UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

En interpret for Asp

Kompendium for INF2100

Stein Krogdahl, Dag Langmyhr

Høsten 2018

Innhold

Figurer

Tabeller

Forord

Dette kompendiet er laget for emnet *INF2100 – Prosjektoppgave i programmering*. Selve kurset er et av de eldste ved Ifi, men innholdet i kurset har allikevel blitt fornyet jevnlig.

Det opprinnelige kurset ble utviklet av *Stein Krogdahl* rundt 1980 og dreide seg om å skrive en kompilator som oversatte det Simula-lignende språket *Minila* til kode for en tenkt datamaskin *Flink*; implementasjonsspråket var Simula. I 1999 gikk man over til å bruke Java som implementasjonsspråk, og i 2007 ble kurset fullstendig renovert av *Dag Langmyhr*: Minila ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen Flink ble avløst av en annen ikkeeksisterende maskin kalt *Rask*. I 2010 ble det besluttet å lage ekte kode for Intel-prosessoren x86 slik at den genererte koden kunne kjøres direkte på en datamaskin. Dette medførte så store endringer i språket RusC at det fikk et nytt navn: *C<* (uttales «c less»). Ønsker om en utvidelse førte i 2012 til at det ble innført datatyper (`int` og `double`) og språket fikk igjen et nytt navn: *Cb* (uttales «c flat»). Tilbakemelding fra studentene avslørte at de syntes det ble veldig mye fikling å lage kode for `double`, så i 2014 ble språket endret enda en gang. Under navnet *AlboC* («A little bit of C») hadde det nå pekere i stedet for flyt-tall.

Nå var det blitt 2015, og hele opplegget gikk gjennom enda en revisjon. Det gjaldt også språket som skulle kompileres: i dette og det etterfølgende året var det et språk som omfattet mesteparten av gode, gamle *Pascal*.

I 2017 ble det en ny gjennomgang. Siden Ifi fra og med denne høsten vil benytte Python som introduksjonsspråk, dukket det opp et ønske om å gi studentene en grundig innføring i hvordan en Python-interpret fungerer. Til dette ble *Asp* (som altså er en mini-Python) utviklet. Basert på erfaringene fra 2017 ble det i 2018 foretatt et par mindre endringer av Asp.

Målet for dette kompendiet er at det sammen med forelesningsplansjene skal gi studentene tilstrekkelig bakgrunn til å kunne gjennomføre prosjektet.

Forfatterne vil ellers takke studentene *Einar Løvhøiden Antonsen*, *Jonny Bekkevold*, *Eivind Alexander Bergem*, *Marius Ekeberg*, *Asbjørn Gaarde*, *Arne Olav Hallingstad*, *Espen Tørressen Hangård*, *Robin Hansen*, *Sigmund Hansen*, *Markus Hauge*, *Simen Heggestøyl*, *Simen Jensen*, *Thor Joramo*, *Morten Kolstad*, *Jan Inge Lamo*, *Brendan Johan Lee*, *Håvard Koller Noren*, *Vegard Nossum*, *Hans Jørgen Nygårdshaug*, *David J Oftedal*, *Mikael Olausson*, *Cathrine Elisabeth Olsen*, *Bendik Rønning Opstad*, *Christian Resell*, *Christian Andre Finnøy Ruud*, *Ryhor Sivuda*, *Yrjab Skrimstad*, *Gaute Solheim*, *Herman Torjussen*, *Christian Tryti*, *Jørgen Vigdal*, *Olga Voronkova*, *Aksel L Webster* og *Sindre Wilting* som har påpekt skrivefeil eller foreslått forbedringer i tidligere utgaver. Om flere studenter gjør dette, vil de også få navnet sitt på trykk.

Blindern, 21. august 2018
Stein Krogdahl *Dag Langmyhr*

Teori er når ingenting virker og alle vet hvorfor. Praksis er når allting virker og ingen vet hvorfor.

I dette kurset kombineres teori og praksis – ingenting virker og ingen vet hvorfor.

— Forfatterne

Kapittel 1

Innledning

1.1 Hva er emnet INF2100?

Emnet INF2100 har betegnelsen *Prosjektoppgave i programmering*, og hovedideen med dette emnet er å ta med studentene på et så stort programmeringsprosjekt som mulig innen rammen av de ti studiepoeng kurset har. Grunnen til at vi satser på ett stort program er at de fleste ting som har å gjøre med strukturering av programmer, objektorientert programmering, oppdeling i moduler etc, ikke oppleves som meningsfylte eller viktige før programmene får en viss størrelse og kompleksitet. Det som sies om slike ting i begynnerkurs, får lett preg av litt livsfjern «programmeringsmoral» fordi man ikke ser behovet for denne måten å tenke på i de små oppgavene man vanligvis rekker å gå gjennom.

Ellers er programmering noe man trenger trening for å bli sikker i. Dette kurset vil derfor ikke innføre så mange nye begreper omkring programmering, men i stedet forsøke å befeste det man allerede har lært, og demonstrere hvordan det kan brukes i forskjellige sammenhenger.

«Det store programmet» som skal lages i løpet av INF2100, er en **interpret**, dvs et program som leser og analyserer et program i et gitt programmeringsspråk, og som deretter utfører det som dette programmet angir skal gjøres. Nedenfor skal vi se nærmere på likheter og forskjeller mellom en interpret og en kompilator.

Selv om vi konsentrerer dette kurset omkring ett større program vil ikke dette kunne bli noe virkelig *stort* program. Ute i den «virkelige» verden blir programmer fort vekk på flere hundre tusen eller endog millioner linjer, og det er først når man skal i gang med å skrive slike programmer, og, ikke minst, senere gjøre endringer i dem, at strukturen av programmene blir helt avgjørende. Det programmet vi skal lage i dette kurset vil typisk bli på drøyt fire tusen linjer.

I dette kompendiet beskrives stort sett bare selve programmerings-oppgaven som skal løses. I tillegg til dette kan det komme ytterligere krav, for eksempel angående bruk av verktøy eller skriftlige arbeider som skal leveres. Dette vil i så fall bli opplyst om på forelesningene og på kursets nettsider.

1.2 Hvorfor lage en interpret?

Når det skulle velges tema for en programmeringsoppgave til dette kurset, var det først og fremst to kriterier som var viktige:

- Oppgaven må være overkommelig å programmere innen kursets ti studiepoeng.
- Programmet må angå en problemstilling som studentene kjenner, slik at det ikke går bort verdifull tid til å forstå hensikten med programmet og dets omgivelser.

I tillegg til dette kan man ønske seg et par ting til:

- Det å lage et program innen et visst anvendelsesområde gir vanligvis også bedre forståelse av området selv. Det er derfor også ønskelig at anvendelsesområdet er hentet fra programmering, slik at denne bivirkningen gir økt forståelse av faget selv.
- Problemområdet bør ha så mange interessante variasjoner at det kan være en god kilde til øvingsoppgaver som kan belyse hovedproblemstillingen.

Ut fra disse kriteriene synes ett felt å peke seg ut som spesielt fristende, nemlig det å skrive en interpret, altså en forenklet versjon av den vanlige Python-interpreten. Dette er en type verktøy som alle som har arbeidet med programmering, har vært borti, og som det også er verdifullt for de fleste å lære litt mer om.

Det å skrive en interpret vil også for de fleste i utgangspunktet virke som en stor og uoversiktlig oppgave. Noe av poenget med kurset er å demonstrere at med en hensiktsmessig oppsplitting av programmet i deler som hver tar ansvaret for en avgrenset del av oppgaven, så kan både de enkelte deler og den helheten de danner, bli høyst medgjørlig. Det er denne erfaringen, og forståelsen av hvordan slik oppdeling kan gjøres på et reelt eksempel, som er det viktigste studentene skal få med seg fra dette kurset.

Vi skal i neste avsnitt se litt mer på hva en interpret er og hvordan den står i forhold til liknende verktøy. Det vil da også raskt bli klart at det å skrive en interpret for et «ekte» programmeringsspråk vil bli en altfor omfattende oppgave. Vi skal derfor forenkle oppgaven en del ved å lage vårt eget lille programmeringsspråk **Asp**. Vi skal i det følgende se litt nærmere på dette og andre elementer som inngår i oppgaven.

1.3 Interpreter, kompilatorer og liknende

Mange som starter på kurset INF2100, har neppe full oversikt over hva en interpret er og hvilken funksjon den har i forbindelse med et programmeringsspråk. Dette vil forhåpentligvis bli mye klarere i løpet av kurset, men for å sette scenen skal vi gi en kort forklaring her.

Grunnen til at man i det hele tatt har interpreter og kompilatorer, er at det er høyst upraktisk å bygge datamaskiner slik at de direkte utfra sin elektronikk kan utføre et program skrevet i et høynivå programmeringsspråk som for eksempel Java, C, C++, Perl eller Python. I stedet er datamaskiner

bygget slik at de kan utføre et begrenset repertoar av nokså enkle instruksjoner, og det blir derved en overkommelig oppgave å lage elektronikk som kan utføre disse. Til gjengjeld kan datamaskiner raskt utføre lange sekvenser av slike instruksjoner, grovt sett med en hastighet av 1–3 milliarder instruksjoner per sekund.

1.3.1 Interpretering

En **interpret** er et program som leser et gitt program og bygger opp en intern representasjon av programmet. Deretter utføres det som angis i denne representasjonen.

Det er flere fordeler ved å utføre programmer på denne måten:

- Siden programmet lagres internt, er det mulig å endre programmet under kjøring. (I vårt programmeringsspråk Asp er det ikke mulig.)
- Når man først har skrevet en interpret, er det enkelt å installere den på andre maskiner, selv om disse har en annen prosessor eller et annet operativsystem.

Den største ulempen ved å interpretere programmer er det går tregere; hvor mye tregere avhenger både av det aktuelle språket og kvaliteten på interpreten. Det er imidlertid vanlig å regne med at en interpret bruker 5–10 ganger så lang tid som kompilert kode.

1.3.2 Kompilering

En annen måte å få utført programmer på er å lage en **kompilator** som *oversetter* programmet til en tilsvarende sekvens av maskininstruksjoner for en gitt datamaskin. En kompilator er altså et program som leser data inn og leverer data fra seg. Dataene det leser inn er et tekstlig program (i det programmeringsspråket denne kompilatoren skal oversette fra), og data det leverer fra seg er en sekvens av maskininstruksjoner for den aktuelle maskinen. Disse maskininstruksjonene vil kompilatoren vanligvis legge på en fil i et passelig format med tanke på at de senere kan kopieres inn i en maskin og bli utført.

Det settet med instruksjoner som en datamaskin kan utføre direkte i elektronikken, kalles maskinens **maskinspråk**, og programmer i dette språket kalles *maskinprogrammer* eller *maskinkode*.

1.3.2.1 Kompilering og kjøring av Java-programmer

En av de opprinnelige ideene ved Java var knyttet til datanett ved at et program skulle kunne kompileres på én maskin for så å kunne sendes over nettet til en hvilken som helst annen maskin (for eksempel som en såkalt *applet*) og bli utført der. For å få til dette definerte man en tenkt datamaskin kalt *Java Virtual Machine* (JVM) og lot kompilatorene produsere maskinkode (gjerne kalt *byte-kode*) for denne maskinen. Det er imidlertid ingen datamaskin som har elektronikk for direkte å utføre slik byte-kode, og maskinen der programmet skal utføres må derfor ha et program som simulerer JVM-maskinen og dens utføring av byte-kode. Vi kan da gjerne si at et slikt simuleringsprogram interpreterer maskinkoden

til JVM-maskinen. I dag har for eksempel de fleste nettlesere (Firefox, Opera, Chrome og andre) innebygget en slik JVM-interpret for å kunne utføre Java-applets når de får disse (ferdig kompilert) over nettet.

Slik interpretering av maskinkode går imidlertid normalt en del saktere enn om man hadde oversatt til «ekte» maskinkode og kjørt den direkte på «hardware». Typisk kan dette for Javas byte-kode gå 2 til 10 ganger så sakte. Etter hvert som Java er blitt mer populært har det derfor også blitt behov for systemer som kjører Java-programmer raskere, og den vanligste måten å gjøre dette på er å utstyre JVM-er med såkalt «Just-In-Time» (JIT)-kompilering. Dette vil si at man i stedet for å interpretere byte-koden, oversetter den videre til den aktuelle maskinkoden umiddelbart før programmet startes opp. Dette kan gjøres for hele programmer, eller for eksempel for klasse etter klasse etterhvert som de tas i bruk første gang.

Man kan selvfølgelig også oversette Java-programmer på mer tradisjonell måte direkte fra Java til maskinkode for en eller annen faktisk maskin, og slike kompilatorer finnes og kan gi meget rask kode. Om man bruker en slik kompilator, mister man imidlertid fordelen med at det kompilerte programmet kan kjøres på alle systemer.

1.4 Oppgaven og dens fire deler

Oppgaven skal løses i fire skritt, hvor alle er obligatoriske oppgaver. Som nevnt kan det utover dette komme krav om for eksempel verktøybruk eller levering av skriftlige tilleggsarbeider, men også dette vil i så fall bli annonsert i god tid.

Hele programmet kan grovt regnet bli på fra fire til fem tusen Java-linjer, alt avhengig av hvor tett man skriver. Vi gir her en rask oversikt over hva de fire delene vil inneholde, men vi kommer fyldig tilbake til hver av dem på forelesningene og i senere kapitler.

1.4.1 Del 1: Skanneren

Første skritt, del 1, består i å få Asps **skanner** til å virke. Skanneren er den modulen som fjerner kommentarer fra programmet, og så deler den gjenstående teksten i en veldefinert sekvens av såkalte **symboler** (på engelsk «tokens»). Symbolene er de «ordene» programmet er bygget opp av, så som *navn*, *tall*, *nøkkelord*, '+', '>=', '(' og alle de andre tegnene og tegnkombinasjonene som har en bestemt betydning i Asp-språket.

Denne «renskårne» sekvensen av symboler vil være det grunnlaget som resten av interpreten eller kompilatoren skal arbeide videre med. Noe av programmet til del 1 vil være ferdig laget eller skissert, og dette vil kunne hentes på angitt sted.

1.4.2 Del 2: Parseren

Del 2 vil ta imot den symbolsekvensen som blir produsert av del 1, og det sentrale arbeidet her vil være å sjekke at denne sekvensen har den formen et riktig Asp-program skal ha (altså, at den følger Asps **syntaks**).

Om alt er i orden, skal del 2 bygge opp et **syntakstre**, en **trestruktur** av objekter som direkte representerer det aktuelle Asp-programmet, altså hvordan det er satt sammen av «expr» inne i «stmt» inne i «func def» osv.

1.4.3 Del 3: Interpretering av uttrykk

I del 3 skal man ta imot et syntakstre for et uttrykk og så evaluere det, dvs beregne resultatverdien. Man må også sjekke at uttrykket ikke har typefeil.

1.4.4 Del 4: Full interpretering

Siste del er å kunne evaluere alle mulige Asp-programmer, dvs programmer med funksjonsdefinisjoner samt setninger med løkker, tester og uttrykk. Dessuten må vi definere et bibliotek med diverse predefinerte funksjoner.

1.5 Krav til samarbeid og gruppetilhørighet

Normalt er det meningen at to personer skal samarbeide om å løse oppgaven. De som samarbeider bør være fra samme øvingsgruppe på kurset. Man bør tidlig begynne å orientere seg for å finne én på gruppen å samarbeide med. Det er også lov å løse oppgaven alene, men dette vil selvfølgelig gi mer arbeid. Om man har en del programmeringserfaring, kan imidlertid dette være et overkommelig alternativ.

Hvis man får samarbeidsproblemer (som at den andre «har meldt seg ut» eller «har tatt all kontroll»), si fra i tide til gruppelærer eller kursledelse, så kan vi se om vi kan hjelpe dere å komme over «krisen». Slikt har skjedd før.

1.6 Kontroll av innlevert arbeid

For å ha en kontroll på at hvert arbeidslag har programmert og testet ut programmene på egen hånd, og at begge medlemmene har vært med i arbeidet, må studentene være forberedt på at gruppelæreren eller kursledelsen forlanger at studenter som har arbeidet sammen, skal kunne redegjøre for oppgitte deler av den interpreten de har skrevet. Med litt støtte og hint skal de for eksempel kunne gjenskape deler av selve programmet på en tavle.

Slik kontroll vil bli foretatt på stikkprøvebasis samt i noen tilfeller der gruppelæreren har sett lite til studentene og dermed ikke har hatt kontroll underveis med studentenes arbeid.

Dessverre har vi tidligere avslørt fusk; derfor ser vi det nødvendig å holde slike overhøringer på slutten av kurset. Dette er altså ingen egentlig eksamen, bare en sjekk på at dere har gjort arbeidet selv. Noe ekstra arbeid for dem som blir innkalt, blir det heller ikke. Når dere har programmert og testet ut programmet, kan dere interpreten deres forlengs, baklengs og med bind for øynene.

Et annet krav er at alle innleverte programmer er vesentlig forskjellig fra alle andre innleveringer. Men om man virkelig gjør jobben selv, får man automatisk et unikt program.

Hvis noen er engstelige for hvor mye de kan samarbeide med andre utenfor sin gruppe, vil vi si:

- Ideer og teknikker kan diskuteres fritt.
- Programkode skal gruppene skrive selv.

Eller sagt på en annen måte: Samarbeid er bra, men kopiering er galt!

Merk at *ingen godkjenning av enkeltdeler er endelig* før den avsluttende runden med slik muntlig kontroll, og denne blir antageligvis holdt en gang rundt begynnelsen av desember.

Du kan lese mer om Ifis regler for kopiering og samarbeid ved obligatoriske oppgaver på nettsiden https://www.uio.no/studier/eksamen/obligatoriske-aktiviteter/mn-ifi-oblig.html. Les dette for å være sikker på at du ikke blir mistenkt for ulovlig kopiering.

1.7 Delta på øvingsgruppene

Ellers vil vi oppfordre studentene til å være aktive på de ukentlige øvingsgruppene. Oppgavene som blir gjennomgått, er meget relevante for skriving av Asp-interpreten. Om man tar en liten titt på oppgavene før gruppetimene, vil man antagelig få svært mye mer ut av gjennomgåelsen.

På gruppa er det helt akseptert å komme med et uartikulert:

> «Jeg forstår ikke hva dette har med saken å gjøre!»

Antageligvis føler da flere det på samme måten, så du gjør gruppa en tjeneste. Og om man synes man har en aha-opplevelse, så det fin støtte både for deg selv og andre om du sier:

> «Aha, det er altså ... som er poenget! Stemmer det?»

Siden det er mange nye begreper å komme inn i, er det viktig å begynne å jobbe med dem så tidlig som mulig i semesteret. Ved så å ta det fram i hodet og oppfriske det noen ganger, vil det neppe ta lang tid før begrepene begynner å komme på plass. Kompendiet sier ganske mye om hvordan oppgaven skal løses, men alle opplysninger om hver programbit står ikke nødvendigvis samlet på ett sted.

Til sist et råd fra tidligere studenter: *Start i tide!*

Kapittel 2

Programmering i Asp

Programmeringsspråket **Asp** er et programmeringsspråk som inneholder de mest sentrale delene av **Python**. Syntaksen er gitt av jernbanediagrammene i figur 2.2 til 2.30 på side 19–26 og bør være lett forståelig for alle som har programmert litt i Python. Et typisk eksempel på et Asp-program er vist i figur 2.1 på neste side.[1]

2.1 Kjøring

Inntil dere selv har laget en Asp-interpret, kan dere benytte referanseinterpreten:

```
$ ~inf2100/asp primes.asp
This is the Ifi Asp interpreter (2018-08-16)
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97 101 103 107 109 113
 127 131 137 139 149 151 157 163 167 173
 179 181 191 193 197 199 211 223 227 229
 233 239 241 251 257 263 269 271 277 281
 283 293 307 311 313 317 331 337 347 349
 353 359 367 373 379 383 389 397 401 409
 419 421 431 433 439 443 449 457 461 463
 467 479 487 491 499 503 509 521 523 541
 547 557 563 569 571 577 587 593 599 601
 607 613 617 619 631 641 643 647 653 659
 661 673 677 683 691 701 709 719 727 733
 739 743 751 757 761 769 773 787 797 809
 811 821 823 827 829 839 853 857 859 863
 877 881 883 887 907 911 919 929 937 941
 947 953 967 971 977 983 991 997
```

Denne finnes på Ifis Linux-maskiner, men dere kan også hente JAR-filen ~inf2100/asp.jar eller http://inf2100.at.ifi.uio.no/oblig/lib/asp.jar og kjøre interpreten på private maskiner.

[1] Du finner kildekoden til dette programmet og også andre nyttige testprogrammer i mappen ~inf2100/oblig/test/ på alle Ifi-maskiner; mappen er også tilgjengelig fra en vilkårlig nettleser som http://inf2100.at.ifi.uio.no/oblig/test/.

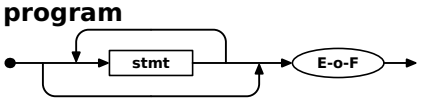
primes.asp

```
1
2  # Finn alle primtall opp til n
3  # ved hjelp av teknikken kalt «Eratosthenes' sil».
4
5  n = 1000
6  primes = [True] * (n+1)
7
8  def find_primes():
9     for i1 in range(2,n+1):
10       i2 = 2 * i1
11       while i2 <= n:
12          primes[i2] = False
13          i2 = i2 + i1
14
15 def w4(n):
16     if n <= 9:
17         return '   ' + str(n)
18     elif n <= 99:
19         return '  ' + str(n)
20     elif n <= 999:
21         return ' ' + str(n)
22     else:
23         return str(n)
24
25
26 def list_primes():
27     n_printed = 0
28     line_buf = ''
29     for i in range(2,n+1):
30        if primes[i]:
31           if n_printed > 0 and n_printed % 10 == 0:
32              print(line_buf)
33              line_buf = ''
34           line_buf = line_buf + w4(i)
35           n_printed = n_printed + 1
36     print(line_buf)
37
38
39 find_primes()
40 list_primes()
```

Figur 2.1: Eksempel på et Asp-program

2.2 Asp-program

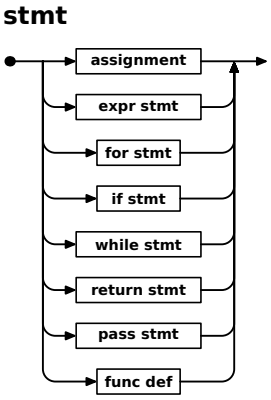
Som vist i figur 2.2, består et Asp-program av en sekvens av setninger (⟨stmt⟩). Symbolet `E-o-f` angir slutt på filen («end of file»).



Figur 2.2: Jernbanediagram for ⟨program⟩

2.2.1 Setninger

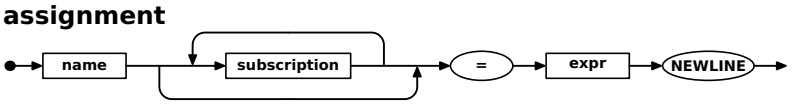
Figur 2.3 viser hva slags setninger man kan bruke i Asp.



Figur 2.3: Jernbanediagram for ⟨stmt⟩

2.2.1.1 Tilordning

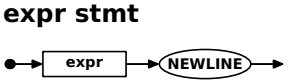
Som i de fleste andre språk, brukes en tilordningssetning til å gi variabler en verdi. Siden Asp har *dynamisk typing*, skal ikke variablene deklareres på forhånd. Les mer om dette i avsnitt 2.3.3 på side 26.



Figur 2.4: Jernbanediagram for ⟨assignment⟩

2.2.1.2 Uttrykk som setning

Et løsrevet uttrykk er også en lovlig setning; dette er spesielt aktuelt når utrykket er et funksjonskall.

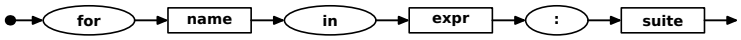


Figur 2.5: Jernbanediagram for ⟨expr stmt⟩

2.2.1.3 For-setninger

Denne formen for løkke går gjennom alle elementene i løkkekontrolluttrykket, som må være en liste.

for stmt

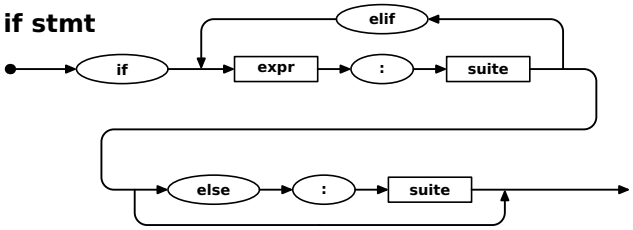


Figur 2.6: Jernbanediagram for ⟨for stmt⟩

2.2.1.4 If-setninger

If-setninger brukes til å velge om setninger skal utføres eller ikke. Se forøvrig avsnitt 2.3.1.1 på side 24 for hva som er lovlige testverdier.

if stmt

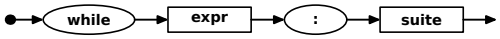


Figur 2.7: Jernbanediagram for ⟨if stmt⟩

2.2.1.5 While-setninger

While-setninger er en annen form for løkkesetning i Asp. Se forøvrig avsnitt 2.3.1.1 på side 24 for hva som er lovlige testverdier.

while stmt

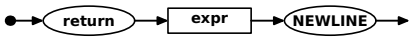


Figur 2.8: Jernbanediagram for ⟨while stmt⟩

2.2.1.6 Return-setninger

Return-setninger brukes til å avslutte utførelsen av en funksjon og angi en resultatverdi.

return stmt

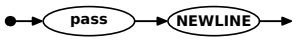


Figur 2.9: Jernbanediagram for ⟨return stmt⟩

2.2.1.7 Pass-setninger

Pass-setninger gjør ingenting; de eksisterer bare for kunne settes der det kreves en setning uten at noe skal gjøres.

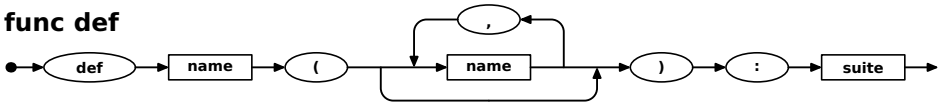
pass stmt



Figur 2.10: Jernbanediagram for ⟨pass stmt⟩

2.2.1.8 Funksjonsdeklarasjoner

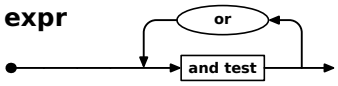
I Asp regnes funksjonsdeklarasjoner som setninger.



Figur 2.11: Jernbanediagram for ⟨func def⟩

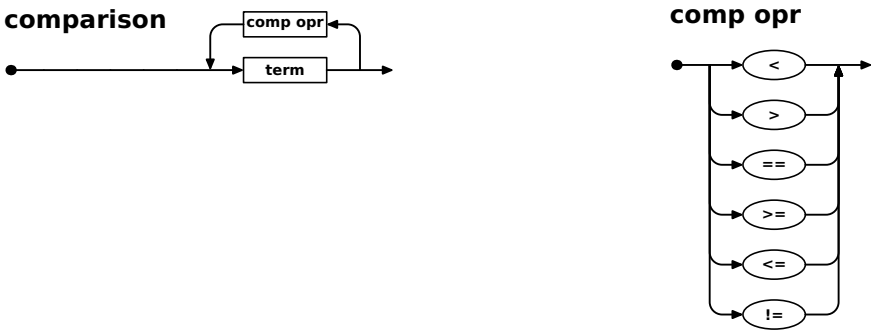
2.2.2 Uttrykk

Et uttrykk beregner en verdi. Det er definert ved hjelp av ganske mange ikketerminaler for å sikre at presedensen[2] blir slik vi ønsker.



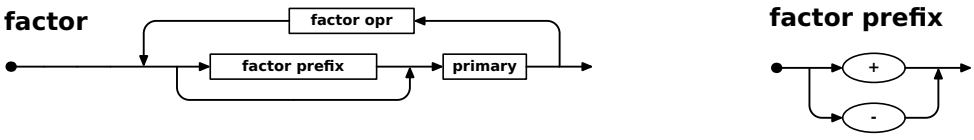
Figur 2.12: Jernbanediagram for ⟨expr⟩

Figur 2.13: Jernbanediagram for ⟨and test⟩ og ⟨not test⟩



Figur 2.14: Jernbanediagram for ⟨comparison⟩ og ⟨comp opr⟩

Figur 2.15: Jernbanediagram for ⟨term⟩ og ⟨term opr⟩

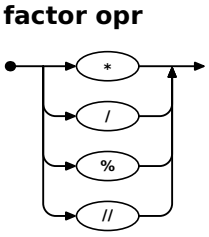


Figur 2.16: Jernbanediagram for ⟨factor⟩ og ⟨factor prefix⟩

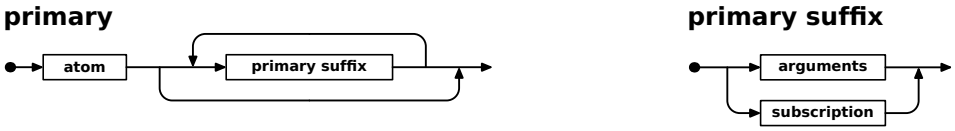
[2] Operatorer har ulik **presedens**, dvs at noen operatorer binder sterkere enn andre. Når vi skriver for eksempel

$$a + b \times c$$

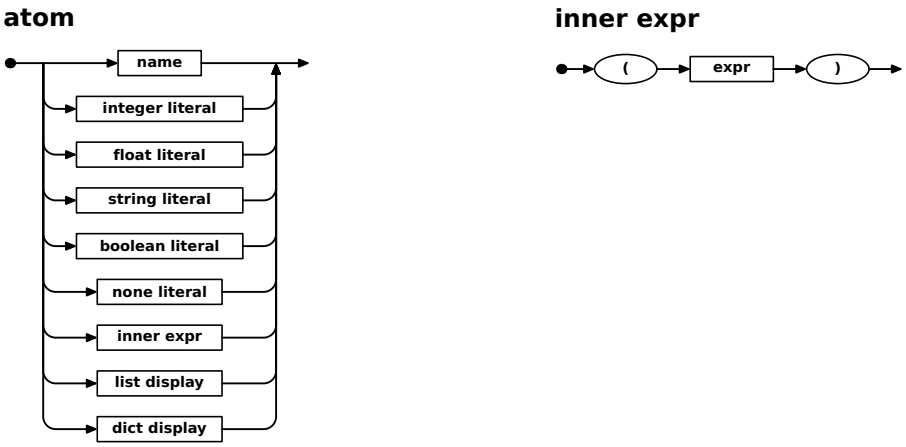
tolkes dette vanligvis som $a + (b \times c)$ fordi $\times$ normalt har høyere presedens enn $+$, dvs $\times$ binder sterkere enn $+$.



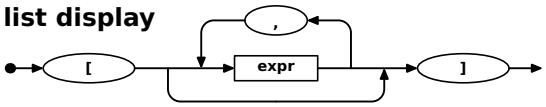
Figur 2.17: Jernbanediagram for ⟨factor opr⟩



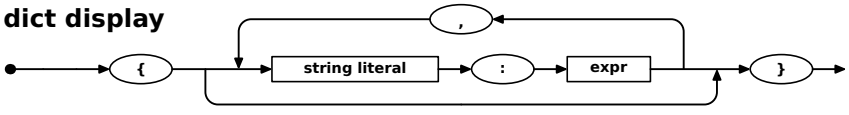
Figur 2.18: Jernbanediagram for ⟨primary⟩ og ⟨primary suffix⟩



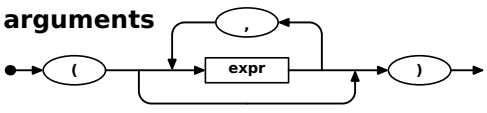
Figur 2.19: Jernbanediagram for ⟨atom⟩ og ⟨inner expr⟩



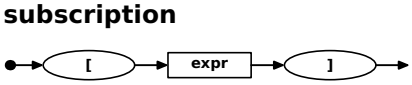
Figur 2.20: Jernbanediagram for ⟨list display⟩



Figur 2.21: Jernbanediagram for ⟨dict display⟩



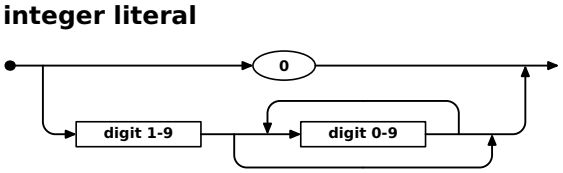
Figur 2.22: Jernbanediagram for ⟨arguments⟩



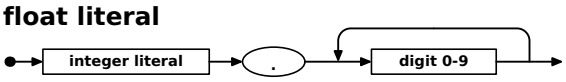
Figur 2.23: Jernbanediagram for ⟨subscription⟩

2.2.2.1 Literaler

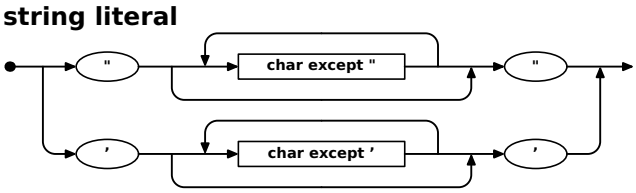
En **literal**[3] er et språkelement som angir en verdi; for eksempel angir «123» alltid heltallsverdien 123.



Figur 2.24: Jernbanediagram for ⟨integer literal⟩



Figur 2.25: Jernbanediagram for ⟨float literal⟩

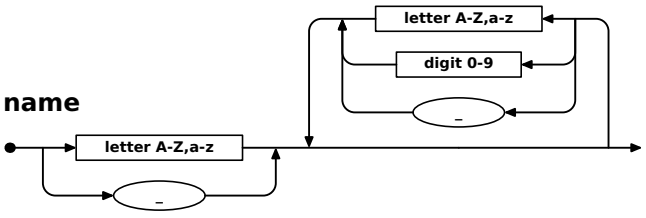


Figur 2.26: Jernbanediagram for ⟨string literal⟩

Figur 2.27: Jernbanediagram for ⟨boolean literal⟩ og ⟨non literal⟩

2.2.2.2 Navn

I Asp benyttes navn til å identifisere variabler og funksjoner.



Figur 2.28: Jernbanediagram for ⟨name⟩

2.3 Spesielle ting i Asp

Noen konstruksjoner i Asp (og følgelig også i Python) kan virke uvante første gang man ser dem.

[3] En **literal** er noe annet enn en **konstant**. En konstant er en navngitt verdi som ikke kan endres mens en literal angir verdien selv.

2.3.1 Typer

Tabell 2.1 gir en oversikt over hvilke typer data i Asp kan ha.

Type	Verdier	Eksempel
bool	Logiske verdier True og False	`True`
dict	Ordbok med verdier	`{'Ja': 17, 'Nei': 0}`
float	Flyt-tall	`3.14159`
func	Funksjoner	`def f(): ...`
int	Heltall	`124`
list	Liste av verdier	`[1, 2, "Ja"]`
none	«Ingenting»-verdien None	`None`
string	Tekster	`"Abrakadabra"`

Tabell 2.1: Typer i Asp

2.3.1.1 Logiske verdier

Språket Asp har en logisk type med verdier `True` og `False`, men det er mye mer fleksibelt i hva det godtar som lovlige logiske verdier i if- og while-setninger eller i uttrykk. Tabell 2.2 angir hva som tillates som logiske verdier.

Type	False	True
bool	`False`	`True`
dict	`{}`	ikke-tomme ordbøker
float	0.0	alle andre verdier
int	0	alle andre verdier
list	`[]`	ikke-tomme lister
none	**None**	—
string	`""`	alle andre tekststrenger

Tabell 2.2: Lovlige logiske verdier i Asp

Tabell 2.3 på side 27 viser at vi har de vanlige operatorene `and`, `or` og `not` for logiske verdier, men resultatet er litt uventet for `and` og `or`: de gir ikke svarene `True` eller `False`, men returnerer i stedet én av de to operandene, slik som dette:

```
"To be" or "not to be" ⇒ "To be"
"Yes" and 3.14         ⇒ 3.14
```

2.3.1.2 Tallverdier

Asp har både heltall og flyt-tall og kan automatisk konvertere fra heltall til flyt-tall ved behov, for eksempel når vi vil addere et tall av hver type.

Tabell 2.3 på side 27 viser at Asp har operatorer for de vanlige fire regneartene, men legg merke til at det finnes to former for divisjon:

`/` er flyt-tallsdivisjon der svaret alltid er et flyt-tall.

in-koder.asp

```
IN_emner = {}  # Start med en tom tabell
IN_emner["IN1000"] = "Introduksjon i objektorientert programmering"
IN_emner["IN1010"] = "Objektorientert programmering"
IN_emner["IN1020"] = "Introduksjon til datateknologi"
IN_emner["IN1030"] = "Systemer, krav og konsekvenser"

emne = input("Gi en emnekode: ")
while emne != "":
   print(emne, "er", IN_emner[emne])
   emne = input("Gi en emnekode: ")
```

Figur 2.29: Eksempel på bruk av ordbøker i Asp

`//` er heltallsdivisjon der svaret alltid er lik et heltall. (Det kan være et heltall som for eksempel 17, men det kan også være et flyt-tall som 3.0, med andre ord et flyt-tall som har 0-er bak desimalpunktet.)

2.3.1.3 Tekstverdier

Tekstverdier kan inneholde vilkårlig mange Unicode-tegn. Tekstliteraler kan angis med enten enkle eller doble anførselstegn (se figur 2.26 på side 23). Den eneste forskjellen på de to er hvilke tegn literalen kan inneholde; ingen av dem kan nemlig inneholde tegnet som brukes som markering. Med andre ord, hvis tekstliteralen vår skal inneholde et dobbelt anførselstegn, må vi bruke enkle anførselstegn rundt literalen.[4]

Asp har ikke særlig mange operatorer for tekster, men det har disse:

`s[`*`i`*`]` kan gi oss enkelttegn fra teksten.[5]

s_1 `+` s_2 skjøter sammen to tekster.

`s * `*`i` lager en tekst bestående av *i* kopier av `s`, for eksempel

```
"abc" * 3 ⇒ "abcabcabc"
```

2.3.1.4 Lister

Istedenfor arrayer har Asp *lister*. De opprettes ved å ta en startliste (ofte med bare ett element) og så kopiere den så mange ganger vi ønsker. Et godt eksempel på lister ser du i figur 2.1 på side 18.

2.3.1.5 Ordbøker

Asp har også *ordbøker* (på engelsk kalt «*dictionaries*») som fungerer som lister som indekseres med tekster i stedet for med heltall; de minner om `HashMap` i Java. Figur 2.29 viser et eksempel på hvordan man kan bruke ordbøker.

2.3.2 Operatorer

Tabell 2.3 på side 27 viser hvilke operatorer som er bygget inn i Asp.

[4] Er det mulig å ha en tekstliteral som inneholder både enkelt og dobbelt anførselstegn? Svaret er nei, men vi kan lage en slik tekstverdi ved å skjøte to tekster.

[5] Noen programmeringsspråk har en egen datatype for enkelttegn; for eksempel har Java typen `char`. Asp har ingen slik type, så et enkelttegn er en tekst med lengde 1.

2.3.2.1 Sammenligninger

Sammenligninger (dvs ==, <= etc) fungerer som normalt, men til forskjell fra programmeringsspråk som Java og C kan de også skjøtes sammen:

`a == b == c ==` ··· er det samme som `a==b and b==c and` ...

2.3.3 Dynamisk typing

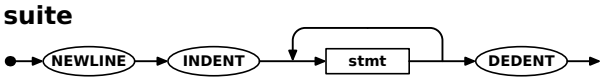
De fleste programmaringsspråk har **statisk typing** som innebærer at alle variabler, funksjoner og parametre har en gitt type som er den samme under hele kjøringen av programmet. Hvis en variabel for eksempel er definert å være av typen `int`, vil den alltid inneholde `int`-verdier. Dette kan kompilatoren bruke til å lage sjekke om programmereren har gjort noen feil; i tillegg vil det gi rask kode.

Noen språk har i stedet **dynamisk typing** som innebærer at variabler ikke er bundet til en spesiell type, men at typen koples til *verdien* som lagres i variabelen; dette innebærer at en variabel kan lagre verdier av først én type og siden en annen type når programmet utføres. Dette gir et mer fleksibelt språk på bekostning av sikkerhet og eksekveringshastighet.

Vårt språk Asp benytter dynamisk typing. I eksemplet i figur 2.31 på side 28 ser vi at variablene `v1` og `v2` ikke deklareres men «oppstår» når de tilordnes en verdi. Likeledes er parametrene `m` og `n` heller ikke deklarert med noen type, og det samme gjelder funksjonen `GCD`.

2.3.4 Indentering av koden

I Asp brukes ikke krøllparenteser til å angi innholdet i en funksjonsdeklarasjon eller en if- eller while-setning slik man gjør det i C, Java og flere andre språk. I stedet brukes innrykk for å angi hvor langt innholdet går. Det er det samme hvor langt man rykker inn; Asp vurderer bare om noen linjer er mer indentert enn andre.



Figur 2.30: Jernbanediagram for ⟨suite⟩

I figur 2.31 på side 28 er vist et eksempel i Asp sammen med det identiske programmet i Java.

2.3.5 Andre ting

2.3.5.1 Kommentarer

Kommentarer skrives slik:

```
# resten av linjen
```

	Resultat	Kommentar
+ float	float	Resultatet er v_1
+ int	int	Resultatet er v_1
- float	float	
- int	int	
float + float	float	
float + int	float	
int + float	float	
int + int	int	
string + string	string	Tekststrengene skjøtes
float - float	float	
float - int	float	
int - float	float	
int - int	int	
float * float	float	
float * int	float	
int * float	float	
int * int	int	
string * int	string	Sett sammen v_2 kopier av v_1
list * int	list	Sett sammen v_2 kopier av v_1
float / float	float	
float / int	float	
int / float	float	
int / int	float	
float // float	float	Beregnes med `Math.floor(v1/v2)`
float // int	float	Beregnes med `Math.floor(v1/v2)`
int // float	float	Beregnes med `Math.floor(v1/v2)`
int // int	int	Beregnes med `Math.floorDiv(v1,v2)`
float % float	float	Beregnes med `v1-v2*Math.floor(v1/v2)`
float % int	float	Beregnes med `v1-v2*Math.floor(v1/v2)`
int % float	float	Beregnes med `v1-v2*Math.floor(v1/v2)`
int % int	int	Beregnes med `Math.floorMod(v1,v2)`
float == float	bool	
float == int	bool	
int == float	bool	
int == int	bool	
string == string	bool	
none == *any*	bool	
any == none	bool	
float < float	bool	
float < int	bool	
int < float	bool	
int < int	bool	
string < string	bool	
not *any*	bool	Se tabell 2.2 på side 24 for logiske verdier
any and *any*	*any*	Oversettes som: `v1 ? v2 : v1`
any or *any*	*any*	Oversettes som: `v1 ? v1 : v2`

Tabell 2.3: Innebygde operatorer i Asp; disse er utelatt:
!= er som ==
<=, > og >= er som <
v_1 og v_2 er henholdsvis første og andre operand.

Asp

```
# A program to compute the greatest common divisor
# of two numbers, i.e., the biggest number by which
# two numbers can be divided without a remainder.

def GCD (m, n):
   if n == 0:
      return m
   else:
      return GCD(n, m % n)

v1 = int(input("A number: "))
v2 = int(input("Another number: "))

res = GCD(v1,v2)
print('GCD('+str(v1)+','+str(v2)+') =', res)
```

Java

```
// A program to compute the greatest common divisor
// of two numbers, i.e., the biggest number by which
// two numbers can be divided without a remainder.

import java.util.Scanner;

class GCD {
    static int gcd(int m, int n) {
        if (n == 0) {
            return m;
        } else {
            return gcd(n, m % n);
        }
    }

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("A number: ");
        int v1 = keyboard.nextInt();
        System.out.print("Another number: ");
        int v2 = keyboard.nextInt();

        int res = gcd(v1,v2);
        System.out.println("GCD("+v1+","+v2+") = "+res);
    }
}
```

Figur 2.31: Indentering i Asp kontra krøllparenteser i Java

2.3.5.2 Tabulering

TAB-er kan brukes til å angi innrykk, og de angir inntil 4 blanke i starten av linjene. [6]

2.4 Predefinerte deklarasjoner

Asp har et ganske lite bibliotek av predefinerte funksjoner i forhold til Python; se tabell 2.4.

Funksjon	Typekrav	Forklaring
`float(`v`)`	$v \in \{\text{int}, \text{float}, \text{string}\}$	Omformer v til en float
`input(`v`)`	$v \in \{\text{string}\}$	Skriver v og leser en linje fra tastaturet; resultatet er en string
`int(`v`)`	$v \in \{\text{int}, \text{float}, \text{string}\}$	Omformer v til en int
`len(`v`)`	$v \in \{\text{string}, \text{dict}, \text{list}\}$	Gir lengden av en string, en dict eller en list; resultatet er en int
`print(`$v_1, v_2, \ldots$`)`	$v_i \in \mathit{any}$	Skriver ut $v_1, v_2, \ldots$ med blank mellom; resultatet er none.
`range(`v_1`,`v_2`)`	$v_1, v_2 \in \{\text{int}\}$	Gir listen $[v_1, \ldots, v_2 - 1]$.
`str(`v`)`	$v \in \mathit{any}$	Omformer v til en string

Tabell 2.4: Asps bibliotek av predefinerte funksjoner

[6] **Tabulator** er en arv fra de gamle skrivemaskinene som hadde denne finessen for enkelt å skrive tabeller; derav navnet. Et tabulatortegn flytter posisjonen frem til neste faste posisjon; i Asp er posisjonene 4, 8, 12, ...; andre språk kan ha andre posisjoner.

2.4.1 Innlesning

I Asp benyttes den predefinerte funksjonen `input` til å lese det brukeren skriver på tastaturet. Parameteren skrives ut som et signal til brukeren om hva som forventes. Du kan se et eksempel på bruken av `input` i figur 2.31 på forrige side.

Hint Resultatet av `input` er alltid en tekst. Hvis man ønsker å lese et tall, må programmereren selv sørge for konvertering med `int` eller `float`.

2.4.2 Utskrift

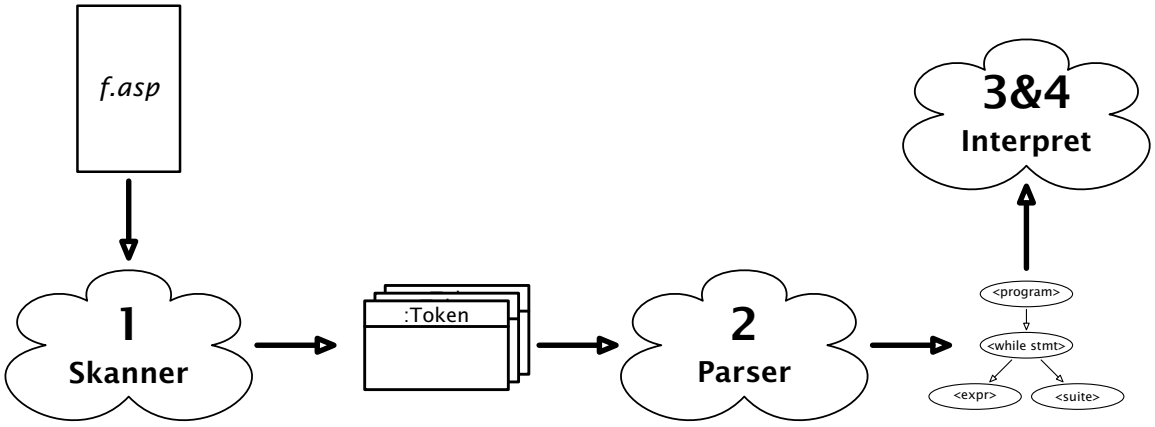
Asp-programmer kan skrive ut ved å benytte den helt spesielle funksjonen `print`. Denne prosedyren er den eneste som kan ha vilkårlig mange parametre. Alle parametrene skrives ut, og de kan være av vilkårlig type (unntatt func). Hvis det er mer enn én parameter, skrives det en blank mellom dem.

Hint Hvis man ønsker å skrive ut flere verdien *uten* den ekstra blanke, kan man selv konvertere alle parametrene til tekst og skjøte dem sammen før man kaller på `print`; se eksemplet i figur 2.31 på forrige side.

Kapittel 3

Prosjektet

Vårt prosjekt består av fire faser, som vist i figur 3.1. Hver av disse fire delene skal innleveres og godkjennes; se kursets nettside for frister.



Figur 3.1: Oversikt over prosjektet

3.1 Diverse informasjon om prosjektet

3.1.1 Basiskode

På emnets nettside ligger **`2100-oblig-2018.zip`** som er nyttig kode å starte med. Lag en egen mappe til prosjektet og legg ZIP-filen der. Gjør så dette:

```
$ cd mappen
$ unzip inf2100-oblig-2018.zip
$ ant
```

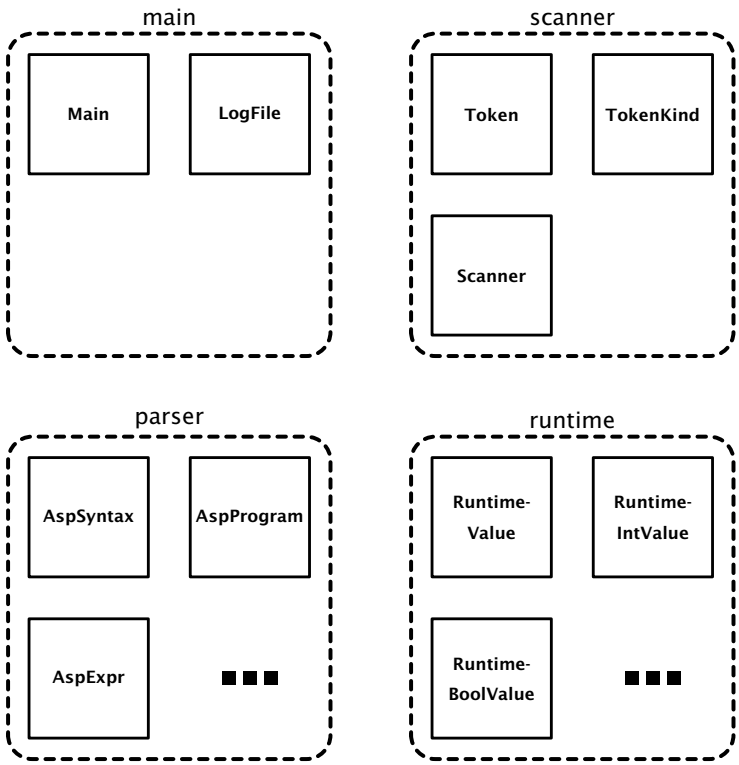
Dette vil resultere i en kjørbar fil `asp.jar` som kan kjøres slik

```
$ java -jar asp.jar minfil.asp
```

men den utleverte koden selvfølgelig ikke vil fungere! Den er bare en basis for å utvikle interpreten. Du kan endre basiskoden litt, men i det store og hele skal den virke likt.

3.1.2 Oppdeling i moduler

Alle større programmer bør deles opp i **moduler**, og i Java gjøres dette med package-mekanismen. Basiskoden er delt opp i fire moduler, som vist i figur 3.2.



Figur 3.2: De fire modulene i interpreten

`main` innholder to sentrale klasser som begge er ferdig programmert:

`Main` er «hovedprogrammet» som styrer hele interpreteringen.

`LogFile` brukes til å opprette en loggfil (se avsnitt 3.1.3).

`scanner` inneholder tre klasser som utgjør skanneren; se avsnitt 3.2 på side 34.

`parser` inneholder (når prosjektet er ferdig) rundt 35 klasser som brukes til å bygge parseringstreet; se avsnitt 3.3 på side 40.

`runtime` inneholder (etter at du har programmert ferdig) et dusin klasser som skal representere verdier av ulike typer under tolkingen av Asp-programmet.

3.1.3 Logging

Som en hjelp under arbeidet, og for enkelt å sjekke om de ulike delene virker, skal koden kunne håndtere loggutskriftene vist i tabell 3.1 på neste side.

3.1.4 Testprogrammer

Til hjelp under arbeidet finnes diverse testprogrammer:

Opsjon	Del	Hva logges
`-logE`	Del 3&4	Hvordan utførelsen av programmet går
`-logP`	Del 2	Hvilke parseringsmetoder som kalles
`-logS`	Del 1	Hvilke symboler som leses av skanneren
`-logY`	Del 2	Utskrift av parseringstreet

Tabell 3.1: Opsjoner for logging

- I mappen ~inf2100/oblig/test/ (som også er tilgjengelig fra en nettleser som http://inf2100.at.ifi.uio.no/oblig/test/) finnes noen Asp-programmer som bør fungere i den forstand at de produserer korrekt utskrift; resultatet av kjøringene er vist i `.out`-filene.
- I mappen ~inf2100/oblig/feil/ (som også er tilgjengelig utenfor Ifi som http://inf2100.at.ifi.uio.no/oblig/feil/) finnes diverse småprogrammer som alle inneholder en feil eller en raritet. Interpreten din bør håndetere disse programmene på samme måte som referanseinterpreten.

3.1.5 På egen datamaskin

Prosjektet er utviklet på Ifis Linux-maskiner, men det er også mulig å gjennomføre programmeringen på egen datamaskin, uansett om den kjører Linux, MacOS eller Windows. Det er imidlertid ditt ansvar at nødvendige verktøy fungerer skikkelig. Du trenger:

`ant` er en overbygning til Java-kompilatoren; den gjør det enkelt å kompilere et system med mange Java-filer. Programmet kan hentes ned fra http://ant.apache.org/bindownload.cgi.

`java` er en Java-interpret (ofte omtalt som «JVM» (Java virtual machine) eller «Java RTE» (Java runtime environment)). Om du installerer `javac` (se neste punkt), får du alltid med `java`.

`javac` er en Java-kompilator; du trenger *Java SE development kit* som kan hentes fra https://java.com/en/download/manual.jsp.

Et redigeringsprogram etter eget valg. Selv foretrekker jeg Emacs som kan hentes fra http://www.gnu.org/software/emacs/, men du kan bruke akkurat hvilket du vil.

3.1.6 Tegnsett

I dag er det spesielt tre tegnkodinger som er i vanlig bruk i Norge:

ISO 8859-1 (også kalt «Latin-1») er et tegnsett der hvert tegn lagres i én byte.

ISO 8859-15 (også kalt «Latin-9») er en lett modernisert variant av ISO 8859-1.

UTF-8 er en lagringsform for **Unicode**-kodingen og bruker 1–4 byte til hvert tegn.

Java-koden bør bare inneholde tegn fra Latin-9, men den ferdige interpreten skal lese og skrive UTF-8.

3.2 Del 1: Skanneren

Skanneren leser programteksten fra en fil og deler den opp i **symboler** (på engelsk «tokens»), omtrent slik vi mennesker leser en tekst ord for ord.

mini.asp

```
1
2 # En hyggelig hilsen
3 navn='Dag'
4 print ('Hei,',navn)
```

Figur 3.3: Et minimalt Asp-program `mini.asp`

Programmet vist i figur 3.3 inneholder for eksempel disse symbolene:

`navn`	`=`	`"Dag"`	`NEWLINE`	`print`	`(`
`"Hei,"`	`,`	`navn`	`)`	`NEWLINE`	`E-o-F`

Legg merke til at den blanke linjen og kommentaren er fjernet, og også all informasjon om blanke tegn mellom symbolene; kun selve symbolene er tilbake. Linjeskift for ikke-blanke linjer er imidlertid bevart siden de er av stor betydning for tolkningen av Asp-programmer. Det er også viktig å ha et symbol for å indikere er det er slutt på filen.[7]

NB! Det er viktig å huske at skanneren kun jobber med å finne symbolene i programkoden; den har ingen forståelse for hva som er et riktig eller fornuftig program. (Det kommer senere.)

3.2.1 Representasjon av symboler

Hvert symbol i Asp-programmet lagres i en instans av klassen `Token` vist i figur 3.4.

Token.java

```
8  public class Token {
9      public TokenKind kind;
10     public String name, stringLit;
11     public long integerLit;
12     public double floatLit;
13     public int lineNum;
           ⋮
61 }
```

Figur 3.4: Klassen `Token`

For hvert symbol må vi angi hva slags symbol det er, og dette angis med en `TokenKind`-referanse; se figur 3.5 på neste side. Legg spesielt merke til de fire siste symbolene som er spesielle for Asp.

[7] «E-o-F» er en vanlig forkortelse for «End of File».

Det er ikke noe i kildefilen som angir at det er slutt på den. Vi oppdager dette ved at metoden som skal lese neste linje, returnerer med en indikasjon på at det ikke var mer å lese.

TokenKind.java

```
5   public enum TokenKind {
6       // Names and literals:
7       nameToken("name"),
8       integerToken("integer literal"),
9       floatToken("float literal"),
10      stringToken("string literal"),
11
12      // Keywords:
13      andToken("and"),
14      asToken("as"),              // Not used in Asp
              ⋮
94      // Format tokens:
95      indentToken("INDENT"),
96      dedentToken("DEDENT"),
97      newLineToken("NEWLINE"),
98      eofToken("E-o-f");
99
100     String image;
101
102     TokenKind(String s) {
103         image = s;
104     }
105
106     public String toString() {
107         return image;
108     }
109 }
```

Figur 3.5: Enum-klassen TokenKind

3.2.2 Skanneren

Selve skanneren er definert av klassen Scanner; se figur 3.6.

Scanner.java

```
9   public class Scanner {
10      private LineNumberReader sourceFile = null;
11      private String curFileName;
12      private ArrayList<Token> curLineTokens = new ArrayList<>();
13      private int indents[] = new int[100];
14      private int numIndents = 0;
15      private final int tabDist = 4;
16
17
18      public Scanner(String fileName) {
19          curFileName = fileName;
20          indents[0] = 0;  numIndents = 1;
21
22          try {
23              sourceFile = new LineNumberReader(
24                              new InputStreamReader(
25                                  new FileInputStream(fileName),
26                                  "UTF-8"));
27          } catch (IOException e) {
28              scannerError("Cannot read " + fileName + "!");
29          }
30      }
              ⋮
321 }
```

Figur 3.6: Klassen Scanner

De viktigste metodene i Scanner er:

readNextLine leser neste linje av Asp-programmet, deler den opp i symbolene og legger alle symbolene i curLineTokens. (Denne metoden er privat og kalles bare fra curToken.)

`curToken` henter *nåværende symbol*; dette symbolet er alltid det første symbolet i `curLineTokens`. Symbolet blir ikke fjernet. Om nødvendig, kaller `curToken` på `readNextLine` for å få lest inn flere linjer.

`readNextToken` fjerner nåværende symbol som altså er første symbol i `curLineTokens`.

`anyEqualToken` sjekker om det finnes et ulest '='-symbol på denne linjen.

`curLineNum` gir nåværende linjes linjenummer.

`findIndent` teller antall blanke i starten av den nåværende linjen; se avsnitt 3.2.2.3 på neste side. (Denne metoden er privat og kalles bare fra `readNextLine`.)

`expandLeadingTabs` omformer innledende TAB-tegn til det riktige antall blanke; se avsnitt 3.2.2.2. (Denne metoden er privat og kalles bare fra `readNextLine`.)

3.2.2.1 Innrykk

Som del av arbeidet med å finne symbolene på en linje må man først

1) fjerne alle innledende TAB-er ved å omforme dem til det riktige antall blanke
2) beregne indenteringen av linjen

Dette er to separate operasjoner som ikke har noe direkte med hverandre å gjøre. Først gjøres den ene, og deretter gjøres den andre på resultatet av den første.

3.2.2.2 Omforme TAB-er til blanke

En kildekodelinje vil typisk starte med blanke og/eller TAB-tegn. For å kunne bestemme indenteringen av en linje (dvs hvor mange blanke den starter med), må vi derfor omforme alle de innledende TAB-ene til det riktige antall blanke etter følgende algoritme:

Omforming av TAB-er til blanke
For hver linje:
1) Sett en teller n til 0.
2) For hvert tegn i linjen:
   - Hvis tegnet er en blank, øk n med 1.
   - Hvis tegnet er en TAB, erstatt den med $4 - (n \bmod 4)$ blanke; øk n tilsvarende.
   - Ved alle andre tegn avsluttes denne løkken.

Hint Denne algoritmen finnes ferdig programmert som del av av basiskoden (se metoden `scanner.Scanner.expandLeadingTabs`), så du trenger ikke gjøre jobben en gang til.

Eksempel Hvis vi bruker en | til å vise starten på linjen og lar tegnet → bety en TAB og tegnet ␣ være en blank, vil omformingen av TAB-er til blanke ha denne effekten:

```
|a = 0          ⇒  |a = 0
|␣a = 0         ⇒  |␣a = 0
⊢→a = 0         ⇒  |␣␣␣␣a = 0
|␣→a = 0        ⇒  |␣␣␣␣a = 0
|␣␣→a = 0       ⇒  |␣␣␣␣a = 0
|␣␣␣→a = 0      ⇒  |␣␣␣␣a = 0
|␣␣␣␣→a = 0     ⇒  |␣␣␣␣␣␣␣␣a = 0
⊢→→a = 0        ⇒  |␣␣␣␣␣␣␣␣a = 0
⊢→→␣a = 0       ⇒  |␣␣␣␣␣␣␣␣␣a = 0
```

TAB-er senere på linjen Siden formålet med omformingen av TAB-ene er å beregne indenteringen, er det nok å fjerne dem fra de innledende delen av linjen. Når vi finner første ikke-blanke tegn på linjen, kan vi stoppe omforming av TAB-ene.

3.2.2.3 Beregne indentering

I Asp er indenteringen veldig viktig, så skanneren må til enhver tid vite om en linje er mer eller mindre indentert enn den foregående. (Vi er ikke interessert i om en indentering er stor eller liten, kun om den større eller mindre enn linjene før og etter.)

Eksempel

```
1 a␣=␣7␣␣␣␣␣␣␣␣␣␣#␣indent␣=␣0,␣nivå␣=␣0
2 if␣a␣==␣0:␣␣␣␣␣␣#␣indent␣=␣0,␣nivå␣=␣0
3 ␣␣b␣=␣-2␣␣␣␣␣␣␣#␣indent␣=␣2,␣nivå␣=␣1
4 ␣␣if␣a␣<␣b:␣␣␣␣␣#␣indent␣=␣2,␣nivå␣=␣1
5 ␣␣␣␣␣␣␣␣b␣=␣b+1␣#␣indent␣=␣8,␣nivå␣=␣2
6 ␣␣else:␣␣␣␣␣␣␣␣#␣indent␣=␣2,␣nivå␣=␣1
7 ␣␣␣b␣=␣b-1␣␣␣␣␣␣#␣indent␣=␣3,␣nivå␣=␣2
8 else:␣␣␣␣␣␣␣␣␣␣#␣indent␣=␣0,␣nivå␣=␣0
9 ␣␣␣␣␣␣␣␣b␣=␣0␣␣␣#␣indent␣=␣8,␣nivå␣=␣1
```

Denne koden inneholder følgende symboler. Hver gang vi går *opp* et indenteringsnivå, genererer vi et INDENT-symbol, og hver gang vi går *ned* et indenteringsnivå, genererer vi et DEDENT-symbol.

```
name:a = int:7 NEWLINE
if name:a == int:0 : NEWLINE
INDENT name:b = - int:2 NEWLINE
if name:a < name:b : NEWLINE
INDENT name:b = name:b + int:1 NEWLINE
DEDENT else : NEWLINE
INDENT name:b = name:b - int:1 NEWLINE
DEDENT DEDENT else : NEWLINE
INDENT name:b = int:0 NEWLINE
DEDENT E-o-F
```

Så, for å gjenta det viktigste:

- Hvis en linje er mer indentert enn den foregående, legger vi et 'INDENT'-symbol først i `curLineTokens`.

- Hvis en linje er mindre indentert enn den foregående, legger vi inn ett eller flere 'DEDENT'-symboler først i `curLineTokens`.

Figurene 3.24 og 3.25 viser dette tydelig.

Nøyaktig hvordan vi skal håndere indentering er gitt av følgende algoritme:

Håndtering av indentering

1) Opprett en stakk `Indents`; se linje 13 `Scanner.java` i figur 3.6 på side 35.
2) Push verdien 0 på `Indents`; se linje 20 i `Scanner.java` i figur 3.6 på side 35.
3) For hver linje:
   (a) Hvis linjen bare inneholder blanke (og eventuelt en kommentar), ignoreres den.
   (b) Omform alle innledende TAB-er til blanke ved å kalle på metoden `expandLeadingTabs`.
   (c) Tell antall innledende blanke `n` ved å kalle på metoden `findIndent`.
   (d) Hvis `n` > `Indents.top`:
      i. Push `n` på `Indents`.
      ii. Legg et 'INDENT'-symbol i `curLineTokens`.
   Ellers, så lenge `n` < `Indents.top`:
      i. Pop `Indents`.
      ii. Legg et 'DEDENT'-symbol i `curLineTokens`.
   Hvis nå `n`≠`Indents.top`, har vi indenteringsfeil.
4) Etter at siste linje er lest:
   For alle verdier på `Indents` som er > 0, legg et 'DEDENT'-symbol i `curLineTokens`.

Hint Hvis du lurer på nøyaktig hva som skjer i ulike situasjoner, skriv små testprogrammer og kjør dem i referanseinterpreten.

3.2.3 Logging

For å sjekke at skanningen fungerer rett, skal interpreten kunne kjøres med opsjonen `-testscanner`. Dette gir logging av to ting til loggfilen:

1) Hver gang `readNextLine` leser inn en ny linje, skal denne linjen logges ved å kalle på `Main.log.noteSource`.

2) Dessuten skal `readNextLine` logge alle symbolene som finnes på linjen ved å kalle på `Main.log.noteToken`.

(Sjekk kildekoden i `Scanner.java` for å se at dette stemmer.)

For å demonstrere hva som ønskes av testutskrift, har jeg laget både et minimalt og litt større Asp-program; se figur 3.7 på neste side og figur 3.23 på side 59. Når interpreten vår kjøres med opsjonen `-testscanner`, skriver den ut logginformasjonen vist i henholdsvis figur 3.7 på neste side og figur 3.24 til 3.25 på side 59–60.

3.2.4 Mål for del 1

Mål for del 1

Programmet skal utvikles slik at opsjonen `-testscanner` produserer loggfiler som vist i figurene 3.7 og 3.24–3.25.

mini.asp

```
1
2  # En hyggelig hilsen
3  navn='Dag'
4  print ('Hei,',navn)
```

```
1     1:
2     2: # En hyggelig hilsen
3     3: navn='Dag'
4  Scanner: name token on line 3: navn
5  Scanner: = token on line 3
6  Scanner: string literal token on line 3: "Dag"
7  Scanner: NEWLINE token on line 3
8     4: print ('Hei,',navn)
9  Scanner: name token on line 4: print
10 Scanner: ( token on line 4
11 Scanner: string literal token on line 4: "Hei,"
12 Scanner: , token on line 4
13 Scanner: name token on line 4: navn
14 Scanner: ) token on line 4
15 Scanner: NEWLINE token on line 4
16 Scanner: E-o-f token
```

Figur 3.7: Loggfil med symbolene skanneren finner i `mini.asp`

3.3 Del 2: Parsering

Denne delen går ut på å skrive en **parser**; en slik parser har to oppgaver:

- sjekke at programmet er korrekt i henhold til språkdefinisjonen (dvs grammatikken, ofte kalt syntaksen) og
- lage et tre som representerer programmet.

Testprogrammet `mini.asp` skal for eksempel gi treet vist i figur 3.8 på neste side.

3.3.1 Implementasjon

Aller først må det defineres en klasse per ikke-terminal («firkantene» i grammatikken), og alle disse må være subklasser av `AspSyntax`. Klassene må inneholde tilstrekkelige deklarasjoner til å kunne representere ikke-terminalen. Som et eksempel er vist klassen `AspAndTest` som representerer ⟨and test⟩; se figur 3.9 på side 42.

Et par ting verdt å merke seg:

- De fem ikke-terminalene ⟨letter A-Z,a-z⟩, ⟨digit 0-9⟩, ⟨digit 1-9⟩, ⟨char except '⟩ og ⟨char except "⟩ er allerede tatt hånd om av skanneren, så de kan vi se bort fra nå.
- Ikke-terminaler som kun er definert som et valg mellom ulike andre ikke-terminaler (som f eks ⟨stmt⟩ og ⟨atom⟩) bør implementeres som en abstrakt klasse, og så bør alternativene være sub-klasser av denne abstrakte klassen.

3.3.2 Parseringen

Den enkleste måte å parsere et Asp-program på er å benytte såkalt «**recursive descent**» og legge inn en metode

```
1  static Xxxx parse(Scanner s) {
2    ...
3  }
```

i alle sub-klassene av `AspSyntax`. Den skal parsere «seg selv» og lagre dette i et objekt; se for eksempel `AspAndTest.parse` i figur 3.9 på side 42. (Metodene `test` og `skip` er nyttige i denne sammenhengen; de er definert i `parser.AspSyntax`-klassen.)

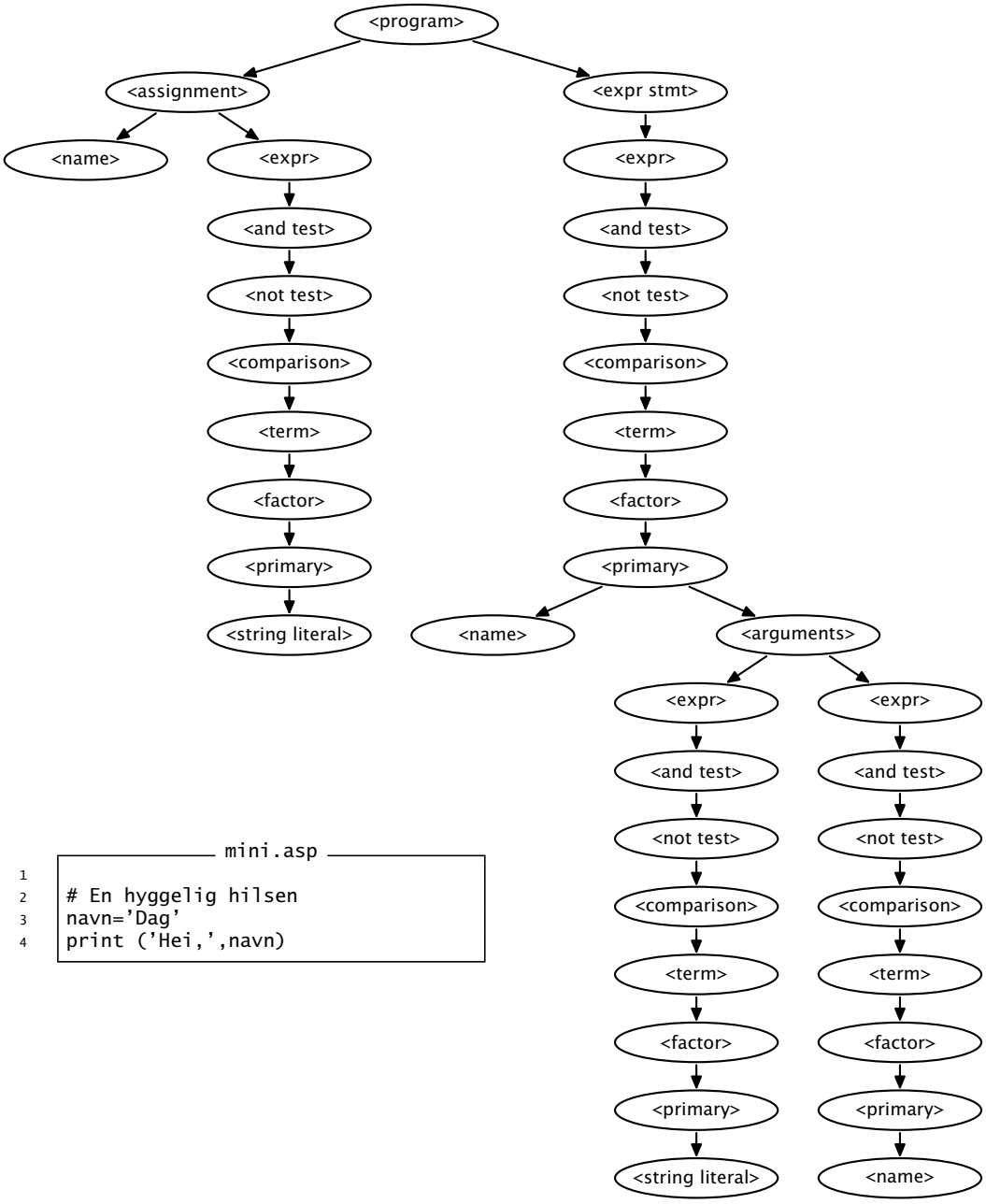
3.3.2.1 Tvetydigheter

Grammatikken til Asp er nesten alltid entydig utifra neste symbol, men ikke alltid. Setningen

```
v[4*(i+2)-1] = a
```

starter med et ⟨name⟩, så den kan være én av to:

1) ⟨assignment⟩
2) ⟨expr stmt⟩



mini.asp

```
1
2  # En hyggelig hilsen
3  navn='Dag'
4  print ('Hei,',navn)
```

Figur 3.8: Syntakstreet laget utifra testprogrammet `mini.asp`

AspAndTest.java

```
1  package no.uio.ifi.asp.parser;
2
3  import java.util.ArrayList;
4
5  import no.uio.ifi.asp.main.*;
6  import no.uio.ifi.asp.runtime.*;
7  import no.uio.ifi.asp.scanner.*;
8  import static no.uio.ifi.asp.scanner.TokenKind.*;
9
10 class AspAndTest extends AspSyntax {
11     ArrayList<AspNotTest> notTests = new ArrayList<>();
12
13     AspAndTest(int n) {
14         super(n);
15     }
16
17     static AspAndTest parse(Scanner s) {
18         enterParser("and test");
19
20         AspAndTest aat = new AspAndTest(s.curLineNum());
21         while (true) {
22             aat.notTests.add(AspNotTest.parse(s));
23             if (s.curToken().kind != andToken) break;
24             skip(s, andToken);
25         }
26
27         leaveParser("and test");
28         return aat;
29     }
30
31     @Override
32     void prettyPrint() {
33         int nPrinted = 0;
34
35         for (AspNotTest ant: notTests) {
36             if (nPrinted > 0)
37                 Main.log.prettyWrite(" and ");
38             ant.prettyPrint();  ++nPrinted;
39         }
40     }
          ⋮
52 }
```

Figur 3.9: Klassen `AspAndTest` (deler relevant for del 2)

For å avgjøre dette, må vi titte fremover på linjen for å se om det finnes en = der eller ikke.[8] Metoden `Scanner.anyEqualToken` er laget for dette formålet.

3.3.3 Syntaksfeil

Ved å benytte denne *recursive descent*-metoden for parsering er det enkelt å finne grammatikkfeil: Når det ikke finnes noe lovlig alternativ i jernbanediagrammene, har vi en feilsituasjon, og vi må kalle `AspSyntax.parserError`. (Metodene `AspSyntax.test` og `AspSyntax.skip` gjør dette automatisk for oss.)

[8] Slik titting forover er ikke helt politisk korrekt etter læreboken for *recursive descent*, men det er likevel ganske vanlig.

```
<program>
   1:
   2: # En hyggelig hilsen
   3: navn='Dag'
  <stmt>
    <assignment>
      <name>
      </name>
      <expr>
        <and test>
          <not test>
            <comparison>
              <term>
                <factor>
                  <primary>
                    <atom>
                      <string literal>
                      </string literal>
                    </atom>
                  </primary>
                </factor>
              </term>
            </comparison>
          </not test>
        </and test>
      </expr>
    </assignment>
  </stmt>
```

Figur 3.10: Loggfil som viser parsering av `mini.asp` (del 1)

3.3.4 Logging

For å sjekke at parseringen går slik den skal (og enda mer for å finne ut hvor langt prosessen er kommet om noe går galt), skal `parse`-metodene kalle på `enterParser` når de starter og så på `leaveParser` når de avslutter. Dette vil gi en oversiktlig opplisting av hvordan parseringen har forløpt.

Våre to vanlige testprogram vist i henholdsvis figur 3.3 på side 34 og figur 3.23 på side 59 vil produsere loggfilene i figurene 3.10–3.11 og figurene 3.26 til 3.32 på side 61 og etterfølgende; disse loggfilene lages når interpreten kjøres med opsjonen `-logP` eller `-testparser`.

3.3.5 Regenerering av programkoden

Logging er imidlertid ikke nok. Selv om parsering forløp feilfritt, kan det hende at parseringstreet ikke er riktig bygget opp. Den enkleste måten å sjekke dette på er å skrive ut det opprinnelige programmet basert på representasjonen i syntakstreet.[9] Dette ordnes best ved å legge inn en metode

```
  void prettyPrint() { ... }
```

i hver subklasse av `AspSyntax`. Resultatet kan sees i figur 3.12 på neste side; legg merke til at det er noen små forskjeller i forhold til originalen: noen blanke er satt inn, og det er brukt andre anførselstegn for tekstliteralene.

[9] En slik automatisk utskrift av et program kalles gjerne «pretty-printing» siden resultatet ofte blir penere enn det en travel programmerer tar seg tid til. Denne finessen var mye vanligere i tiden før man fikk interaktive datamaskiner og gode redigeringsprogrammer.

```
   4: print ('Hei,',navn)
  <stmt>
    <expr stmt>
      <expr>
        <and test>
          <not test>
            <comparison>
              <term>
                <factor>
                  <primary>
                    <atom>
                      <name>
                      </name>
                    </atom>
                    <primary suffix>
                      <arguments>
                        <expr>
                          <and test>
                            <not test>
                              <comparison>
                                <term>
                                  <factor>
                                    <primary>
                                      <atom>
                                        <string literal>
                                        </string literal>
                                      </atom>
                                    </primary>
                                  </factor>
                                </term>
                              </comparison>
                            </not test>
                          </and test>
                        </expr>
                        <expr>
                          <and test>
                            <not test>
                              <comparison>
                                <term>
                                  <factor>
                                    <primary>
                                      <atom>
                                        <name>
                                        </name>
                                      </atom>
                                    </primary>
                                  </factor>
                                </term>
                              </comparison>
                            </not test>
                          </and test>
                        </expr>
                      </arguments>
                    </primary suffix>
                  </primary>
                </factor>
              </term>
            </comparison>
          </not test>
        </and test>
      </expr>
    </expr stmt>
  </stmt>
</program>
```

Figur 3.11: Loggfil som viser parsering av `mini.asp` (del 2)

```
PP> navn = "Dag"
PP> print("Hei,", navn)
```

Figur 3.12: Loggfil med «skjønnskrift» av `mini.asp`

Mål for del 2

Programmet skal implementere parsering og også utskrift av det lagrede programmet; med andre ord skal opsjonen `-testparser` *gi utskrift som vist i figurene 3.10–3.12 og 3.26–3.33.*

3.4 Del 3: Evaluering av uttrykk

Den tredje delen av prosjektet er å implementere evaluering av uttrykk. Dette gjøres ved å utstyre klassene som implementerer uttrykk eller deluttrykk med en metode

```
1   RuntimeValue eval(RuntimeScope curScope) throws RuntimeReturnValue
```

(Klassen `RuntimeValue` forklares i avsnitt 3.4.1, klassen `RuntimeScope` i avsnitt 3.5.2.1 på side 55 og klassen `RuntimeReturnValue` i avsnitt 3.5.4.3 på side 57.)

Den komplette klassen `AspAndTest` med `eval`-metode er vist i figur 3.13 på neste side.

3.4.1 Verdier

Alle uttrykk skal gi en *verdi* som svar, og siden Asp har dynamisk typing og derfor skal sjekke typene under kjøring, må verdiene også inneholde en angivelse av verdiens type og dermed hvilke operasjoner som er lov for den. Dette løses enklest ved å benytte objektorientert programmering og la alle verdiene skal være av en subklasse av `runtime.RuntimeValue` som er vist i figur 3.14 på side 48, slik for eksempel `RuntimeBoolValue` som implementerer verdiene til `False` og `True` (se figur 3.15 på side 49).

3.4.2 Metoder

`RuntimeValue` og dens subklasser har tre ulike grupper metoder. Alle metodene defineres i den abstrakte klassen `RuntimeValue` som en feilsituasjon, og så kan de enkelte subklassene redefinere dem til å gjøre noe fornuftig om det er mulig.

3.4.2.1 Metoder med informasjon

Følgende virtuelle metoder gir informasjon om den aktuelle verdien:

`typeName` gir navnet på verdiens type.

`showInfo` gir verdien på en form som egner seg for intern bruk, dvs til feilsjekking og -utskrift.

`toString` gir verdien på en form som egner seg for utskrift under kjøring, for eksempel å skrive ut med `print`-funksjonen.

3.4.2.2 Metoder med Java-verdier

Disse metodene gir en Java-verdi som interpreten kan bruke til ulike formål; hvis den ønskete verdien er umulig å fremskaffe, skal interpreten stoppe med en feilmelding.

`getBoolValue` gir en boolsk verdi. (Vær oppmerksom på at svært mange typer verdier kan tolkes som en lovlig logisk verdi; se tabell 2.2 på side 24.)

AspAndTest.java

```
1  package no.uio.ifi.asp.parser;
2
3  import java.util.ArrayList;
4
5  import no.uio.ifi.asp.main.*;
6  import no.uio.ifi.asp.runtime.*;
7  import no.uio.ifi.asp.scanner.*;
8  import static no.uio.ifi.asp.scanner.TokenKind.*;
9
10 class AspAndTest extends AspSyntax {
11     ArrayList<AspNotTest> notTests = new ArrayList<>();
12
13     AspAndTest(int n) {
14         super(n);
15     }
16
17     static AspAndTest parse(Scanner s) {
18         enterParser("and test");
19
20         AspAndTest aat = new AspAndTest(s.curLineNum());
21         while (true) {
22             aat.notTests.add(AspNotTest.parse(s));
23             if (s.curToken().kind != andToken) break;
24             skip(s, andToken);
25         }
26
27         leaveParser("and test");
28         return aat;
29     }
30
31     @Override
32     void prettyPrint() {
33         int nPrinted = 0;
34
35         for (AspNotTest ant: notTests) {
36             if (nPrinted > 0)
37                 Main.log.prettyWrite(" and ");
38             ant.prettyPrint();  ++nPrinted;
39         }
40     }
41
42     @Override
43     RuntimeValue eval(RuntimeScope curScope) throws RuntimeReturnValue {
44         RuntimeValue v = notTests.get(0).eval(curScope);
45         for (int i = 1;  i < notTests.size();  ++i) {
46             if (! v.getBoolValue("and operand",this))
47                 return v;
48             v = notTests.get(i).eval(curScope);
49         }
50         return v;
51     }
52 }
```

Figur 3.13: Klassen AspAndTest

RuntimeValue.java

```
7
8   public abstract class RuntimeValue {
9       abstract protected String typeName();
10
11      public String showInfo() {
12          return toString();
13      }
14
15      // For parts 3 and 4:
16
17      public boolean getBoolValue(String what, AspSyntax where) {
18          runtimeError("Type error: "+what+" is not a Boolean!", where);
19          return false;  // Required by the compiler!
20      }
21
22      public double getFloatValue(String what, AspSyntax where) {
23          runtimeError("Type error: "+what+" is not a float!", where);
24          return 0.0;  // Required by the compiler!
25      }
26
27      public long getIntValue(String what, AspSyntax where) {
28          runtimeError("Type error: "+what+" is not an integer!", where);
29          return 0;  // Required by the compiler!
30      }
31
32      public String getStringValue(String what, AspSyntax where) {
33          runtimeError("Type error: "+what+" is not a text string!", where);
34          return null;  // Required by the compiler!
35      }
36
37      // For part 3:
38
39      public RuntimeValue evalAdd(RuntimeValue v, AspSyntax where) {
40          runtimeError("'+' undefined for "+typeName()+"!", where);
41          return null;  // Required by the compiler!
42      }
              ⋮
98
99      public RuntimeValue evalNot(AspSyntax where) {
100         runtimeError("'not' undefined for "+typeName()+"!", where);
101         return null;  // Required by the compiler!
102     }
              ⋮
123
124     // General:
125
126     public static void runtimeError(String message, int lNum) {
127         Main.error("Asp runtime error on line " + lNum + ": " + message);
128     }
129
130     public static void runtimeError(String message, AspSyntax where) {
131         runtimeError(message, where.lineNum);
132     }
133
134     // For part 4:
135
136     public void evalAssignElem(RuntimeValue inx, RuntimeValue val, AspSyntax where) {
137         runtimeError("assigning to an element not allowed for "+typeName()+"!", where);
138     }
139
140     public RuntimeValue evalFuncCall(ArrayList<RuntimeValue> actualParams,
141                                      AspSyntax where) {
142         runtimeError("'Function call (...)' undefined for "+typeName()+"!", where);
143         return null;  // Required by the compiler!
144     }
145 }
```

Figur 3.14: Klassen `RuntimeValue`

RuntimeBoolValue.java

```
1  package no.uio.ifi.asp.runtime;
2
3  import no.uio.ifi.asp.main.*;
4  import no.uio.ifi.asp.parser.AspSyntax;
5
6  public class RuntimeBoolValue extends RuntimeValue {
7      boolean boolValue;
8
9      public RuntimeBoolValue(boolean v) {
10         boolValue = v;
11     }
12
13
14     @Override
15     protected String typeName() {
16         return "boolean";
17     }
18
19
20     @Override
21     public String toString() {
22         return (boolValue ? "True" : "False");
23     }
24
25
26     @Override
27     public boolean getBoolValue(String what, AspSyntax where) {
28         return boolValue;
29     }
30
31
32     @Override
33     public RuntimeValue evalEqual(RuntimeValue v, AspSyntax where) {
34         if (v instanceof RuntimeNoneValue) {
35             return new RuntimeBoolValue(false);
36         }
37         runtimeError("Type error for ==.", where);
38         return null;  // Required by the compiler
39     }
40
41
42     @Override
43     public RuntimeValue evalNot(AspSyntax where) {
44         return new RuntimeBoolValue(! boolValue);
45     }
46
47
48     @Override
49     public RuntimeValue evalNotEqual(RuntimeValue v, AspSyntax where) {
50         if (v instanceof RuntimeNoneValue) {
51             return new RuntimeBoolValue(true);
52         }
53         runtimeError("Type error for !=.", where);
54         return null;  // Required by the compiler
55     }
56 }
```

Figur 3.15: Klassen RuntimeBoolValue

`getFloatValue` gir en flyt-tallsverdi som en `double`. (Husk at heltall automatisk konverteres til flyt-tall.)

`getIntValue` gir en heltallsverdi som en `long`.

`getStringValue` gir en tekststreng som en `String`.

3.4.2.3 Metoder for Asp-operatorer

Bruk av de aller fleste operatorene i Asp implementeres ved et kall på en egnet metode; for eksempel vil den binære operatoren + implementeres med `evalAdd` som finnes for heltall, flyt-tall og tekststrenger. En oversikt over alle operatorene og deres implementasjonsmetode er vist i tabell 3.2.

Operator	Implementasjon	Resultat
`a + b`	`evalAdd`	`a` + `b`
`a / b`	`evalDivide`	Flyttallsverdien `a` / `b`
`a == b`	`evalEqual`	Er `a` = `b`?
`a > b`	`evalGreater`	Er `a` > `b`?
`a >= b`	`evalGreaterEqual`	Er `a` ≥ `b`?
`a // b`	`evalIntDivide`	⌊`a` / `b`⌋
`a < b`	`evalLess`	Er `a` < `b`?
`a <= b`	`evalLessEqual`	Er `a` ≤ `b`?
`a % b`	`evalModulo`	`a` mod `b`
`a * b`	`evalMultiply`	`a` × `b`
`- a`	`evalNegate`	− `a`
`not a`	`evalNot`	¬ `a`
`a != b`	`evalNotEqual`	Er `a` ≠ `b`?
`+ a`	`evalPositive`	+ `a`
`a - b`	`evalSubtract`	`a` − `b`

Tabell 3.2: Asp-operatorer og deres implementasjonsmetode

Om operatoren ikke er lov for den aktuelle typen, skal interpreten gi en feilmelding og stoppe, og dette er allerede angitt som standardmetode i supertypen `RuntimeValue`.

Noen av disse metodene implementer Asp-konstruksjor som man vanligvis ikke tenker på som operatorer:

`evalLen` brukes av biblioteksfunksjonen `len`; se avsnitt 3.5.5 på side 58.

`evalSubscription` benyttes for å hente et element med angitt indeks fra en liste eller en ordbok.

`evalAssignElem` plasserer en verdi i en liste eller ordbok.

`evalFuncCall` utfører et kall på en funksjon.

3.4.3 Sporing av kjøringen

For enkelt å kunne sjekke at beregningen av uttrykk og deluttrykk skjer korrekt, er det vanlig å legge inn en mulighet for **sporing** (på engelsk

«**tracing**»). I vår interpret skjer dette ved å benytte opsjonen `-logT` eller `-testexpr`. Da vil `main.Main.doTestExpr` kalle på `log.traceEval` for hvert uttrykk den beregner.

3.4.4 Et eksempel

Når vi jobber med del 3, kan vi ikke teste på komplette Asp-programmer, kun på linjer med uttrykk uten variabler eller funksjoner, som vist i figurene 3.16 og 3.18.

mini-expr.asp

```
1
2  "Noen eksempler på <expr>:"
3  1 + 2
4  2 + 2 == 4
```

Figur 3.16: Noen enkle Asp-uttrykk

Når vi kjører slike filer med kommendoen «`java -jar asp.jar -testexpr mini-expr.asp`», blir resultatet som vist i figurene 3.17 og 3.19 på side 53.

```
1  PP> "Noen eksempler på <expr>:" ==>
2  Trace line 2: 'Noen eksempler på <expr>:'
3  PP> 1 + 2 ==>
4  Trace line 3: 3
5  PP> 2 + 2 == 4 ==>
6  Trace line 4: True
```

Figur 3.17: Sporingslogg fra kjøring av `mini-expr.asp`

Mål for del 3

Programmet skal implementere evaluering av uttrykk uten navn (dvs uten variabler og funksjoner); opsjonen `-testexpr` skal gi sporingsinformasjon som vist i figurene 3.17 og 3.19.

expressions.asp

```
"Testing integer expressions:"
42
-1017
-2 + 5 * 7
100 % (-2 % (+2 * 5) + 18)
1234567890 // 1000000

"Testing float expressions:"
42.0
-1017.1
-2 + 5.0 * 7
100 % (-2 % (+2 * 5.0) + 18)
1234567890.0 // 1000000.0

"Testing string expressions"
"Abc"
"x" * 10
"α" + "-"*5 + "ω"

"Testing boolean expressions"
not False
"To be" or "not to be"
"Yes" and 3.14
False or True or 144 or "?"

"Testing comparisons"
1 <= 2 <= 3
"a" != "b" > "b" != 99
3.14 > 2.718281828459045 > 0

"Testing lists"
[]
[22, "w", [-1,+1], 3.14159265]
[101,102,103][1]
[-1,0,1]*(2)
"Abcdef"[0]

"Testing dictionaries"
{"A": "a", "B": 1+2}
{"Ja": 1, "Nei": 0}["Ja"]
```

Figur 3.18: Noen litt mer avanserte Asp-uttrykk

```
PP> "Testing integer expressions:" ==>
Trace line 1: 'Testing integer expressions:'
PP> 42 ==>
Trace line 2: 42
PP> - 1017 ==>
Trace line 3: -1017
PP> - 2 + 5 * 7 ==>
Trace line 4: 33
PP> 100 % (- 2 % (+ 2 * 5) + 18) ==>
Trace line 5: 22
PP> 1234567890 // 1000000 ==>
Trace line 6: 1234
PP> "Testing float expressions:" ==>
Trace line 8: 'Testing float expressions:'
PP> 42.000000 ==>
Trace line 9: 42.0
PP> - 1017.100000 ==>
Trace line 10: -1017.1
PP> - 2 + 5.000000 * 7 ==>
Trace line 11: 33.0
PP> 100 % (- 2 % (+ 2 * 5.000000) + 18) ==>
Trace line 12: 22.0
PP> 1234567890.000000 // 1000000.000000 ==>
Trace line 13: 1234.0
PP> "Testing string expressions" ==>
Trace line 15: 'Testing string expressions'
PP> "Abc" ==>
Trace line 16: 'Abc'
PP> "x" * 10 ==>
Trace line 17: 'xxxxxxxxxx'
PP> "α" + "-" * 5 + "ω" ==>
Trace line 18: 'α-----ω'
PP> "Testing boolean expressions" ==>
Trace line 20: 'Testing boolean expressions'
PP> not False ==>
Trace line 21: True
PP> "To be" or "not to be" ==>
Trace line 22: 'To be'
PP> "Yes" and 3.140000 ==>
Trace line 23: 3.14
PP> False or True or 144 or "?" ==>
Trace line 24: True
PP> "Testing comparisons" ==>
Trace line 26: 'Testing comparisons'
PP> 1 <= 2 <= 3 ==>
Trace line 27: True
PP> "a" != "b" > "b" != 99 ==>
Trace line 28: False
PP> 3.140000 > 2.718282 > 0 ==>
Trace line 29: True
PP> "Testing lists" ==>
Trace line 31: 'Testing lists'
PP> [] ==>
Trace line 32: []
PP> [22, "w", [- 1, + 1], 3.141593] ==>
Trace line 33: [22, 'w', [-1, 1], 3.14159265]
PP> [101, 102, 103][1] ==>
Trace line 34: 102
PP> [- 1, 0, 1] * (2) ==>
Trace line 35: [-1, 0, 1, -1, 0, 1]
PP> "Abcdef"[0] ==>
Trace line 36: 'A'
PP> "Testing dictionaries" ==>
Trace line 38: 'Testing dictionaries'
PP> {"A":"a", "B":1 + 2} ==>
Trace line 39: {'A': 'a', 'B': 3}
PP> {"Ja":1, "Nei":0}["Ja"] ==>
Trace line 40: 1
```

Figur 3.19: Sporingslogg fra kjøring av `expressions.asp`

3.5 Del 4: Evaluering av setninger og funksjoner

Siste del av prosjektet er å legge inn det som mangler når det gjelder evaluering av Asp-programmer.

3.5.1 Setninger

Evaluering av setninger er rimelig rett frem å implementere.

3.5.1.1 Tilordningssetninger

Disse setningene er omtalt i avsnitt 3.5.3 på side 56.

3.5.1.2 Uttrykkssetninger

Disse setningene er bare uttrykk, og de ble implementert i del 3 av prosjektet.

3.5.1.3 For-setninger

Slike setninger evaluerer først kontrolluttrykket, som må være en liste. Deretter blir innmaten i løkken utført så mange ganger som listens lengden tilsier, og løkkevariabelen tilordnes riktig listelement for hvert gjennomløp.

3.5.1.4 If-setninger

Her må testuttrykkene evalueres etter tur til interpreten finner ett som er `True` (i henhold til tabell 2.2 på side 24) og så tolke det tilhørende alternativet.

3.5.1.5 While-setning

Dette er en løkkesetning, så interpreten må først beregne testuttrykket. Om det er `True` (i henhold til tabell 2.2 på side 24), utføres løkkeinnmaten før testuttrykket beregnes på nytt; først når testuttrykket beregnes til `False`, er while-setningen ferdig.

3.5.1.6 Return-setning

Denne setningen blir forklart i avsnitt 3.5.4.3 på side 57.

3.5.1.7 Pass-setning

Denne setningen gjør absolutt ingenting, så den er triviell å implementere.

3.5.1.8 Funksjonsdefinisjoner

Dette omtales i avsnitt 3.5.4 på side 56.

3.5.2 Variabler

I Asp deklareres ikke variabler; de oppstår automatisk første gang de tilordnes en verdi.

RuntimeScope.java

```
10 public class RuntimeScope {
11     private RuntimeScope outer;
12     private HashMap<String,RuntimeValue> decls = new HashMap<>();
13 
14     public RuntimeScope() {
15         outer = null;
16     }
17 
18 
19     public RuntimeScope(RuntimeScope oScope) {
20         outer = oScope;
21     }
22 
23 
24     public void assign(String id, RuntimeValue val) {
25         decls.put(id, val);
26     }
27 
28 
29     public RuntimeValue find(String id, AspSyntax where) {
30         RuntimeValue v = decls.get(id);
31         if (v != null)
32             return v;
33         if (outer != null)
34             return outer.find(id, where);
35 
36         RuntimeValue.runtimeError("Name " + id + " not defined!", where);
37         return null;  // Required by the compiler.
38     }
39 }
```

Figur 3.20: Klassen `RuntimeScope`

3.5.2.1 Skop

For å holde orden på hvilke variabler som er deklarert til enhver tid, benyttes objekter av klassen `runtime.RuntimeScope` som er vist i figur 3.20. De to viktigste metodene er:

`assign` tilordner (og oppretter, om den ikke finnes før) en verdi til en variabel.

`find` finner frem verdien tilordnet et gitt navn.

Når et Asp-program starter, finnes det to skop:

1) Biblioteket, som inneholder predefinerte funksjoner; se avsnitt 3.5.5 på side 58.

2) Hovedprogrammet, som initielt er tomt.

Siden vil hvert funksjonskall opprette et nytt skop for sine parametre og variabler; se avsnitt 3.5.4.2 på side 57.

Skopene ligger inni hverandre. Ytterst ligger biblioteket og innenfor det ligger hovedprogrammet. Elementet `outer` angir hvilket skop som ligger utenfor. Grunnen til dette er at man i et gitt skop kan referere til navn som er deklarert i eget skop men også i ytre skop. Se for eksempel på figur 2.1 på side 18:

- `n_printed` defineres i funksjonens skop i linje 27, så bruken i linje 31 refererer til denne lokale variabelen.

- `n` defineres i hovedprogrammets skop i linje 5, så bruken i linje 29 refererer dit.
- `str` er predefinert i biblioteket, så bruken i linje 17 angir den definisjonen.

Legg forøvrig merke til at definisjoner kan skygge for hverandre. I tillegg til den globale definisjonen av `n` i linje 5 er `n` også definert som parameter i linje 15. Bruken i linje 16 gjelder da definisjonen i det nærmeste skopet, med andre ord den i linje 15.[10]

3.5.3 Tilordning til variabler

Tilordning til enkle variabler er trivielt å implementere; det er bare å benytte metoden `assign` i det inneværende skopet, og det er gitt som parameter til `eval`-metoden.

3.5.3.1 Tilordning til mer kompliserte variabler

Som vist i definisjonen til ⟨assignment⟩ (se figur 2.4 på side 19), kan en venstreside i en tilordning være ganske sammesatt og involvere én eller flere indekser til lister og/eller ordbøker:

```
f[88]["Ja"][3] = True
```

Da må vi gjøre følgende:

1) For alle indekser unntatt den siste: Slå opp i listen eller ordboken på samme måte som da vi beregnet uttrykk.
2) Kall på siste verdi sin `evalAssignElem` for å foreta tilordningen.

3.5.4 Funksjoner

Funksjoner er det mest intrikate vi skal ta oss av i dette prosjektet, men om vi holder tungen rett i munnen, bør det gå rimelig greit. Det viktigste er å få en full forståelse over hva som skal skje før man begynner å skrive kode.

3.5.4.1 Definisjon av funksjoner

I Asp (som i Python) regnes en funksjonsdeklarasjon som en form for tilordning, så definisjonen av `GCD` i figur 3.23 på side 59 kan betraktes som noe à la

```
GCD = ⟨funksjonsverdi⟩
```

Klassen `RuntimeFunc` må du skrive selv, men den må være en subklasse av `RuntimeValue`.

[10] Kan vi i det hele tatt i `w4` få tilgang til den globale `n`. Svaret er nei.

3.5.4.2 Kall på funksjoner

Vi kan se at vi har et funksjonskall ved at det finnes et `AspArguments`-objekt i syntakstreet. Selve håndteringen av kallet kan enten skje der eller i `AspPrimary` om du foretrekker det. Uansett bør følgende skje:[11]

1) De aktuelle parametrene[12] beregnes og legges i en `ArrayList`.
2) Funksjonens `evalFuncCall` kalles med to parametre:
   (a) listen med aktuelle paramtre
   (b) kallets sted i syntakstreet (for å kunne gi korrekte feilmeldinger)
3) `RuntimeFunc.evalFuncCall` må så ta seg av kallet:
   (a) Sjekk at antallet aktuelle parametre er det samme som antallet formelle parametre.
   (b) Opprett et nytt `RuntimeScope`-objekt. Dette skopets `outer` skal være det skopet der funksjonen ble deklarert.
   (c) Gå gjennom parameterlisten og tilordne alle de aktuelle parameterverdiene til de formelle parametrene.
   (d) Kall funksjonens `runFunction` (med det nye skopet som parameter) slik at den kan utføre innmaten av funksjonen.

Og det er stort sett det som skal til.

3.5.4.3 `return`-setningen

Denne setningen skal avslutte kjøringen av den nåværende funksjonen, og den skal samtidig angi resultatverdien. Problemet er at return-setningen kan ligge inni en if-setning som er inni en while-setning som er inni ...

Dette ordnes enkelt med unntaksmekanismen i Java. Det eneste `return`-setningen behøver å gjøre, er å beregne resultatverdien og så `throw`-e en `RuntimeReturnValue`. Denne klassen er vist i figur 3.21.

RuntimeReturnValue.java

```
1  package no.uio.ifi.asp.runtime;
2
3  // For part 4:
4
5  public class RuntimeReturnValue extends Exception {
6      public int lineNum;
7      RuntimeValue value;
8
9      public RuntimeReturnValue(RuntimeValue v, int lNum) {
10         value = v;  lineNum = lNum;
11     }
12 }
```

Figur 3.21: Klassen `RuntimeReturnValue`

Hvis man nå sørger for at koden som kaller på `runFunction` `catch`-er `RuntimeReturnValue`, har man det som skal til.

[11] Beskrivelsen av funksjonskall er med vilje litt vag slik at du har mulighet til implementere dette på din egen måte.

[12] Betegnelsen **aktuell parameter** angir en parameter som står i funksjons*kallet*, mens **formell parameter** betegner en parameter som står i funksjons*definisjonen*.

3.5.5 Biblioteket

Som nevnt skal biblioteket eksistere når programutførelsen starter, så vi må opprette det. Det er rett og slett et `RuntimeScope`-objekt med de syv funksjonene fra tabell 2.4 på side 28. For hver av dem oppretter vi et objekt som er av en anonym subklasse av `RuntimeFunc` der `evalFuncCall` er byttet ut med en spesiallaget metode, for eksempel som vist i figur 3.22.

RuntimeLibrary.java

```
77          // len
78          assign("len", new RuntimeFunc("len") {
79                  @Override
80                  public RuntimeValue evalFuncCall(
81                          ArrayList<RuntimeValue> actualParams,
82                          AspSyntax where) {
83                      checkNumParams(actualParams, 1, "len", where);
84                      return actualParams.get(0).evalLen(where);
85                  }});
```

Figur 3.22: Fra klassen `RuntimeLibrary`

3.5.6 Sporing

Også setninger skal kunne spores ved at alle setningen kaller på `AspSyntax.trace` for å fortelle hva de gjør; se for eksempel figur 3.34 på side 68.

Mål for del 4

Programmet skal implementere resten av Asp slik at programmer som vist i figur 3.23 på neste side gir sporingsinformasjon som vist i figur 3.34 på side 68 når input er tallene 30 og 75.

3.6 Et litt større eksempel

gcd.asp

```
1
2   # A program to compute the greatest common divisor
3   # of two numbers, i.e., the biggest number by which
4   # two numbers can be divided without a remainder.
5
6   def GCD (m, n):
7      if n == 0:
8         return m
9      else:
10        return GCD(n, m % n)
11
12  v1 = int(input("A number: "))
13  v2 = int(input("Another number: "))
14
15  res = GCD(v1,v2)
16  print('GCD('+str(v1)+','+str(v2)+') =', res)
```

Figur 3.23: Et litt større Asp-program `gcd.asp`

```
1     1:
2     2: # A program to compute the greatest common divisor
3     3: # of two numbers, i.e., the biggest number by which
4     4: # two numbers can be divided without a remainder.
5     5:
6     6: def GCD (m, n):
7   Scanner: def token on line 6
8   Scanner: name token on line 6: GCD
9   Scanner: ( token on line 6
10  Scanner: name token on line 6: m
11  Scanner: , token on line 6
12  Scanner: name token on line 6: n
13  Scanner: ) token on line 6
14  Scanner: : token on line 6
15  Scanner: NEWLINE token on line 6
16     7:    if n == 0:
17  Scanner: INDENT token on line 7
18  Scanner: if token on line 7
19  Scanner: name token on line 7: n
20  Scanner: == token on line 7
21  Scanner: integer literal token on line 7: 0
22  Scanner: : token on line 7
23  Scanner: NEWLINE token on line 7
24     8:       return m
25  Scanner: INDENT token on line 8
26  Scanner: return token on line 8
27  Scanner: name token on line 8: m
28  Scanner: NEWLINE token on line 8
29     9:    else:
30  Scanner: DEDENT token on line 9
31  Scanner: else token on line 9
32  Scanner: : token on line 9
33  Scanner: NEWLINE token on line 9
```

Figur 3.24: Loggfil som demonstrerer hvilke symboler skanneren finner i `gcd.asp` (del 1)

```
34    10:       return GCD(n, m % n)
35  Scanner: INDENT token on line 10
36  Scanner: return token on line 10
37  Scanner: name token on line 10: GCD
38  Scanner: ( token on line 10
39  Scanner: name token on line 10: n
40  Scanner: , token on line 10
41  Scanner: name token on line 10: m
42  Scanner: % token on line 10
43  Scanner: name token on line 10: n
44  Scanner: ) token on line 10
45  Scanner: NEWLINE token on line 10
46    11:
47    12: v1 = int(input("A number: "))
48  Scanner: DEDENT token on line 12
49  Scanner: DEDENT token on line 12
50  Scanner: name token on line 12: v1
51  Scanner: = token on line 12
52  Scanner: name token on line 12: int
53  Scanner: ( token on line 12
54  Scanner: name token on line 12: input
55  Scanner: ( token on line 12
56  Scanner: string literal token on line 12: "A number: "
57  Scanner: ) token on line 12
58  Scanner: ) token on line 12
59  Scanner: NEWLINE token on line 12
60    13: v2 = int(input("Another number: "))
61  Scanner: name token on line 13: v2
62  Scanner: = token on line 13
63  Scanner: name token on line 13: int
64  Scanner: ( token on line 13
65  Scanner: name token on line 13: input
66  Scanner: ( token on line 13
67  Scanner: string literal token on line 13: "Another number: "
68  Scanner: ) token on line 13
69  Scanner: ) token on line 13
70  Scanner: NEWLINE token on line 13
71    14:
72    15: res = GCD(v1,v2)
73  Scanner: name token on line 15: res
74  Scanner: = token on line 15
75  Scanner: name token on line 15: GCD
76  Scanner: ( token on line 15
77  Scanner: name token on line 15: v1
78  Scanner: , token on line 15
79  Scanner: name token on line 15: v2
80  Scanner: ) token on line 15
81  Scanner: NEWLINE token on line 15
82    16: print('GCD('+str(v1)+','+str(v2)+') =', res)
83  Scanner: name token on line 16: print
84  Scanner: ( token on line 16
85  Scanner: string literal token on line 16: "GCD("
86  Scanner: + token on line 16
87  Scanner: name token on line 16: str
88  Scanner: ( token on line 16
89  Scanner: name token on line 16: v1
90  Scanner: ) token on line 16
91  Scanner: + token on line 16
92  Scanner: string literal token on line 16: ","
93  Scanner: + token on line 16
94  Scanner: name token on line 16: str
95  Scanner: ( token on line 16
96  Scanner: name token on line 16: v2
97  Scanner: ) token on line 16
98  Scanner: + token on line 16
99  Scanner: string literal token on line 16: ") ="
100 Scanner: , token on line 16
101 Scanner: name token on line 16: res
102 Scanner: ) token on line 16
103 Scanner: NEWLINE token on line 16
104 Scanner: E-o-f token
```

Figur 3.25: Loggfil som demonstrerer hvilke symboler skanneren finner i `gcd.asp` (del 2)

```
<program>
   1:
   2: # A program to compute the greatest common divisor
   3: # of two numbers, i.e., the biggest number by which
   4: # two numbers can be divided without a remainder.
   5:
   6: def GCD (m, n):
  <stmt>
    <func def>
      <name>
      </name>
      <name>
      </name>
      <name>
      </name>
      <suite>
   7:    if n == 0:
        <stmt>
          <if stmt>
            <expr>
              <and test>
                <not test>
                  <comparison>
                    <term>
                      <factor>
                        <primary>
                          <atom>
                            <name>
                            </name>
                          </atom>
                        </primary>
                      </factor>
                    </term>
                    <comp opr>
                    </comp opr>
                    <term>
                      <factor>
                        <primary>
                          <atom>
                            <integer literal>
                            </integer literal>
                          </atom>
                        </primary>
                      </factor>
                    </term>
                  </comparison>
                </not test>
              </and test>
            </expr>
            <suite>
   8:         return m
              <stmt>
                <return stmt>
                  <expr>
                    <and test>
                      <not test>
                        <comparison>
                          <term>
                            <factor>
                              <primary>
                                <atom>
                                  <name>
                                  </name>
                                </atom>
                              </primary>
                            </factor>
                          </term>
                        </comparison>
                      </not test>
                    </and test>
                  </expr>
                </return stmt>
              </stmt>
   9:    else:
            </suite>
```

Figur 3.26: Loggfil som viser parsering av `gcd.asp` (del 1)

```
76              <suite>
77    10:       return GCD(n, m % n)
78                <stmt>
79                  <return stmt>
80                    <expr>
81                      <and test>
82                        <not test>
83                          <comparison>
84                            <term>
85                              <factor>
86                                <primary>
87                                  <atom>
88                                    <name>
89                                    </name>
90                                  </atom>
91                                  <primary suffix>
92                                    <arguments>
93                                      <expr>
94                                        <and test>
95                                          <not test>
96                                            <comparison>
97                                              <term>
98                                                <factor>
99                                                  <primary>
100                                                   <atom>
101                                                     <name>
102                                                     </name>
103                                                   </atom>
104                                                 </primary>
105                                               </factor>
106                                             </term>
107                                           </comparison>
108                                         </not test>
109                                       </and test>
110                                     </expr>
111                                     <expr>
112                                       <and test>
113                                         <not test>
114                                           <comparison>
115                                             <term>
116                                               <factor>
117                                                 <primary>
118                                                   <atom>
119                                                     <name>
120                                                     </name>
121                                                   </atom>
122                                                 </primary>
123                                                 <factor opr>
124                                                 </factor opr>
125                                                 <primary>
126                                                   <atom>
127                                                     <name>
128                                                     </name>
129                                                   </atom>
130                                                 </primary>
131                                               </factor>
132                                             </term>
133                                           </comparison>
134                                         </not test>
135                                       </and test>
136                                     </expr>
137                                   </arguments>
138                                 </primary suffix>
139                               </primary>
140                             </factor>
141                           </term>
142                         </comparison>
143                       </not test>
144                     </and test>
145                   </expr>
146                 </return stmt>
147               </stmt>
148   11:
149   12: v1 = int(input("A number: "))
150             </suite>
```

Figur 3.27: Loggfil som viser parsering av `gcd.asp` (del 2)

```
151           </if stmt>
152         </stmt>
153       </suite>
154     </func def>
155   </stmt>
156   <stmt>
157     <assignment>
158       <name>
159       </name>
160       <expr>
161         <and test>
162           <not test>
163             <comparison>
164               <term>
165                 <factor>
166                   <primary>
167                     <atom>
168                       <name>
169                       </name>
170                     </atom>
171                     <primary suffix>
172                       <arguments>
173                         <expr>
174                           <and test>
175                             <not test>
176                               <comparison>
177                                 <term>
178                                   <factor>
179                                     <primary>
180                                       <atom>
181                                         <name>
182                                         </name>
183                                       </atom>
184                                       <primary suffix>
185                                         <arguments>
186                                           <expr>
187                                             <and test>
188                                               <not test>
189                                                 <comparison>
190                                                   <term>
191                                                     <factor>
192                                                       <primary>
193                                                         <atom>
194                                                           <string literal>
195                                                           </string literal>
196                                                         </atom>
197                                                       </primary>
198                                                     </factor>
199                                                   </term>
200                                                 </comparison>
201                                               </not test>
202                                             </and test>
203                                           </expr>
204                                         </arguments>
205                                       </primary suffix>
206                                     </primary>
207                                   </factor>
208                                 </term>
209                               </comparison>
210                             </not test>
211                           </and test>
212                         </expr>
213                       </arguments>
214                     </primary suffix>
215                   </primary>
216                 </factor>
217               </term>
218             </comparison>
219           </not test>
220         </and test>
221       </expr>
222     </assignment>
223   </stmt>
224   13: v2 = int(input("Another number: "))
225   <stmt>
```

Figur 3.28: Loggfil som viser parsering av `gcd.asp` (del 3)

```
226     <assignment>
227       <name>
228       </name>
229       <expr>
230         <and test>
231           <not test>
232             <comparison>
233               <term>
234                 <factor>
235                   <primary>
236                     <atom>
237                       <name>
238                       </name>
239                     </atom>
240                     <primary suffix>
241                       <arguments>
242                         <expr>
243                           <and test>
244                             <not test>
245                               <comparison>
246                                 <term>
247                                   <factor>
248                                     <primary>
249                                       <atom>
250                                         <name>
251                                         </name>
252                                       </atom>
253                                       <primary suffix>
254                                         <arguments>
255                                           <expr>
256                                             <and test>
257                                               <not test>
258                                                 <comparison>
259                                                   <term>
260                                                     <factor>
261                                                       <primary>
262                                                         <atom>
263                                                           <string literal>
264                                                           </string literal>
265                                                         </atom>
266                                                       </primary>
267                                                     </factor>
268                                                   </term>
269                                                 </comparison>
270                                               </not test>
271                                             </and test>
272                                           </expr>
273                                         </arguments>
274                                       </primary suffix>
275                                     </primary>
276                                   </factor>
277                                 </term>
278                               </comparison>
279                             </not test>
280                           </and test>
281                         </expr>
282                       </arguments>
283                     </primary suffix>
284                   </primary>
285                 </factor>
286               </term>
287             </comparison>
288           </not test>
289         </and test>
290       </expr>
291     </assignment>
292   </stmt>
293   14:
294   15: res = GCD(v1,v2)
295   <stmt>
296     <assignment>
297       <name>
298       </name>
299       <expr>
300         <and test>
```

Figur 3.29: Loggfil som viser parsering av `gcd.asp` (del 4)

```
301           <not test>
302             <comparison>
303               <term>
304                 <factor>
305                   <primary>
306                     <atom>
307                       <name>
308                       </name>
309                     </atom>
310                     <primary suffix>
311                       <arguments>
312                         <expr>
313                           <and test>
314                             <not test>
315                               <comparison>
316                                 <term>
317                                   <factor>
318                                     <primary>
319                                       <atom>
320                                         <name>
321                                         </name>
322                                       </atom>
323                                     </primary>
324                                   </factor>
325                                 </term>
326                               </comparison>
327                             </not test>
328                           </and test>
329                         </expr>
330                         <expr>
331                           <and test>
332                             <not test>
333                               <comparison>
334                                 <term>
335                                   <factor>
336                                     <primary>
337                                       <atom>
338                                         <name>
339                                         </name>
340                                       </atom>
341                                     </primary>
342                                   </factor>
343                                 </term>
344                               </comparison>
345                             </not test>
346                           </and test>
347                         </expr>
348                       </arguments>
349                     </primary suffix>
350                   </primary>
351                 </factor>
352               </term>
353             </comparison>
354           </not test>
355         </and test>
356       </expr>
357     </assignment>
358   </stmt>
359   16: print('GCD('+str(v1)+','+str(v2)+') =', res)
360   <stmt>
361     <expr stmt>
362       <expr>
363         <and test>
364           <not test>
365             <comparison>
366               <term>
367                 <factor>
368                   <primary>
369                     <atom>
370                       <name>
371                       </name>
372                     </atom>
373                     <primary suffix>
374                       <arguments>
375                         <expr>
```

Figur 3.30: Loggfil som viser parsering av `gcd.asp` (del 5)

```
376                           <and test>
377                             <not test>
378                               <comparison>
379                                 <term>
380                                   <factor>
381                                     <primary>
382                                       <atom>
383                                         <string literal>
384                                         </string literal>
385                                       </atom>
386                                     </primary>
387                                   </factor>
388                                   <term opr>
389                                   </term opr>
390                                   <factor>
391                                     <primary>
392                                       <atom>
393                                         <name>
394                                         </name>
395                                       </atom>
396                                       <primary suffix>
397                                         <arguments>
398                                           <expr>
399                                             <and test>
400                                               <not test>
401                                                 <comparison>
402                                                   <term>
403                                                     <factor>
404                                                       <primary>
405                                                         <atom>
406                                                           <name>
407                                                           </name>
408                                                         </atom>
409                                                       </primary>
410                                                     </factor>
411                                                   </term>
412                                                 </comparison>
413                                               </not test>
414                                             </and test>
415                                           </expr>
416                                         </arguments>
417                                       </primary suffix>
418                                     </primary>
419                                   </factor>
420                                   <term opr>
421                                   </term opr>
422                                   <factor>
423                                     <primary>
424                                       <atom>
425                                         <string literal>
426                                         </string literal>
427                                       </atom>
428                                     </primary>
429                                   </factor>
430                                   <term opr>
431                                   </term opr>
432                                   <factor>
433                                     <primary>
434                                       <atom>
435                                         <name>
436                                         </name>
437                                       </atom>
438                                       <primary suffix>
439                                         <arguments>
440                                           <expr>
441                                             <and test>
442                                               <not test>
443                                                 <comparison>
444                                                   <term>
445                                                     <factor>
446                                                       <primary>
447                                                         <atom>
448                                                           <name>
449                                                           </name>
450                                                         </atom>
```

Figur 3.31: Loggfil som viser parsering av `gcd.asp` (del 6)

```
                                                    </primary>
                                                  </factor>
                                                </term>
                                              </comparison>
                                            </not test>
                                          </and test>
                                        </expr>
                                      </arguments>
                                    </primary suffix>
                                  </primary>
                                </factor>
                                <term opr>
                                </term opr>
                                <factor>
                                  <primary>
                                    <atom>
                                      <string literal>
                                      </string literal>
                                    </atom>
                                  </primary>
                                </factor>
                              </term>
                            </comparison>
                          </not test>
                        </and test>
                      </expr>
                      <expr>
                        <and test>
                          <not test>
                            <comparison>
                              <term>
                                <factor>
                                  <primary>
                                    <atom>
                                      <name>
                                      </name>
                                    </atom>
                                  </primary>
                                </factor>
                              </term>
                            </comparison>
                          </not test>
                        </and test>
                      </expr>
                    </arguments>
                  </primary suffix>
                </primary>
              </factor>
            </term>
          </comparison>
        </not test>
      </and test>
    </expr>
  </expr stmt>
</stmt>
</program>
```

Figur 3.32: Loggfil som viser parsering av `gcd.asp` (del 7)

```
PP> def GCD (m, n):
PP>   if n == 0:
PP>     return m
PP>   else:
PP>     return GCD(n, m % n)
PP>
PP> v1 = int(input("A number: "))
PP> v2 = int(input("Another number: "))
PP> res = GCD(v1, v2)
PP> print("GCD(" + str(v1) + "," + str(v2) + ") =", res)
```

Figur 3.33: Loggfil med «skjønnskrift» av `gcd.asp`

```
1  Trace line 6: def GCD
2  Trace line 12: Call function input with params ['A number: ']
3  Trace line 12: Call function int with params ['30']
4  Trace line 12: v1 = 30
5  Trace line 13: Call function input with params ['Another number: ']
6  Trace line 13: Call function int with params ['75']
7  Trace line 13: v2 = 75
8  Trace line 15: Call function GCD with params [30, 75]
9  Trace line 7: else: ...
10 Trace line 10: Call function GCD with params [75, 30]
11 Trace line 7: else: ...
12 Trace line 10: Call function GCD with params [30, 15]
13 Trace line 7: else: ...
14 Trace line 10: Call function GCD with params [15, 0]
15 Trace line 7: if True alt #1: ...
16 Trace line 8: return 15
17 Trace line 10: return 15
18 Trace line 10: return 15
19 Trace line 10: return 15
20 Trace line 15: res = 15
21 Trace line 16: Call function str with params [30]
22 Trace line 16: Call function str with params [75]
23 Trace line 16: Call function print with params ['GCD(30,75) =', 15]
24 Trace line 16: None
```

Figur 3.34: Sporingslogg fra kjøring av `gcd.asp`

Kapittel 4

Programmeringsstil

4.1 Suns anbefalte Java-stil

Datafirmaet Sun, som utviklet Java, har også tanker om hvordan Java-koden bør se ut. Dette er uttrykt i et lite skriv på 24 sider som kan hentes fra http://java.sun.com/docs/codeconv/CodeConventions.pdf. Her er hovedpunktene.

4.1.1 Klasser

Hver klasse bør ligge i sin egen kildefil; unntatt er private klasser som «tilhører» en vanlig klasse.

Klasse-filer bør inneholde følgende (i denne rekkefølgen):

1) En kommentar med de aller viktigste opplysningene om filen:

```
/*
 * Klassens navn
 *
 * Versjonsinformasjon
 *
 * Copyrightangivelse
 */
```

2) Alle `import`-spesifikasjonene.

3) JavaDoc-kommentar for klassen. (JavaDoc er beskrevet i avsnitt 5.1 på side 73.)

4) Selve klassen.

4.1.2 Variabler

Variabler bør deklareres én og én på hver linje:

```
int level;
int size;
```

De bør komme først i {}-blokken (dvs før alle setningene), men lokale `for`-indekser er helt OK:

Type navn	Kapitalisering	Hva slags ord	Eksempel
Klasser	XxxxXxxx	Substantiv som beskriver objektene	IfStatement
Metoder	xxxxXxxx	Verb som angir hva metoden gjør	readToken
Variabler	xxxxXxxx	Korte substantiver; «bruk-og-kast-variabler» kan være på én bokstav	curToken, i
Konstanter	XXXX_XX	Substantiv	MAX_MEMORY

Tabell 4.1: Suns forslag til navnevalg i Java-programmer

```
1 for (int i = 1;  i <= 10;  ++i) {
2     ...
3 }
```

Om man kan initialisere variablene samtidig med deklarasjonen, er det en fordel.

4.1.3 Setninger

Enkle setninger bør stå én og én på hver linje:

```
1 i = 1;
2 j = 2;
```

De ulike sammensatte setningene skal se ut slik figur 4.1 på neste side viser. De skal alltid ha {} rundt innmaten, og innmaten skal indenteres 4 posisjoner.

4.1.4 Navn

Navn bør velges slik det er angitt i tabell 4.1.

4.1.5 Utseende

4.1.5.1 Linjelengde og linjedeling

Linjene bør ikke være mer enn 80 tegn lange, og kommentarer ikke lenger enn 70 tegn.

En linje som er for lang, bør deles

- etter et komma eller
- før en operator (som + eller &&).

Linjedelen etter delingspunktet bør indenteres likt med starten av uttrykket som ble delt.

```
do {
    setninger;
} while (uttrykk);

for (init;  betingelse;  oppdatering) {
    setninger;
}

if (uttrykk) {
    setninger;
}

if (uttrykk) {
    setninger;
} else {
    setninger;
}

if (uttrykk) {
    setninger;
} else if (uttrykk) {
    setninger;
} else if (uttrykk) {
    setninger;
}

return uttrykk;

switch (uttrykk) {
case xxx:
   setninger;
   break;

case xxx:
    setninger;
    break;

default:
    setninger;
    break;
}

try {
    setninger;
} catch (ExceptionClass e) {
    setninger;
}

while (uttrykk) {
    setninger;
}
```

Figur 4.1: Suns forslag til hvordan setninger bør skrives

4.1.5.2 Blanke linjer

Sett inn doble blanke linjer

- mellom klasser.

Sett inn enkle blanke linjer

- mellom metoder,
- mellom variabeldeklarasjonene og første setning i metoder eller
- mellom ulike deler av en metode.

4.1.5.3 Mellomrom

Sett inn mellomrom

- etter kommaer i parameterlister,
- rundt binære operatorer:

```
1 if (x < a + 1) {
```

(men ikke etter unære operatorer: `-a`)

- ved typekonvertering:

```
1 (int) x
```

Kapittel 5

Dokumentasjon

5.1 JavaDoc

Sun har også laget et opplegg for dokumentasjon av programmer. Hovedtankene er

1) Brukeren skriver kommentarer i hver Java-pakke, -klasse og -metode i henhold til visse regler.

2) Et eget program `javadoc` leser kodefilene og bygger opp et helt nett av HTML-filer med dokumentasjonen.

Et typisk eksempel på JavaDoc-dokumentasjon er den som beskriver Javas enorme bibliotek: http://java.sun.com/javase/7/docs/api/.

5.1.1 Hvordan skrive JavaDoc-kommentarer

Det er ikke vanskelig å skrive JavaDoc-kommentarer. Her er en kort innføring til hvordan det skal gjøres; den fulle beskrivelsen finnes på nettsiden http://java.sun.com/j2se/javadoc/writingdoccomments/.

En JavaDoc-kommentarer for en klasse ser slik ut:

```
1  /**
2   * Én setning som kort beskriver klassen
3   * Mer forklaring
4   *      ⋮
5   * @author  navn
6   * @author  navn
7   * @version dato
8   */
```

Legg spesielt merke til den doble stjernen på første linje – det er den som angir at dette er en JavaDoc-kommentar og ikke bare en vanlig kommentar.

JavaDoc-kommentarer for metoder følger nesten samme oppsettet:

```
1  /**
2   * Én setning som kort beskriver metoden
3   * Ytterligere kommentarer
4   *      ⋮
```

```
 * @param  navn1 Kort beskrivelse av parameteren
 * @param  navn2 Kort beskrivelse av parameteren
 * @return Kort beskrivelse av returverdien
 * @see    navn3
 */
```

Her er det viktig at den første setningen kort og presist forteller hva metoden gjør. Denne setningen vil bli brukt i metodeoversikten.

Ellers er verdt å merke seg at kommentaren skrives i HTML-kode, så man kan bruke konstruksjoner som `<i>...</i>` eller `<table>...</table>` om man ønsker det.

5.1.2 Eksempel

I figur 5.1 kan vi se en Java-metode med dokumentasjon.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Figur 5.1: Java-kode med JavaDoc-kommentarer

5.2 «Lesbar programmering»

Lesbar programmering («literate programming») er oppfunnet av Donald Knuth, forfatteren av *The art of computer programming* og opphavsmannen til TEX. Hovedtanken er at programmer først og fremst skal skrives slik at mennesker kan lese dem; datamaskiner klarer å «forstå» alt så lenge programmet er korrekt. Dette innebærer følgende:

- Programkoden og dokumentasjonen skrives som en enhet.
- Programmet deles opp i passende små navngitte enheter som legges inn i dokumentasjonen. Slike enheter kan referere til andre enheter.
- Programmet skrives i den rekkefølgen som er enklest for leseren å forstå.
- Dokumentasjonen skrives i et dokumentasjonsspråk (som LATEX) og kan benytte alle tilgjengelige typografiske hjelpemidler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

- Det kan automatisk lages oversikter og klasser, funksjoner og variabler: hvor de deklareres og hvor de brukes.

Ut ifra kildekoden («web-koden») kan man så lage

1) et dokument som kan skrives ut og
2) en kompilerbar kildekode.

5.2.1 Et eksempel

Som eksempel skal vi bruke en implementasjon av boblesortering. Fremgangsmåten er som følger:

1) Skriv kildefilen `bubble.w0` (vist i figur 5.2 og 5.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.
2) Bruk programmet `weave0`[13] til å lage det ferdige dokumentet som er vist i figur 5.4–5.7:

```
$ weave0 -l c -e -o bubble.tex bubble.w0
$ ltx2pdf bubble.tex
```

3) Bruk `tangle0` til å lage et kjørbart program:

```
$ tangle0 -o bubble.c bubble.w0
$ gcc -c bubble.c
```

[13] Dette eksemplet bruker Dags versjon av lesbar programmering kalt web_0; for mer informasjon, se http://dag.at.ifi.uio.no/public/doc/web0.pdf.

bubble.w0 del 1

```
\documentclass[12pt,a4paper]{webzero}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{amssymb,mathpazo,textcomp}

\title{Bubble sort}
\author{Dag Langmyhr\\ Department of Informatics\\
  University of Oslo\\[5pt] \texttt{dag@ifi.uio.no}}

\begin{document}
\maketitle

\noindent This short article describes \emph{bubble
  sort}, which quite probably is the easiest sorting
method to understand and implement.
Although far from being the most efficient one, it is
useful as an example when teaching sorting algorithms.

Let us write a function \texttt{bubble} in C which sorts
an array \texttt{a} with \texttt{n} elements. In other
words, the array \texttt{a} should satisfy the following
condition when \texttt{bubble} exits:
\[
  \forall i, j \in \mathbb{N}: 0 \leq i < j < \mathtt{n}
  \Rightarrow \mathtt{a}[i] \leq \mathtt{a}[j]
\]


<<bubble sort>>=
void bubble(int a[], int n)
{
  <<local variables>>

  <<use bubble sort>>
}
@
Bubble sorting is done by making several passes through
the array, each time letting the larger elements
``bubble'' up. This is repeated until the array is
completely sorted.

<<use bubble sort>>=
do {
  <<perform bubbling>>
} while (<<not sorted>>);
@
```

Figur 5.2: «Lesbar programmering» — kildefilen `bubble.w0` del 1

bubble.w0 del 2

```
47  Each pass through the array consists of looking at
48  every pair of adjacent elements;\footnote{We could, on the
49    average, double the execution speed of \texttt{bubble} by
50    reducing the range of the \texttt{for}-loop by~1 each time.
51    Since a simple implementation is the main issue, however,
52    this improvement was omitted.} if the two are in
53  the wrong sorting order, they are swapped:
54  <<perform bubbling>>=
55  <<initialize>>
56  for (i=0; i<n-1; ++i)
57    if (a[i]>a[i+1]) { <<swap a[i] and a[i+1]>> }
58  @
59  The \texttt{for}-loop needs an index variable
60  \texttt{i}:
61
62  <<local var...>>=
63  int i;
64  @
65  Swapping two array elements is done in the standard way
66  using an auxiliary variable \texttt{temp}. We also
67  increment a swap counter named \texttt{n\_swaps}.
68
69  <<swap ...>>=
70  temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;
71  ++n_swaps;
72  @
73  The variables \texttt{temp} and \texttt{n\_swaps}
74  must also be declared:
75
76  <<local var...>>=
77  int temp, n_swaps;
78  @
79  The variable \texttt{n\_swaps} counts the number of
80  swaps performed during one ``bubbling'' pass.
81  It must be initialized prior to each pass.
82
83  <<initialize>>=
84  n_swaps = 0;
85  @
86  If no swaps were made during the ``bubbling'' pass,
87  the array is sorted.
88
89  <<not sorted>>=
90  n_swaps > 0
91  @
92
93  \wzvarindex  \wzmetaindex
94  \end{document}
```

Figur 5.3: «Lesbar programmering» — kildefilen bubble.w0 del 2

Bubble sort

Dag Langmyhr
Department of Informatics
University of Oslo
dag@ifi.uio.no

August 17, 2018

This short article describes *bubble sort*, which quite probably is the easiest sorting method to understand and implement. Although far from being the most efficient one, it is useful as an example when teaching sorting algorithms.

Let us write a function `bubble` in C which sorts an array `a` with `n` elements. In other words, the array `a` should satisfy the following condition when `bubble` exits:

$$\forall i, j \in \mathbb{N} : 0 \le i < j < \mathtt{n} \Rightarrow \mathtt{a}[i] \le \mathtt{a}[j]$$

#1 ⟨*bubble sort*⟩ ≡

```
1  void bubble(int a[], int n)
2  {
3      ⟨local variables #4 (p.1)⟩
4
5      ⟨use bubble sort #2 (p.1)⟩
6  }
```

(This code is not used.)

Bubble sorting is done by making several passes through the array, each time letting the larger elements "bubble" up. This is repeated until the array is completely sorted.

#2 ⟨*use bubble sort*⟩ ≡

```
7  do {
8      ⟨perform bubbling #3 (p.1)⟩
9  } while (⟨not sorted #7 (p.2)⟩);
```

(This code is used in #1 (p.1).)

Each pass through the array consists of looking at every pair of adjacent elements;[1] if the two are in the wrong sorting order, they are swapped:

#3 ⟨*perform bubbling*⟩ ≡

```
10 ⟨initialize #6 (p.2)⟩
11 for (i=0; i<n-1; ++i)
12     if (a[i]>a[i+1]) { ⟨swap a[i] and a[i+1] #5 (p.2)⟩ }
```

(This code is used in #2 (p.1).)

The `for`-loop needs an index variable `i`:

#4 ⟨*local variables*⟩ ≡

```
13 int i;
```

(This code is extended in #4a (p.2). It is used in #1 (p.1).)

[1] We could, on the average, double the execution speed of `bubble` by reducing the range of the `for`-loop by 1 each time. Since a simple implementation is the main issue, however, this improvement was omitted.

File: *bubble.w0* page 1

Figur 5.4: «Lesbar programmering» — utskrift side 1

Swapping two array elements is done in the standard way using an auxiliary variable `temp`. We also increment a swap counter named `n_swaps`.

#5 ⟨*swap a[i] and a[i+1]*⟩ ≡

```
14  temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;
15  ++n_swaps;
```

(This code is used in #3 (p.1).)

The variables `temp` and `n_swaps` must also be declared:

#4a ⟨*local variables* #4 (p.1)⟩ +≡

```
16  int temp, n_swaps;
```

The variable `n_swaps` counts the number of swaps performed during one "bubbling" pass. It must be initialized prior to each pass.

#6 ⟨*initialize*⟩ ≡

```
17  n_swaps = 0;
```

(This code is used in #3 (p.1).)

If no swaps were made during the "bubbling" pass, the array is sorted.

#7 ⟨*not sorted*⟩ ≡

```
18  n_swaps > 0
```

(This code is used in #2 (p.1).)

File: *bubble.w0* page 2

Figur 5.5: «Lesbar programmering» — utskrift side 2

Variables

A

`a` 1, 12, 14

I

`i` 11, 12, 13, 14

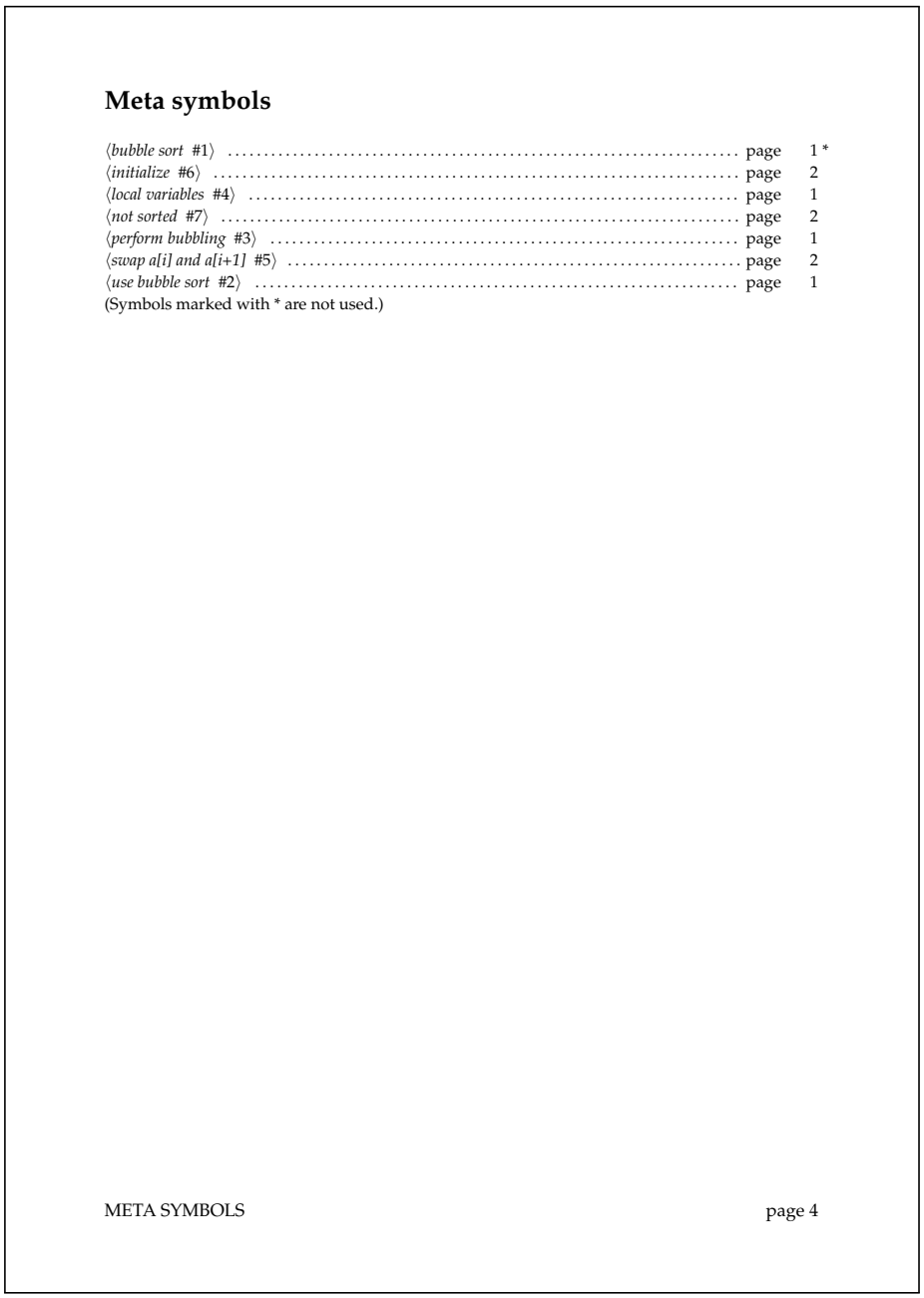
N

`n` 1, 11
`n_swaps` 15, 16, 17, 18

T

`temp` 14, 16

Figur 5.6: «Lesbar programmering» — utskrift side 3



Meta symbols

⟨*bubble sort* #1⟩ ..... page 1 *
⟨*initialize* #6⟩ ..... page 2
⟨*local variables* #4⟩ ..... page 1
⟨*not sorted* #7⟩ ..... page 2
⟨*perform bubbling* #3⟩ ..... page 1
⟨*swap a[i] and a[i+1]* #5⟩ ..... page 2
⟨*use bubble sort* #2⟩ ..... page 1
(Symbols marked with * are not used.)

META SYMBOLS page 4

Figur 5.7: «Lesbar programmering» — utskrift side 4

Register