# Laboration 3 : Ett ramverk för evaluering av uttryck

## 1 Syfte

- Användning av generiska klasser och gränssnitt.
- Avancerad mjukvarudesign.
- Allmän kännedom om RPN, BNF, uttryck, höger- och vänstervärden, m.m.

## 2 Uppgift

Vi skall implementera ett ramverk för evaluering av uttryck. Uttrycken kan t. ex. se ut som;

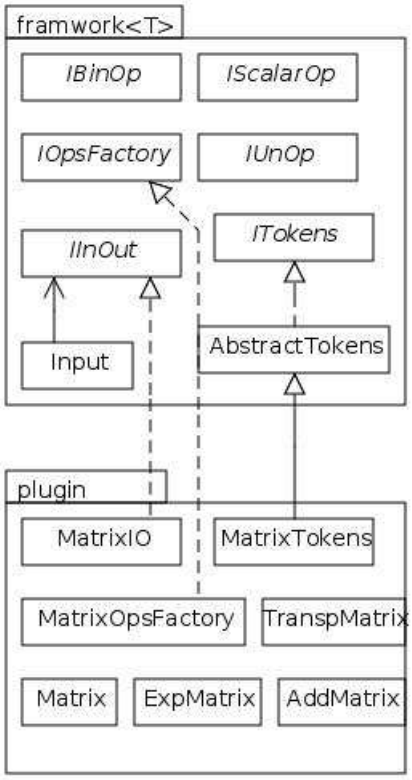
```
a b +
```

- Uttrycken skrivs i postfix notation (operatorn sist)
- a och b är variabler. Variablernas värden kan vara av godtycklig typ. Tanken är att typen skall vara "strukturerad" t.ex. vektorer, matriser, kvaternioner eller dylikt (behöver inte vara numeriska typer, kan tänka sig operationer på strängar eller bilder ...)

För att använda ramverket implementerar användaren (en annan programmerare) det typberoende delarna (plugin).

Självklart skall detta vara så enkelt som möjligt d.v.s. så mycket som möjligt skall skötas av ramverket. Efter implementationen kan slutanvändaren (icke-programmerare) använda ramverkat för olika beräkningar.

# 3 Övergripande Design




Figur 1: Ramverket och en plugin för Matriser.

## 3.1 framework<T>

Ramverket definierar de gränssnitt de typberoende delarna måste implementera utifrån den givna typen T samt en del typoberoende operationer.

- IBinOp, IUnOp och IScalarOp är de gränssnitt operatorerna måste implementera
  - IBinOp är en binär operation ( T x T -> T)
  - IUnOp är en unär operation ( T -> T )
  - IScalarOp definierar en skälar operation på typen ( k x T -> T, k:Number).
- IOpsFactory är en fabrik som dels definierar vilka strängar (tokens) som representerar operatorer samt mappningen mellan dessa och implementering av operatorerna.
- IInOut definierar metoder för att skriva och läsa typen till/från fil eller standard in- och utströmmar.
- Input är en klass som sköter inmatning av typen (finns även motvarande Output, visas ej).
- ITokens definierar en klassifikation av olika operander utifrån något strängmönster (reguljära uttryck). AbstractTokens innehåller en default-implementation, se vidare Uttryck.

## 3.2 plugin

I detta fallet är T vald till typen Matrix.

- MatrixIO implementerar läsning och skrivning av matriset till olika strömmar. Används av Input och Output.
- MatrixTokens utökar eller återanvänder de fördefinierade strängmönstren.
- MatrixOpsFactory mappar t.ex. "+" till en klass (metod) som sköter addition av matriser.
- Matrix är instansen av typen T (se vidare plugin manual nedan).
- AddMatrix är en binär operation, TranspMatrix är en unär operation och ExpMatrix är an skalär operation på typen.

# 4 Uttryck

Uttrycken måste som sagt skrivas i postfix notation. Detta gör evalueringen enkel, se t.ex. Wikipedia > Postfix notation. De fördefinierade uttryckens syntax ges av Figur 2 (se Wikipedia, Backus-Naur Form). Det är möjligt att samla ett godtyckligt antal uttryck i en fil (ett program).

## 4.1 Aritmetiska uttryck

De binära och unära operatorerna motsvarar i detta fall de "vanliga". Obs! Att olika uttryck kan ge samma resultat;

```
 // Addera tre a
a a a + +
// Addera tre a
a a + a +
```

## 4.2 Tilldelning

Tilldelning är ett uttryck. Värdet för tilldelning fungerar på samma sätt som i Java. Följande är korrekt;

```
a b = c d = +
```

Värdet av a b = är b och värdet av c d = är d. Resultatet blir alltså summan av b och d. Tilldelning har precis som vanligt sidoeffekten att a ges värdet b och c ges värdet av d.

### 4.2.1 Vänster respektive högervärden

Följande är fel;

```
a b + c =
```

Vi kan inte tilldela c till resultatet av a b +. Resultatet är ett (höger)värde inte en vänstervärder (= variabel = minnesplats). Korrekt är däremot;

```
c a b + =
```

Variabeln c tilldelas värdet av a b +.

## 4.3 Uttryck med < och >

De binära io-operatorerna <, > representerar in- och utmatning. Operander till dess måste vara ett vänstervärde samt ett filnamn eller symbolen för standard io ("_") . Korrekta exempel;

```
a _  >      // Skriv a till stdout
a _  <      // Läs a från stdin
a a.mtx <   // Läs från fil a.mtx
a a.mtx >   // Skriv a till fil a.mtx
```

Felaktiga exempel;

```
_ a <      // Kan inte skriva till standar io
a.mtx _ <  // Kan inte skriva till fil
```

Värdet av operatorerna är "det in/utmatade". Följande är korrekt;

```
a _ < b _ < +
```

Värdena är summan av de inlästa a och b variablerna.

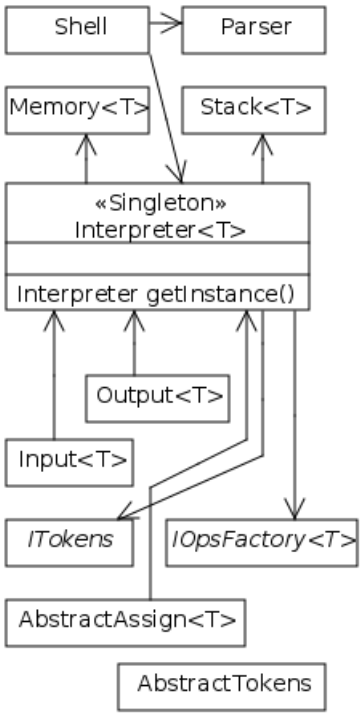
## 4.4 Felaktiga uttryck

Felaktiva uttryck hanteras under evalueringen, vi får runtime-fel. Se vidare nedan.

```
expr     ::= operand operand binop |
             operand iooperand ioop |
             operand unop |
             operand number scalarop
operand  ::= variable | expr   (obs! inga literaler)
iooperand::= stdio | file
binop    ::= "+" | "-" | "*" | "=" | ioop
ioop     ::=  "<" | ">"
unop     ::= "!"
scalarop ::= "^"
variable ::= börjar på bokstav, därefter bokstäver eller siffror
stdio    ::= "_"
file     ::= variable"."variable
```

Figur 2: BNF för uttryck

# 5 Detaljerad Design Ramverket




Figur 3: Design ramverket (vissa gränssnitt utlämnade,se bild ovan).

- Shell är en kommandoradstolk som läser in och skriver ut strängar (värden omvandlade till strängar). Används då slutanvändaren vill arbeta interaktivt och mata in/skriva ut uttryck. Förutom uttryck hanteras följande kommandon;
  - bye, quit, end avslutar ramverket.
  - exec *filnamn,* läser in en fil och exekverar uttrycken i denna (kör ett program).
- Parser delar upp strängar i tokens (delsträngar), t.ex. "+", "myVar" eller ">" (och läser bort irrelevanta saker som nyrad, kommentarer, ...
- Memory används för att spara variabelnamn och värdet för dessa.
- Stack används för evaluering enligt RPN.
- Interpreter sköter själva beräkningen av uttrycken m.h.a. Memory

och Stack samt implementationerna av ITokens och IOpsFactory. All hantering av felaktiga uttryck sköts också här. Input och Output behöver interpretatorn, detta löses genom att den implementeras som en Singleton.

- Input och Output sköter IO enligt tidigare.
- AbstractAssign<T> hanterar evaluering och sidoeffekter vid tilldelning.
- AbstractTokens som tidigare.

### 5.1 Evaluering av uttryck

Interpreter får en token-lista och läser denna tills listan är tom. Då listan är tom skall exakt ett värde ligga på stacktoppen. Detta är det slutgiltiga värdet av uttrycket. Vi kan använda en mycket enkel algoritm enligt Figur 4.

Lägg märke till att mellanresultat alltid finns på stacken men aldrig i minnet.

Att vi skapar ett nollvärde beror på att för vissa tokens är värdet ointressant (en dummy). Exempel;

```
a _ <
```

1. a är en variabel. Denna sparas i minnet med ett "nollvärde" (eftersom den f.n. saknar värde). Värdet pushas dessutom på stacken.
2. Vid _ händer samma sak.
3. Vid < (binär operator) hämtas två värden från stacken. Värdet till vänster måste vara en variabel, kan kontrolleras genom att slå upp namnet "a"' i minnet och/eller använda metoderna i ITokens. Till höger skall stå "_" eller ett filnamn, kan kontrolleras på samma sätt. Exekvering av <-operationen innebär att a's nollvärde ersätts med ett inläst värde. Stdin eller filnamn och värdena för dessa stryks ur minnet.

### 5.2 Felmeddelanden

Ramverket skall meddela så exakt som möjligt vad eventuella fel beror på (saknas operator operand, felaktigt vänstervärde, ... Några exempel (första '>' är shell prompten);

```
> a
Bad tokenlist. Too short
> a a
Missing operator after a
> a +
Missing operand for +
> a a + a =
Bad leftvalue
> _ a <
Bad leftvalue
```

## 6 Plugin manual

För att skapa en plugin gör man följande;

- Implementera typen T. Typen bör följa den kanoniska formen (clone, equals, toString...).
- Implementera IInOut.
- Implementera operationer på typen utifrån IBinOp, IUnop och IScalarOp.
  - Tilldelning implementeras genom att subklassa AbstractAssign och override:a getCopy(). Metoden skall leverera en djup kopia av argumentet.
  - IO operationer sköts med att de färdiga klasserna Input

```
while( finns fler tokens ){
  if( operator ){
    // Hämta operander från stack
    // Hämta impl. av operator från fabrik
    // Exekvera impl.
    // Pusha resultatvärde på stack
  } else if (operand ){
    if( operand finns i minne){
      // Hämta värde
    }else{
      // Skapa nytt "nollvärde"
      // Spara operand/nollvärde i minnet
    }
    // Pusha värde av på stack.
  } else {
    exception
  }
}
```

Figur 4: Algoritm för evaluering av uttryck.

och Output. Dessa måste få en implementation av IInOut som konstruktor-argument.

- Implementera IOpsFactory och bestäm mappning mellan tokens och operationer.
- Subklassa AbstractTokens. Vid behov override:a metoder.

För att starta ramverket (programmet) skriver man kommandot run.sh i en terminal. Som argument måste man ge de fullt kvalificerade klassnamnen för implementationerna av IOpsFactory och ITokens. Som ett tredje valfritt argument kan man ange en fil med uttryck (ett program). Anges ingen fil startar programmet kommandotolken för interaktiv körning.

```
$ run.sh factoryClassName\
      tokensClassName [programfil]
```

## 6.1 Matris-pluginen

Vi måste ha en plugin för att verifiera att ramverket fungerar. Pluginen skall hantera glesa matriser (d.v.s. de flesta element är 0). Matriselementens typer kan variera vi gör därför en generisk matrisklass.

Bara element skilda från noll sparas i den bakomliggande representationen. Så fort man slår upp ett element som saknas (inom matrisens min/max rad/kolumn) returneras värdet noll.

### 6.1.1 Operationer

Vissa operationer på matriser är beroende av matchande rader/kolumner på matriser. Se Wikipedia. Filformat för matriser finns givet (för$<$ och $>$ operatorerna). Följande operationer (förutom IO operationerns) skall finnas; tilldelning, addition, multiplikation, skalär multipliktion, exponentiering.

## 7 Utökning av språket

(BONUSPOÄNG) Försök att utöka språket med t.ex. en if-sats. För ideér se programmeringsspråket Forth (Wikipedia).

## 8 Arbetsgång

1. Läs in er på refererade artiklar.
2. Hämta startkod (ej körbar).
3. Välj en lämplig representation för matriser och gör klart den påbörjade typen för double-matriser (Matrix.java). Testa denna och testa MatrixDoubleIO.java.
4. Vi skall börja med IO operationerna. Tyvärr är dessa ganska intrasslade. För att det skall vara möjligt att testa dessa måste;
   a) Input och Output-klasserna vara körbara (dessa använder MatrixDoubleIO). Input klassen är klar ni behöver bara implementera Output.
   b) MatrixDoubleOpsFactory kunna leverera Input och Output objekt (då $<$ eller $>$ dyker upp i token-listan).
   c) Interpretatorn måste vara körbar och kunna evaluera ett uttryck typ "a _ $<$ ".

   Arbeta parallellt med dessa klasser och försök få den första testen i TestInterpretator att fungera (testInputSquare). För att få setUp-metoden att fungera måste Interpretatorn vara en Singleton.
5. Arbeta vidare med operationer och uttryck enligt de tester som finns i TestInterpreter och TestInterpreterException. Tester för transponering och exponentiering saknas. Lägg till.
6. Gör klart Main. Testa att köra hela "program" t.ex. arim.mc

## 9 Redovisning

Som tidigare.

**Inlämningsdatum** Se kurssidan.