An interpretation for Asp

Compendium for INF2100

Stein Krogdahl, Dag Langmyhr

Fall 2017

Page 3

Contents

Page 4

Page 5

figures

Page 6

tables

Page 8

Preface

This compendium is made subject *INF2100 - Project in*

programming. The course itself is one of the oldest at Ifi, but the content of The course has been renewed regularly.

The original course was developed by *Stein Krogdahl* around 1980 and turned about compiling a compiler that translated the Simula-like language *Minila* to code for an imaginary computer *Flink;* The implementation language was Simula. In 1999, you went to use Java as the implementation language, and in 2007 the course was completely renovated by *Day Langemyhr:* Minila was replaced by a minimal variation of C called *Rusc* and computer Flink was replaced by another non-existent machine called *Fast*. In 2010 it became decided to create real code for the Intel x86 processor so that it generated The code could be run directly on a computer. This resulted in such a great deal changes in language Rusc that given a new name: *C <(pronounced* "c loading"). Desiring an extension led to the introduction of data types in 2012 (int and double), and the language was again a new name: *Cb* (pronounced "c flat"). Feedback from the students revealed that they thought it was very much fooling to make code for double, so in 2014 the language was changed once more. Under the name *AlboC* ("A little bit of C") had now pointers instead of flow numbers.

Now it was 2015, and the whole plan went through another revision. The also applied to the language to be compiled: in this and the following year there was a language which included most of the good old *Pascal*.

In 2017 a new review was made. Since Ifi from this fall will use Python as an introductory language, it appeared a desire to give students a thorough introduction to how a Python interpreter works. To this was *Asp* (which thus is a mini-Python) developed.

The aim of this compendium is to do that together with the lecture plans shall provide the students with sufficient background to be able to complete the project tet.

The authors would otherwise like to thank the students *Einar Løvhøiden Antonsen, Jonny Bekkevold, Eivind Alexander Bergem, Marius Ekeberg, Arne Olav HallingGlass City, Espen Tørressen Hangård, Sigmund Hansen, Simen Heggestøyl, Simen Jensen, Thor Joramo, Morten Kolstad, Jan Inge Lamo, Brendan Johan Lee, Havard Koller Noren, Vegard Nossum, Hans Jørgen Nygårdshaug, David J Oftedal, Mikael Olausson, Catherine Elisabeth Olsen Bendik Rønning OptiMix City, Christian Resell, Christian Other Finnøy Ruud, Ryhor Sivuda, Yrjab Skrim Stad, Herman Torjussen Christian Tryti, Jørgen Vigdal Olga Voronkova, Aksel L Webster* and *Sindre Wilting* who have pointed out typos or FORE hit improvements in previous editions. If more students do this, they will Also get their name on tap.

PREFACE

Blindern, September 6, 2017
Stein Krogdahl *Day Langmyhr*

Theory is when nothing works and everyone know why. Practice is when everything works and no one knows why.

In this course, theory and theory combine practice - nothing works and no one know why.

- The authors

Chapter 1

Introduction

1.1 What is the subject INF2100?

The course INF2100 the designation *Programming project,* and
The main idea of this topic is to bring the students to such a large extent
programming project as possible within the framework of the ten credits
the course has. The reason we focus on a big program is that most
things that are related to structuring programs, object oriented
programming, division into modules etc, are not perceived as meaningful
or important before the programs get a certain size and complexity. The
As mentioned in the first course, these things are easily affected by a bit of life remover
"Programming moral" because you do not see the need for this way to
Think about the small tasks you usually go through.

Otherwise, programming is something you need training to be safe in.
This course will therefore not introduce so many new concepts around
programming, but instead try to consolidate what has already been learned,
and demonstrate how it can be used in different contexts.

The "big program" to be created during the INF2100 is one
interpret, ie a program that reads and analyzes a program in a given
programming language, and which then performs it as this program
indicates to be done. Below we will look at similarities and differences
between an interpret and a compiler.

Although we concentrate this course on a larger program will not
this could be something really *great* program. Out in the "real" world will be
Programs quickly away on hundreds of thousands or even millions of lines, and
It's only when you start writing such programs, and not
At least, later make changes in them that the structure of the programs becomes
absolutely crucial. The program we will make in this course will typically be on
just over four thousand lines.

In this compendium, only the programming-
the task to be solved. In addition to this, additional requirements may arise,
for example regarding the use of tools or written work as intended
delivered. This will be explained in the lectures and in the course
websites.

1.2 Why make a interpreter?

When selecting a theme for a programming assignment for this course,
there were first and foremost two criteria that were important:

> The assignment must be affordable to program within the ten
> credits.
>
> The program must concern a problem that the students know,
> so it does not go away valuable time to understand the purpose of
> the program and its surroundings.

In addition to this, you may wish to:

> Creating a program within a certain scope usually provides
> also better understanding of the area itself. It is therefore also desirable
> that the scope is obtained from programming, so that this one
> The side effect gives an increased understanding of the subject itself.
>
> The problem area should have as many interesting variations as it can
> be a good source of exercises that can illuminate the main problem-
> the position.

Based on these criteria, a field appears to be particularly tempting, namely to write an interpret, ie a simplified version of the usual one Python Interpreter. This is a type of tool that everyone has worked with programming has been beyond and, as it is also valuable for most to learn a little more about

The writing of an interpret will also, for most people, basically seem to be one big and unobtrusive task. Some of the points of the course are to demonstrate that with a proper breakdown of the program in parts that each takes The responsibility for a limited part of the task, both the individual parts and The entirety they form becomes highly conclusive. It is this experience, and The understanding of how such division can be done on a real example, such as It is the most important students to get away from this course.

In the next section we will look at what an interpretation is and how it stands compared to similar tools. It will also soon become clear that writing An interpretation for a "real" programming language will be an excessive one task. We should therefore simplify the task a part by making our own little one programming language Asp. We will look into this in the following and other elements included in the assignment.

1.3 Interpreter, compilers and the like

Many who start the course INF2100 have hardly a full overview what an interpretation is and what function it has in connection with one programming language. This will hopefully be much clearer during course, but to put the stage we will give a brief explanation here.

The reason that you at all have interpreters and compilers is that is highly inconvenient to build computers so that they directly from their elect-ronikk can perform a program written in a high-level programming language such as Java, C, C ++, Perl or Python. Instead, computers are

built so that they can perform a limited repertoire of rather simple instructions making it an affordable task to make electronics like can perform these. In return, computers can quickly perform long sequences-Looks out of such instructions, roughly speaking at a rate of 1-3 billion instructions per second.

1.3.1 interpreting

An interpret is a program that reads a given program and builds one internal representation of the program. Then, as indicated in this representation.

There are several advantages to performing programs this way:

- Since the program is stored internally, it is possible to change the program while driving. (In our programming language Asp, it is not possible.)
- Once you have written an interpreter, it is easy to install it other machines, even if they have another processor or another operating system.

The main disadvantage of interpreting programs is slower; how much slower depends on both the language and the quality of Interpreter. However, it is common to expect an interpret user 5-10 times as long as compiled code.

1.3.2 Compiling

Another way to get done programs is to create a compiler like *translates* the program into a corresponding sequence of machine instructions for a given computer. A compiler is thus a program that reads data enter and deliver data from it. The data it reads is a textual program (in it programming language this compiler should translate from), and data It delivers a sequence of machine instructions for the current one machine. These machine instructions will usually be compiled by the compiler A file in a customizable format, given that they can later be copied into one machine and be done.

That set of instructions that a computer can perform directly in electronics, called the machine's machine language, and programs in this language called *machine programs* or *machine code*.

1.3.2.1 Compiling and running Java programs

One of the original ideas of Java was connected to computer networks by one program could be compiled on one machine so that it could be sent over the net to any other machine (such as a so-called *applet)* and be carried out there. To get to this one defined one plan computer called *the Java Virtual Machine* (JVM) and let compilers produce machine code (often called *byte code)* for this machine. The However, there is no computer that has electronics for direct execution such byte code, and the machine in which the application is to be executed must therefore have one program that simulates the JVM and its execution of byte code. We could you say that such a simulation program interprets the machine code

to the JVM machine. Today, for example, most browsers (Firefox, Opera, Chrome and others) embedded such a JVM interpreter in order to perform Java-applets when they get these (done compiled) over the web.

However, such a machine code interpretation is usually slower than if you had translated into "real" machine code and drove it directly on «hardware». Typically, this for Java's byte code can be 2 to 10 times so slow. As Java has become more popular, therefore, it also has need systems that run Java applications faster and it The most common way to do this is to equip JVMs with so-called «Just-In-Time »(JIT) compilation. This means that instead of interpreting the byte-code, translates it on to the current machine code immediately before The program is booted. This can be done for entire programs, or for example for class by class as they are first taken.

You can of course also translate Java programs into more traditional way directly from Java to machine code for one or other actual machine, and such compilers exist and can provide very fast code. If you use Such a compiler, however, loses the advantage of compiling it The program can run on all systems.

1.4 The mission and its quarters

The assignment will be solved in four steps, all of which are mandatory tasks. As Besides this, it may be necessary for example to use tools or delivery of written additional work, but this will also be the case announced in good time.

The entire program can roughly rain from four to five thousand Java lines.
Everything depends on how close you are writing. We give you a quick overview of what
The four parts will contain, but we will return to each of them
at the lectures and in subsequent chapters.

1.4.1 Part 1: Scanner

The first step, part 1, consists in getting Asp's scanner to work. The scanner
is the module that removes comments from the program and then shares
The remaining text in a well-defined sequence of so-called symbols (on
English «tokens»). The symbols are the "words" program is built up
off, such as *names, numbers, keywords,* '+', '> =', '(' and all the other characters and
character combinations that have a particular meaning in the Asp language.

This "pure-cut" sequence of symbols will be the basis of
The rest of the interpreter or compiler will continue working with. Some of
The program for Part 1 will be completed or outlined, and this will be possible
picked up at the specified location.

1.4.2 Part 2: Parser

Part 2 will accept the symbol sequence that is produced by Part 1, and that
The central work here will be to check that this sequence has that shape
A proper Asp program must have (that is, it follows Asp's syntax).

If everything is ok, part 2 must build a syntax tree, a tree structure
of objects that directly represent the current Asp program, ie
how it is composed of "expr" inside "stmt" inside "func def" etc.

1.4.3 Part 3: interpreting expressions

In part 3 you will receive a syntax for an expression and then evaluate it, ie
calculate the earnings value. You must also check that the expression is not incorrect.

1.4.4 Part 4: Full interpreting

The last part is to evaluate all possible Asp programs, ie programs
with functional definitions and sentences with loops, tests and expressions.
In addition, we need to define a library of various predefined features.

1.5 Requirements for collaboration and group affiliation

Normally, two people are supposed to work together to resolve
task. Those who work together should be from the same exercise group
course. One should start to get up early to find one on the group
to cooperate with. It is also allowed to solve the task alone, but this will
of course, give more work. If you have some programming experience,
however, this can be an affordable option.

If you get collaborative problems (like the other "got out"
or "have taken all control"), say from time to class teacher or course management, so
can we see if we can help you get over the "crisis". This has happened before.

1.6 Inspection of filed work

To check that each team team has programmed and tested

out the programs on their own and that both members have been
With the work, the students must be prepared for the group teacher or
Course management requires students who have worked together to be able to
Explain the given parts of the interpreter they have written. With a little bit
support and hints should they, for example, be able to recreate parts of the actual
the program on a blackboard.

Such checks will be conducted on a random basis and in some cases there
The group teacher has seen little to the students and thus has not been in control
along the way with the students' work.

Unfortunately, we have previously disclosed cheating; therefore we see it necessary to
hold such consultations at the end of the course. This is no one really
exam, just a check that you have done the work yourself. Some extra work
For those who are called, it will not be. Once you have programmed
and tested the program, can you interpret them longer, backwards and
with eyes closed.

Another requirement is that all submitted programs are significantly different from
all other submissions. But if you really do the job yourself, you get
automatically a unique program.

Page 16

If anyone is anxious about how much they can cooperate with others outside
his group, we want to say:

> Ideas and techniques can be discussed freely.
>
> Program code should groups write themselves.

Or put another way: Cooperation is good, but copying is wrong!

Note that *no approval of the individual components is finally* before the closing
round with such an oral check and this is probably held once
around the beginning of December.

You can read more about Ifis's rules for copying and collaboration by ob-
ligatoriske tasks on the website https://www.uio.no/studier/eksamen/
Obligatory activities / mn-ifi-oblig.html . Read this to be sure
that you are not suspected of illegal copying.

1.7 Attend training groups

Otherwise, we will encourage students to be active on the weekly
practice groups. The tasks under review are very relevant to
writing of the Asp interpreter If you take a little look at the tasks before
group tutorials, you will probably get much more out of the review.

In the group it is completely accepted to make an unarticipated:

> "I do not understand what this has to do with the case!"

Probably, more people feel the same way, so you do the group one
service. And if you think you have an aha experience, it looks nice
both for yourself and others if you say:

> "Ah, that's the way. . . that's the point! Is that right?"

Since there are many new concepts to come into, it's important to begin
to work with them as early as possible during the semester. Then I'll put it in

head and refresh it sometimes, it will hardly take long before the terms begins to come into place. The compendium says quite a bit about how The task must be solved, but all information about each programbit is not available necessarily gathered in one place.

Finally, the advice of former students: *Start on time!*

Chapter 2

Programming in Asp

The programming language Asp is a programming language that contains the most central parts of Python. The syntax is given by the railway diagram-flows in Figure 2.2 to 2.30 on page 19 - 27 and should be easily understandable for everyone who have programmed a little in Python. A typical example of an Asp program is shown in Figure 2.1 on the next page . [1]

2.1 Driving

Until you have created an Asp interpreter, you can use the reference inter-Pret:

```
$ ~ inf2100 / asp primes.asp
This is the Ifi Asp Interpreter (2017-06-23)
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97 101 103 107 109 113
 127 131 137 139 149 151 157 163 167 173
 179 181 191 193 197 199 211 223 227 229
 233 239 241 251 257 263 269 271 277 281
 283 293 307 311 313 317 331 337 347 349
 353 359 367 373 379 383 389 397 401 409
 419 421 431 433 439 443 449 457 461 463
 467 479 487 491 499 503 509 521 523 541
 547 557 563 569 571 577 587 593 599 601
 607 613 617 619 631 641 643 647 653 659
 661 673 677 683 691 701 709 719 727 733
 739 743 751 757 761 769 773 787 797 809
 811 821 823 827 829 839 853 857 859 863
 877 881 883 887 907 911 919 929 937 941
 947 953 967 971 977 983 991 997
```

This must be run on one of Ifis Linux machines. To protect the code, run interpret as a special user so you must take care of the following:

1) The Asp file must be readable to all. (Remember that this means that the whole The folder structure down to the file must be accessible to everyone.)

2) If you are going to generate a log file (see section 3.1.3 on page 32) must The folder must be writeable to all. It is therefore advisable to add these Asp applications in a separate folder.

1 You can find the source code for this program and also other useful test applications in the folder ~ INF2100 / oblig / test / on all Ifi computers; The folder is also available from any one browser http://inf2100.at.ifi.uio.no/oblig/test/ .

CHAPTER 2 PROGRAMMING IN ASP

primes.asp

```
1
2   # Find all prime numbers up to n
3   # using the technique called "Eratosthenes' sieve".
4
5   n = 1000
6   primes = [true] * (n + 1)
7
8   def find_primes ():
9       i1 = 2
10      while i1 <= n:
11          i2 = 2 * i1
12          while i2 <= n:
13              primes [i2] = false
14              i2 = i2 + i1
15          i1 = i1 + 1
16
17  def w4 (n):
18      if n <= 9:
19          return '       '+ str (n)
20      elif n <= 99:
21          return '     '+ str (n)
22      elif n <= 999:
23          return '' + str (n)
24      else:
25          return str (n)
26
27
28  def list_primes ():
29      n_printed = 0
30      line_buf = ''
31      i = 2
32      while i <= n:
33          if primes [i]:
34              if n_printed> 0 and n_printed% 10 == 0:
35                  print (line_buf)
36                  line_buf = ''
37              line_buf = line_buf + w4 (i)
38              n_printed = n_printed + 1
39          i = i + 1
40      print (line_buf)
41
42
43  find_primes ()
44  list_primes ()
```

Figure 2.1: Example of an Asp program

Page 18

Page 19

2.2 Asp program

As shown in Figure 2.2 , an Asp program consists of a sequence of sentences (<Stmt>). The symbol Eof indicates the end of the file ("end of file").

program

stmt **EEC**

Figure 2.2: Railway Diagram for <application>

2.2.1 Sentences

Figure 2.3 shows what kind of phrases you can use in Asp.

stmt

assignment
expr stmt
if stmt
while stmt
return stmt
pass stmt
func def

Figure 2.3: Rail Diagram <stmt>

2.2.1.1 Assignment

As in most other languages, an assignment application is used to provide variables a value. Since Asp has *dynamic typing*, shall not variables declared in advance. Read more about this in section 2.3.3 on page 27 .

assignment

name **subscription** **=** **expr** **NEWLINE**

Figure 2.4: Rail Diagram <Assignment>

2.2.1.2 Expression sentence

A detached expression is also a legal sentence; This is especially relevant when the expression is a function call.

expr stmt

expr **NEWLINE**

Figure 2.5: Rail Diagram <expr stmt>

2.2.1.3 If statements

If statements are used to choose whether or not to make sentences. See also Section 2.3.1.1 on page 24 for what is legal test values.

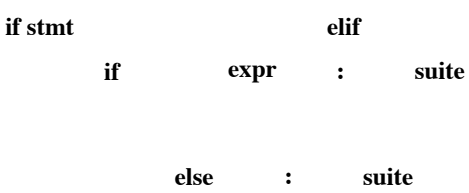


Figure 2.6: Rail Diagram <IF stmt>

2.2.1.4 While statements

While sentences are the only loop direction in Asp. See other av- section 2.3.1.1 on page 24 for what is legal test values.

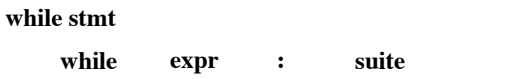


Figure 2.7: Rail Diagram <while stmt>

2.2.1.5 Return statements

Return statements are used to quit the execution of a function and enter one result value.

Figure 2.8: Rail Diagram <return stmt>

2.2.1.6 Pass-phrases

Pass phrases do nothing; they exist only to be put there a sentence is required without anything to be done.

Figure 2.9: Rail Diagram <stepping stmt>

2.2.1.7 of functions

In Asp, functional declarations are considered sentences.

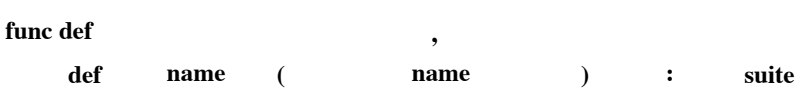


Figure 2.10: Rail Diagram <func def>

2.2.2 Expression

An expression calculates a value. It is defined with the help of quite a few open-air terminals to ensure the presence [2] is the way we want.

Figure 2.11: Rail Diagram <expr>

Figure 2.12: Rail Diagram <and test> and <not test>

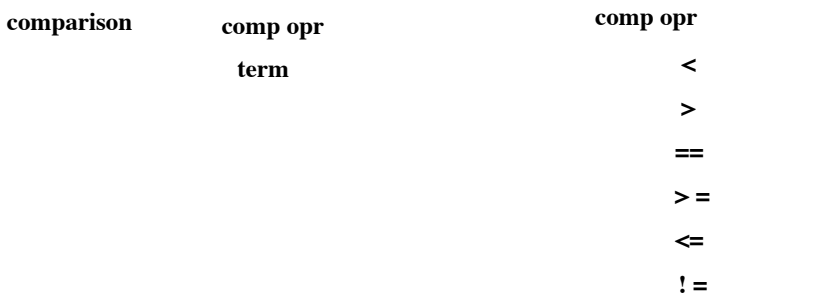


Figure 2.13: Rail Diagram <Comparison> and <comp opr>

Figure 2.14: Rail Diagram <term> and <term opr>

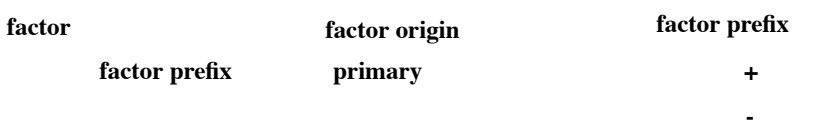


Figure 2.15: Rail Diagram <factor> and <factor prefix>

2.2.2.1 Literals

A literal [3] is a language item indicating a value; for example, enter "123" always the integer value 123.

2 Operators have different precedence, meaning that some operators binds stronger than others. When we writes for example

$$a + b \times c$$

interpreted generally as $a + (b \times c)$ because × normally has higher precedence than the +, ie × binds stronger than +.

3 A literal is other than a constant. A constant is a named value that can not changes while a letter indicates the value itself.

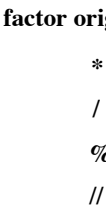


Figure 2.16: Rail Diagram <factor opr>

Figure 2.17: Rail Diagram <primary> and <primary suffix>

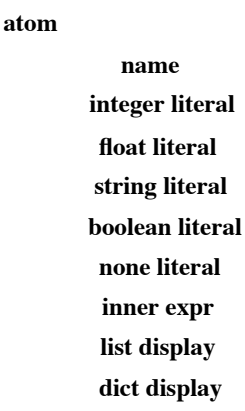


Figure 2.18: Rail Diagram <atom>

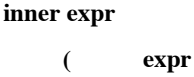


Figure 2.19: Rail Diagram <inner expr>

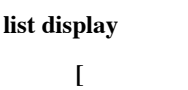


Figure 2.20: Rail Diagram <list display>

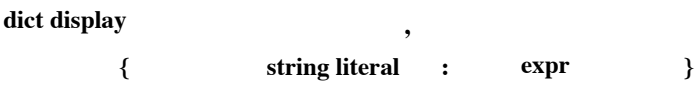


Figure 2.21: Rail Diagram <dict display>

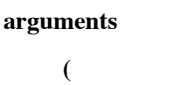


Figure 2.22: Rail Diagram <arguments>

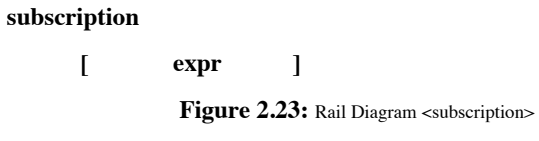


Figure 2.23: Rail Diagram <subscription>

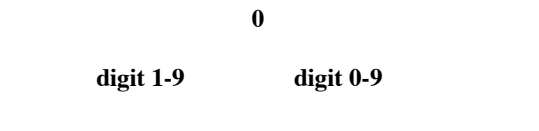


Figure 2.24: Rail Diagram <integer literal>

Figure 2.25: Rail Diagram <float literal>

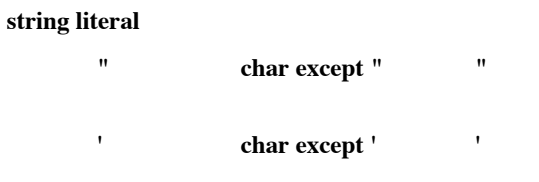


Figure 2.26: Rail Diagram <string literal>

Figure 2.27: Rail Diagram <boolean literal> and <non literal>

2.2.2.2 Name

In Asp, names are used to identify variables and functions.

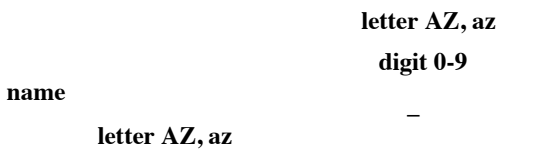


Figure 2.28: Rail Diagram <name>

2.3 Special items in Asp

Some constructions in Asp (and consequently also in Python) may seem unimportant The first time you see them.

2.3.1 Types

Table 2.1 on the next page provides an overview of the types of data in Asp may have.

Type	values	Example
bool	Logical values True and False	Threaten
dict	Value Table	{'Yes': 17, 'No': 0}
float	Flow of	3.14159
func	features	def f (): ...
int	integer	124
list	List of values	[1, 2, "yes"]
none	The "Nothing" value None	none
string	Texts	"Abracadabra"

Table 2.1: Types of Asp

2.3.1.1 Logical values

The language Asp has a logical type with values True and False, but that's a lot more flexible in what it accepts as legal logical values in if statements or in expression. Table 2.2 indicates what is allowed by logical values.

Type	false	Threaten
bool	false	Threaten
dict	{}	non-empty tables
float	0.0	all other values
int	0	all other values
list	[]	non-empty lists
none	none	-
string	""	all other text strings

Table 2.2: Accepted logical values in Asp

Table 2.3 on page 26 shows that we have the usual operators and, or, not for logiske verdier, men resultatet er litt uventet for and og or: de gir ikke svarene True eller False, men returnerer i stedet én av de to operandene, slik som dette:

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

2.3.1.2 Tallverdier

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

Tabell 2.3 på side 26 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.)

in-koder.asp
IN_emner = {} # Start med en tom tabell

```
IN_emner["IN1000"] = "Introduksjon i objektorientert programmering"
IN_emner["IN1010"] = "Objektorientert programmering"
IN_emner["IN1020"] = "Introduksjon til datateknologi"
IN_emner["IN1030"] = "Systemer, krav og konsekvenser"

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

Figur 2.29: Eksempel på bruk av tabeller i Asp

2.3.1.3 Tekstverdier

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

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

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

s_1 + s_2 skjøter sammen to tekster.

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

"abc" * 3 ⇒ "abcabcabc"

2.3.1.4 Lister

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

2.3.1.5 Tabeller

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

2.3.2 Operatorer

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

2.3.2.1 Sammenligninger

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

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

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

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

	Resultat	Comment
+ float	float	Resultatet er v_1

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

2.3.3 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. This kan kompilatoren bruke til å lage sjekke om programmereren har gjort noen feil; i tillegg vil det gi rask kode.

Noen språk har i stedet dynamisk typing som innebærer at variabler ikke

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

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

2.3.4 Indentering av koden

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

suite

NEWLINEINDENT stmt DEDENT

Figur 2.30: Jernbanediagram for ⟨suite⟩

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

2.3.5 Andre ting

2.3.5.1 Kommentarer

Kommentarer skrives slik:

```
# resten av linjen
```

2.3.5.2 Tabulering

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

2.4 Predefinerte deklarasjoner

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

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

Asp

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

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

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

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

Java

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

import java.util.Scanner;

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

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

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

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

Figur 2.31: Indentering i Asp kontra krøllparenteser i Java

Funksjon	Typekrav	Forklaring
float(*v*)	*v* ∈ {int , float , string}	Omformer *v* til en float
input(*v*)	*v* ∈ {string}	Skriver *v* og leser en linje fra tastaturet; resultatet er en string
int(*v*)	*v* ∈ {int , float , string}	Omformer *v* til en int
len(*v*)	*v* ∈ {string , dict , list}	Gir lengden av en string, en dict eller en list; resultatet er en int
print(v_1 ,v_2 ,...)	$v_i \in any$	Skriver ut v_1 ,v_2 ,... med blank mellom; resultatet er none.
str(*v*)	*v* ∈ *any*	Omformer *v* til en string

Tabell 2.4: Asps bibliotek av predefinerte funksjoner

2.4.1 Innlesning

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

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

2.4.2 Utskrift

Asp-programmer kan skrive ut ved å benytte den helt spesielle funksjonen print. Denne prosedyren er den eneste som kan ha vilkårlig mange parametre. Alle parametrene skrives ut, og de kan være av vilkårlig type

(unntatt func). Hvis det er mer enn én parameter, skrives det en blank mellom dem.

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

Page 30

Side 30

Page 31

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.

f.asp 3&4

Interpret

Figur 3.1: Oversikt over prosjektet

3.1 Diverse informasjon om prosjektet

3.1.1 Basiskode

På emnets nettside ligger **2100-oblig-2017.zip** som er nyttig kode å starte med. Lag en egen mappe til prosjektet og legg ZIP-filen der. Gjør så dette:

```
$ cd mappen
$ unzip inf2100-oblig-2017.zip
$ ant
```

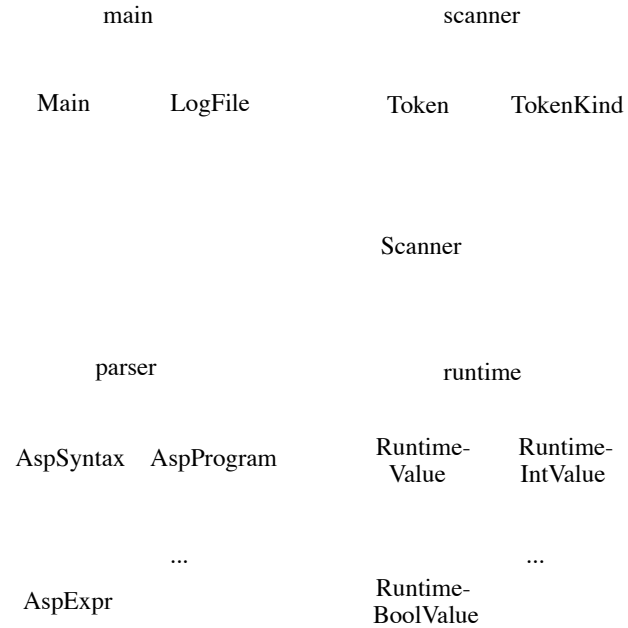
Dette vil resultere i en kjørbar fil asp.jar som kan kjøres slik

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

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

3.1.2 Oppdeling i moduler

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



Figur 3.2: De fire modulene i interpreten

main innholder to sentrale klasser som begge er ferdig programmert:

Main er «hovedprogrammet» som styrer hele interpreteringen.

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

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

parser inneholder (når prosjektet er ferdig) rundt 35 klasser som brukes til å bygge parseringstreet; se avsnitt 3.3 på side 39 .

runtime inneholder et dusin klasser som skal representere verdier av ulike typer under tolkingen av Asp-programmet.

3.1.3 Logging

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

3.1.4 Testprogrammer

Til hjelp under arbeidet finnes diverse testprogrammer:

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

Tabell 3.1: Opsjoner for logging

I mappen ~inf2100/oblig/test/ (som også er tilgjengelig fra en nettleser som http://inf2100.at.ifi.uio.no/oblig/test/) finnes noen Asp-programmer som bør fungere i den forstand at de produserer korrekt utskrift; resultatet av kjøringene er vist i .res-filene.

I mappen ~inf2100/oblig/feil/ (som også er tilgjengelig utenfor Ifi som http://inf2100.at.ifi.uio.no/oblig/feil/) finnes diverse småprogrammer som alle inneholder en feil eller en raritet. Interpreten din bør håndetere disse programmene på samme måte som referanseinterpreten.

3.1.5 På egen datamaskin

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

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

java er en Java-interpret (ofte omtalt som «JVM» (Java virtual machine)

eller «Java RTE» (Java runtime environment)). Om du installerer javac (se neste punkt), får du alltid med java.

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

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

3.1.6 Tegnsett

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

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

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

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

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

3.2 Del 1: Skanneren

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

mini.asp

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

Figur 3.3: Et minimalt Asp-program mini.asp

Programmet vist i figur 3.3 inneholder for eksempel disse symbolene:

navn = "Dag" NEWLINE print (

"Hei," , navn) NEWLINE EoF

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

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

3.2.1 Representasjon av symboler

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

Token.java

```
5
```

```
6  public class Token {
7      public TokenKind kind;
8      public String name, stringLit;
9      public long integerLit;
10     public double floatLit;
11     public int lineNum;
               ...

59 }
```

Figur 3.4: Klassen Token

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

7 «EoF» er en vanlig forkortelse for «End of File».

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

TokenKind.java

```
2
3  public enum TokenKind {
4      // Names and literals:
5      nameToken("name"),
6      integerToken("integer literal"),
7      floatToken("float literal"),
8      stringToken("string literal"),
9
10     // Keywords:
11     andToken("and"),
12     asToken("as"),
           ...

92     // Format tokens:
93     indentToken("INDENT"),
94     dedentToken("DEDENT"),
95     newLineToken("NEWLINE"),
96     eofToken("Eof");
97
98     String image;
99
100    TokenKind(String s) {
101        image = s;
102    }
103
104    public String toString() {
105        return image;
106    }
107 }
```

Figur 3.5: Enum-klassen TokenKind

3.2.2 Skanneren

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

Scanner.java

```
8
9  public class Scanner {
10     private LineNumberReader sourceFile = null;
11     private String curFileName;
12     private ArrayList<Token> curLineTokens = new ArrayList<>();
13     private int indents[] = new int[100];
14     private int numIndents = 0;
15     private final int tabDist = 4;
16
17
18     public Scanner(String fileName) {
19         curFileName = fileName;
20         indents[0] = 0;      numIndents = 1;
21
22         try {
               sourceFile = new LineNumberReader(
```

```
24                                  new InputStreamReader(
25                                      new FileInputStream(fileName),
26                                      "UTF-8"));
27              } catch (IOException e) {
28                  scannerError("Cannot read " + fileName + "!");
29              }
30          }
                ...

370         }
371     }
```

Figur 3.6: Klassen Scanner

De viktigste metodene i Scanner er:

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

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

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

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

curLineNum gir nåværende linjes linjenummer.

findIndent teller antall blanke i starten av den nåværende linjen; se avsnitt 3.2.2.1 . (Denne metoden er privat og kalles bare fra readNextLine.)

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

3.2.2.1 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.)

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

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

Figurene 3.24 og 3.25 viser dette tydelig.

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

Håndtering av indentering
1) Opprett en stakk Indents.
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 T AB -er til blanke.
(c) Tell antall innledende blanke: n.
(d) Hvis n > Indents.top:
i. Push n på Indents.

ii. Legg et 'INDENT'-symbol i curLineTokens.
Hvis n < Indents.top:
Så lenge n < Indents.top:
i. Pop Indents.
ii. Legg et 'DEDENT'-symbol i curLineTokens.
Hvis nå n≠Indents.top, har vi indeteringsfeil.
4) Etter at siste linje er lest:
(a) For alle verdier på Indents som er > 0 , legg et 'DEDENT'-symbol i curLineTokens.

Side 36

Page 37

mini.asp

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

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

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

3.2.2.2 Omforme T AB -er til blanke

En kildekodelinje vil typisk starte med blanke og/eller T AB -tegn. To kunne bestemme indenteringen av en linje (dvs hvor mange blanke den starter med), må vi derfor omforme alle de innledende T AB -ene til det riktige antall blanke etter følgende algoritme: [8]

Omforming av T AB -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 T AB , erstatt den med $4 - (n \bmod 4)$ blanke; øk *n* tilsvarende.
Ved alle andre tegn avsluttes denne løkken.

3.2.3 Logging

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

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

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

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

For å demonstrere hva som ønskes av testutskrift, har jeg laget både et minimalt og litt større Asp-program; se figur 3.7 og figur 3.23 på side 57 . Når interpreten vår kjøres med opsjonen -testscanner, skriver den ut

sidene 57 – 58 sjonen vist i henholdsvis figur 3.7 og figur 3.24 til 3.25 på

8 Denne algoritmen ser bare på de innledende blanke og T AB -er for det er bare de som er interessante når det gjelder å bestemme indenteringen av en linje.

3.2.4 Mål for del 1

Mål for del 1

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

3.3 Del 2: Parsering

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

sjekke at programmet er korrekt i henhold til språkdefinisjonen (dvs grammatikken, ofte kalt syntaksen) og

lage et tre som representerer programmet.

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

3.3.1 Implementasjon

Aller først må det defineres en klasse per ikke-terminal («firkantene» i grammatikken), og alle disse må være subklasser av AspSyntax. Klassene må inneholde tilstrekkelige deklarasjoner til å kunne representere ikke-terminalen. Som et eksempel er vist klassen AspAndTest som representerer 〈and test〉 ; se figur 3.9 på side 41 .

Et par ting verdt å merke seg:

De fem ikke-terminalene 〈letter AZ,az〉 , 〈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 valgt mellom ulike andre ikke-terminaler (som f eks 〈stmt〉 og 〈atom〉) bør implementeres som en abstrakt klasse, og så bør alternativene være sub-klasser av denne abstrakte klassen.

3.3.2 Parseringen

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

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

i alle sub-klassene av AspSyntax. Den skal parsere «seg selv» og lagre dette i et objekt; se for eksempel AspAndTest.parse i figur 3.9 på side 41 . (Metodene test og skip er nyttige i denne sammenhengen; de er definert i parser.AspSyntax-klassen.)

3.3.2.1 Tvetydigheter

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

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

starter med et 〈name〉 , så den kan være én av to:

1) 〈assignment〉

Page 40

KAPITTEL 3 PROSJEKTET

mini.asp

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

Figur 3.8: Syntakstreet laget utifra testprogrammet mini.asp

Side 40

Page 41

AspAndTest.java

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

52 }
```

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

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

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

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

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

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

3.3.4 Logging

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

Våre to vanlige testprogram vist i henholdsvis figur 3.3 på side 34 og figur 3.23 på side 57 vil produsere loggfilene i figurene 3.10 – 3.11 og figurene 3.26 til 3.32 på side 59 og etterfølgende; disse loggfilene lages når interpreten kjøres med opsjonen -logP eller -testparser.

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

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

i hver subklasse av AspSyntax.

Mål for del 2

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

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.

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

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

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

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

3.4 Del 3: Evaluering av uttrykk

Den tredje delen 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 53 og klassen RuntimeReturnValue i avsnitt 3.5.4.3 på side 55 .)

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

3.4.1 Verdier

Alle uttrykk skal gi en *verdi* som svar, og siden Asp har dynamisk typing og derfor skal sjekke typene under kjøring, må verdiene også inneholde en angivelse av verdiens type og dermed hvilke operasjoner som er lov for den. Dette løses enklest ved å benytte objektorientert programmering og la alle verdiene skal være av en subklasse av runtime.RuntimeValue som er vist i figur 3.14 på side 46 , slik for eksempel RuntimeBoolValue som implementerer verdiene til False og True (se figur 3.15 på side 47).

3.4.2 Metoder

RuntimeValue og dens subklasser har tre ulike grupper metoder.

3.4.2.1 Metoder med informasjon

Følgende virtuelle metoder gir informasjon om den aktuelle verdien:

typeName gir navnet på verdiens type.

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

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

3.4.2.2 Metoder med Java-verdier

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

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

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.

AspAndTest.java

```
1   package no.uio.ifi.asp.parser;
2
```

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

Figur 3.13: Klassen AspAndTest

RuntimeValue.java

```
7
8  public abstract class RuntimeValue {
9      abstract protected String typeName();
10
11     public String showInfo() {
12         return toString();
13     }
14
15     // For parts 3 and 4:
16
17     public boolean getBoolValue(String what, AspSyntax where) {
18         runtimeError("Type error: "+what+" is not a Boolean!", where);
19         return false;       // Required by the compiler!
```

```
20      }
21      public double getFloatValue(String what, AspSyntax where) {
22          runtimeError("Type error: "+what+" is not a float!", where);
23          return 0.0;      // Required by the compiler!
24      }
25
26      public long getIntValue(String what, AspSyntax where) {
27          runtimeError("Type error: "+what+" is not an integer!", where);
28          return 0;     // Required by the compiler!
29      }
30
31      public String getStringValue(String what, AspSyntax where) {
32          runtimeError("Type error: "+what+" is not a text string!", where);
33          return null;      // Required by the compiler!
34      }
35
36      // For part 3:
37
38      public RuntimeValue evalAdd(RuntimeValue v, AspSyntax where) {
39          runtimeError("'+' undefined for "+typeName()+"!", where);
40          return null;      // Required by the compiler!
41      }
42          ...

99      public RuntimeValue evalNot(AspSyntax where) {
100         runtimeError("'not' undefined for "+typeName()+"!", where);
101         return null;      // Required by the compiler!
102     }
            ...

124     // General:
125
126     public static void runtimeError(String message, AspSyntax where) {
127         String m = "Asp runtime error on line " + where.lineNum + ": " + message;
128         Main.error(m);
129     }
130
131     // For part 4:
132
133     public void evalAssignElem(RuntimeValue inx, RuntimeValue val, AspSyntax where) {
134         runtimeError("subscription undefined for "+typeName()+"!", where);
135     }
136
137     public RuntimeValue evalFuncCall(ArrayList<RuntimeValue> actualParams,
138                                      AspSyntax where) {
139         runtimeError("'Function call (...)' undefined for "+typeName()+"!", where);
140         return null;      // Required by the compiler!
141     }
142 }
```

Figur 3.14: Klassen RuntimeValue

Side 46

Page 47

3.4 DEL 3: EVALUERING AV UTTRYKK

RuntimeBoolValue.java

```
1   package no.uio.ifi.asp.runtime;
2
3   import no.uio.ifi.asp.parser.AspSyntax;
4
5   public class RuntimeBoolValue extends RuntimeValue {
6       boolean boolValue;
7
8       public RuntimeBoolValue(boolean v) {
9           boolValue = v;
10      }
11
12
13      @Override
14      protected String typeName() {
15          return "boolean";
16      }
```

```
17
18
19          @Override
20          public String toString() {
21              return (boolValue ? "True" : "False");
22          }
23
24
25          @Override
26          public boolean getBoolValue(String what, AspSyntax where) {
27              return boolValue;
28          }
29
30
31          @Override
32          public RuntimeValue evalEqual(RuntimeValue v, AspSyntax where) {
33              if (v instanceof RuntimeNoneValue) {
34                  return new RuntimeBoolValue(false);
35              } else {
36                  return new RuntimeBoolValue(
37                          boolValue == v.getBoolValue("== operand",where));
38              }
39          }
40
41
42          @Override
43          public RuntimeValue evalNot(AspSyntax where) {
44              return new RuntimeBoolValue(! boolValue);
45          }
46
47
48          @Override
49          public RuntimeValue evalNotEqual(RuntimeValue v, AspSyntax where) {
50              if (v instanceof RuntimeNoneValue) {
51                  return new RuntimeBoolValue(true);
52              } else {
53                  return new RuntimeBoolValue(
54                          boolValue != v.getBoolValue("!= operand",where));
55              }
56          }
57      }
```

Figur 3.15: Klassen RuntimeBoolValue

3.4.2.3 Metoder for Asp-operatorer

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

Operator	Implementasjon	Resultat
a + b	evalAdd	a + b
a / b	evalDivide	Flyttallsverdien a / b
a == b	evalEqual	Er a = b?
a > b	evalGreater	Er a > b?
a >= b	evalGreaterEqual	Er a ≥ b?
a // b	evalIntDivide	⌊a / b⌋
a < b	evalLess	Er a < b?
a <= b	evalLessEqual	Er a ≤ b?
a % b	evalModulo	a mod b
a * b	evalMultiply	a × b

- a	evalNegate	– a
not a	evalNot	¬ a
a != b	evalNotEqual	Er a = b?
+ a	evalPositive	+ a
a - b	evalSubtract	a – b

Tabell 3.2: Asp-operatorer og deres implementasjonsmetode

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

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

evalLen brukes av biblioteksfunksjonen len; se avsnitt 3.5.5 på side 56 .

evalSubscription benyttes for å hente et element med angitt index fra en liste.

evalAssignElem plasserer en verdi i en liste eller tabell.

evalFuncCall utfører et kall på en funksjon.

3.4.3 Sporing av kjøringen

For enkelt å kunne sjekke at beregningen av uttrykk og deluttrykk skjer korrekt, er det vanlig å legge inn en mulighet for sporing (på engelsk «Tracing»). I vår interpret skjer dette ved å benytte opsjonen -testexpr. Da vil main.Main.doTestExpr kalle på log.traceEval for hvert uttrykk den beregner.

3.4.4 Et eksempel

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

mini-expr.asp

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

Figur 3.16: Noen enkle Asp-uttrykk

Når vi kjører slike filer med kommendoen «java -jar asp.jar -testexpr mini-expr.asp», blir resultatet som vist i figurene 3.17 og 3.19 på side 51 .

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

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

Mål for del 3

Programmet skal implementere evaluering at uttrykk

uten navn (dvs uten variabler og funksjoner), resultatet av testuttrykket skal gi sporingsinformasjon som vist i figurene 3.17 og 3.19 på side 51 .

expressions.asp

```
1  "Testing integer expressions:"
2  42
3  -1017
4  -2 + 5 * 7
5  100 % (-2 % (+2 * 5) + 18)
6  1234567890 // 1000000
7
8  "Testing float expressions:"
9  42.0
10 -1017.1
11 -2 + 5.0 * 7
12 100 % (-2 % (+2 * 5.0) + 18)
13 1234567890.0 // 1000000.0
14
15 "Testing string expressions"
16 "Abc"
17 "α " + "-"*5 + " ω "
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  "a" != "b" > "b" != 99
27  3.14 > 2.718281828459045 > 0
28
29  "Testing lists"
30  [[
31  [22, "w", [-1,+1], 3.14159265]
32  [101,102,103][1]
33  [-1,0,1]*(2)
34  "Abcdef"[0]
35
36  "Testing dictionaries"
37  {"A": "a", "B": 1+2}
38  {"Ja": 1, "Nei": 0}["Ja"]
```

Figur 3.18: Noen litt mer avanserte Asp-uttrykk

Side 50

Page 51

3.4 DEL 3: EVALUERING AV UTTRYKK

```
1   PP> "Testing integer expressions:" ==>
2   Trace line 1: 'Testing integer expressions:'
3   PP> 42 ==>
4   Trace line 2: 42
5   PP> - 1017 ==>
6   Trace line 3: -1017
7   PP> - 2 + 5 * 7 ==>
8   Trace line 4: 33
9   PP> 100 % (- 2 % (+ 2 * 5) + 18) ==>
10  Trace line 5: 22
11  PP> 1234567890 // 1000000 ==>
12  Trace line 6: 1234
13  PP> "Testing float expressions:" ==>
14  Trace line 8: 'Testing float expressions:'
15  PP> 42.000000 ==>
16  Trace line 9: 42.0
17  PP> - 1017.100000 ==>
18  Trace line 10: -1017.1
19  PP> - 2 + 5.000000 * 7 ==>
20  Trace line 11: 33.0
21  PP> 100 % (- 2 % (+ 2 * 5.000000) + 18) ==>
22  Trace line 12: 22.0
23  PP> 1234567890.000000 // 1000000.000000 ==>
24  Trace line 13: 1234.0
25  PP> "Testing string expressions" ==>
26  Trace line 15: 'Testing string expressions'
27  PP> "Abc" ==>
28  Trace line 16: 'Abc'
29  PP> " α " + "-" * 5 + " ω " ==>
30  Trace line 17: ' α ----- ω '
31  PP> "Testing boolean expressions" ==>
32  Trace line 19: 'Testing boolean expressions'
33  PP> not False ==>
34  Trace line 20: True
35  PP> "To be" or "not to be" ==>
36  Trace line 21: 'To be'
37  PP> "Yes" and 3.140000 ==>
38  Trace line 22: 3.14
39  PP> False or True or 144 or "?" ==>
40  Trace line 23: True
41  PP> "Testing comparisons" ==>
42  Trace line 25: 'Testing comparisons'
43  PP> 1 <= 2 <= 3 ==>
44  Trace line 26: True
```

```
PP> "a" != 27 "False" != 99 ==>
PP> 3.140000 > 2.718282 > 0 ==>
Trace line 28: True
PP> "Testing lists" ==>
Trace line 30: 'Testing lists'
PP> [] ==>
Trace line 31: []
PP> [22, "w", [-1, +1], 3.141593] ==>
Trace line 32: [22, 'w', [-1, 1], 3.14159265]
PP> [101, 102, 103][1] ==>
Trace line 33: 102
PP> [-1, 0, 1] * (2) ==>
Trace line 34: [-1, 0, 1, -1, 0, 1]
PP> "Abcdef"[0] ==>
Trace line 35: 'A'
PP> "Testing dictionaries" ==>
Trace line 37: 'Testing dictionaries'
PP> {"A":"a", "B":1 + 2} ==>
Trace line 38: {'A': 'a', 'B': 3}
PP> {"Ja":1, "Nei":0}["Ja"] ==>
Trace line 39: 1
```

Figur 3.19: Sporingslogg fra kjøring av expressions.asp

3.5 Del 4: Evaluering av setninger og funksjoner

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

3.5.1 Setninger

Evaluering av setninger er rimelig rett-frem å implementere.

3.5.1.1 Tilordningssetninger

Disse setningene er omtalt i avsnitt 3.5.3 på side 54 .

3.5.1.2 Uttrykkssetninger

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

3.5.1.3 If-setninger

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

3.5.1.4 While-setning

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

3.5.1.5 Return-setning

Denne setningen blir forklart i avsnitt 3.5.4.3 på side 55 .

3.5.1.6 Pass-setning

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

3.5.1.7 Funksjonsdefinisjoner

Dette omtales i avsnitt 3.5.4 på side 54 .

3.5.2 Variabler

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

RuntimeScope.java

```
1  package no.uio.ifi.asp.runtime;
2
3  // For part 4:
4
5  import java.util.HashMap;
6
7  import no.uio.ifi.asp.parser.AspSyntax;
8
9  public class RuntimeScope {
10     RuntimeScope outer;
11     HashMap<String,RuntimeValue> decls = new HashMap<>();
12
13     public RuntimeScope() {
14         outer = null;
15     }
16
17
18     public RuntimeScope(RuntimeScope oScope) {
19         outer = oScope;
20     }
21
22
23     public void assign(String id, RuntimeValue val) {
24         decls.put(id, val);
25     }
26
27
28     public RuntimeValue find(String id, AspSyntax where) {
29         RuntimeValue v = decls.get(id);
30         if (v != null)
31             return v;
32         if (outer != null)
33             return outer.find(id, where);
34
35         RuntimeValue.runtimeError("Name " + id + " not defined!", where);
36         return null;      // Required by the compiler.
37     }
38 }
```

Figur 3.20: Klassen RuntimeScope

3.5.2.1 Skop

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

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

find finner frem verdien tilrodnet et gitt navn.

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

1) Biblioteket, som inneholder predefinerte funksjoner; se avsnitt 3.5.5 på side 56 .

2) Hovedprogrammet, som initielt er tomt.

Siden vil hvert funksjonskall opprette et nytt skop for sine parametre og variabler; se avsnitt 3.5.4.2 på side 55 .

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

er deklarert i eget skop men også i ytre skop. Se for eksempel på figur 2.1 på side 18 :

n_printed defineres i funksjonens skop i linje 29, så bruken i linje 34 refererer til denne lokale variabelen.

n defineres i hovedprogrammets skop i linje 5, så bruken i linje 32 refererer dit.

str er predefinert i biblioteket, så bruken i linje 23 angir den definisjonen.

Legg forøvrig merke til at definisjoner kan skygge for hverandre. In addition til den globale definisjonen av ni linje 5 er n også definert som parameter i linje 17. Bruken i linje 19 gjelder da definisjonen i det nærmeste skopet, med andre ord den i linje 17. [11]

3.5.3 Tilordning til variabler

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

3.5.3.1 Tilordning til mer kompliserte variabler

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

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

Da må vi gjøre følgende:

1) For alle indekser unntatt den siste: Slå opp i listen eller tabellen på samme måte som da vi beregnet uttrykk.

2) Kall på siste verdi sin evalAssignElem for å foreta tilordningen.

3.5.4 Funksjoner

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

3.5.4.1 Definisjon av funksjoner

I Asp (som i Python) regnes en funksjonsdeklarasjon som en form for

tilordning, så definisjonen av GCD i figur 3.23 på side 57 kan betraktes som noe à la

GCD = 〈funksjonsverdi〉

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

11 Kan vi i det hele tatt i w4 få tilgang til den globale n. Svaret er nei.

Side 54

Page 55

3.5.4.2 Kall på funksjoner

Funksjonskall skjer i klassen til 〈arguments〉, og følgende bør skje: 12

1) De aktuelle parametrene 13 beregnes og legges i en ArrayList.

2) Funksjonens evalFuncCall kalles med to parametre:

   (a) listen med aktuelle paramtre

   (b) kallets sted i syntakstreet (for å kunne gi korrekte feilmeldinger)

3) RuntimeFunc.evalFuncCall må så ta seg av kallet:

   (a) Sjekk at antallet aktuelle parametre er det samme som antallet formelle parametre.

   (b) Opprett et nytt RuntimeScope-objekt. Dette skopets outer skal være det skopet der funksjonen ble deklarert.

   (c) Gå gjennom parameterlisten og tilordne alle de aktuelle parameterverdiene til de formelle parametrene.

   (d) Kall funksjonens runFunction (med det nye skopet som parameter) slik at den kan utføre innmaten av funksjonen.

Og det er stort sett det som skal til.

3.5.4.3 return -setningen

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

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

RuntimeReturnValue.java

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

Figur 3.21: Klassen RuntimeReturnValue

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

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

13 Betegnelsen aktuell parameter angir de parametrene som står i funksjons *kallet* , mens formell parameter betegner de parametrene som står i funksjons *definisjonen* .

3.5.5 Biblioteket

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

RuntimeLibrary.java

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

Figur 3.22: Fra klassen RuntimeLibrary

3.5.6 Sporing

Også setninger skal kunne spores ved at alle setningen kaller på AspSyntax.trace for å fortelle hva de gjør; se for eksempel figur 3.34 på side 66 .

Mål for del 4

Programmet skal implementere resten av Asp slik at programmer som vist i figur 3.23 på neste side gir sporingsinformasjon som vist i figur 3.34 på side 66 når input er tallene 30 og 75.

3.6 Et litt større eksempel

gcd.pas

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

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

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

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

Page 58


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

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

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

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

```
            <suite>
10:         return GCD(n, m % n)
              <stmt>
                <return stmt>
                  <expr>
                    <and test>
                      <not test>
                        <comparison>
                          <term>
                            <factor>
                              <primary>
                                <atom>
                                  <name>
                                  </name>
                                </atom>
                                <primary suffix>
                                  <arguments>
                                    <expr>
                                      <and test>
                                        <not test>
                                          <comparison>
                                            <term>
                                              <factor>
                                                <primary>
                                                  <atom>
                                                    <name>
                                                    </name>
                                                  </atom>
                                                </primary>
                                              </factor>
                                            </term>
                                          </comparison>
                                        </not test>
                                      </and test>
                                    </expr>
                                    <expr>
                                      <and test>
                                        <not test>
                                          <comparison>
                                            <term>
                                              <factor>
                                                <primary>
                                                  <atom>
                                                    <name>
                                                    </name>
                                                  </atom>
                                                </primary>
                                                <factor opr>
                                                </factor opr>
                                                <primary>
                                                  <atom>
                                                    <name>
                                                    </name>
                                                  </atom>
                                                </primary>
                                              </factor>
                                            </term>
                                          </comparison>
                                        </not test>
                                      </and test>
                                    </expr>
                                  </arguments>
                                </primary suffix>
                              </primary>
                            </factor>
                          </term>
                        </comparison>
                      </not test>
                    </and test>
                  </expr>
                </return stmt>
              </stmt>
11:
12: v1 = int(input("A number: "))
            </suite>
```

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

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

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


```
226       <assignment>
227         <name>
            </name>
```

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

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


```
301             <not test>
302               <comparison>
303                 <term>
304                   <factor>
305                     <primary>
306                       <atom>
307                         <name>
308                         </name>
309                       </atom>
```

```
                    <primary suffix>
                      <arguments>
                        <expr>
                          <and test>
                            <not test>
                              <comparison>
                                <term>
                                  <factor>
                                    <primary>
                                      <atom>
                                        <name>
                                        </name>
                                      </atom>
                                    </primary>
                                  </factor>
                                </term>
                              </comparison>
                            </not test>
                          </and test>
                        </expr>
                        <expr>
                          <and test>
                            <not test>
                              <comparison>
                                <term>
                                  <factor>
                                    <primary>
                                      <atom>
                                        <name>
                                        </name>
                                      </atom>
                                    </primary>
                                  </factor>
                                </term>
                              </comparison>
                            </not test>
                          </and test>
                        </expr>
                      </arguments>
                    </primary suffix>
                  </primary>
                </factor>
              </term>
            </comparison>
          </not test>
        </and test>
      </expr>
    </assignment>
  </stmt>
  16: print('GCD('+str(v1)+','+str(v2)+') =', res)
  <stmt>
    <expr stmt>
      <expr>
        <and test>
          <not test>
            <comparison>
              <term>
                <factor>
                  <primary>
                    <atom>
                      <name>
                      </name>
                    </atom>
                    <primary suffix>
                      <arguments>
                        <expr>
```

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


```
                          <and test>
                            <not test>
                              <comparison>
                                <term>
                                  <factor>
                                    <primary>
                                      <atom>
                                        <string literal>
                                        </string literal>
                                      </atom>
                                    </primary>
                                  </factor>
                                  <term opr>
                                  </term opr>
                                  <factor>
                                    <primary>
```

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

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


```
451                                                   </primary>
452                                                 </factor>
453                                               </term>
454                                             </comparison>
455                                           </not test>
456                                         </and test>
457                                       </expr>
458                                     </arguments>
459                                   </primary suffix>
460                                 </primary>
461                               </factor>
462                               <term opr>
463                               </term opr>
464                               <factor>
465                                 <primary>
466                                   <atom>
467                                     <string literal>
468                                     </string literal>
469                                   </atom>
470                                 </primary>
471                               </factor>
```

```
                                    </comparison>
                                </not test>
                            </and test>
                        </expr>
                        <expr>
                            <and test>
                                <not test>
                                    <comparison>
                                        <term>
                                            <factor>
                                                <primary>
                                                    <atom>
                                                        <name>
                                                        </name>
                                                    </atom>
                                                </primary>
                                            </factor>
                                        </term>
                                    </comparison>
                                </not test>
                            </and test>
                        </expr>
                    </arguments>
                </primary suffix>
            </primary>
        </factor>
    </term>
</comparison>
```

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

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

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


```
Trace line 6: def GCD
Trace line 12: Call function input with params ['A number: ']
Trace line 12: Call function int with params ['30']
```

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

Figur 3.34: Sporingslogg fra kjøring av gcd.asp

Kapittel 4

Programmeringsstil

4.1 Suns anbefalte Java-stil

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

4.1.1 Klasser

Hver klasse bør ligge i sin egen kildefil; unntatt er private klasser som kun brukes av en vanlig klasse.

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

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

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

2) Alle import-spesifikasjonene.

3) JavaDoc-kommentar for klassen. (JavaDoc er beskrevet i avsnitt 5.1 på side 71 .)

4) Selve klassen.

4.1.2 Variabler

Variabler bør deklareres én og én på hver linje:

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

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

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

Tabell 4.1: Suns forslag til navnevalg i Java-programmer

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

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:

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

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

4.1.4 Navn

Navn bør velges slik det er angitt i tabell 4.1 .

4.1.5 Utseende

4.1.5.1 Linjelengde og linjedeling

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

En linje som er for lang, bør deles

etter et komma eller

før en operator (som + eller &&).

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

Side 68

Page 69


```
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.

Side 69

Page 70

KAPITTEL 4 PROGRAMMERINGSSTIL

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
```

Side 70

Page 71

Kapittel 5

Documentation

5.1 JavaDoc

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

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

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

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

5.1.1 Hvordan skrive JavaDoc-kommentarer

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

En JavaDoc-kommentarer for en klasse ser slik ut:

```
1  /**
2   * Én setning som kort beskriver klassen
3   * Mer forklaring
4   *       ⋮
                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 ... 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 TEX. Hovedtanken er at programmer først og fremst skal skrives slik at mennesker kan lese dem; datamaskiner klarer å «forstå» alt så lenge programmet er korrekt. Dette innebærer følgende:

Programkoden og dokumentasjonen skrives som en enhet.

Programmet deles opp i passende små navngitte enheter som legges inn i dokumentasjonen. Slike enheter kan referere til andre enheter.

Programmet skrives i den rekkefølgen som er enklest for leseren å forstå.

Dokumentasjonen skrives i et dokumentasjonsspråk (som L A TEX) og kan benytte alle tilgjengelige typografiske hjelpemidler som figurer, matematiske formler, fotnoter, kapittelinndeling, fontskifte og annet.

Side 72

Page 73

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

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

1) et dokument som kan skrives ut og

2) en kompilerbar kildekode.

5.2.1 Et eksempel

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

1) Skriv kildefilen bubble.w0 (vist i figur 5.2 og 5.3). Dette gjøres med en vanlig tekstbehandler som for eksempel Emacs.

2) Bruk programmet weave0 [14] til å lage det ferdige dokumentet som er vist i figur 5.4 – 5.7 :

```
$ weave0 -lc -e -o bubble.tex bubble.w0
$ ltx2pdf bubble.tex
```

3) Bruk tangle0 til å lage et kjørbart program:

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

14 Dette eksemplet bruker Dags versjon av lesbar programmering kalt web ₀ ; for mer informasjon, se http://dag.at.ifi.uio.no/public/doc/web0.pdf .

bubble.w0 del 1

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

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

\begin{document}
\maketitle

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

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


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

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

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

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

Side 74

Page 75

bubble.w0 del 2

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

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

Page 76

KAPITTEL 5 DOCUMENTATION

Bubble sort

Dag Langmyhr
Department of Informatics
University of Oslo
dag@ifi.uio.no
May 12, 2017

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:

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

#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 a (p.2). It is used in #1 (p.1).)

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

File: bubble.w0 page 1

Figur 5.4: «Lesbar programmering» — utskrift side 1

Side 76

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:

#4 ⟨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

Figur 5.6: «Lesbar programmering» — utskrift side 3

Meta symbols

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

META SYMBOLS page 4

Figur 5.7: «Lesbar programmering» — utskrift side 4

Side 79

Page 81

Register

Side 82