Dagens tema: del 1

- De ulike modulene i prosjektet vårt
  - Pakker i Java
- Hva gjør en Skanner?
- Klassene Token og TokenKind
  - Enum-klasser i Java
- Klassene Scanner og Main
- Feilmeldinger
- Testutskrifter
  - Klassen LogFile
- Praktiske råd

Hvordan skriver man et større program som en kompilator/interpret?

- Det bør deles opp i passe store deler.

Hvordan bør et program deles opp?

- Hvilken oppdeling virker naturlig?
- Hvilken oppdeling gir få aksesser mellom modulene?
- Hvordan flyter data?

Innledende oppgave

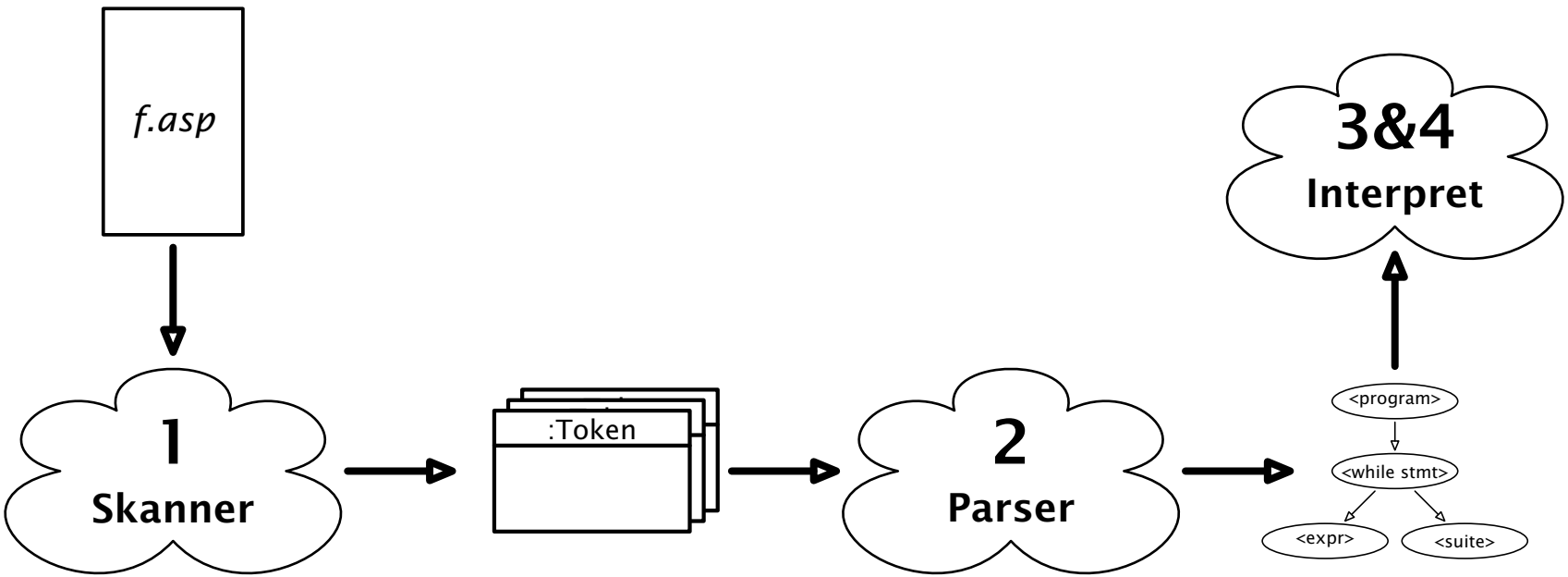
Hva må gjøres for å interpretere et enkelt program som dette?

print.asp

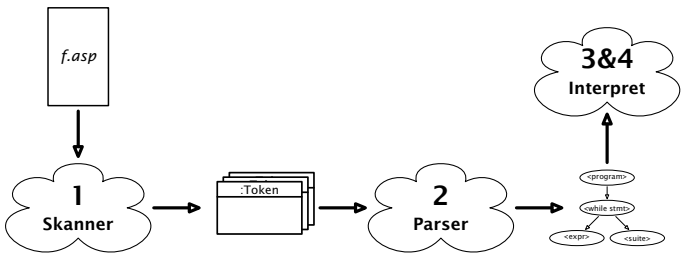
```
print(1+2)
```

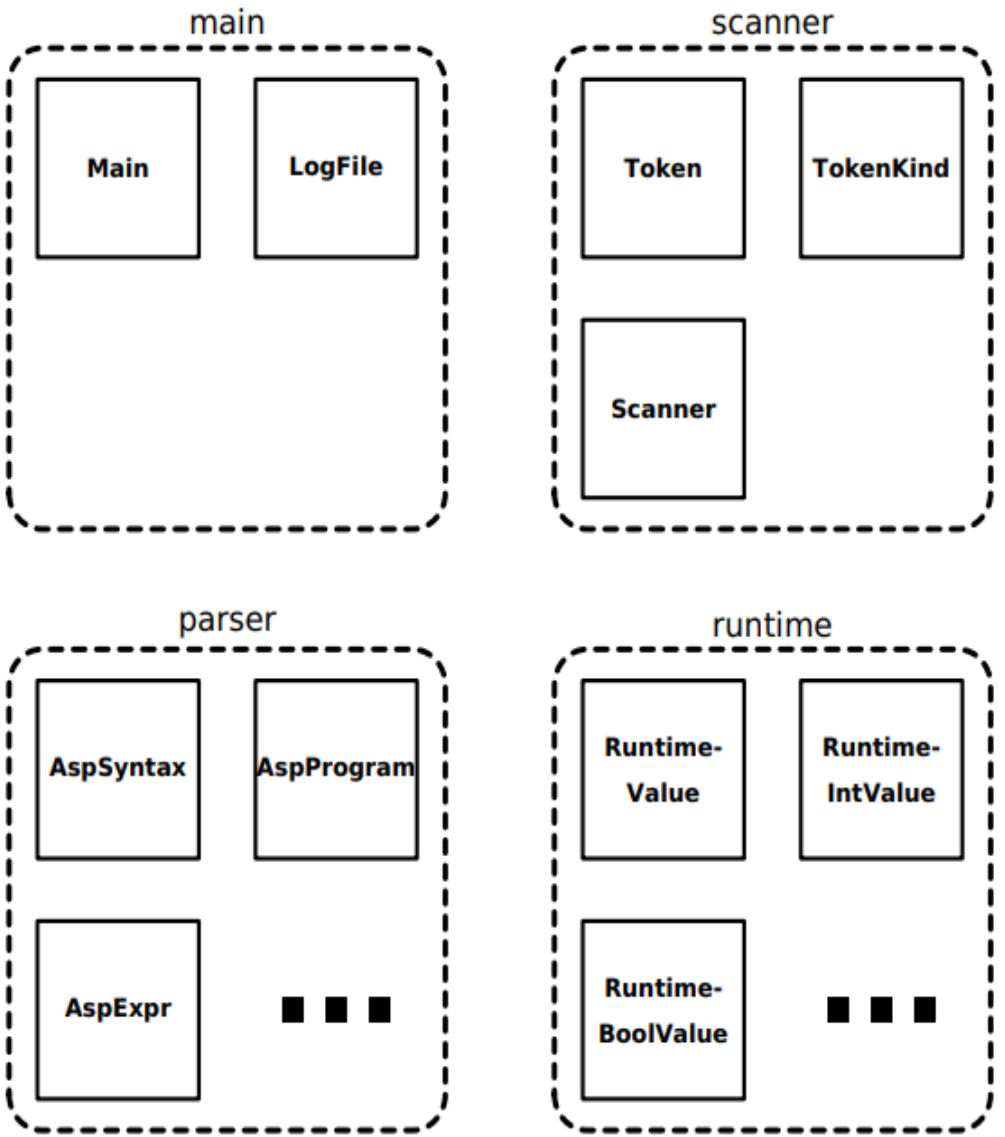
Beskriv så nøyaktig som mulig hvilke deloppgaver interpreten må utføre.

Prosjektet


f.asp
1
Skanner
:Token
2
Parser
3&4
Interpret
<program>
<while stmt>
<expr>
<suite>

Hvordan dele opp et stort prosjekt?





Moduler i Java

Noen programmeringsspråk har mekanismer for store moduler — men langt fra alle. Java har package.

Alle filene som skal inngå i Java-pakken *navn* starter med «package *navn*».

Eksempel

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Hver pakke består av et antall klasser.

Entydig pakkenavn

For å sikre et helt entydig pakkenavn, bør det innledes med eiers internettdomene i omvendt rekkefølge.
I IN2030 heter pakkene for eksempel:

no.uio.ifi.asp.xxx

Under kompileringen må klassene ligge i en mappestruktur som heter det samme som leddene i pakkenavnet. I IN2030 heter filene for eksempel:

no/uio/ifi/asp/scanner/Scanner.java

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Vi kan hente klasser fra alle pakker så lenge de finnes i CLASSPATH:

B.java

```
class B {
    public static void main (String arg[]) {
        System.out.println(P1.A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

For å unngå å skrive pakkenavnet så mange ganger, kan vi importere klasser fra pakker:

B.java

```
import P1.A;

class B {
    public static void main (String arg[]) {
        System.out.println(A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Vi kan også importere alle klassene fra en pakke:

B.java

```
import P1.*;

class B {
    public static void main (String arg[]) {
        System.out.println(A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

En siste mulighet er å importere *statiske* deklarasjoner fra en klasse:

B.java

```
import static P1.A.*;

class B {
    public static void main (String arg[]) {
        System.out.println(x);
    }
}
```

Oppsummering import

Import-setning	Bruk
< ingen >	P1.A.x
import P1.A;	A.x
import P1.*;	A.x
import static P1.A.*;	x

Beskyttelse av klasser

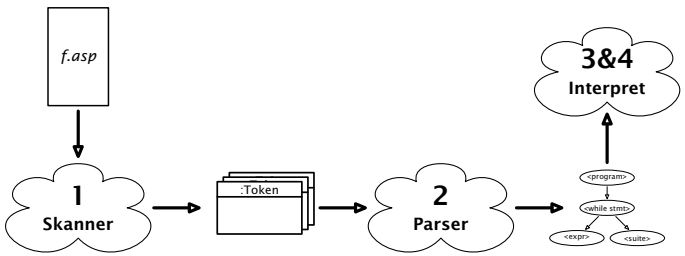
— er usynlig utenfor pakken.

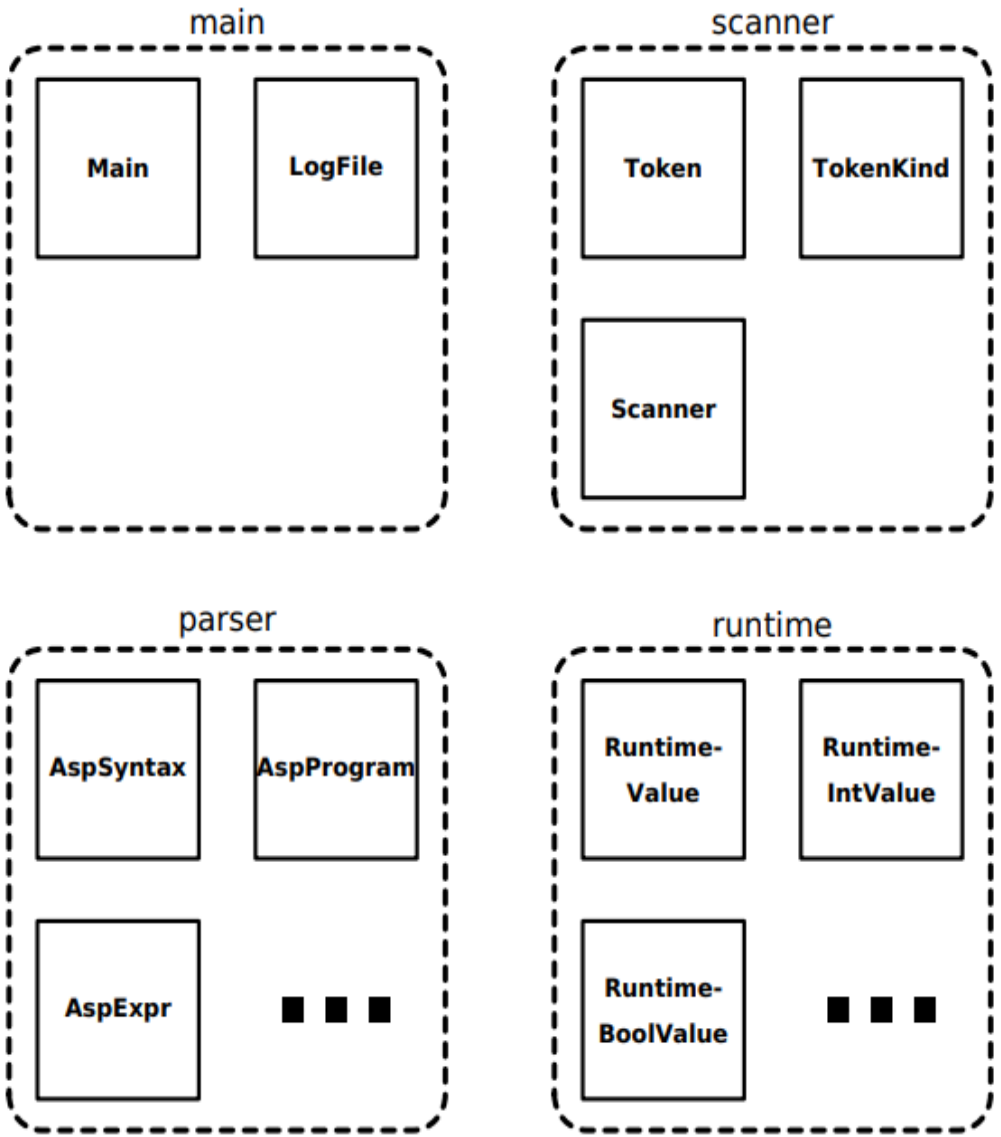
`public` kan brukes fra andre pakker.

Beskyttelse av klasseelementer

	public	protected	< ingen >	private
I klassen	✓	✓	✓	✓
I pakken	✓	✓	✓	
I subklasse (samme pakke)	✓	✓	✓	
I subklasse (annen pakke)	✓	✓		
Ellers	✓			

Våre pakker


f.asp
1
Skanner
:Token
2
Parser
3&4
Interpret


main
Main
LogFile
scanner
Token
TokenKind
Scanner
parser
AspSyntax
AspProgram
AspExpr
runtime
Runtime-Value
Runtime-IntValue
Runtime-BoolValue

Skanner

En kompilator/interpret *kan* lese og tolke en program tegn for tegn, men det er mye lettere om det kan gjøres *symbol for symbol*. Dette ordner en **skanner**.

En skanner gjør følgende:

- Leser programkoden fra en fil
- Fjerner alle kommentarer
- Deler resten av teksten opp i symboler («tokens»)

```
1
2   # En hyggelig hilsen
3   emne='IN2030'
4   print ('Velkommen til',emne,'!')
```

deles opp på denne måten:

`emne` `=` `'IN2030'` `NEWLINE`

`print` `(` `'Velkommen til'` `,` `emne`
`,` `'!'` `)` `NEWLINE`

`E-o-F`

```
1
2  # En hyggelig hilsen
3  emne='IN2030'
4  print ('Velkommen til',emne,'!')
```

og blir til disse symbolene (tokens):

`name:emne` `=` `string:"IN2030"` `NEWLINE`

`name:print` `(` `string:"Velkommen til"` `,` `name:emne`
`,` `string:"!"` `)` `NEWLINE`

`E-o-F`

```
1  if b:
2     while c:
3        c = False
4  else:
5     b = 0
```

har disse symbolene (tokens):

```
if  name:b  :  NEWLINE
INDENT  while  name:c  :  NEWLINE
INDENT  name:c  =  False  NEWLINE
DEDENT  DEDENT  else  :  NEWLINE
INDENT  name:b  =  int:0  NEWLINE
DEDENT  E-o-F
```

I vår skanner

Vår skanner kan levere de **symbolene** som er definert i klassen Token:

```
public class Token {
    public TokenKind kind;
    public String name, stringLit;
    public long integerLit;
    public double floatLit;
    public int lineNum;

      ⋮
```

I klassen Token lagres typen symbol (token) ved hjelp av klassen TokenKind. Denne er definert i en enum-klasse.

Enum-klasser

Noen ganger har man diskrete data som kun kan ha et begrenset antall fast definerte verdier:

Kortfarge Kløver, ruter, hjerter, spar

Tippetegn Hjemmeseier, uavgjort, borteseier

Ukedag Mandag, tirsdag, onsdag, torsdag, fredag, lørdag, søndag

Å representere disse med heltall er en halvgod løsning.

Java tilbyr **enum-klasser**:

Tippetegn.java

```
enum Tippetegn {
    Hjemmeseier, Uavgjort, Borteseier;
    // ...
}
```

Dette er **syntaktisk sukker** for

Tippetegn.java

```
class Tippetegn extends java.lang.Enum {
    public static final Tippetegn
        Hjemmeseier = new Tippetegn(),
        Uavgjort = new Tippetegn(),
        Borteseier = new Tippetegn();
    // ...
}
```

Hva kan vi gjøre med enum-klasser?

- Opprette verdier
  (`Tippetegn rekke[] new Tippetegn[12+1]`)
- Tilordne verdier
  (`rekke[i] = Tippetegn.Uavgjort`)
- Sjekke på likhet og ulikhet
  (`rekke[1] == Tippetegn.Borteseier`)
- Velge blant alternativer
  (`switch (rekke[1]) { case Uavgjort: ...}`)
- Skrive ut objektet
  (`System.out.println(rekke[1])`
  som er det samme som
  `System.out.println(rekke[1].toString())`)

I vår skanner

Vår skanner kan levere de **symbolene** som er definert i klassen Token:

```
public class Token {
    public TokenKind kind;
    public String name, stringLit;
    public long integerLit;
    public double floatLit;
    public int lineNum;

       ⋮
```

I klassen Token lagres typen symbol (token) ved hjelp av klassen TokenKind. Denne er definert i en enum-klasse.

TokenKind er definert i en enum-klasse:

```
public enum TokenKind {
    // 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
    assertToken("assert"),      // Not used in Asp
        .
        .
        .
    // Format tokens:
    indentToken("INDENT"),
    dedentToken("DEDENT"),
    newLineToken("NEWLINE"),
    eofToken("E-o-f");

    String image;

    TokenKind(String s) {
        image = s;
    }
    public String toString() {
        return image;
    }
```

Den sentrale metoden i del 1

Klassen `Scanner` skal lese linje for linje fra asp-filen og dele den opp i `Token`-objekter som lagres i en liste `curLineTokens`.

Hovedmetoden for å gjøre dette er `readNextLine`.

`readNextLine` skal altså gjøre følgende:

1. Innledende TAB-er oversettes til blanke.
2. Hvis linjen er tom (eventuelt blanke), ignoreres den.
3. Hvis linjen bare inneholder en kommentar (dvs første ikke-blanke tegn er en '#'), ignoreres den.
4. Indentering beregnes, og INDENT/DEDENT-er legges i `curLineTokens`.
5. Gå gjennom linjen:
   1. Blanke tegn og TAB-er ignoreres.
   2. En '#' angir at resten av linjen skal ignoreres.
   3. Andre tegn angir starten på et nytt symbol. Finn ut hvor mange tegn som inngår i symbolet. Lag et Token-objekt og legg det i `curLineTokens`.
6. Til slutt legges et NEWLINE-objekt i `curLineTokens`.

Hovedprogrammet `Main.java`

Noen ganger er det nyttig å ha data og metoder som er globale for prosjektet vårt. Disse legger vi i `Main`-objektet som `public static`.

```
package no.uio.ifi.asp.main;

import no.uio.ifi.asp.parser.AspExpr;
import no.uio.ifi.asp.parser.AspProgram;
import no.uio.ifi.asp.parser.AspSyntax;
import no.uio.ifi.asp.runtime.*;
import no.uio.ifi.asp.scanner.*;
import static no.uio.ifi.asp.scanner.TokenKind.*;

public class Main {
    public static final String version = "2023-08-21";
    public static LogFile log = null;
```

Modulen «main»

```
public static void main(String arg[]) {
    String fileName = null, baseFilename = null;
    boolean testExpr = false, testParser = false, testScanner = false,
    logE = false, logP = false, logS = false, logY = false;

    System.out.println("This is the IN2030 Asp interpreter (" +
                       version + ")");



    for (String a: arg) {
        if (a.equals("-logE")) {
            logE = true;
        } else if (a.equals("-logP")) {
            logP = true;
        } else if (a.equals("-logS")) {
            logS = true;
        } else if (a.equals("-logY")) {
            logY = true;
        } else if (a.equals("-testexpr")) {
            testExpr = true;
        } else if (a.equals("-testparser")) {
            testParser = true;
        } else if (a.equals("-testscanner")) {
            testScanner = true;
        } else if (a.startsWith("-")) {
            usage();
        } else if (fileName != null) {
            usage();
        } else {
            fileName = a;
        }
    }
    if (fileName == null) usage();
```

Modulen «main»

```
baseFilename = fileName;
if (baseFilename.endsWith(".asp"))
    baseFilename = baseFilename.substring(0,baseFilename.length()-4);
else if (baseFilename.endsWith(".py"))
    baseFilename = baseFilename.substring(0,baseFilename.length()-3);

log = new LogFile(baseFilename+".log");
if (logE || testExpr) log.doLogEval = true;
if (logP || testParser) log.doLogParser = true;
if (logS || testScanner) log.doLogScanner = true;
if (logY || testExpr || testParser) log.doLogPrettyPrint = true;

Scanner s = new Scanner(fileName);
if (testScanner)
    doTestScanner(s);
else if (testParser)
    doTestParser(s);
else if (testExpr)
    doTestExpr(s);
else
    doRunInterpreter(s);
```

```
private static void doTestScanner(Scanner s) {
do {
    s.readNextToken();
} while (s.curToken().kind != eofToken);
```

Klassen Scanner

```
public class Scanner {
    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 + "!");
        }
    }
```

Vi henter symboler fra den lokale bufferen `curLineTokens` med metoden `curToken`:

```
public Token curToken() {
    while (curLineTokens.isEmpty()) {
        readNextLine();
    }
    return curLineTokens.get(0);
}
```

Når vi er ferdige med et symbol, angir vi det med `readNextToken`:

```
public void readNextToken() {
    if (! curLineTokens.isEmpty())
        curLineTokens.remove(0);
}
```

Hver linje skal altså bli lest av `readNextLine` som splitter linjen i `Token`-objekter.

Indentering

Metoden `Scanner.findIndent` finner indenteringen:

```
    private int findIndent(String s) {
        int indent = 0;

        while (indent<s.length() && s.charAt(indent)==' ') indent++;
        return indent;
    }
```

For å kunne beregne innrykket, må vi også ta hensyn til TAB-er i starten av linjen — se algoritme i figur 3.7 i kompendiet samt ukeoppgavene.

Algoritme for å holde orden på indenteringen

1. Opprett en stakk `Indents` og push verdien 0 på den.
2. For hver linje:
   1. Hvis linjen bare inneholder blanke (og eventuelt en kommentar), ignoreres den.
   2. Omform alle innledende tab-er til blanke.
   3. Tell antall innledende blanke: `n`.
   4. Hvis `n > Indents.peek()`:
      — Push `n` på `Indents`.
      — Legg et 'INDENT'-symbol i `curLineTokens`.
   5. Så lenge `n < Indents.peek()`:
      — Pop `Indents`.
      — Legg et 'DEDENT'-symbol i `curLineTokens`.
   6. Hvis nå `n`≠`Indents.peek()`, har vi indenteringsfeil.
3. Etter at siste linje er lest:
   For alle verdier på `Indents` som er > 0, legg et 'DEDENT'-symbol i `curLineTokens`.

For å få interpreten til å fungere trenger vi i tillegg å legge inn

- feilmeldinger, feks ved syntaksfeil
- testutskrifter for å sjekke at ting fungerer som det skal

Hva er en god feilmelding?

Ubrukelig

```
ERROR: Syntax error detected!
```

En god del bedre

```
ERROR: Syntax error found in line 217.
```

Enda litt bedre

```
ERROR: Syntax error found in line 217:
    if x = y+1:
---------^
```

Melding med mening

Meldingen bør fortelle hva som er galt:

```
ERROR in line 217: Illegal expression.
    if x = y+1:
```

Den beste meldingen

Meldingen bør angi hvorledes kompilatoren/interpreten «tenker»:

```
Asp parser error on line 217: Expected : but found =
```

Feil

Hva gjør man med feil?

- Før prøvde man å finne så mange feil som mulig.
- Vi skal stoppe med melding ved første feil.

Metoden Main.error

```
public static void error(String message) {
    System.out.println();
    System.err.println(message);
    if (log != null) log.noteError(message);
    System.exit(1);
}
```

Metoden Scanner.scannerError

```
private void scannerError(String message) {
    String m = "Asp scanner error";
    if (curLineNum() > 0)
        m += " on line " + curLineNum();
    m += ": " + message;

    Main.error(m);
}
```

Testutskrifter

Alle vil gjøre feil under arbeidet med interpreten. For enklere å oppdage feilene når de skjer, skal det legges inn ulike typer testutskrifter som kan slås av og på:

Opsjon	Alternativt	Hva dumpes?	Del
–logS	–testscanner	Skanneren	1
–logP	–testparser	Parseringen	2
–logY	–testparser	«Pretty-printing»	2
–logE	–testexpr	Eksekveringen	3&4

Klassen main/LogFile

Det er enkelt å slå av og på logging.

```
public class LogFile {
    public boolean doLogEval = false,
        doLogParser = false,
        doLogPrettyPrint = false,
        doLogScanner = false;

    public void noteSourceLine(int lineNum, String line) {
        if (doLogParser || doLogScanner) {
            writeLogLine(String.format("%4d: %s",lineNum,line));
        }
    }

    public void noteToken(Token tok) {
        if (doLogScanner)
            writeLogLine("Scanner: " + tok.showInfo());
    }
```

```
1
2  # En hyggelig hilsen
3  emne='IN2030'
4  print ('Velkommen til',emne,'!')
```

skal bli til disse symbolene (tokens):

`name:emne` `=` `string:"IN2030"` `NEWLINE`

`name:print` `(` `string:"Velkommen til"` `,` `name:emne`
`,` `string:"!"` `)` `NEWLINE`

`E-o-F`

Klassen LogFile

```
$ ~inf2100/asp -testscanner mini.asp
$ more mini.log
   1:
   2: # En hyggelig hilsen
   3: emne='IN2030'
Scanner: name token on line 3: emne
Scanner: = token on line 3
Scanner: string literal token on line 3: "IN2030"
Scanner: NEWLINE token on line 3
   4: print ('Velkommen til',emne,'!')
Scanner: name token on line 4: print
Scanner: ( token on line 4
Scanner: string literal token on line 4: "Velkommen til"
Scanner: , token on line 4
Scanner: name token on line 4: emne
Scanner: , token on line 4
Scanner: string literal token on line 4: "!"
Scanner: ) token on line 4
Scanner: NEWLINE token on line 4
Scanner: E-o-f token
```

```
1
2   # En hyggelig hilsen
3   emne='IN2030'
4   print ('Velkommen til',emne,'!')
```

Klassen LogFile

På semestersiden og i mappen ~inf2100/oblig/ finnes det diverse testprogrammer:

- obligatorisk: programmer som interpreten *skal* fungere på
- test: ekstra test-programmer
- feil: test-programmer med ulike typer feil

I tillegg: Lag egne (små) testprogrammer!

Mål for del 1

1. Hent ned, pakk ut og kompiler prekoden.
2. Gjør nødvendige endringer slik at skanneren fungerer og at den skriver ut loggmeldinger som vist når interpreten kjøres med opsjonen **–testscanner**.

Skanneren er . . .

- . . . dum! Den lager symboler uten tanke på sammenhengen.
- . . . grådig! Den vil ha så lange symboler som mulig; for eksempel:

`... ifa ...` blir til `name:ifa`

Siste råd?

- En stor del av jobben er å skjønne basiskoden, resten er å programmere `Scanner.readNextLine`.
- Les kompendiet! Nøye!
- Det er lov å endre basiskoden litt, men det må eventuelt begrunnes i en kommentar.
- Bruk alt du vil fra Java-biblioteket (men `Tokenizer` frarådes).
- Bruk gruppetimene og Discourse.
- Begynn i tide!