

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

En interpret for Asp

Kompendium for IN2030

Stein Krogdahl(†), Dag Langmyhr
Høsten 2021



Innhold

Forord	9
1 Innledning	11
1.1 Hva er emnet IN2030?	11
1.2 Hvorfor lage en interpret?	12
1.3 Interpreter, kompilatorer og liknende	12
1.3.1 Interpreting	13
1.3.2 Kompilering	13
1.4 Oppgaven og dens fire deler	14
1.4.1 Del 1: Skanneren	14
1.4.2 Del 2: Parseren	15
1.4.3 Del 3: Interpreting av uttrykk	15
1.4.4 Del 4: Full interpreting	15
1.5 Krav til samarbeid og gruppetilhørighet	15
1.6 Kontroll av innlevert arbeid	15
1.7 Delta på øvingsgruppene	16
2 Programmering i Asp	17
2.1 Kjøring	17
2.2 Asp-program	19
2.2.1 Setninger	19
2.2.2 Uttrykk	21
2.3 Spesielle ting i Asp	24
2.3.1 Typer	25
2.3.2 Operatorer	26
2.3.3 Dynamisk typing	28
2.3.4 Indentering av koden	28
2.3.5 Andre ting	29
2.4 Predefinerte deklarasjoner	29
2.4.1 Innlesning	29
2.4.2 Utskrift	30
3 Prosjektet	31
3.1 Diverse informasjon om prosjektet	31
3.1.1 Basiskode	31
3.1.2 Oppdeling i moduler	32
3.1.3 Logging	32
3.1.4 Testprogrammer	33
3.1.5 På egen datamaskin	33
3.1.6 Tegnsett	34

3.1.7	Innlevering	34
3.2	Del 1: Skanneren	35
3.2.1	Representasjon av symboler	35
3.2.2	Skanneren	36
3.2.3	Logging	40
3.2.4	Mål for del 1	40
3.3	Del 2: Parsering	41
3.3.1	Implementasjon	41
3.3.2	Parseringen	41
3.3.3	Syntaksfeil	44
3.3.4	Logging	44
3.3.5	Regenerering av programkoden	44
3.4	Del 3: Evaluering av uttrykk	47
3.4.1	Verdier	47
3.4.2	Metoder	47
3.4.3	Sporing av kjøringen	53
3.4.4	Et eksempel	53
3.5	Del 4: Evaluering av setninger og funksjoner	56
3.5.1	Setninger	56
3.5.2	Variabler	57
3.5.3	Tilordning til variabler	59
3.5.4	Funksjoner	59
3.5.5	Biblioteket	61
3.5.6	Sporing	61
3.6	Et litt større eksempel	62
4	Programmeringsstil	73
4.1	Suns anbefalte Java-stil	73
4.1.1	Klasser	73
4.1.2	Variabler	73
4.1.3	Setninger	74
4.1.4	Navn	74
4.1.5	Utseende	74
5	Dokumentasjon	77
5.1	JavaDoc	77
5.1.1	Hvordan skrive JavaDoc-kommentarer	77
5.1.2	Eksempel	78
5.2	«Lesbar programmering»	78
5.2.1	Et eksempel	79
Register		87

Figurer

2.1	Eksempel på et Asp-program	18
2.2	Jernbanediagram for <code>{program}</code>	19
2.3	Jernbanediagram for <code>{stmt}</code>	19
2.4	Jernbanediagram for <code>{small stmt list}</code> og <code>{small stmt}</code>	19
2.5	Jernbanediagram for <code>{assignment}</code>	19
2.6	Jernbanediagram for <code>{expr stmt}</code> og <code>{return stmt}</code>	20
2.7	Jernbanediagram for <code>{pass stmt}</code> og <code>{global stmt}</code>	20
2.8	Jernbanediagram for <code>{compound stmt}</code>	20
2.9	Jernbanediagram for <code>{if stmt}</code>	20
2.10	Jernbanediagram for <code>{suite}</code>	20
2.11	Jernbanediagram for <code>{for stmt}</code>	21
2.12	Jernbanediagram for <code>{while stmt}</code>	21
2.13	Jernbanediagram for <code>{func def}</code>	21
2.14	Jernbanediagram for <code>{expr}</code>	21
2.15	Jernbanediagram for <code>{and test}</code> og <code>{not test}</code>	21
2.16	Jernbanediagram for <code>{comparison}</code> og <code>{comp opr}</code>	22
2.17	Jernbanediagram for <code>{term}</code> og <code>{term opr}</code>	22
2.18	Jernbanediagram for <code>{factor}</code> og <code>{factor prefix}</code>	22
2.19	Jernbanediagram for <code>{factor opr}</code>	22
2.20	Jernbanediagram for <code>{primary}</code> og <code>{primary suffix}</code>	22
2.21	Jernbanediagram for <code>{atom}</code> og <code>{inner expr}</code>	23
2.22	Jernbanediagram for <code>{list display}</code>	23
2.23	Jernbanediagram for <code>{dict display}</code>	23
2.24	Jernbanediagram for <code>{arguments}</code> og <code>{subscription}</code>	23
2.25	Jernbanediagram for <code>{integer literal}</code>	24
2.26	Jernbanediagram for <code>{float literal}</code>	24
2.27	Jernbanediagram for <code>{string literal}</code>	24
2.28	Jernbanediagram for <code>{boolean literal}</code> og <code>{none literal}</code>	24
2.29	Jernbanediagram for <code>{name}</code>	24
2.30	Indentering i Asp kontra krøllparenteser i Java	29
3.1	Oversikt over prosjektet	31
3.2	De fire modulene i interpreten	32
3.3	Et minimalt Asp-program <code>mini.asp</code>	35
3.4	Klassen <code>Token</code>	35
3.5	Enum-klassen <code>TokenKind</code>	36
3.6	Klassen <code>Scanner</code>	37
3.7	Algoritme for omforming av TAB-er til blanke	38
3.8	Eksempel på ekspansjon av TAB-er	38

3.9	Algoritme for håndtering av indentering	39
3.10	Skanning av <code>mini.asp</code>	40
3.11	Syntakstreet laget utifra det lille testprogrammet <code>mini.asp</code>	42
3.12	Klassen <code>AspAndTest</code>	43
3.13	Parsering av <code>mini.asp</code> (del 1)	45
3.14	Parsering av <code>mini.asp</code> (del 2)	46
3.15	Utskrift av treet til <code>mini.asp</code>	46
3.16	Klassen <code>AspAndTest</code>	48
3.17	Klassen <code>RuntimeValue</code>	49
3.18	Klassen <code>RuntimeBoolValue</code>	50
3.19	Noen enkle Asp-uttrykk	53
3.20	Sporingslogg fra kjøring av <code>mini-expr.asp</code>	53
3.21	Noen litt mer avanserte Asp-uttrykk	54
3.22	Sporingslogg fra kjøring av <code>expressions.asp</code>	55
3.23	Klassen <code>RuntimeScope</code>	57
3.24	Eksempel på bruk av skop	58
3.25	Demonstrasjon av <code>global</code>	59
3.26	Klassen <code>RuntimeReturnValue</code>	61
3.27	Fra klassen <code>RuntimeLibrary</code>	61
3.28	Et litt større Asp-program <code>palindrom.asp</code>	62
3.29	Skanning av <code>palindrom.asp</code> (del 1)	62
3.30	Skanning av <code>palindrom.asp</code> (del 2)	63
3.31	Parsering av <code>palindrom.asp</code> (del 1)	64
3.32	Parsering av <code>palindrom.asp</code> (del 2)	65
3.33	Parsering av <code>palindrom.asp</code> (del 3)	66
3.34	Parsering av <code>palindrom.asp</code> (del 4)	67
3.35	Parsering av <code>palindrom.asp</code> (del 5)	68
3.36	Parsering av <code>palindrom.asp</code> (del 6)	69
3.37	Parsering av <code>palindrom.asp</code> (del 7)	70
3.38	Utskrift av syntakstreet til <code>palindrom.asp</code>	71
3.39	Sporingslogg fra kjøring av <code>palindrom.asp</code>	71
4.1	Suns forslag til hvordan setninger bør skrives	75
5.1	Java-kode med JavaDoc-kommentarer	78
5.2	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 1	80
5.3	«Lesbar programmering» – kildefilen <code>bubble.w0</code> del 2	81
5.4	«Lesbar programmering» – utskrift side 1	82
5.5	«Lesbar programmering» – utskrift side 2	83
5.6	«Lesbar programmering» – utskrift side 3	84
5.7	«Lesbar programmering» – utskrift side 4	85

Tabeller

2.1	Typer i Asp	25
2.2	Lovlige logiske verdier i Asp	25
2.3	Innebygde operatorer i Asp	27
2.4	Asps bibliotek av predefinerte funksjoner	30
3.1	Opsjoner for logging	33
3.2	Asp-operatorer og deres implementasjonsmetode	51
4.1	Suns forslag til navnevalg i Java-programmer	74

Forord

Dette kompendiet er laget for emnet *IN2030 – Prosjektoppgave i programmering*. Selve kurset er et av de eldste ved Ifi, og nummeret har skiftet flere ganger, men innholdet i kurset har alltid vært variasjoner over samme tema.

Det opprinnelige kurset ble utviklet av *Stein Krogdahl* (1945–2021) 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 renoveret av *Dag Langmyhr*: *Minila* ble erstattet av en minimal variant av C kalt *RusC* og datamaskinen *Flink* ble avløst av en annen ikkeksisterende 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 gikk over til å 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 årene 2018–2021 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.

FORORD

Forfatterne vil ellers takke studentene *David Daja Andersen, Einar Løvhøi-den Antonsen, Jonny Bekkevold, Eivind Alexander Bergem, Marius Ekeberg, Henning Grande, Asbjørn Gaarde, Arne Olav Hallingstad, Espen Tørressen Hangård, Robin Hansen, Sigmund Hansen, Markus Hauge, Simen Heggestøyl, Celina Jakobsen, Robin Espinosa Jelle, Simen Jensen, Thor Joramo, Per Magne Kirkhus, 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, Lars Christian Hovtun Palm, Christian Resell, Christian Andre Finnøy Ruud, Ryhor Sivuda, Sarek Høverstad Skotåm, Yrjab Skrimstad, Gaute Solheim, Mika Sundland, 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, 18. juli 2021
Stein Krogdahl(+) Dag Langmyhr

Teori er når ingenting virker og alle vet hvorfor. Praksis er når altting 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 IN2030?

Emnet IN2030 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 IN2030, 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 programmeringsoppgaven 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 oppsplittning 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 IN2030, 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 interepeter og kompilatorer, er at det er høyst upraktisk å bygge datamaskiner slik at de direkte utfra sin elektroteknikk 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 Interpreting

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 minnet 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 og andre) innebygget en slik JVM-interpret for å kunne utføre Java-applets når de får disse (ferdig kompilert) over nettet.

Slik interperetering 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 interperitere 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 annonseret 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 interpereten 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: Interpreting 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 interpreting

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». Slik 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 gruppetime, 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 tjjeneste. Og om man synes man har en aha-opplevelse, så er 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 2.29 på side 19–24 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 ukedag.asp
This is the IN2030 Asp interpreter (2021-07-18)
Oppgi et år: 2021
Oppgi en måned: desember
Oppgi en dag: 24
24. desember 2021 er en fredag
```

Denne finnes på Ifis Linux-maskiner, men dere kan også hente JAR-filen `~inf2100/asp.jar` eller <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/nedlasting/asp.jar> og kjøre interpreten på private maskiner. Da startes interpreten slik:

```
$ java -jar asp.jar ukedag.asp
```

¹ Du finner kildekoden til dette programmet og også andre nyttige testprogrammer i mappen `~inf2100/oblig/test/` på alle Ifis Linux-maskiner; mappen er også tilgjengelig fra en vilkårlig nettleser som <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/test>.

```

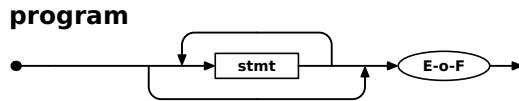
1      _____ ukedag.asp
2  # Program som ber brukeren oppgi en dato og
3  # skriver ut hvilken ukedag den faller på.
4
5  ukedag = ["man", "tirs", "ons", "tors", "fre", "lør", "søn"]
6  m_leng = [0, 31, 28, 31, 30, 31, 31, 30, 31, 30, 31]
7
8  m_navn = [0]*(12+1)
9  m_navn[ 1] = "januar"; m_navn[ 2] = "februar"; m_navn[ 3] = "mars";
10 m_navn[ 4] = "april";   m_navn[ 5] = "mai";     m_navn[ 6] = "juni";
11 m_navn[ 7] = "juli";    m_navn[ 8] = "august";  m_navn[ 9] = "september";
12 m_navn[10] = "oktober"; m_navn[11] = "november"; m_navn[12] = "desember";
13
14 # Gitt et månedsnavn, finn månedens nummer.
15 # Svar 0 om det ikke er et lovlig månedsnavn.
16 def finn_maaned (m_id):
17     for m_ix in range(1,len(m_navn)):
18         if m_navn[m_ix] == m_id: return m_ix
19     return 0
20
21 # Les et månedsnavn; fortsett til lovlig navn.
22 def les_maaned () :
23     while True:
24         m_id = input("Oppgi en måned: ")
25         m_num = finn_maaned(m_id)
26         if m_num > 0: return m_num
27         print("Ulovlig måned!")
28
29 # Er angitte år et skuddår?
30 def er_skuddaar (aa):
31     return aa%4==0 and aa%100 or aa%400==0
32
33 # Beregn antall dager i en gitt måned i et gitt år.
34 def finn_m_leng (m, aa):
35     if m==2 and er_skuddaar(aa): return 29
36     return m_leng[m]
37
38 # Beregn antall dager fra 1. januar 1900 til angitt dato:
39 def finn_dag_nr (aa, m, d):
40     d_nr = d
41
42 # Tell dagene i årene før angitte dato:
43 for aax in range(1900,aa):
44     d_nr = d_nr + 365;
45     if er_skuddaar(aax): d_nr = d_nr + 1
46
47 # Tell dagene i månedene før:
48 for mx in range(1,m):
49     d_nr = d_nr + finn_m_leng(mx,aa)
50
51 return d_nr
52
53 # Hovedprogrammet:
54 aar = int(input("Oppgi et år: "))
55 maaned = les_maaned()
56 dag = int(input("Oppgi en dag: "))
57 u_dag = ukedag[(6+finn_dag_nr(aar,maaned,dag))%7] + "dag"
58 print(str(dag)+".", m_navn[maaned], aar, "er en", u_dag)

```

Figur 2.1: Eksempel på et Asp-program

2.2 Asp-program

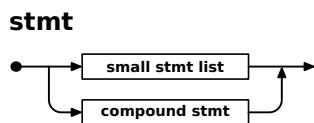
Som vist i figur 2.2, består et Asp-program av en sekvens av setninger (`{stmt}`). Tomme linjer er også lov. Symbolet `E-o-f` angir slutt på filen («end of file»).



Figur 2.2: Jernbanediagram for `{program}`

2.2.1 Setninger

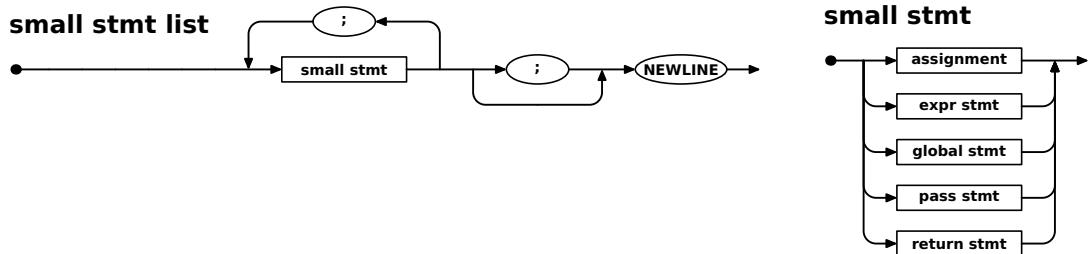
Figur 2.3 viser hva slags setninger man kan bruke i Asp.



Figur 2.3: Jernbanediagram for `{stmt}`

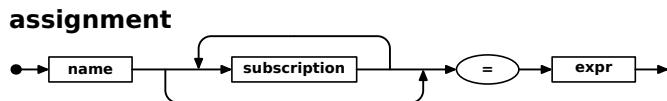
2.2.1.1 Enkle setninger

Enkle setninger er gjerne korte, og da er det lov å ha flere på samme linje med semikolon mellom.



Figur 2.4: Jernbanediagram for `{small stmt list}` og `{small stmt}`

Tilordning Som i de fleste andre språk, brukes en tilordningssetning til å gi variabler en verdi. Siden Asp har *dynamisk typing*, skal ikke variablene deklarereres på forhånd. Les mer om dette i avsnitt 2.3.3 på side 28.



Figur 2.5: Jernbanediagram for `{assignment}`

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

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



Figur 2.6: Jernbanediagram for {expr stmt} og {return stmt}

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



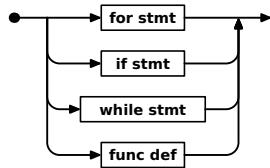
Figur 2.7: Jernbanediagram for {pass stmt} og {global stmt}

Global-setninger Med denne setningen kan man angi at noen variabler skal være globale. Det står mer om dette i avsnitt 3.5.2.2 på side 58.

2.2.1.2 Sammensatte setninger

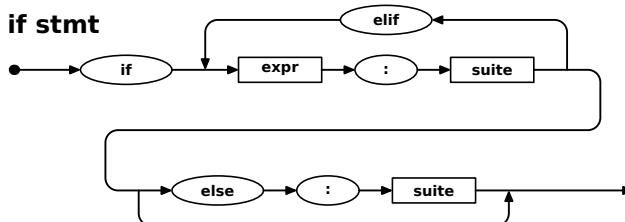
Sammensatte setninger inneholder andre setninger.

compound stmt



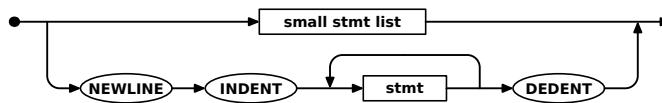
Figur 2.8: Jernbanediagram for {compound stmt}

If-setninger If-setninger brukes til å velge om setninger skal utføres eller ikke. Se forøvrig avsnitt 2.3.1.1 på side 25 for hva som er lovlige testverdier.



Figur 2.9: Jernbanediagram for {if stmt}

suite



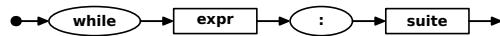
Figur 2.10: Jernbanediagram for {suite}

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

for stmt

Figur 2.11: Jernbanediagram for {for stmt}

While-setninger While-setninger er en annen form for løkkesetning i Asp. Se forøvrig avsnitt 2.3.1.1 på side 25 for hva som er lovlige testverdier.

while stmt

Figur 2.12: Jernbanediagram for {while stmt}

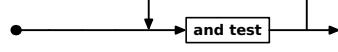
Funksjonsdeklarasjoner I Asp regnes funksjonsdeklarasjoner som setninger.

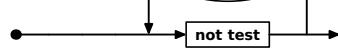
func def

Figur 2.13: 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 presedensen² blir slik vi ønsker.

expr

Figur 2.14: Jernbanediagram for {expr}

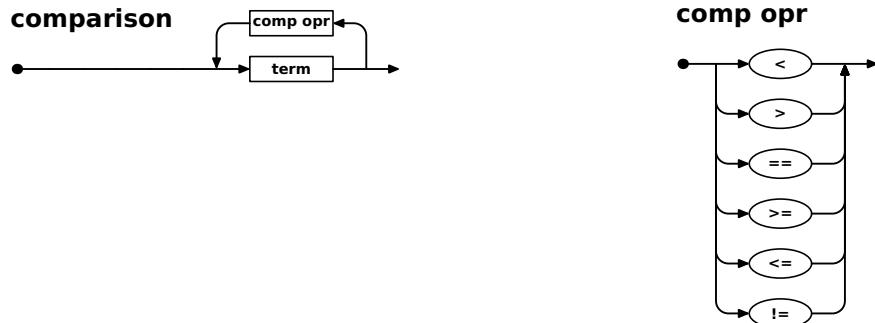
and test

not test

Figur 2.15: Jernbanediagram for {and test} og {not test}

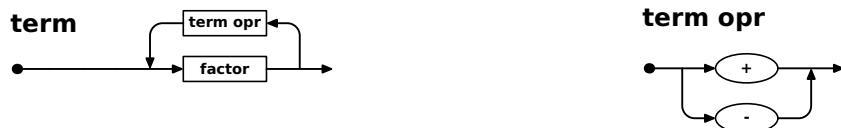
² Operatorer har ulik **presedens**, dvs at noen operatorer binder sterkere enn andre. Når vi skriver for eksempel

$$a + b \times c$$

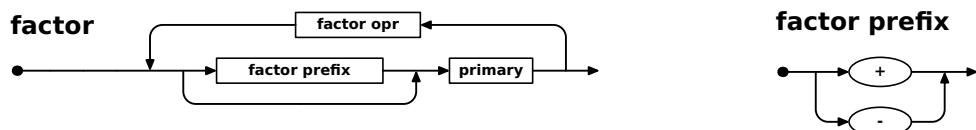
tolkes dette vanligvis som $a + (b \times c)$ fordi \times normalt har høyere presedens enn $+$, dvs \times binder sterkere enn $+$.



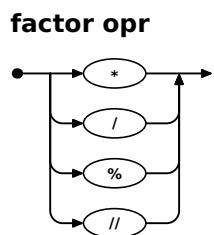
Figur 2.16: Jernbanediagram for {comparison} og {comp opr}



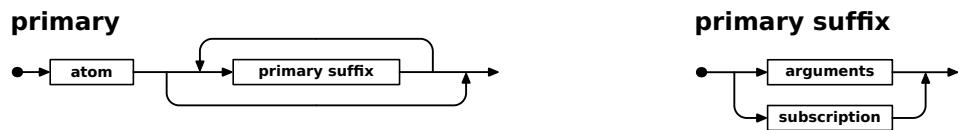
Figur 2.17: Jernbanediagram for {term} og {term opr}



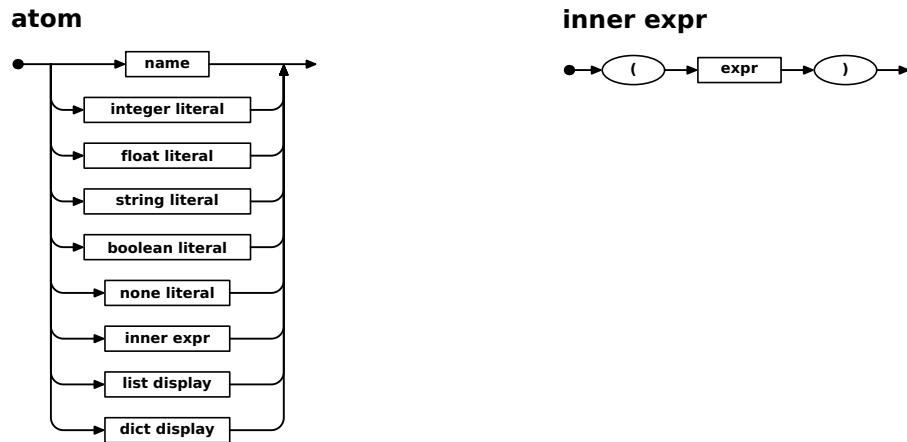
Figur 2.18: Jernbanediagram for {factor} og {factor prefix}



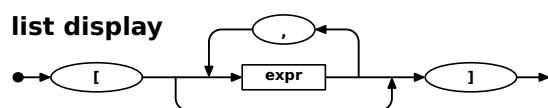
Figur 2.19: Jernbanediagram for {factor opr}



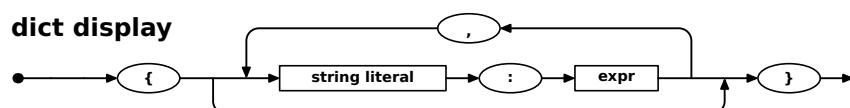
Figur 2.20: Jernbanediagram for {primary} og {primary suffix}



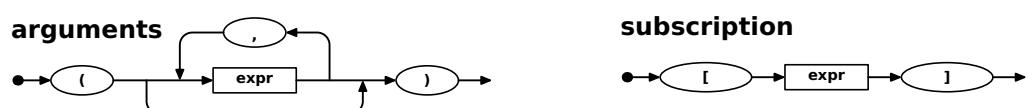
Figur 2.21: Jernbanediagram for {atom} og {inner expr}



Figur 2.22: Jernbanediagram for {list display}



Figur 2.23: Jernbanediagram for {dict display}

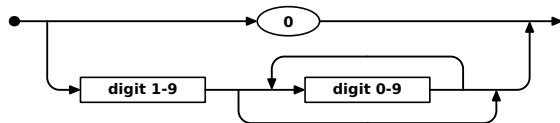


Figur 2.24: Jernbanediagram for {arguments} og {subscription}

2.2.2.1 Literaler

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

integer literal



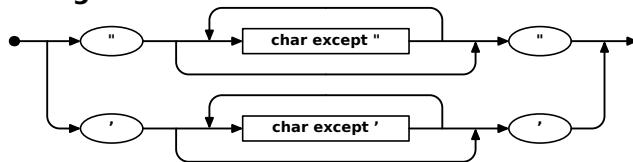
Figur 2.25: Jernbanediagram for {integer literal}

float literal



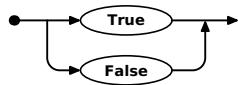
Figur 2.26: Jernbanediagram for {float literal}

string literal



Figur 2.27: Jernbanediagram for {string literal}

boolean literal



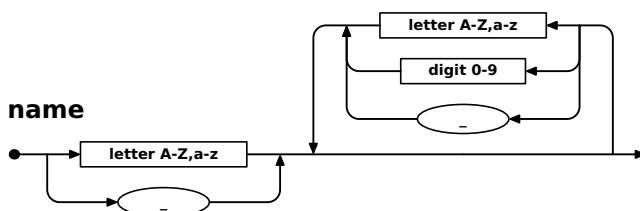
none literal



Figur 2.28: Jernbanediagram for {boolean literal} og {none literal}

2.2.2.2 Navn

I Asp benyttes navn til å identifisere variabler og funksjoner.



Figur 2.29: 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 alltid 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
```

Dette forklares nærmere i avsnitt 3.4.2.4 på side 52.

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.
// 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 2.27 på side 24).⁴ 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.⁵

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

- s[i]** kan gi oss enkelttegn fra teksten.⁶
- s₁ + s₂** 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

Istedentfor 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.

2.3.2 Operatorer

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

⁴ Dessverre finnes det flere tegn som ligner på disse anførselstegnene. I Asp-kode skal vi bruke disse:

- ' (ASCII 39=27₁₆ og Unicode U+0027) finnes til venstre for Enter-tasten på norske PC-tastaturer og til venstre for 1-tasten på et Mac-tastatur.
- " (ASCII 34=22₁₆ og Unicode U+0022) finnes på 2-tasten (med skift) på både norske PC- og Mac-tastaturer.

⁵ 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.

⁶ 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.

	Resultat	Kommentar
+ float + int	float int	Resultatet er v_1 Resultatet er v_1
- float - int	float int	
float + float float + int int + float int + int string + string	float float float int string	Tekststrengene skjøtes
float - float float - int int - float int - int	float float float int	
float * float float * int int * float int * int string * int list * int	float float float int string list	Sett sammen v_2 kopier av v_1 Sett sammen v_2 kopier av v_1
float / float float / int int / float int / int	float float float float	
float // float float // int int // float int // int	float float float int	Beregnes med <code>Math.floor(v1/v2)</code> Beregnes med <code>Math.floor(v1/v2)</code> Beregnes med <code>Math.floor(v1/v2)</code> Beregnes med <code>Math.floorDiv(v1,v2)</code>
float % float float % int int % float int % int	float float float int	Beregnes med $v_1 - v_2 * \text{Math.floor}(v_1/v_2)$ Beregnes med $v_1 - v_2 * \text{Math.floor}(v_1/v_2)$ Beregnes med $v_1 - v_2 * \text{Math.floor}(v_1/v_2)$ Beregnes med <code>Math.floorMod(v1,v2)</code>
float == float float == int int == float int == int string == string none == any any == none	bool bool bool bool bool bool bool	
float < float float < int int < float int < int string < string	bool bool bool bool bool	
not any	bool	Se tabell 2.2 på side 25 for logiske verdier
any and any	any	Oversettes som: $v_1 ? v_2 : v_1$
any or any	any	Oversettes som: $v_1 ? v_1 : v_2$

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.

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 ...`

En mer detaljert forklaring kommer i avsnitt [3.4.2.5 på side 52](#).

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 sjekke om programmeren 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 variablene; 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 [2.30 på neste side](#) ser vi at variablene v1 og v2 ikke deklarereres 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.

I figur [2.30 på neste side](#) er vist et eksempel i Asp sammen med det identiske programmet i Java.

2.4 PREDEFINERTE DEKLARASJONER

Asp	Java
<pre># 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. # # Example: gcd(32,36) = 4 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)</pre>	<pre>// 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. // // Example: gcd(32,36) = 4 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); } }</pre>

Figur 2.30: Indentering i Asp kontra krøllparenteser i Java

2.3.5 Andre ting

2.3.5.1 Kommentarer

Kommentarer skrives slik:

```
# resten av linjen
```

2.3.5.2 Tabulering

TAB-er kan brukes til å angi innrykk, og de angir inntil 4 blanke i starten av linjene.⁷

2.4 Predefinerte deklarasjoner

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

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 2.30.

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`.

⁷ **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.

Funksjon	Typekrav	Forklaring
<code>float(v)</code>	$v \in \{\text{int, float, string}\}$	Omformer v til en float
<code>input(v)</code>	$v \in \{\text{string}\}$	Skriver v og leser en linje fra tastaturet; resultatet er en string
<code>int(v)</code>	$v \in \{\text{int, float, string}\}$	Omformer v til en int
<code>len(v)</code>	$v \in \{\text{string, dict, list}\}$	Gir lengden av en string, en dict eller en list; resultatet er en int
<code>print(v₁, v₂, ...)</code>	$v_i \in \text{any}$	Skriver ut v_1, v_2, \dots med blank mellom; resultatet er none.
<code>range(v₁, v₂)</code>	$v_1, v_2 \in \{\text{int}\}$	Gir en liste med verdiene $[v_1, \dots, v_2 - 1]$.
<code>str(v)</code>	$v \in \text{any}$	Omformer v til en string

Tabell 2.4: Asps bibliotek av predefinerte funksjoner

2.4.2 Utskrift

Asp-programmer kan skrive ut ved å benytte den helt spesielle funksjonen `print`. Denne funksjonen 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 verdier 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 2.30 på forrige side.

Hint Husk at flyttall sjeldent lagres helt nøyaktig, så man kan risikere at verdien 1,3 skrives ut på ulike måter:

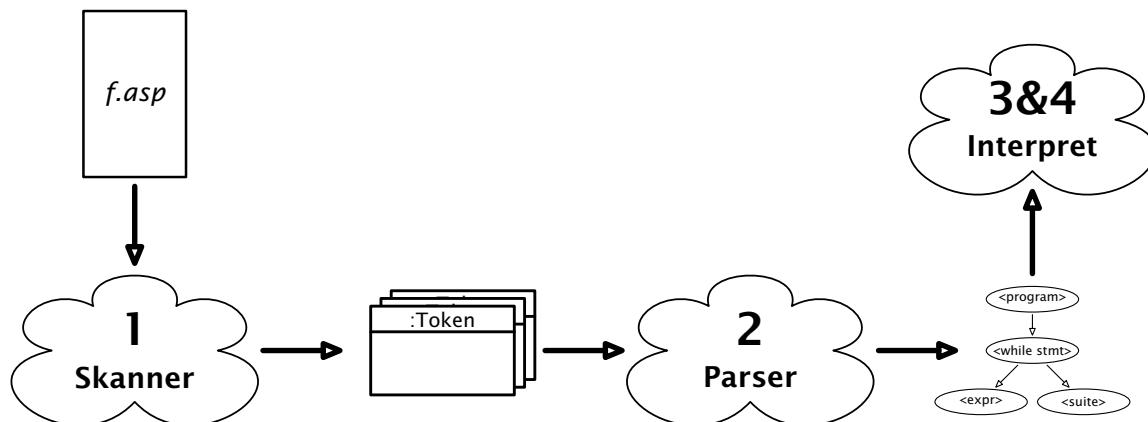
1.3 1.300000 1.3000000000000003 1.2999999999999998

Alle utskrifter som ligger nær den ekte verdien, blir godtatt.

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 **in2030-oblig-2021.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 in2030-oblig-2021.zip
$ ant
```

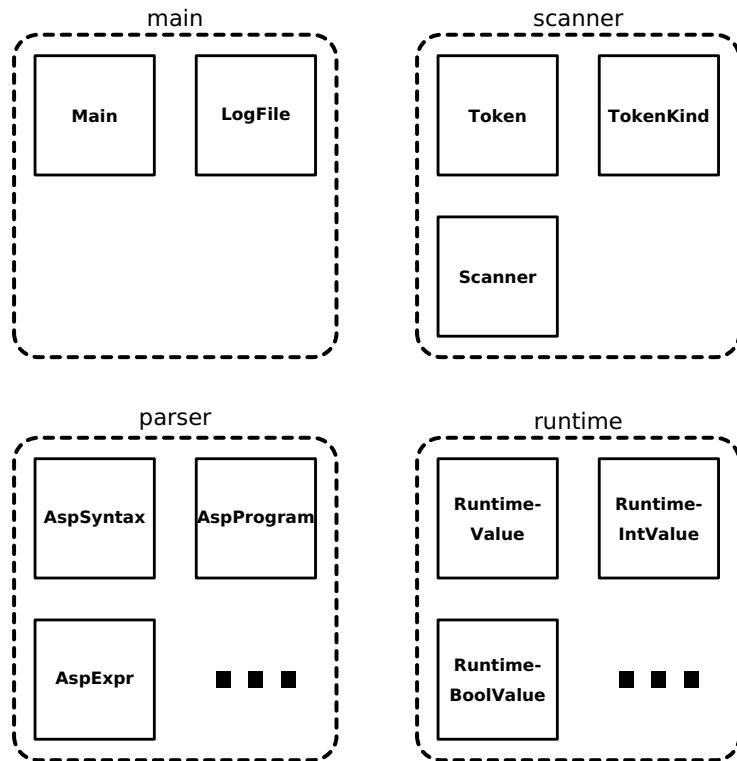
Dette vil resultere i en kjørbar fil *asp.jar* som kan kjøres slik

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

men den utleverte koden vil selvfølgelig ikke 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 35.

parser inneholder (når prosjektet er ferdig) rundt 35 klasser som brukes til å analysere Asp-koden og bygge parseringstreet; se avsnitt 3.3 på side 41.

runtime inneholder (etter at du har programmert ferdig) et dusin klasser som skal representerer 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.

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 parсерingstreet

Tabell 3.1: Opsjoner for logging

3.1.4 Testprogrammer

Til hjelp under arbeidet finnes diverse testprogrammer:

- I mappen [~inf2100/oblig/obligatorisk/](https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/obligatorisk/) (som også er tilgjengelig fra en nettleser som <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/obligatorisk>) ligger diverse Asp-programmer som interpretern din må kunne håndtere riktig for å få godkjent del 1, 2 og 4 av prosjektet.
- I mappen [~inf2100/oblig/obligatorisk-del3/](https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/obligatorisk-del3/) (som også er tilgjengelig fra en nettleser som <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/obligatorisk-del-3>) ligger tilsvarende for del 3 av prosjektet.
- I mappen [~inf2100/oblig/test/](https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/test/) (som også er tilgjengelig fra en nettleser som <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/test>) finnes noen flere Asp-programmer som du kan bruke til å teste interpretern din.
- I mappen [~inf2100/oblig/feil/](https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/feil/) (som også er tilgjengelig utenfor Ifi som <https://www.uio.no/studier/emner/matnat/ifi/IN2030/h21/feil>) finnes diverse småprogrammer som alle inneholder en feil. Interpretern din bør håndtere disse programmene og gi en fornuftig feilmelding (gjerne den samme som referanseinterpreteren).

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 er installert og 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* versjon 8 eller nyere; den 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 fire tegnkoder som er i vanlig bruk i Norge:

ASCII er en gammel 7-bits tegnkode som kan brukes overalt. Alle de tre andre kodingene nevnt her er utvidelser av ASCII.

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 din bør bare inneholde ASCII-tegn, men den ferdige interpreten skal lese og skrive UTF-8.

3.1.7 Innlevering

Når en del av prosjektet er ferdig for innlevering, må dere gjøre kommandoen

```
$ ant zip
```

Resultatet er filen `asp.zip` som leveres i Devilry (<https://devilry.ifi.uio.no>).

Hvis dere er to om arbeidet, må dere opprette en gruppe i Devilry før dere leverer deres felles besvarelse. Det må dere gjøre ved hver innlevering.

3.1.7.1 Registrering i Inspera

Når alle fire delene av prosjektet er godkjent, må det registreres i Inspera (<https://ui0.inspera.no>). Send inn den samme ZIP-filen som dere leverte i siste innlevering i Devilry.

Hvis dere har jobbet som en gruppe med to personer, må begge logge inn i Inspera og levele filen.

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.

```
1 ━━━━━━ mini.asp ━━━━━━
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:



Legg merke til at den blanke linjen og kommentaren er fjernet, og også all informasjon om blanke tegn mellom symbolene; vi har nå bare selve symbolene igjen. 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 at det er slutt på filen.⁸

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.

```
1 ━━━━━━ Token.java ━━━━━━
2 public class Token {
3     public TokenKind kind;
4     public String name, stringLit;
5     public long integerLit;
6     public double floatLit;
7     public int lineNumber;
8     :
9 }
```

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 («Format tokens») som ikke så mange programmeringsspråk har.

⁸ «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.

```
public enum TokenKind {                                     TokenKind.java
    // Names and literals:
    nameToken("name"),
    integerToken("integer literal"),
    floatToken("float literal"),
    stringToken("string literal"),

    // Keywords (including those used in Python 3):
    andToken("and"),
    asToken("as"),           // Not used in Asp
    :
    // Format tokens:
    indentToken("INDENT"),
    dedentToken("DEDENT"),
    newLineToken("NEWLINE"),
    eofToken("E-o-f");

    String image;

    TokenKind(String s) {
        image = s;
    }

    @Override
    public String toString() {
        return image;
    }
}
```

Figur 3.5: Enum-klassen TokenKind

3.2.2 Skanneren

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

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 i den aktuelle setningen.

curLineNum gir nåværende linjes linjenummer.

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

findIndent teller antall blanke i starten av den nåværende linjen; se avsnitt 3.2.2.3 på side 38. (Denne metoden er privat og kalles bare fra readNextLine.)

```

public class Scanner {                               Scanner.java
    private LineNumberReader sourceFile = null;
    private String curFileName;
    private ArrayList<Token> curLineTokens = new ArrayList<>();
    private Stack<Integer> indents = new Stack<>();
    private final int TABDIST = 4;

    public Scanner(String fileName) {
        curFileName = fileName;
        indents.push(0);

        try {
            sourceFile = new LineNumberReader(
                new InputStreamReader(
                    new FileInputStream(fileName),
                    "UTF-8"));
        } catch (IOException e) {
            scannerError("Cannot read " + fileName + "!");
        }
    }

    :

    public boolean anyEqualToken() {
        for (Token t: curLineTokens) {
            if (t.kind == equalToken) return true;
            if (t.kind == semicolonToken) return false;
        }
        return false;
    }
}

```

Figur 3.6: Klassen Scanner

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 algoritmen i figur 3.7 på neste side.

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 effekten vist i figur 3.8 på neste side.

TAB-er senere på linjen Siden formålet med omformingen av TAB-ene er å beregne indenteringen, er det nok å fjerne dem fra den innledende

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 $4 - (n \text{ mod } 4)$ blanke; øk n tilsvarende.
 - Ved alle andre tegn avsluttes denne løkken.

Figur 3.7: Algoritme for omforming av TAB-er til blanke

$$\begin{array}{ll}
 |a = 0 & \Rightarrow |a = 0 \\
 |_a = 0 & \Rightarrow |_a = 0 \\
 |\rightarrow a = 0 & \Rightarrow |_____a = 0 \\
 |_a = 0 & \Rightarrow |_____a = 0 \\
 |_a = 0 & \Rightarrow |_____a = 0 \\
 |_a = 0 & \Rightarrow |_____a = 0 \\
 |\rightarrow a = 0 & \Rightarrow |_____a = 0 \\
 |\rightarrow a = 0 & \Rightarrow |_____a = 0 \\
 |\rightarrow a = 0 & \Rightarrow |_____a = 0
 \end{array}$$

Figur 3.8: Eksempel på ekspansjon av TAB-er

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 3.29 og 3.30 viser dette tydelig; se side 62.

Nøyaktig hvordan vi skal håndtere indentering er gitt av algoritmen i figur 3.9.

Håndtering av indentering

- 1) Opprett en stakk `indents`.
- 2) Push verdien 0 på `indents`.
- 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 $n > \text{indents.peek()}$:
 - i. Push n på `indents`.
 - ii. Legg et 'INDENT'-symbol i `curlineTokens`.
 - Ellers, så lenge $n < \text{indents.peek}()$:
 - i. Pop `indents`.
 - ii. Legg et 'DEDENT'-symbol i `curlineTokens`.

Hvis nå $n \neq \text{indents.peek}()$, har vi indenteringsfeil.
- 4) Etter at siste linje er lest:
For alle verdier på `indents` som er > 0 , legg et 'DEDENT'-symbol i `curlineTokens`.

Figur 3.9: Algoritme for håndtering av indentering

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

KAPITTEL 3 PROSJEKTET

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

Figur 3.10: Loggfil med symbolene som skanneren finner i mini.asp

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.noteSourceLine`.
- 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 3.10 og figur 3.28 på side 62. Når interpreten vår kjøres med opsjonen `-testscanner`, skriver den ut logginformasjonen vist i henholdsvis figur 3.10 og figur 3.29 til 3.30 på side 62–63.

3.2.4 Mål for del 1

Mål for del 1

Programmet skal utvikles slik at opsjonen `-testscanner` produserer loggfiler som ligner på de som er vist i figurene 3.10 og 3.29–3.30. Programmet skal fungere for alle testfilene i mappen `~inf2100/oblig/obligatorisk/`.

3.3 Del 2: Parsing

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 3.11 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 tilstrekkelig med deklarasjoner til å kunne representer ikke-terminalen. Som et eksempel er vist klassen `AspAndTest` som representerer (and test); se figur 3.12 på side 43.

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 subklasser av denne abstrakte klassen.

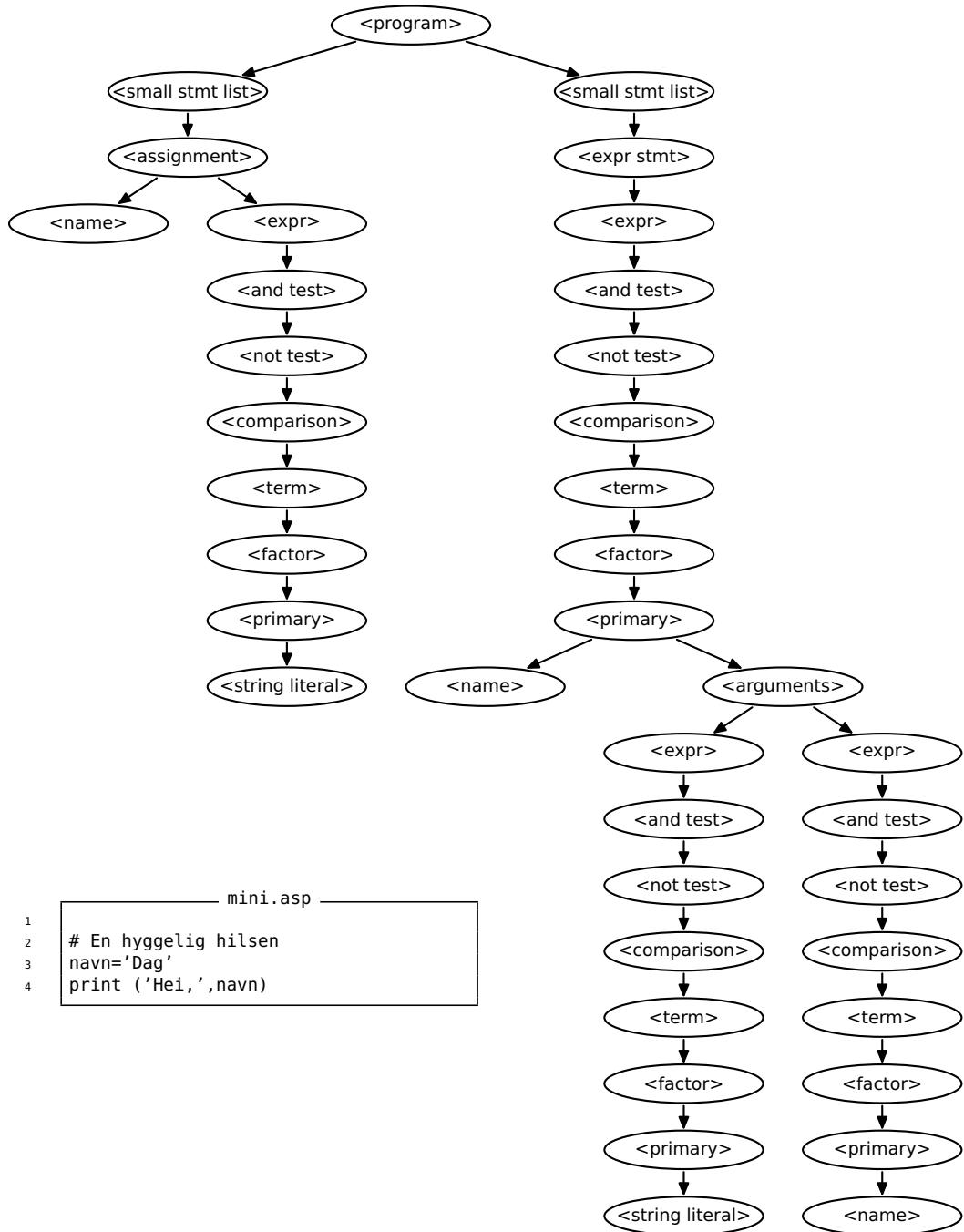
3.3.2 Parseringen

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

```

1 static Xxx 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 3.12 på side 43. (Metodene `test` og `skip` er nyttige i denne sammenhengen; de er definert i `parser.Asyntax`-klassen.)



Figur 3.11: Syntakstreet laget utifra det lille testprogrammet
mini.asp

```
AspAndTest.java
package no.uio.ifi.asp.parser;

import java.util.ArrayList;

import no.uio.ifi.asp.main.*;
import no.uio.ifi.asp.runtime.*;
import no.uio.ifi.asp.scanner.*;
import static no.uio.ifi.asp.scanner.TokenKind.*;

class AspAndTest extends AspSyntax {
    ArrayList<AspNotTest> notTests = new ArrayList<>();

    AspAndTest(int n) {
        super(n);
    }

    static AspAndTest parse(Scanner s) {
        enterParser("and test");

        AspAndTest aat = new AspAndTest(s.curLineNum());
        while (true) {
            aat.notTests.add(AspNotTest.parse(s));
            if (s.curToken().kind != andToken) break;
            skip(s, andToken);
        }

        leaveParser("and test");
        return aat;
    }

    @Override
    void prettyPrint() {
        int nPrinted = 0;

        for (AspNotTest ant: notTests) {
            if (nPrinted > 0)
                prettyWrite(" and ");
            ant.prettyPrint();  ++nPrinted;
        }
    }
}
```

Figur 3.12: Klassen AspAndTest (deler relevant for del 2)

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}

For å avgjøre dette, må vi titte fremover på linjen for å se om det finnes en = der eller ikke.⁹ Metoden Scanner.anyEqualToken er laget for dette formålet.

3.3.3 Syntaksfeil

En stor fordel ved å benytte denne *recursive descent*-teknikken for parsering er at det er meget enkelt å finne grammatikkfeil: Når det ikke finnes noe lovlige alternativ i jernbanediagrammene, har vi en feilsituasjon, og vi må kalle AspSyntax.parserError. (Metodene AspSyntax.test og AspSyntax.skip gjør dette automatisk for oss.)

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 35 og figur 3.28 på side 62 vil produsere loggfilene i figurene 3.13–3.14 og figurene 3.31 til 3.37 på side 64 og etterfølgende; disse loggfilene lages når interpreteren kjøres med operasjonen -logP eller -testparser.

3.3.5 Regenerering av programkoden

Logging er imidlertid ikke nok. Selv om parseringen forløp feilfritt, kan det hende at parсерingstreet ikke er riktig bygget opp. Den enkleste måten å sjekke dette på er å skrive ut det opprinnelige programmet basert på representasjonen i syntakstreet.¹⁰ Dette ordnes best ved å legge inn en metode

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

⁹ Slik titting forover er ikke helt politisk korrekt etter læreboken for *recursive descent*, men det er likevel ganske vanlig.

¹⁰ 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.

```

1  <program>
2    1:
3      2: # En hyggelig hilsen
4      3: navn='Dag'
5    <stmt>
6      <small_stmt_list>
7        <small_stmt>
8          <assignment>
9            <name>
10           </name>
11           <expr>
12             <and test>
13               <not test>
14                 <comparison>
15                   <term>
16                     <factor>
17                       <primary>
18                         <atom>
19                           <string_literal>
20                             </string_literal>
21                           </atom>
22                         </primary>
23                       </factor>
24                     </term>
25                   </comparison>
26                 </not test>
27               </and test>
28             </expr>
29           </assignment>
30         </small_stmt>
31       </small_stmt_list>
32     </stmt>

```

Figur 3.13: Loggfil som viser parsering av `mini.asp` (del 1)

i hver subklasse av `AspSyntax`. Resultatet kan sees i figur 3.15 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. Det er slik det skal være.

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 3.13–3.15 og 3.31–3.38. Programmet skal fungere for alle testfilene i mappen ~inf2100/oblig/obligatorisk/.

KAPITTEL 3 PROSJEKTET

```
33  4: print ('Hei,',navn)
34  <stmt>
35  <small stmt list>
36  <small stmt>
37  <expr stmt>
38  <expr>
39  <and test>
40  <not test>
41  <comparison>
42  <term>
43  <factor>
44  <primary>
45  <atom>
46  <name>
47  </name>
48  </atom>
49  <primary suffix>
50  <arguments>
51  <expr>
52  <and test>
53  <not test>
54  <comparison>
55  <term>
56  <factor>
57  <primary>
58  <atom>
59  <string literal>
60  </string literal>
61  </atom>
62  </primary>
63  </factor>
64  </term>
65  </comparison>
66  </not test>
67  </and test>
68  </expr>
69  <and test>
70  <not test>
71  <comparison>
72  <term>
73  <factor>
74  <primary>
75  <atom>
76  <name>
77  </name>
78  </atom>
79  </primary>
80  </factor>
81  </term>
82  </comparison>
83  </not test>
84  </and test>
85  </expr>
86  </arguments>
87  </primary suffix>
88  </primary>
89  </factor>
90  </term>
91  </comparison>
92  </not test>
93  </and test>
94  </expr>
95  </expr stmt>
96  </small stmt>
97  </small stmt list>
98  </stmt>
99  </program>
```

Figur 3.14: Loggfil som viser parsering av mini.asp (del 2)

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

Figur 3.15: Loggfil med «skjønnskrift» av mini.asp

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 57](#) og klassen `RuntimeReturnValue` i avsnitt [3.5.4.3 på side 60](#).)

Den komplette klassen `AspAndTest` med `eval`-metode er vist i figur [3.16 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 programering og la alle verdiene være av en subklasse av `runtime.RuntimeValue` som er vist i figur [3.17 på side 49](#). Et eksempel er `RuntimeBoolValue` som implementerer verdiene til `False` og `True` (se figur [3.18 på side 50](#)).

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.¹¹

¹¹ Metodene `showInfo` og `toString` vil nesten alltid gi samme resultat, men ikke alltid. For eksempel vil `runtimeStringValue.showInfo` ha med anførselstegnene rundt tekststrengen mens `runtimeStringValue.toString` vil droppe dem.

```
AspAndTest.java
// © 2021 Dag Langmyhr, Institutt for informatikk, Universitetet i Oslo

package no.uio.ifi.asp.parser;

import java.util.ArrayList;

import no.uio.ifi.asp.main.*;
import no.uio.ifi.asp.runtime.*;
import no.uio.ifi.asp.scanner.*;
import static no.uio.ifi.asp.scanner.TokenKind.*;

class AspAndTest extends AspSyntax {
    ArrayList<AspNotTest> notTests = new ArrayList<>();

    AspAndTest(int n) {
        super(n);
    }

    static AspAndTest parse(Scanner s) {
        enterParser("and test");

        AspAndTest aat = new AspAndTest(s.curLineNum());
        while (true) {
            aat.notTests.add(AspNotTest.parse(s));
            if (s.curToken().kind != andToken) break;
            skip(s, andToken);
        }

        leaveParser("and test");
        return aat;
    }

    @Override
    void prettyPrint() {
        int nPrinted = 0;

        for (AspNotTest ant: notTests) {
            if (nPrinted > 0)
                prettyWrite(" and ");
            ant.prettyPrint(); ++nPrinted;
        }
    }

    @Override
    RuntimeValue eval(RuntimeScope curScope) throws RuntimeReturnValue {
        RuntimeValue v = notTests.get(0).eval(curScope);
        for (int i = 1; i < notTests.size(); ++i) {
            if (! v.getBoolValue("and operand",this))
                return v;
            v = notTests.get(i).eval(curScope);
        }
        return v;
    }
}
```

Figur 3.16: Klassen AspAndTest

3.4 DEL 3: EVALUERING AV UTTRYKK

RuntimeValue.java

```
public abstract class RuntimeValue {
    abstract String typeName();

    public String showInfo() {
        return toString();
    }

    // For parts 3 and 4:

    public boolean getBoolValue(String what, AspSyntax where) {
        runtimeError("Type error: "+what+" is not a Boolean!", where);
        return false; // Required by the compiler!
    }

    public double getFloatValue(String what, AspSyntax where) {
        runtimeError("Type error: "+what+" is not a float!", where);
        return 0.0; // Required by the compiler!
    }

    public long getIntValue(String what, AspSyntax where) {
        runtimeError("Type error: "+what+" is not an integer!", where);
        return 0; // Required by the compiler!
    }

    public String getStringValue(String what, AspSyntax where) {
        runtimeError("Type error: "+what+" is not a text string!", where);
        return null; // Required by the compiler!
    }

    // For part 3:

    public RuntimeValue evalAdd(RuntimeValue v, AspSyntax where) {
        runtimeError("'" undefined for "+typeName()+"!", where);
        return null; // Required by the compiler!
    }

    :

    public RuntimeValue evalNot(AspSyntax where) {
        runtimeError("not undefined for "+typeName()+"!", where);
        return null; // Required by the compiler!
    }

    :

    // General:

    public static void runtimeError(String message, int lNum) {
        Main.error("Asp runtime error on line " + lNum + ": " + message);
    }

    public static void runtimeError(String message, AspSyntax where) {
        runtimeError(message, where.lineNum);
    }

    // For part 4:

    public void evalAssignElem(RuntimeValue inx, RuntimeValue val, AspSyntax where) {
        runtimeError("Assigning to an element not allowed for "+typeName()+"!", where);
    }

    public RuntimeValue evalFuncCall(ArrayList<RuntimeValue> actualParams,
                                    AspSyntax where) {
        runtimeError("Function call '(...)' undefined for "+typeName()+"!", where);
        return null; // Required by the compiler!
    }
}
```

Figur 3.17: Klassen RuntimeValue

```
RuntimeBoolValue.java
// © 2021 Dag Langmyhr, Institutt for informatikk, Universitetet i Oslo

package no.uio.ifi.asp.runtime;

import no.uio.ifi.asp.main.*;
import no.uio.ifi.asp.parser.AspSyntax;

public class RuntimeBoolValue extends RuntimeValue {
    boolean boolValue;

    public RuntimeBoolValue(boolean v) {
        boolValue = v;
    }

    @Override
    String typeName() {
        return "boolean";
    }

    @Override
    public String toString() {
        return (boolValue ? "True" : "False");
    }

    @Override
    public boolean getBoolValue(String what, AspSyntax where) {
        return boolValue;
    }

    @Override
    public RuntimeValue evalEqual(RuntimeValue v, AspSyntax where) {
        if (v instanceof RuntimeNoneValue) {
            return new RuntimeBoolValue(false);
        }
        runtimeError("Type error for ==.", where);
        return null; // Required by the compiler
    }

    @Override
    public RuntimeValue evalNot(AspSyntax where) {
        return new RuntimeBoolValue(! boolValue);
    }

    @Override
    public RuntimeValue evalNotEqual(RuntimeValue v, AspSyntax where) {
        if (v instanceof RuntimeNoneValue) {
            return new RuntimeBoolValue(true);
        }
        runtimeError("Type error for !=.", where);
        return null; // Required by the compiler
    }
}
```

Figur 3.18: Klassen RuntimeBoolValue

3.4.2.2 Metoder med Java-verdier

Disse metodene gir en Java-verdi som interpreteren kan bruke til ulike formål; hvis den ønskede verdien er umulig å fremstappe, skal interpreteren bruke standarddefinisjonen som er å 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 25.)

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 interpreteren gi en feilmelding og stoppe, og dette er allerede angitt som standardmetode i supertypen RuntimeValue.

Noen av disse metodene implementerer Asp-konstruksjoner som man vanligvis ikke tenker på som operatorer:

evalLen brukes av biblioteksfunksjonen len; se avsnitt 3.5.5 på side 61.

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

evalAssignElem plasserer en verdi i en liste eller ordbok.

evalFuncCall utfører et kall på en funksjon.

3.4.2.4 Evaluering av and og or

Operatoren **and** er spesiell i det at den kan settes sammen i en kjede, for eksempel

$v_1 \text{ and } v_2 \text{ and } v_3 \text{ and } \dots \text{ and } v_n$

Her evalueres operandene v_1 , v_2 etc inntil man finner en operand som er **False** (og husk å bruke tabell 2.2 på side 25 når det gjelder å avgjøre hva som er **True** og **False**); denne evaluerte operanden er da svaret for hele and-sekvensen. Hvis ingen v_i er **False**, er resultatet siste operand v_n .

Det tilsvarende gjelder for **or**, bortsett fra at evalueringen skal avsluttes første gang man finner noe som er **True**.

Eksempel Konstruksjonen

$v_1 \text{ and } v_2 \text{ and } v_3 \text{ and } \dots \text{ and } v_n$

evalueres slik:

- 1) Beregn først v_1 ; hvis den er **False**, er evalueringen avsluttet og svaret er v_1 .
- 2) Hvis vi ikke er ferdig, beregnes så v_2 ; hvis den er **False**, er evalueringen avsluttet og svaret er v_2 .
- 3) \dots
- 4) Om vi ennå ikke er ferdig, beregnes til sist siste operand v_n ; den er svaret, uansett om den er **True** eller **False**.

Se også på koden i figur 3.16 på side 48.

3.4.2.5 Evaluering av sammenligninger

Sammenligninger kan også kjedes sammen, for eksempel

$v_1 \leq v_2 < v_3 == \dots != v_n$

Dette skal evalueres som om brukeren hadde skrevet

$(v_1 \leq v_2) \text{ and } (v_2 < v_3) \text{ and } (v_3 == v_4) \text{ and } \dots \text{ and } (v_{n-1} != v_n)$

i stedet.¹² Med andre ord vil hver enkelt sammenligning bli evaluert inntil noen gir **False** som svar; i så fall er dette resultatet av hele sammenligningskjeden. Hvis alle enkeltsammenligningene gir **True** som svar, er resultatet **True** for hele kjeden.

¹² Dette er ikke helt sant. I en sammenligningskjede blir aldri noen operander evaluert mer enn én gang så resultatet av beregningen lagres i en skjult midlertidig variabel.

Eksempel Konstruksjonen
 $v_1 \leq v_2 < v_3 == v_4$

evalueres slik:

- 1) Først beregnes v_1 og v_2 og så $v_1 \leq v_2$; hvis resultatet er False, avsluttes evalueringen av hele sammenligningen med svaret False.
- 2) Så beregnes v_3 og deretter $v_2 < v_3$; hvis resultatet er False, avsluttes evalueringen av hele sammenligningen med svaret False.
- 3) Så beregnes v_4 og deretter $v_3 == v_4$, og dette blir svaret på hele sammenligningen.

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 -logE 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 3.19 og 3.21.

```
mini-expr.asp
1 "Noen eksempler på <expr>:"
2 1 + 2
3 2 + 2 == 4
```

Figur 3.19: Noen enkle Asp-uttrykk

Når vi kjører slike filer med kommandoen «java -jar asp.jar -testexpr mini-expr.asp», blir resultatet som vist i figurene 3.20 og 3.22 på side 55.

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

Figur 3.20: 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 3.20 og 3.22. (Operasjoner med flyt-tall behøver ikke gi nøyaktig samme svar som referanseinterpretene.) Programmet ditt skal fungere for alle filene i mappen [-inf2100/oblig/obligatorisk-del3/](#).

```
expressions.asp
1 "Testing integer expressions:"
2 42
3 -1017
4 -2 + 5 * 7
5 8 // 2 * (2+2)
6
7 "Testing float expressions:"
8 42.0
9 -1017.1
10 1.4142 * 1.4142
11 -2 + 5.0 * 7
12
13 "Testing string expressions"
14 "Abc"
15 "x" * 10
16 "Abra" + "ka" + "dabra"
17 'abcdefghijkl'[4]
18
19 "Testing boolean expressions"
20 not False
21 "To be" or "not to be"
22 "Yes" and 3.14
23 False or True or 144 or "?"
24
25 "Testing comparisons"
26 1 < 0
27 1 <= 2 <= 3
28
29 "Testing lists"
30 []
31 [-1,0,1]*(2)
32 [101,102,103][1]
33
34 "Testing dictionaries"
35 {"A": "a", "B": 1+2}
36 {"Ja": 1, "Nei": 0}["Ja"]
```

Figur 3.21: Noen litt mer avanserte Asp-uttrykk

3.4 DEL 3: EVALUERING AV UTTRYKK

```
1 "Testing integer expressions:" ==>
2 Trace line 1: 'Testing integer expressions:'
3 42 ==>
4 Trace line 2: 42
5 - 1017 ==>
6 Trace line 3: -1017
7 - 2 + 5 * 7 ==>
8 Trace line 4: 33
9 8 // 2 * (2 + 2) ==>
10 Trace line 5: 16
11 "Testing float expressions:" ==>
12 Trace line 7: 'Testing float expressions:'
13 42.000000 ==>
14 Trace line 8: 42.0
15 - 1017.100000 ==>
16 Trace line 9: -1017.1
17 1.414200 * 1.414200 ==>
18 Trace line 10: 1.9999616399999998
19 - 2 + 5.000000 * 7 ==>
20 Trace line 11: 33.0
21 "Testing string expressions" ==>
22 Trace line 13: 'Testing string expressions'
23 "Abc" ==>
24 Trace line 14: 'Abc'
25 "x" * 10 ==>
26 Trace line 15: 'xxxxxxxxxx'
27 "Abra" + "ka" + "dabra" ==>
28 Trace line 16: 'Abrakadabra'
29 "abcdefghijkl"[4] ==>
30 Trace line 17: 'e'
31 "Testing boolean expressions" ==>
32 Trace line 19: 'Testing boolean expressions'
33 not False ==>
34 Trace line 20: True
35 "To be" or "not to be" ==>
36 Trace line 21: 'To be'
37 "Yes" and 3.140000 ==>
38 Trace line 22: 3.14
39 False or True or 144 or "?" ==>
40 Trace line 23: True
41 "Testing comparisons" ==>
42 Trace line 25: 'Testing comparisons'
43 1 < 0 ==>
44 Trace line 26: False
45 1 <= 2 <= 3 ==>
46 Trace line 27: True
47 "Testing lists" ==>
48 Trace line 29: 'Testing lists'
49 [] ==>
50 Trace line 30: []
51 [- 1, 0, 1] * (2) ==>
52 Trace line 31: [-1, 0, 1, -1, 0, 1]
53 [101, 102, 103][1] ==>
54 Trace line 32: 102
55 "Testing dictionaries" ==>
56 Trace line 34: 'Testing dictionaries'
57 {"A":"a", "B":1 + 2} ==>
58 Trace line 35: {'A': 'a', 'B': 3}
59 {"Ja":1, "Nei":0}["Ja"] ==>
60 Trace line 36: 1
```

Figur 3.22: 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 lett frem å implementere.

3.5.1.1 Tilordningssetninger

Disse setningene er omtalt i avsnitt [3.5.3 på side 59](#).

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 innmatten i løkken utført så mange ganger som listens lengde 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 25](#)) og så tolke det tilhørende alternativet. Hvis ingen tester gir True, skal else-alternativet utføres.

3.5.1.5 While-setning

Dette er en løkke-setning, så interpreten må først beregne testuttrykket. Om det er True (i henhold til tabell [2.2 på side 25](#)), utføres løkkeinnmatten 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 60](#).

3.5.1.7 Pass-setning

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

3.5.1.8 Global-setningen

Denne setningen blir forklart i avsnitt [3.5.2.2 på side 58](#).

3.5.1.9 Funksjonsdefinisjoner

Dette omtales i avsnitt [3.5.4 på side 59](#).

3.5.2 Variabler

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

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 3.23.

```
public class RuntimeScope {                                         RuntimeScope.java
    private RuntimeScope outer;
    private HashMap<String,RuntimeValue> decls = new HashMap<>();
    private ArrayList<String> globalNames = new ArrayList<>();

    :

    public void assign(String id, RuntimeValue val) {
        decls.put(id, val);
    }

    public RuntimeValue find(String id, AspSyntax where) {
        if (globalNames.contains(id)) {
            RuntimeValue v = Main.globalScope.decls.get(id);
            if (v != null) return v;
        } else {
            RuntimeScope scope = this;
            while (scope != null) {
                RuntimeValue v = scope.decls.get(id);
                if (v != null) return v;
                scope = scope.outer;
            }
        }
        RuntimeValue.runtimeError("Name " + id + " not defined!", where);
        return null; // Required by the compiler.
    }
}
```

Figur 3.23: Klassen RuntimeScope

De to viktigste metodene er:

assign tilordner en verdi til en variabel i det *nåværende skopet*. Hvis variablene ikke allerede finnes, blir den opprettet.

find finner frem verdien tilordnet et gitt navn. Eventuelle *global*-setninger (se avsnitt 3.5.2.2 på neste side) avgjør om det skal letes i alle skop eller kun i det globale skopet.

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

- 1) *Biblioteket*, som inneholder predefinerte funksjoner; se avsnitt 3.5.5 på side 61.
- 2) *Globalt skop* (hovedprogrammet), som initielt er tomt. Variabelen `main.Main.globalScope` peker på dette.

Siden vil hvert funksjonskall opprette et nytt skop for sine parametre og variabler; se avsnitt 3.5.4.2 på side 60.

Skopene ligger inni hverandre. Ytterst ligger biblioteket og innenfor det ligger det globale skopet. Elementet `outer` i ethvert skop angir hvilket skop som ligger utenfor. Grunnen til dette er at man i et gitt skop kan referere

til navn som er deklarert ikke bare i eget skop men også i alle ytre skop. Se for eksempel på programmet `primes.asp` i figur 3.24:

```

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

```

Figur 3.24: Eksempel på bruk av skop

- `n_printed` defineres i funksjonens skop i linje 21, så bruken i linje 25 refererer til denne lokale variabelen.
- `n` defineres i hovedprogrammets skop i linje 5, så bruken i linje 23 refererer dit.
- `str` er predefinert i biblioteket, så bruken i linje 16 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.

3.5.2.2 Direkte adgang til globalt skop

Noen ganger ønsker en Asp-programmerer å endre globale variabler, og da trenger man i Asp (som i Python) global-setninger.¹³ Dette er vist i figur 3.25 på neste side. Det vanlige (vist til venstre) er at tilordning oppretter eller endrer en variabel i det nåværende skopet, uansett om det

¹³ Man trenger ikke global-setninger om man bare skal lese globale variabler, kun når de skal endres.

3.5 DEL 4: EVALUERING AV SETNINGER OG FUNKSJONER

finnes en variabel i ytre skop med samme navn. Med en global-setning (vist til høyre) angir programmeren at når man i det lokale skopet endrer eller oppretter a, er det a i det globale skopet man mener.

```
a = 1
def f():
    a = 2
f()
print("a =", a)
_____
a = 1

def f():
    global a
    a = 2
f()
print("a =", a)
_____
a = 2
```

Figur 3.25: Demonstrasjon av global

For å holde orden på global-spesifikasjoner har RuntimeScope en liste globalNames over navn som er definert som globale i akkurat det skopet.

Evaluering av global-setninger er rett og slett å legge de oppgitte navnene inn i det nåværende skopets globalNames.

3.5.3 Tilordning til variabler

Tilordning til enkle variabler er ganske enkelt å implementere:

- Hvis variabelens navn er nevnt i det nåværende skopets globalNames, skal man bruke metoden Main.globalScope.assign for å endre det globale skopet.
- Hvis ikke, skal man bruke assign i det nåværende skopet.

3.5.3.1 Tilordning til mer kompliserte variabler

Som vist i definisjonen til {assignment} (se figur 2.5 på side 19), kan en venstreside i en tilordning være ganske sammesatt og involvere én eller flere indeks til lister og/eller ordbøker:

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

Da må vi gjøre følgende:

- 1) For alle indeks unntatt den siste: Slå opp i listen eller ordboken på samme måte som når vi beregner uttrykk.
- 2) Kall på siste verdi sin evalAssignElem for å foreta tilordningen.

3.5.4 Funksjoner

Funksjoner er det mest intrikate vi skal ta for oss i dette prosjektet, men om vi holder tungen rett i munnen, bør det gå rimelig greit. Det viktigste er å få en full forståelse for 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 2.30 på side 29 kan betraktes som noe å la

```
GCD = {funksjonsverdi}
```

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

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 ske der eller i AspPrimary om du foretrekker det. Uansett bør følgende ske:¹⁴

- 1) De aktuelle parametrene¹⁵ beregnes og legges i en ArrayList.
- 2) Funksjonens evalFuncCall kalles med to parametre:
 - (a) listen med aktuelle parametre
 - (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 3.26 på neste side.

¹⁴ Beskrivelsen av funksjonskall er med vilje litt vag slik at du har mulighet til implementere dette på din egen måte.

¹⁵ Betegnelsen **aktuell parameter** angir en parameter som står i funksjonskallet, mens **formell parameter** betegner en parameter som står i funksjonsdefinisjonen.

3.5 DEL 4: EVALUERING AV SETNINGER OG FUNKSJONER

```
RuntimeReturnValue.java  
// © 2021 Dag Langmyhr, Institutt for informatikk, Universitetet i Oslo  
package no.uio.ifi.asp.runtime;  
  
// For part 4:  
  
public class RuntimeReturnValue extends Exception {  
    public int lineNumber;  
    public RuntimeValue value;  
  
    public RuntimeReturnValue(RuntimeValue v, int lNum) {  
        value = v; lineNumber = lNum;  
    }  
}
```

Figur 3.26: Klassen RuntimeReturnValue

Hvis man nå sørger for at koden som kaller på runFunction catch-er RuntimeReturnValue, har man det som skal til.¹⁶

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 funksjonene fra tabell 2.4 på side 30. 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 3.27.

```
RuntimeLibrary.java  
assign("len", new RuntimeFunc("len") {  
    @Override  
    public RuntimeValue evalFuncCall(  
        ArrayList<RuntimeValue> actualParams,  
        AspSyntax where) {  
        checkNumParams(actualParams, 1, "len", where);  
        return actualParams.get(0).evalLen(where);  
    }});
```

Figur 3.27: Fra klassen RuntimeLibrary

3.5.6 Sporing

Også setninger skal kunne spores ved at alle setningene kaller på AspSyntax.trace for å fortelle hva de gjør; se for eksempel figur 3.39 på side 71.

Mål for del 4

Programmet skal implementere resten av Asp slik at programmer som vist i figur 3.28 på neste side gir sporingsinformasjon som vist i figur 3.39 på side 71 når input er ordet "racecar". Programmet skal fungere for alle testfilene i mappen `~inf2100/oblig/obligatorisk/`.

¹⁶ Hva om funksjonen avsluttes uten noen return-setning? Da er det riktig å la funksjonsevalue-ring gi et RuntimeNoneValue-objekt som svar.

3.6 Et litt større eksempel

```
palindrom.asp
1 # A palindrome is a word that reads the same
2 # forwards and backwards, like OBO and RACECAR.
3
4 def is_a_palindrome (word):
5     i1 = 0; i2 = len(word)-1
6     while i1 < i2:
7         if word[i1] != word[i2]: return False
8         i1 = i1 + 1; i2 = i2 - 1
9
10    return True
11
12 query = input("A word: ")
13 print("'" +query+ "'", is_a_palindrome(query))
```

Figur 3.28: Et litt større Asp-program palindrom.asp

```
1:
2: # A palindrome is a word that reads the same
3: # forwards and backwards, like OBO and RACECAR.
4:
5: def is_a_palindrome (word):
6: Scanner: def token on line 5
7: Scanner: name token on line 5: is_a_palindrome
8: Scanner: ( token on line 5
9: Scanner: name token on line 5: word
10: Scanner: ) token on line 5
11: Scanner: : token on line 5
12: Scanner: NEWLINE token on line 5
13:     6: il = 0; i2 = len(word)-1
14: Scanner: INDENT token on line 6
15: Scanner: name token on line 6: i1
16: Scanner: = token on line 6
17: Scanner: integer literal token on line 6: 0
18: Scanner: ; token on line 6
19: Scanner: name token on line 6: i2
20: Scanner: = token on line 6
21: Scanner: name token on line 6: len
22: Scanner: ( token on line 6
23: Scanner: name token on line 6: word
24: Scanner: ) token on line 6
25: Scanner: - token on line 6
26: Scanner: integer literal token on line 6: 1
27: Scanner: NEWLINE token on line 6
28:     7: while i1 < i2:
29: Scanner: while token on line 7
30: Scanner: name token on line 7: i1
31: Scanner: < token on line 7
32: Scanner: name token on line 7: i2
33: Scanner: : token on line 7
34: Scanner: NEWLINE token on line 7
```

Figur 3.29: Loggfil som demonstrerer hvilke symboler skanneren finner i palindrom.asp (del 1)

```

35     8:         if word[i1] != word[i2]: return False
36 Scanner: INDENT token on line 8
37 Scanner: if token on line 8
38 Scanner: name token on line 8: word
39 Scanner: [ token on line 8
40 Scanner: name token on line 8: i1
41 Scanner: ] token on line 8
42 Scanner: != token on line 8
43 Scanner: name token on line 8: word
44 Scanner: [ token on line 8
45 Scanner: name token on line 8: i2
46 Scanner: ] token on line 8
47 Scanner: : token on line 8
48 Scanner: return token on line 8
49 Scanner: False token on line 8
50 Scanner: NEWLINE token on line 8
51     9:         i1 = i1 + 1; i2 = i2 - 1
52 Scanner: name token on line 9: i1
53 Scanner: = token on line 9
54 Scanner: name token on line 9: i1
55 Scanner: + token on line 9
56 Scanner: integer literal token on line 9: 1
57 Scanner: ; token on line 9
58 Scanner: name token on line 9: i2
59 Scanner: = token on line 9
60 Scanner: name token on line 9: i2
61 Scanner: - token on line 9
62 Scanner: integer literal token on line 9: 1
63 Scanner: NEWLINE token on line 9
64     10:    return True
65 Scanner: DEDENT token on line 10
66 Scanner: return token on line 10
67 Scanner: True token on line 10
68 Scanner: NEWLINE token on line 10
69     11:
70     12: query = input("A word: ")
71 Scanner: DEDENT token on line 12
72 Scanner: name token on line 12: query
73 Scanner: = token on line 12
74 Scanner: name token on line 12: input
75 Scanner: ( token on line 12
76 Scanner: string literal token on line 12: "A word: "
77 Scanner: ) token on line 12
78 Scanner: NEWLINE token on line 12
79     13: print("'" +query+ "'", is_a_palindrome(query))
80 Scanner: name token on line 13: print
81 Scanner: ( token on line 13
82 Scanner: string literal token on line 13: """
83 Scanner: + token on line 13
84 Scanner: name token on line 13: query
85 Scanner: + token on line 13
86 Scanner: string literal token on line 13: '":'
87 Scanner: , token on line 13
88 Scanner: name token on line 13: is_a_palindrome
89 Scanner: ( token on line 13
90 Scanner: name token on line 13: query
91 Scanner: ) token on line 13
92 Scanner: ) token on line 13
93 Scanner: NEWLINE token on line 13
94 Scanner: E-o-f token

```

Figur 3.30: Loggfil som demonstrerer hvilke symboler skanneren finner i palindrom.asp (del 2)

KAPITTEL 3 PROSJEKTET

```
1  <program>
2    1:
3      2: # A palindrome is a word that reads the same
4      3: # forwards and backwards, like OBO and RACECAR.
5      4:
6      5: def is_a_palindrome (word):
7        <stmt>
8          <compound stmt>
9            <func def>
10           <name>
11             </name>
12           <name>
13             </name>
14           <suite>
15         6:   il = 0;  i2 = len(word)-1
16         <stmt>
17           <small stmt list>
18             <small stmt>
19               <assignment>
20                 <name>
21                   </name>
22                 <expr>
23                   <and test>
24                     <not test>
25                       <comparison>
26                         <term>
27                           <factor>
28                             <primary>
29                               <atom>
30                                 <integer literal>
31                                   </integer literal>
32                                 </atom>
33                               </primary>
34                             </factor>
35                           </term>
36                         </comparison>
37                       </not test>
38                     </and test>
39                   </expr>
40                 </assignment>
41               </small stmt>
42             <small stmt>
43               <assignment>
44                 <name>
45                   </name>
46                 <expr>
47                   <and test>
48                     <not test>
49                       <comparison>
50                         <term>
51                           <factor>
52                             <primary>
53                               <atom>
54                                 <name>
55                                   </name>
56                                 </atom>
57                               <primary suffix>
58                                 <arguments>
59                                   <expr>
60                                     <and test>
61                                       <not test>
62                                         <comparison>
63                                           <term>
64                                             <factor>
65                                               <primary>
66                                                 <atom>
67                                                   <name>
68                                                     </name>
69                                                   </atom>
70                                                 </primary>
71                                               </factor>
72                                             </term>
73                                           </comparison>
74                                         </not test>
75                                       </and test>
```

Figur 3.31: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 1)

```
76          </expr>
77      </arguments>
78      </primary suffix>
79      </primary>
80      </factor>
81      <term opr>
82      </term opr>
83      <factor>
84      <primary>
85      <atom>
86      <integer literal>
87      </integer literal>
88      </atom>
89      </primary>
90      </factor>
91      </term>
92      </comparison>
93      </not test>
94      </and test>
95      </expr>
96      </assignment>
97      </small stmt>
98      </small stmt list>
99      </stmt>
100 7: while i1 < i2:
101      <stmt>
102      <compound stmt>
103      <while stmt>
104      <expr>
105      <and test>
106      <not test>
107      <comparison>
108      <term>
109      <factor>
110      <primary>
111      <atom>
112      <name>
113      </name>
114      </atom>
115      </primary>
116      </factor>
117      </term>
118      <comp opr>
119      </comp opr>
120      <term>
121      <factor>
122      <primary>
123      <atom>
124      <name>
125      </name>
126      </atom>
127      </primary>
128      </factor>
129      </term>
130      </comparison>
131      </not test>
132      </and test>
133      </expr>
134      <suite>
135 8: if word[i1] != word[i2]: return False
136      <stmt>
137      <compound stmt>
138      <if stmt>
139      <expr>
140      <and test>
141      <not test>
142      <comparison>
143      <term>
144      <factor>
145      <primary>
146      <atom>
147      <name>
148      </name>
149      </atom>
150      <primary suffix>
```

Figur 3.32: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 2)

KAPITTEL 3 PROSJEKTET

```
151                      <subscription>
152                          <expr>
153                              <and test>
154                                  <not test>
155                                      <comparison>
156                                          <term>
157                                              <factor>
158                                                  <primary>
159                                                      <atom>
160                                                          <name>
161                                                              </name>
162                                                      </atom>
163                                                  </primary>
164                                              </factor>
165                                          </term>
166                                      </comparison>
167                                  </not test>
168                              </and test>
169                          </expr>
170                      </subscription>
171                  </primary suffix>
172              </primary>
173          </factor>
174      </term>
175  <comp opr>
176  </comp opr>
177  <term>
178      <factor>
179          <primary>
180              <atom>
181                  <name>
182                      </name>
183                  </atom>
184          <primary suffix>
185              <subscription>
186                  <expr>
187                      <and test>
188                          <not test>
189                              <comparison>
190                                  <term>
191                                      <factor>
192                                          <primary>
193                                              <atom>
194                                                  <name>
195                                                      </name>
196                                              </atom>
197                                          </primary>
198                                      </factor>
199                                  </term>
200                              </comparison>
201                          </not test>
202                      </and test>
203                  </expr>
204              </subscription>
205          </primary suffix>
206      </primary>
207  </factor>
208      </term>
209  </comparison>
210  </not test>
211  </and test>
212 </expr>
213 <suite>
214     <small stmt list>
215         <small stmt>
216             <return stmt>
217                 <expr>
218                     <and test>
219                         <not test>
220                             <comparison>
221                                 <term>
222                                     <factor>
223                                         <primary>
224                                             <atom>
225                                                 <boolean literal>
```

Figur 3.33: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 3)

```
226                                     </boolean literal>
227                                         </atom>
228                                         </primary>
229                                         </factor>
230                                         </term>
231                                         </comparison>
232                                         </not test>
233                                         </and test>
234                                         </expr>
235                                         </return stmt>
236                                         </small stmt>
237                                         </small stmt list>
238                                         </suite>
239 9:      i1 = i1 + 1;  i2 = i2 - 1
240                                         </if stmt>
241                                         </compound stmt>
242                                         <stmt>
243                                         <small stmt list>
244                                         <small stmt>
245                                         <assignment>
246                                         <name>
247                                         </name>
248                                         <expr>
249                                         <and test>
250                                         <not test>
251                                         <comparison>
252                                         <term>
253                                         <factor>
254                                         <primary>
255                                         <atom>
256                                         <name>
257                                         </name>
258                                         </atom>
259                                         </primary>
260                                         </factor>
261                                         <term opr>
262                                         </term opr>
263                                         <factor>
264                                         <primary>
265                                         <atom>
266                                         <integer literal>
267                                         </integer literal>
268                                         </atom>
269                                         </primary>
270                                         </factor>
271                                         </term>
272                                         </comparison>
273                                         </not test>
274                                         </and test>
275                                         </expr>
276                                         <assignment>
277                                         <small stmt>
278                                         <small stmt>
279                                         <assignment>
280                                         <name>
281                                         </name>
282                                         <expr>
283                                         <and test>
284                                         <not test>
285                                         <comparison>
286                                         <term>
287                                         <factor>
288                                         <primary>
289                                         <atom>
290                                         <name>
291                                         </name>
292                                         </atom>
293                                         </primary>
294                                         </factor>
295                                         <term opr>
296                                         </term opr>
297                                         <factor>
298                                         <primary>
299                                         <atom>
```

Figur 3.34: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 4)

KAPITTEL 3 PROSJEKTET

```
301                               <integer literal>
302                               </integer literal>
303                               </atom>
304                               </primary>
305                               </factor>
306                               </term>
307                               </comparison>
308                               </not test>
309                               </and test>
310                               </expr>
311                               </assignment>
312                               </small stmt>
313                               </small stmt list>
314                               </stmt>
315 10:    return True
316                               </suite>
317                               </while stmt>
318                               </compound stmt>
319                               </stmt>
320                               <stmt>
321                               <small stmt list>
322                               <small stmt>
323                               <return stmt>
324                               <expr>
325                               <and test>
326                               <not test>
327                               <comparison>
328                               <term>
329                               <factor>
330                               <primary>
331                               <atom>
332                               <boolean literal>
333                               </boolean literal>
334                               </atom>
335                               </primary>
336                               </factor>
337                               </term>
338                               </comparison>
339                               </not test>
340                               </and test>
341                               </expr>
342                               </return stmt>
343                               </small stmt>
344                               </small stmt list>
345                               </stmt>
346 11:
347 12: query = input("A word: ")
348                               </suite>
349                               </func def>
350                               </compound stmt>
351                               </stmt>
352                               <stmt>
353                               <small stmt list>
354                               <small stmt>
355                               <assignment>
356                               <name>
357                               </name>
358                               <expr>
359                               <and test>
360                               <not test>
361                               <comparison>
362                               <term>
363                               <factor>
364                               <primary>
365                               <atom>
366                               <name>
367                               </name>
368                               </atom>
369                               <primary suffix>
370                               <arguments>
371                               <expr>
372                               <and test>
373                               <not test>
374                               <comparison>
375                               <term>
```

Figur 3.35: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 5)

```

376           <factor>
377             <primary>
378               <atom>
379                 <string literal>
380                   </string literal>
381                 </atom>
382               </primary>
383             </factor>
384           </term>
385         </comparison>
386       </not test>
387     </and test>
388   </expr>
389   </arguments>
390   </primary suffix>
391   </primary>
392   </factor>
393   </term>
394   </comparison>
395   </not test>
396   </and test>
397 </expr>
398 </assignment>
399   <small stmt>
400   <small stmt list>
401 </stmt>
402 13: print(''+query+':', is_a_palindrome(query))
403 <stmt>
404   <small stmt list>
405     <small stmt>
406       <expr stmt>
407         <expr>
408           <and test>
409             <not test>
410               <comparison>
411                 <term>
412                   <factor>
413                     <primary>
414                       <atom>
415                         <name>
416                           </name>
417                         </atom>
418                     <primary suffix>
419                       <arguments>
420                         <expr>
421                           <and test>
422                             <not test>
423                               <comparison>
424                                 <term>
425                                   <factor>
426                                     <primary>
427                                       <atom>
428                                         <string literal>
429                                           </string literal>
430                                         </atom>
431                                       </primary>
432                                     </factor>
433                                   <term opr>
434                                     </term opr>
435                                   <factor>
436                                     <primary>
437                                       <atom>
438                                         <name>
439                                           </name>
440                                         </atom>
441                                       </primary>
442                                     </factor>
443                                   <term opr>
444                                     </term opr>
445                                   <factor>
446                                     <primary>
447                                       <atom>
448                                         <string literal>
449                                           </string literal>
450                                         </atom>
451                                       </primary>
452                                     </factor>
453                                   </term>
454     </comparison>

```

Figur 3.36: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 6)

```
455                      </not test>
456                  </and test>
457              </expr>
458          <expr>
459          <and test>
460          <not test>
461          <comparison>
462          <term>
463          <factor>
464          <primary>
465          <atom>
466          <name>
467          </name>
468          </atom>
469          <primary suffix>
470          <arguments>
471          <expr>
472          <and test>
473          <not test>
474          <comparison>
475          <term>
476          <factor>
477          <primary>
478          <atom>
479          <name>
480          </name>
481          </atom>
482          <primary>
483          </factor>
484          </term>
485          </comparison>
486          </not test>
487          </and test>
488      </expr>
489      <arguments>
490      </primary suffix>
491      </primary>
492      <factor>
493      </term>
494      <comparison>
495      <not test>
496      </and test>
497      </expr>
498      <arguments>
499      </primary suffix>
500      </primary>
501      <factor>
502      </term>
503      <comparison>
504      <not test>
505      </and test>
506      </expr>
507      </expr stmt>
508      </small stmt>
509      </small stmt list>
510      </stmt>
511  </program>
```

Figur 3.37: Loggfil som viser parseringen av testprogrammet palindrom.asp (del 7)

3.6 ET LITT STØRRE EKSEMPEL

```
def is_a_palindrome (word):
    i1 = 0; i2 = len(word) - 1
    while i1 < i2:
        if word[i1] != word[i2]: return False
        i1 = i1 + 1; i2 = i2 - 1
    return True

query = input("A word: ")
print("'" + query + "'", is_a_palindrome(query))
```

Figur 3.38: Loggfil med «skjønnsskrift» av palindrom.asp

```
1 Trace line 5: def is_a_palindrome
2 Trace line 12: Call function input with params ['A word: ']
3 Trace line 12: query = 'racecar'
4 Trace line 13: Call function is_a_palindrome with params ['racecar']
5 Trace line 6: i1 = 0
6 Trace line 6: Call function len with params ['racecar']
7 Trace line 6: i2 = 6
8 Trace line 7: while True: ...
9 Trace line 9: i1 = 1
10 Trace line 9: i2 = 5
11 Trace line 7: while True: ...
12 Trace line 9: i1 = 2
13 Trace line 9: i2 = 4
14 Trace line 7: while True: ...
15 Trace line 9: i1 = 3
16 Trace line 9: i2 = 3
17 Trace line 7: while False:
18 Trace line 10: return True
19 Trace line 13: Call function print with params ['"racecar":', True]
20 Trace line 13: None
```

Figur 3.39: Sporingslogg fra kjøring av palindrom.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 <https://www.oracle.com/technetwork/java/codeconventions-150003.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:

```
1  /*
2   * Klassens navn
3   *
4   * Versjonsinformasjon
5   *
6   * Copyrightangivelse
7   */
```

- 2) Alle import-spesifikasjonene.
- 3) JavaDoc-kommentar for klassen. (JavaDoc er beskrevet i avsnitt [5.1 på side 77.](#))
- 4) Selve klassen.

4.1.2 Variabler

Variabler bør deklarereres én og én på hver linje:

```
1 int level;
2 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	XxxxxXxxx	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 innmatten 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.

```

1  do {
2      setninger;
3  } while (uttrykk);
4
5  for (init; betingelse; oppdatering) {
6      setninger;
7  }
8
9  if (uttrykk) {
10     setninger;
11 }
12
13 if (uttrykk) {
14     setninger;
15 } else {
16     setninger;
17 }
18
19 if (uttrykk) {
20     setninger;
21 } else if (uttrykk) {
22     setninger;
23 } else if (uttrykk) {
24     setninger;
25 }
26
27 return uttrykk;
28
29 switch (uttrykk) {
30 case xxx:
31     setninger;
32     break;
33
34 case xxx:
35     setninger;
36     break;
37
38 default:
39     setninger;
40     break;
41 }
42
43 try {
44     setninger;
45 } catch (ExceptionClass e) {
46     setninger;
47 }
48
49 while (uttrykk) {
50     setninger;
51 }

```

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 kodelinene 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 <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html>.

En JavaDoc-kommentar 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  *      :
5  * @param navn1 Kort beskrivelse av parameteren
6  * @param navn2 Kort beskrivelse av parameteren
7  * @return Kort beskrivelse av returverdien
8  * @see    navn3
9  */
```

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.

```
2 /**
3  * Returns an Image object that can then be painted on the screen.
4  * The url argument must specify an absolute {@link URL}. The name
5  * argument is a specifier that is relative to the url argument.
6  * <p>
7  * This method always returns immediately, whether or not the
8  * image exists. When this applet attempts to draw the image on
9  * the screen, the data will be loaded. The graphics primitives
10 * that draw the image will incrementally paint on the screen.
11 *
12 * @param url an absolute URL giving the base location of the image
13 * @param name the location of the image, relative to the url argument
14 * @return the image at the specified URL
15 * @see Image
16 */
17 public Image getImage(URL url, String name) {
18     try {
19         return getImage(new URL(url, name));
20     } catch (MalformedURLException e) {
21         return null;
22     }
23 }
```

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 *T_EX*. 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 hjelpeidler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.
- Det kan automatisk lages oversikter og klasser, funksjoner og variabler: hvor de deklarereres 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 `weave17` til å lage det ferdige dokumentet som er vist i figur 5.4–5.7:

```
1 $ weave0 -l c -e -o bubble.tex bubble.w0
2 $ ltx bubble.tex
```

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

```
1 $ tangle0 -o bubble.c bubble.w0
2 $ gcc -c bubble.c
```

¹⁷ Dette eksemplet bruker Dags versjon av lesbar programmering kalt `web0`; for mer informasjon, se <https://www.mn.uio.no/ifi/personer/vit/dag/public/doc/web0.pdf>.

bubble.w0 del 1

```

1 \documentclass[12pt,a4paper]{webzero}
2 \usepackage[latin1]{inputenc}
3 \usepackage[T1]{fontenc}
4 \usepackage{amssymb,mathpazo,textcomp}
5
6 \title{Bubble sort}
7 \author{Dag Langmyhr\\ Department of Informatics\\
8 University of Oslo\\ [5pt] \texttt{dag@ifi.uio.no}}
9
10 \begin{document}
11 \maketitle
12
13 \noindent This short article describes \emph{bubble
14 sort}, which quite probably is the easiest sorting
15 method to understand and implement.
16 Although far from being the most efficient one, it is
17 useful as an example when teaching sorting algorithms.
18
19 Let us write a function \texttt{bubble} in C which sorts
20 an array \texttt{a} with \texttt{n} elements. In other
21 words, the array \texttt{a} should satisfy the following
22 condition when \texttt{bubble} exits:
23 [
24   \forall i, j \in \mathbb{N}: 0 \leq i < j < n
25   \Rightarrow a[i] \leq a[j]
26 ]
27
28
29 <<bubble sort>>=
30 void bubble(int a[], int n)
31 {
32   <<local variables>>
33
34   <<use bubble sort>>
35 }
36 @
37 Bubble sorting is done by making several passes through
38 the array, each time letting the larger elements
39 "bubble" up. This is repeated until the array is
40 completely sorted.
41
42 <<use bubble sort>>=
43 do {
44   <<perform bubbling>>
45 } while (<<not sorted>>);
46 @

```

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 \wzmacroindex
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

July 23, 2020

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:

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

```
#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 #4(p.2). It is used in #1(p.1).)
```

¹We 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.

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

a1, 12, 14

I

i 11, 12, 13, 14

N

n1, 11

n_swaps 15, 16, 17, 18

T

temp 14, 16

VARIABLES

page 3

Figur 5.6: «Lesbar programmering» — utskrift side 3

Macro names

<i>(bubble sort #1)</i>	page 1 *
<i>(initialize #6)</i>	page 2
<i>(local variables #4)</i>	page 1
<i>(not sorted #7)</i>	page 2
<i>(perform bubbling #3)</i>	page 1
<i>(swap $a[i]$ and $a[i+1]$ #5)</i>	page 2
<i>(use bubble sort #2)</i>	page 1

(Macro names marked with * are not used internally.)

Figur 5.7: «Lesbar programmering» — utskrift side 4

Register

Aktuell parameter, 60

ant, 33

Asp, 17

Dictionary, 26

Dynamisk typing, 28

Formell parameter, 60

Interpret, 11, 13

java, 33

javac, 33

JavaDoc, 77

Kompilator, 13

Konstant, 24

Linux, 33

Liste, 26

Literal, 24

MacOS, 33

Moduler, 32

Ordbok, 26

Package, 32

Parser, 41

Parsering, 41

Presedens, 21

Programmeringsstil, 73

Python, 17

Recursive descent, 41

Skanner, 14

Sporing, 53

Statisk typing, 28

Symboler, 14, 35

Syntaks, 15

Syntakstre, 15

Tabulator, 29

Tokens, 14, 35

Tracing, 53

Unicode, 34

Windows, 33

