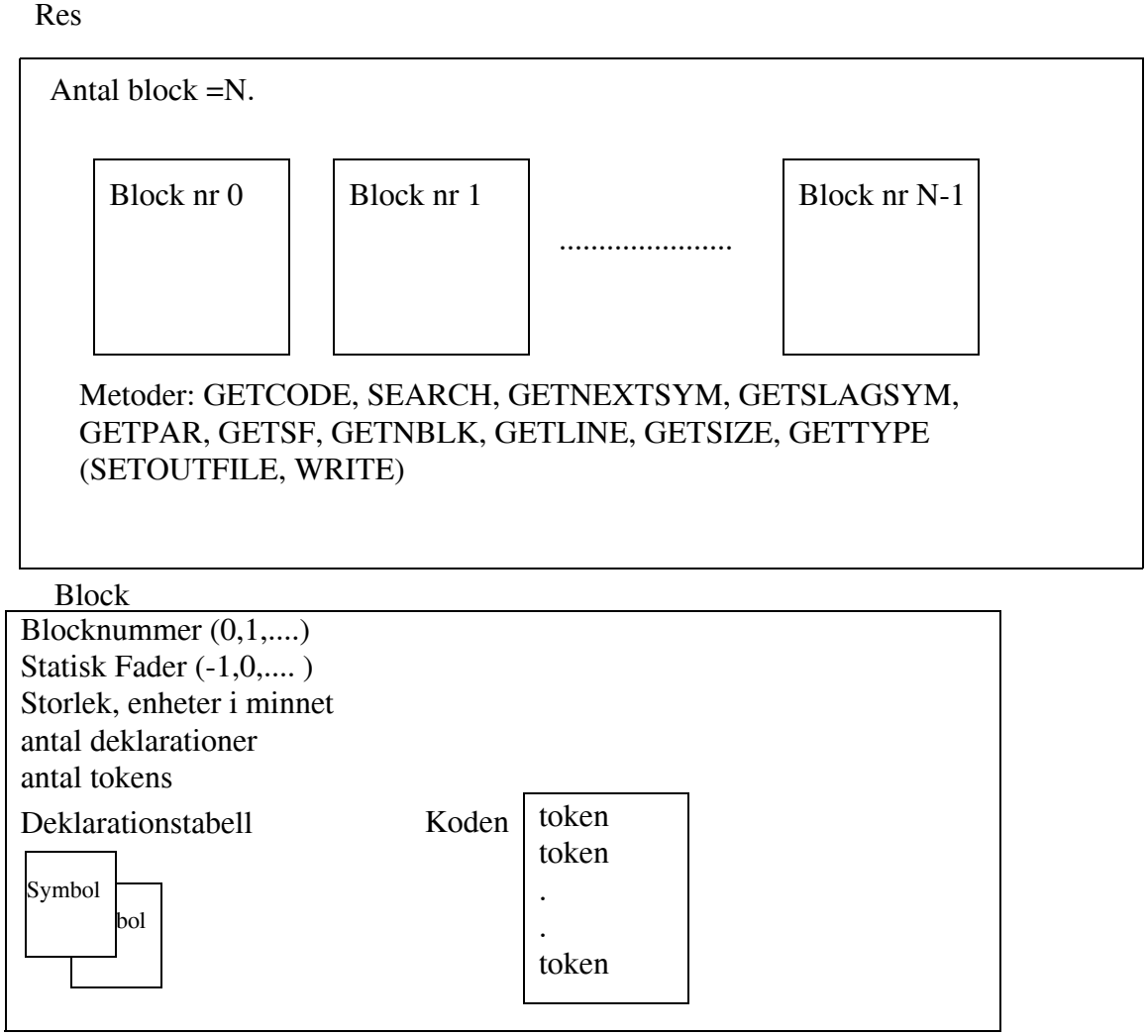
Version 1.0
080416 Göran Fries

Hjälpmedel för interpretatorkonstruktion(dv1211)

En interpretator är ganska lik en kompilator, med en stor skillnad, kompilatorn genererar kod som körs senare medan interpretatorn exekverar koden direkt. Fördelen med kompilerad kod är att den är snabbare när den körs, medan en interpretator flexibelt kan översätta och köra ny kod omedelbart. I vår uppgift ska vi ha ett relativt normalstort språk som vi ska konstruera en interpretator för. Att fullständigt göra detta är en klart för stor uppgift för en sådan kurs, vi gör därför ett antal begränsningar. Detta medför att vi inte kan köra vår interpretator så att man ser fördelarna med interpreterad kod, men vi kan se hur man konstruerar en sådan.
En översättare kan ses göra sitt jobb i ett antal faser. Första fas är en överföring av koden i textformat till en mera hanterbar kod (lexikalisk analys), en andra fas som är särskilt värdefull i ett blockorienterat språk är en block och deklarationsanalys. Efter denna har vi koden i ett format som består av en sekvens av block, där varje block innehåller en symboltabell (deklarationer) och en sekvens av tokens (den exekverbara koden). Nästa fas är byggande av ett syntaxträd och generering av mellankod, ibland uppdelat i två faser. I en kompilering genereras här verklig mellankod, medan en interpretator snarare exekverar denna kod direkt. En interpretator är här färdig, medan en kompilator har en ytterligare fas som från mellankod genererar binärkod för aktuell maskin (back end).
När en kompilator översätter kod så görs detta en gång, även om koden vid körning kommer att exekveras ett antal gånger (tex. en funktion). En interpretator kommer att översätta koden varje gång den körs. För att detta inte ska bli alltför långsamt sparar man kanske någon mellanform så att man inte behöver gå tillbaks till textrepresentationen varje gång. Man kan till och med spara binärkod så att man vid upprepning kör i stort kompilerad kod, skillnaden mellan interpretering och kompilering suddas då delvis ut. Tar vi java som exempel så översätts normalt programmet till bytekod som sedan interpreteras. Man kan också koppla in en sk. just in time (JIT) kompilering som sparar binärkod och vid nästa anrop av en funktion eller kodavsnitt så körs den kompilerade koden.
För att underlätta för er så interpreterar ni inte källkoden i textformat, utan ni interpreterar resultatet efter block och deklarationsanalysen.Tillgängligt finns ett Pass1 till en kompilator eller interpretator. Detta pass skulle normalt infogats i interpretatorn, men då det är skrivet i java (Simula variant finns) så skulle detta ju tvinga er att använda detta språk, därför kan passet köras fristående och generera en textfil som ni startar med att läsa och bygga upp en lämplig objektstruktur från. Ni kan då använda ert favoritspråk.
Programmet p1 översätter ett program i källkod (textfil) till en sekvens av block, där blocket är en sekvens av deklarationer följt av en sekvens av tokens som representerar den “exekverbara” delen av blocket. Resultatet av p1 skrivs ut på en fil enligt nedan beskrivet format. Denna fil är således en normalt läsbar textfil.
Interpretatorn måste naturligtvis implementera ett gränssnitt mot p1, lämpligen konstruerar man ett objekt som innehåller resultatet från p1, detta får sitt innehåll från p1:s resultatfil. Objektet kan lämpligen ha följande struktur och metoder. Förutom detta objekt kommer man också att behöva objekt av typerna Symbol och Token, dessa används både i p1:s resultat och i interpretatorn.
Alla block identifieras av ett blockid, som är ett heltal. Blocken numreras från 0 och uppåt till N-1. Det fiktiva block som kan antas omge resten av blocken har nummer -1, men finns inte representerat någonstans. Symboler identifieras av ett symid som också det är ett heltal. Strukturen framgår av följande figurer.

Objekttypen Res
(lämplig utformning)
Metoder:
GETCODE(blkid)-->Token
Denna metod ger nästa token från blockets koddel. Första gången får man således första token i blocket, nästa gång får man följande och så vidare. Då sista token är läst får man vid följande anrop null som resultat. Resultatet är ett objekt av typen Token. Observera att det finns särskilda tokens som representerar ny rad i källkoden, dessa bör vara osynliga då man läser nästa token, dvs metoden returnerar aldrig ett "radtoken". Då man påträffar ett sådant hanterar man detta "internt" och läser nästa token tills man finner ett "riktigt".
SEARCH(blkid,symid) --> Symbol
Här söker man i angivet block upp angiven symbol, och om den finns får man symbolen som resultat. Detta är ett objekt av typen Symbol. Finns inte symbolen i angivet block blir resultatet null, man får då själv eventuellt söka vidare i omgivande block.
(GETNEXTSYM(blkid) -->Symbol) *Denna metod behövs troligen inte.*
I stället för att söka upp en symbol i ett block kan man läsa alla symboler i blocket på samma sätt om man kan läsa nästa token. Man läser symbolerna en i taget tills där inte finns fler. Fortsätter man att läsa får man null som resultat. Med hjälp av denna metod kan man läsa deklarationerna sekventiellt en gång från blocket, därefter kan man inte läsa fler med denna metod. Vill man läsa flera gånger är man hänvisad till att läsa efter slag, se nedan.
GETSLAGSYM(blkid,slag) --> Symbol *Behövs vid fältdeklarationer.*
Här läser man sekventiellt symbolerna i ett block, men nu bara av ett visst slag. Så länge man läser av samma slag som senast får man nästa, då man byter slag eller läst slut återvänder man till första symbolen. Med hjälp av denna metod kan man således läsa symbolerna flera gånger.



Denna metod kommer man att behöva för att läsa alla fältdeklarationer i ett block, dvs av slag array. Fältdeklarationer bör interpreteras, för att utföra "allokering" i början av blocket.
GETPAR(blkid,nr) -->Symbol
Parametrar till en funktion är deklarerade i funktionens block, men har ingen symid som är känd utanför blocket och man kan således inte söka upp dessa med SEARCH, man kan istället söka upp parameter med ett visst nummer. När man kompilerar ett funktionsanrop måste man överföra parametrar, man befinner sig då normalt i ett annat block än funktionen och känner då inte till parametrarnas namn (dummy).
GETSF(blkid) -->blkid
Används för att få blockid till den statiska fadern till angivet block. Statisk fader till block 0 är block -1, som inte finns med i Res. Denna metod används då man vid sökning av deklaration måste gå uppåt i kjedjan av omgivande block.
GETSIZE(blkid) --> heltal
Ger storleken av ett blocks datadel, dvs hur många enheter minne behövs. Storleken räknas i hur mycket utrymme en variabel (heltal, flyttal, pekare) tar. OBS! funktioner tar här ingen plats!
GETNBLK --> heltal
Ger antalet block.
GETLINE --> heltal
Ge radnummer, dvs det radnummer som fanns i senaste påträffade radtoken (syns lämpligen ej vid vanlig läsning av tokens).
Internt i Res har man lämpligen block enligt figur, här har man de metoder som behövs för att definiera ovanstående metoder i Res.

Objekttypen Symbol

En symbol har följande innehåll:

```
!
!          _______________
!          | Symbol Id   |     The name code (1,2,.....)
!          | Type        |     The type (void,int,real,...)
!          | Kind        |     The kind (simple, array, func, ...)
!          | Info 1      |     Information depending on kind
!          | Info 2      |     Information depending on kind
!          | Info 3      |    Information depending on kind
!          | RelAdr      |    Relative address or zero
!          |_____________|
!                          Info 1,2,3:s meaning depending on kind:
!                                  kind=simple|funcval
!                                        not used
!                                  kind=array
!                                        Info 1: number of indeces
!                                        Info 2: not used
!                                        Info 3: qualification(if ref)/0
!                                  kind=function
!                                        Info 1: number of parameters
!                                        Info 2: defining block index
!                                        Info 3: qualification(if ref)/0
!
!
```

Objekttypen Token

Ett token representerar egentligen ett lexem. Ett token har en tokentyp samt ett värde, nedan kallat CODE, dessutom finns en textsträng. Denna används mest för debugging, men då det gäller tal ligger själva värdet där. Tokentypen id används både för reserverade ord och för användardefinierade enheter, oftast variabel och funktionsnamn. Heltal, flyttal och text är när man i koden direkt anger ett värde (ex. x+2, då blir 2 ett sådant token oftast kallat literal). Typen op anger en operator som +,-, :=, men även ; och andra skiljetecken. Typen fel står för ett av pass1 genererat token då passet påträffat ett fel. Observera att de flesta fel bryr sig inte pass1 om! Typen rad används för att kunna knyta påträffade fel till viss rad i ursprungstexten och är normalt osynligt. Typen call anger att här finns ett block som brutits ut ur sitt ursprungsblock av block och deklarationsanalysen.
GETTYPE--> heltal
Här får man typen dvs något av id, heltal, flyttal,.....call, enligt tabellen nedan..
GETID--> heltal
om typen är id så får man den identifierande koden annars får man -1.
GETCODE--> heltal
Om typen är op får man operatorns kod, dvs vilken operator det är. Operatorerna har koder från 1 till 16 i följande sekvens ; ( ) : , := = <> < <= > >= + - * /
Om det inte är något op-token blir resultatet -1.
GETLINE--> heltal
Ger ett radnummer om token är ett radnummertoken. Sådana skall ni inte se, utan de bör tas omhand internt, i Res metod GETTOKEN.
GETBLKNR--> heltal
Om vi har ett call block token returneras nummret på det block som anropas, annars ges -1.
GETNAME--> text
Ger alltid textsträngen oavsett typ.
GETINT--> heltal
GETREAL--> flyttal
Dessa ger talvärdet om det är ett token av rätt typ.

Token

TYPE - typ av token
CODE - “värde”
STRING- text

Metoder: GETID, GETINT, GETREAL, GETCODE, GETBLKNR,GETLINE, GETTYPE
(WRITE)

1 - id	symid	namnet
2 - heltal	?	talet som text
3 - flyttal	?	talet som text
4 - text	?	texten
5 - op	opkod	operatorn som text
6 - fel	felkod	“fel”
7 - rad	radnr	“line#”
10 - call	blockid	“Callblock”

Att använda programmet p1.
Man får köra p1 som översätter källtexten till en text i annat format. Man får sedan skriva ett interface som läser denna text och bygger upp de strukturer man behöver, lämpligen enligt beskrivningen ovan. I den resulterande texten har källprogrammet delats upp i ett antal block, där inre block brutits ut. Ett program blir således en sekvens av block, som inte innehåller några inkapslade block. Texten ser ut på följande sätt:

###PROGRAM###
n
Block-0
Block-1

Block-(n-1)
###PROGRAMSLUT###

Rader med ett eller flera # är egentligen redundant information, men gör det lättläsligare, dessutom kan inläsningsprogrammet kontrollera om strukturen är rätt eller inte. I principskissen ovan är n på andra raden ett heltal (textrepresentation), som anger antalet block i programmet. Därefter följer en sekvens av block, där varje block har följande utseende:

##BLOCK##
blkid sfblkid m d t
#DEKLARATIONER#
Symbol-1
Symbol-2

Symbol-d
#KOD#
Token-1
Token-2

Token-t
##BLOCKSLUT##

Andra raden i blocket innehåller fem heltal som ska tolkas på följande sätt: blkid är "namnet" på blocket, sfblkid är motsvarande namn på det statiskt omgivande blocket (Statisk Fader), m är det antal enheter (ord om 4 bytes) som behövs i minnet för blockets data under exekvering, d är det antal deklarationer som blocket innehåller, t slutligen är det antal tokens som blockets koddel innehåller. Varje deklaration representeras av en rad som anger symbolen och dess innehåll. En symbol beskrivs enligt följande: symid, typ, slag, info1, info2, info3, reladr, extnamn, gränser, namn. Alla dessa fram tom. reladr är heltal. extnamn är för er ** , men kan vara en

namnsträng om vi tillåter externt definierade enheter. gränser anges som # om det inte gäller en array, annars ges index, låg, hög här. För er är index alltid 1, låg och hög är tal, men kan vara mer komplicerat. Slutligen finns namn med som är den deklarerade enhetens namnsträng, som finns med av avlusningsskäl. Vad de olika fälten betyder kan ni se ovan, där en symbol är beskriven. Koddelen består av ett antal tokens, en per rad, enligt följande: kod, typ, sträng. Här är kod och typ heltal, medan sträng är en text. Betydelsen av de olika fälten är beskrivna under metod 2 ovan.

Användning av program.

När man använder program av typ p1 är det lämpligt att man håller sig till vissa konventioner. Den källkod man ska kompilera bör finnas i en fil med namn av viss typ. Lämpligen använder vi följande, vi antar att vi vill kalla programmet "prog". Vi har då källkoden i prog.prg, p1 lämnar sitt resultat i prog.p1, därefter kommer interpretatorn som ju inte generar någon kod ut, den kör ju programmet.
Mitt p1 arbetar enligt: p1 prog läser prog.prg och producerar prog.p1
OBS! Programmet p1 klarar ett språk som innehåller betydligt mer än vad ni ska klara, därför finns ett antal reserverade ord och operatorer som inte är aktuella för er! Detta ska ni normalt inte behöova bry er om.

Reserverade ord och opkoder.

**** Token Konstanter ****
T_ID=1,T_FIX=2,T_FLOAT=3,T_TEXT=4,T_OP=5,T_ERROR=6;
T_LINE=7,T_CALL=10;
TC_NO=0,TC_SEMI=1,TC_LPAR=2,TC_RPAR=3,TC_KOL=4,TC_COM=5;
TC_BECOME=6,TC_EQUAL=7,TC_NEQ=8,TC_LESS=9,TC_LE=10;
TC_GT=11,TC_GE=12,TC_PLUS=13,TC_MINUS=14,TC_MUL=15,TC_DIV=16;

**** Reserverade ord ****
integer ID_BEGIN=1,ID_END=2,ID_ARRAY=3,ID_FUNCTION=4,ID_INTEGER=5;
integer ID_REAL=6,ID_TEXT=7,ID_WHILE=8,ID_DO=9,ID_IF=10,ID_THEN=11;
integer ID_ELSE=12,ID_IS=13,ID_EXTERNAL=14,ID_OBJECT=15,ID_POINTER=16;
integer ID_NEW=17,ID_DYN=18,ID_MAX=ID_DYN;

Första användarsymbol får då koden 19

!**** Slag i symboler****
K_SIMPLE=0,K_ARRAY=1,K_FUNC=2,K_FUNCVAL=3,K_OBJECT=4,K_REF=5;

Här kommer lite om använda felkoder.

integer E_Sem=1,E_Id=2,E_Lpar=3,E_Rpar=4,E_ResId=5;
integer E_FunHead=6,E_Ext=7,E_Obj=8,E_MissSpec=9,E_OverSpec=10;
integer E_ArrLim=11,E_MultDec=12;

```
ErrMsg(0):-"Unspecified error";
ErrMsg(E_Sem):-"Missing ;";
ErrMsg(E_Id):-"Identifier expected";
ErrMsg(E_Lpar):-"Left ( expected";
ErrMsg(E_Rpar):-"Right )expected";
ErrMsg(E_ResId):-"Illegal use of reserved identifier";
ErrMsg(E_FunHead):-"Illegal function head";
ErrMsg(E_Ext):-"External expected";
ErrMsg(E_Obj):-"Illegal object structure";
ErrMsg(E_MissSpec):-"Missing specification";
ErrMsg(E_OverSpec):-"Too many specifications";
ErrMsg(E_ArrLim):-"Illegal array limit";
ErrMsg(E_MultDec):-"Multiple declaration";
```

Ett exempel på p1:s resultat.

```
begin
    integer I,K;
    real X;
    text H;
    integer array A(2:9);
    integer function Add(X,Y); integer X,Y; Add:=X+Y;
    I:=2;
    X:=4.5;
    K:=I+X;
    begin
            integer I;
            I:=1;
    end;
    H:="Hej";
    write(Add(I,K));
end
```

Detta program innehåller tre block. Det yttre blocket, dvs hela programmet. Sedan kommer funktionen Add som är ett block även om det inte finns något begin ... end, utan endast en sats. Slutligen finns det ett inre block begin integer .... end.

```
###PROGRAM###
   3
##BLOCK##
 0 -1  5  6 42
#DEKLARATIONER#
  19   1   0   0   0   0   0 **   #   I
  20   1   0   0   0   0   1 **   #   K
  21   2   0   0   0   0   2 **   #   X
  22   3   0   0   0   0   3 **   #   H
  23   1   1   1   0   0   4 **    1  2  9  A
  24   1   2   2   1   0   0 **   #   Add
#KOD#
   3   7   Line#
   4   7   Line#
```

```
  5   7   Line#
  6   7   Line#
  7   7   Line#
 19   1   I
  6   5   :=
  2   2   2
  1   5   ;
  8   7   Line#
 21   1   X
  6   5   :=
  5   3   4.5
  1   5   ;
  9   7   Line#
 20   1   K
  6   5   :=
 19   1   I
 13   5   +
 21   1   X
  1   5   ;
 10   7   Line#
 11   7   Line#
  2  10   Callblock
 14   7   Line#
 22   1   H
  6   5   :=
  0   4   Hej
  1   5   ;
 15   7   Line#
 26   1   write
  2   5   (
 24   1   Add
  2   5   (
 19   1   I
  5   5   ,
 20   1   K
  3   5   )
  3   5   )
  1   5   ;
 16   7   Line#
  2   1   end
##BLOCKSLUT##
##BLOCK##
 1  0  3  3  6
#DEKLARATIONER#
  24   1   3   0   0   0   0  **   #   Add
  21   1   0   0   0   0   1  **   #   X
  25   1   0   0   0   0   2  **   #   Y


#KOD#
  24   1   Add
   6   5   :=
  21   1   X
  13   5   +
  25   1   Y
   1   5   ;
##BLOCKSLUT##
##BLOCK##
 2  0  1  1  7
#DEKLARATIONER#
  19   1   0   0   0   0   0  **   #   I
#KOD#
  12   7   Line#
```

```
  19   1   I
   6   5   :=
   1   2   1
   1   5   ;
  13   7   Line#
   2   1   end
##BLOCKSLUT##
###PROGRAMSLUT###
```