Enigma

Et programmeringssprog

Forfatter:
Niels Serup X. Y
Vejleder:
A B C

Fag:
Programmering
Uddannelsessted:
N M

25. maj 2010

Indholdsfortegnelse

Indledning

Når man laver et nyt program, der skal kunne køre på en computer, bruger man i langt de fleste tilfælde et eller flere programmeringssprog. Programmeringssprog tillader, at ens kode kan være mere abstrakt i forhold til ren maskinkode og assembly. I løbet af de sidste cirka 25 år er der kommet mange nye programmeringsprog, hver med sine fordele og ulemper. Størstedelen af de programmeringssprog, der bliver brugt i dag — C, C++, Java, Python, Ruby, PHP etc.[1] — minder rent syntaxmæssigt en del om hinanden, og selv om Python og Ruby har tilføjet anderledes måder at gøre visse ting på, virker de mest udbredte sprog stadigvæk ret ens. Det betyder selvfølgelig, at det er nemmere at lære nye sprog, men det er også lidt kedeligt. Området trænger til lidt fornyelse.

Mit problem er altså, at der mangler adspredelse indenfor programmeringssprogssyntax.

Man kan tænke sig frem til flere løsninger. En nem løsning ville være at lave et program, der kunne hjælpe brugeren af programmet til at finde nye, esoteriske programmeringssprog ved brug af internettet og indtastede stikord. En anden løsning ville være at lave et program, der kunne generere semitilfældige syntaxvariationer. En tredje løsning ville være at lave et nyt programmeringssprog.

Jeg har valgt at lave et nyt programmeringssprog fra bunden og implementere en parser for det i Java. Dette er muligt indenfor projektets tidsramme, idet jeg begrænser mig til at lave et *simpelt* sprog – og altså intet i nærheden af fx Python. Jeg kalder dette sprog for...

ENIGMA

Sproget

En enigma er en svær gåde. Det er ENIGMA ikke. Hvis sproget var meget mere simpelt end det er nu, ville man ikke kunne bruge det til noget. Dette er dog ikke tilfældet, og det er fuldt ud muligt at beskrive velfungerende programmer med dets ellers begrænsede syntax.

Syntax

Syntaxen i ENIGMA er lidt anderledes end normen indenfor programmeringssprog dikterer. Mens de fleste sprog bruger typen

```
variabelnavn = værdi;
```

til at "gemme" værdier i en variabel, og

```
funktionsnavn(variabler);
```

[1] Langpop [2010]

til at køre en funktion, vender Enigma det om. I Enigma gemmer man værdier ved at skrive

```
værdi = variabelnavn;
```

...og man kører en funktion ved at skrive

```
variabler ! funktion;
```

Dette er det mest grundlæggende i Enigma, og egentlig er der ikke meget mere end det. 2 andre operatører, "|" og "*", findes også, men de er mindre vigtige og eksisterer kun for at gøre visse ting nemmere. Strengt taget er de ikke nødvendige.

I Enigma separares variabler med mellemrum. Hvis man vil sende 3 argumenter til en funktion, er det altså efter denne syntax:

```
var1 var2 var3 ! kode;
```

Og hvis man vil gemme 5 værdier i en listevariabel, foregår det sådan her:

```
var1 var2 var3 var4 var5 = liste;
```

Dette kan bruges som "genveje". De to følgende eksempler resulterer i det samme:

```
#Den lange
var1 var2 var3 ! kode1;
var1 var2 var3 ! kode2;
var1 var2 var3 ! kode3;

#Den korte
var1 var2 var3 = liste;
liste ! kode1;
liste ! kode2;
liste ! kode3;
```

I visse tilfælde kan "!"-operatøren godt have lidt mangler, hvilket resulterer i lang kode:

```
a ! b = c;
d c ! e = c;
```

Dette løser "|"-operatøren. Den agerer som en samler, der linker to statements sammen. Næste kodesnippet gør det samme som den før:

```
a ! b | d temp ! e = c;
```

Her betyder `temp` den værdi, som `b` returnerer og som før blev gemt i `c` som et mellemtrin.

Det er ikke nødvendigt med whitespace mellem variabler og operatører, og `x ! y` er altså det samme som `x!y`.

Den fjerde og sidste operatør, "*" fungerer ligesom "=", men mens "=" fungerer lokalt, virker "*" globalt. Mere om det senere.

Typer

Der er 8 typer variabler i Enigma. Modsat almindelige programmeringssprog er der ingen simple typer såsom `int`, `char` og `float`. Alt er objekter. De 8 typer er:

- CODE
- BUILTINCODE
- STRING
- NUMBER
- BOOLEAN
- LIST
- FILE
- NONE

`CODE` og BUILTINCODE kan modtage og returnere data, men `CODE` kommer fra koden af det Enigmaprogram, der kører, mens `BUILTINCODE` kører direkte fra interpreteren. De inbygedde funktioner er:

str, num, list, bool, code, repr, type, len, clone, slice, loop, open, close, read, write, greater, lesser, equal, and, or, not, act, system, add, subtract, multiply, divide, mod, pow, log, random, abs, round, floor, ceil, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh

Et `CODE`-objekt betegnes sådan:

```
{...kode...}
```

Når man kører en Enigmafil, kører man faktisk bare et `CODE`-objekt.

`STRING` dækker over strenge af tegn. Det betegnes sådan:

```
"tekst..."
```

`NUMBER` dækker over tal. Et tal kan være så langt, som ens computers hukommelse tillader. Internt er det ikke en simpel double, men en kompleks datastruktur. Tal skrives som tal...

`BOOLEAN` dækker over *`true`* og *`false`*. Den bliver brugt meget i Enigma.

`LIST` er en speciel type variabel i Enigma. Man kan ikke eksplicit lave en liste — den kommer af sig selv, når det er nødvendigt. Egentlig er `LIST` hovedsageligt brugt til internt at holde styr på variabler, men man kan sagtens interagere med lister.

`FILE` dækker over filer. Enigma startes altid med 3 filer: stdout, stderr og stdin. En fil kan laves vha. den indbyggede funktion `open`, der tager et filnavn som det første argument.

Sidst og med mindst værdi er `NONE`. Den eksisterer som et abstrakt svar på ingeting. Nogle indbyggede funktioner returnerer denne værdi. Den kan skrives direkte i kode som `none`.

Enigma har en række indbyggede konstanter af forskellige typer. De er:

args, return, temp, stdin, stderr, stdout, zero, true, false, none, @pi, @e, cwd, cpd, fnm

args indeholder argumenter til en funktion når den bliver kaldt, *return* læses af Enigma når noget kode er færdigt, *zero* er en fil der kan skrives til uden der sker noget, *cwd* er navnet på den sti, som Enigma køres fra, *cpd* er navnet på den sti, som ens program ligger i, og *fnm* er stien samt filnavnet på programmet. Resten er selvforklarende.

Funktionslogik

En vigtig del af Enigma er dens funktionsopbygning. For det første er der fuld support for funktioner i funktioner (i funktioner i funktioner etc.). Der er ikke nogen grænser hvad det angår.

Det er også vigtigt at funktioner kan tage argumenter, og i Enigma kan funktioner tage alt fra 0 til uendeligt argumenter — ikke at funktionerne er tvunget til at læse argumenterne.

En central del af funktioner — kodestykker — er at der skal være et slags hierarki. Hvis man har to funktioner, f1 og f2, og f2 er inden i f1, så skal f2 have adgang til variablerne i f1, men f1 skal ikke have adgang til variablerne i f2. Det sidste forhindres også af at alle variabler i en funktions hierarki slettes når funktionen slutter.

Nogle gange kan det dog være et problem, at en funktion ikke gemmer variabler for eftertiden. Ved hjælp af den førnævnte operatør "*" kan man ændre på denne opførsel. Den følgende kode forklarer dette:

```
#x forsvinder
{37 = x;} = kode;
!kode;

#x bliver
{37 * x;} = kode;
!kode;
```

Som genvej kan man også bare putte en "*" foran sin kode (som `*37 = x;`). Dette har samme effekt, men det gælder alle steder i koden.

Det kan være lidt besværligt at skulle trække variabler ud fra `args` i begyndelsen af et nyt kodestykke, og derfor er der defineret en genvej der ser sådan ud:

```
/navn1,navn2/kode...
```

Her er `navn1` den først sendte variabel til funktionen (eller none hvis der ikke er nogen) og navn2 den anden (eller none hvis der ikke er nogen. Ved at bruge "/.../"-genvejen kan man spare flere linjer kode.

De indbyggede funktioner fungerer internt anderledes end brugerdefinerede kodestykker, men umiddelbart kan det godt føles som om de opfører sig ens. Og det er kun godt.

Mindre detaljer

Kommentarer

Kommentarer skrives ved brug af syntaxen:

```
#kommentar
```

If og while

I normale programmeringssprog har især *if* og *while* en hel syntaxdel for dem selv. I Enigma er de bare funktioner.

C-like kode:

```
if (x == 2) {
  kode();
}

while (y == 3) {
  kode();
}
```

Enigmakode:

```
x 2 ! equal | temp kode ! act;

{y 3 ! equal = r; r kode ! act; r = return;} ! loop;
```

Interpreteren

Når man kører et Enigmaprogram skal det ikke kunne ses hvilket sprog interpreteren er skrevet i. Dette er fordi Enigma skal kunne implementeres i flere sprog. Man kan også implementere Enigma i Enigma for den sags skyld.

Da den indtil videre eneste Enigmainterpreter er skrevet i Java, er det den kode der skal gennemgås nu.

Programstruktur

Da Java er et meget objektorienteret sprog, er interpreteren delt op i objekter. Der er 7 klasser:

- EnigmaMain
- EnigmaLogic
- EnigmaBuiltinMethods
- EnigmaVariable
- EnigmaVariableDict
- EnigmaVariableDictList
- EnigmaVariableType

Når Enigma køres, loades EnigmaMain, som så starter EnigmaLogic, som så kører ens program.

Programstart

EnigmaMain består af en statisk main-klasse, der udfører forskellige handlinger baseret på de argumenter man sender til programmet. Den følger GNU-standarderne med hensyn til "command line interfaces"[2]. I sin simpleste form kan det køres ved bare at skrive:

```
java enigma.EnigmaMain sti/til/program.ngm argument1 argumnent2 ... argumentN
```

Hvis der er specificeret et program, loader Enigma EnigmaLogic og kører derefter programmet.

Variabler

I Enigma -implementationen bruges der en speciel klasse til at holde styr på værdier: EnigmaVariable. EnigmaVariable består af et enum — EnigmaVariableType — og en værdi — Object.

[2] GNU [2010]

Disse variabler gemmes i HashMaps (EnigmaVariableDict), så de kan blive associeret med en tekststreng. Disse HashMaps gemmes så i ArrayLists (EnigmaVariableDictList), så der kan komme et hierarki.

Indbyggede metoder

Typen BUILTINCODE dækker i Java-implementationen over de metoder i EnigmaBuiltinMethods, som er skrevet i Java, men som kan køres fra en Enigma -fil. Dette kræver at en streng er linket til funktionen. Umiddelbart er dette ikke muligt i Java, da pointers altid er implicitte. Man kan ikke selv lave en reference til en metode, og derfor må man bede Java om at gøre det for en via dets Reflection API[3].

Når EnigmaLogic startes, kreeres alle referencer til de indbyggede funktioner via en simpel løkke. Alle indbyggede funktioner tager imod en ArrayList som eneste argument. Dette er for at gøre det nemmere at have med at gøre. Kommentarerne ved de indbyggede funktioner følger Javadoc-konventionen[4].

Tal

I stedet for at bruge longs og doubles bruger Enigma Javas egen BigDecimal, der ikke er begrænset i størrelse af andet end brugerens computer. Dette sikrer at der ikke sker overflow errors og at der ikke opstår underlige afrundinger[5].

Logikken

Selve hjertet af Enigma -implementationen ligger i EnigmaLogic. I EnigmaLogic ligger nemlig metoden *parseCode*, som forstår Enigma -kode. Denne funktion skal tage højde for en del, idet Enigma -kode, simpel som den er, trods alt stadig har et syntax.

parseCode får bl.a. en kodestreng som argument, og den kodestreng læser den så tegn for tegn, om end den nogle gange læser det samme tegn to gange og nogle gange springer et tegn over. Denne funktion kan forklares ved hjælp af det følgende stykke pseudekode (meget forsimplet):

$kode = String$
$dict = EnigmaVariableDict$
$vars = EnigmaVariableDictList$
$position = 0$
$vartype = 0$

[3] Sun [2010b]
[4] Se http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javadoc.html#documentationcomments
[5] Sun [2010a]

```
actiontype = 0
while position < code.length() do
  if vartype = 0 then
    if kode[position] er et ligegyldigt tegn then
      position = position + 1
    else if kode[position] er et vigtigt tegn then
      if actiontype = '=' || actiontype = '*' then
        Sæt værdien fra det tidligere variabelsæt til det nuværende.
        Gem de nye værdier i de specificerede variabler.
      else if actiontype = '!' then
        Send data til kode (indbygget eller ej).
      end if
      if kode[position] medfører en handling der udvider ens statement then
        Lav en ny liste til nye variabler.
      end if
      if kode[position] = '=' then
        actiontype = '='
      else if kode[position] = '*' then
        actiontype = '*'
      else if kode[position] = '!' then
        actiontype = '!'
      else if kode[position] = ';' || kode[position] = '|' then
        Reset diverse variabler, så der er plads til at begynde forfra.
      end if
      position = position + 1
    else
      if kode[position] = '{' then
        vartype = 1
      else if kode[position] = '"' then
        vartype = 2
      else
        vartype = 3
      end if
    end if
  else if vartype = 1 then
    Parse kodestreng og gem
  else if vartype = 2 then
    Parse tekststreng og gem
  else if vartype = 3 then
    if streng minder om tal then
      Gem som tal
    else
      Få fat i den variabel, som strengen linker til, og gem.
    end if
  end if
end while
return dict["return"]
```

Programtests

Enigma er lidt komplekst at implementere, og det kan derfor være en god ide at teste det lidt, så man får udryddet de største bugs. Den bedste måde at teste en interpreter på er selvfølgelig ved at køre Enigma -programmer.

Et eksempel er dette lille program, der udregner fakultetstal:

```
{/num/
  1 = res;
  {/n,r/
    r n ! multiply;
    n 1 ! subtract;
    n 1 ! greater = return;
  } = facfunc;
  facfunc num res ! loop;
  res = return;
} = factorial;

5 ! factorial | stdout "5! =" temp ! write;
```

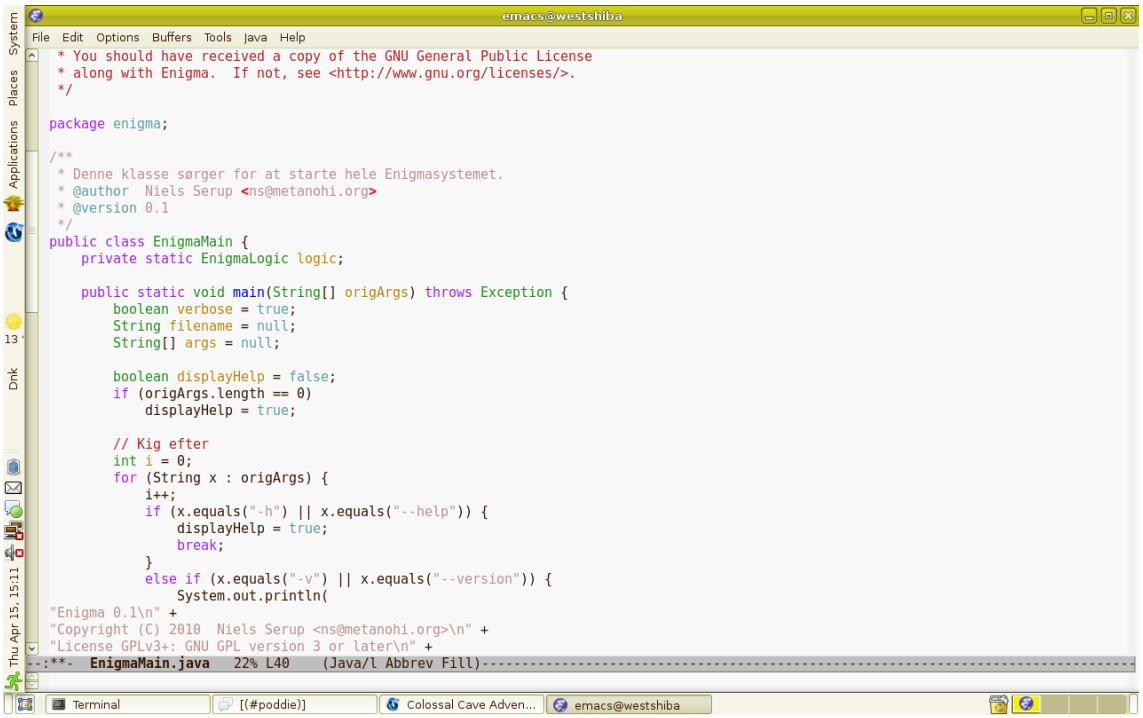
Dette — og mange andre — viste hvad der virkede og hvad der ikke virkede. Faktisk hjalp det. Nogle gange var der fejl i noget, og så kunne man fikse det.

Processen

Før jeg overhovedet begyndte at programmere, skrev jeg en masse testprogrammer, som jeg kørte i mit hovede. Jeg overvejede hvordan syntaxen skulle være og hvilke funktioner der skulle være bygget ind i interpreteren. Først var der kun "=" og "!", og det varede en del tid før "|" og "*" kom til. Selve grundideen fik jeg dog hurtigt styr på, og efter nogle ugers grublen begyndte jeg så at lave programmet.

I programbygningsfasen havde jeg først ikke særlig meget overblik. Tanker som "skal jeg lægge dette i en klasse for sig selv?" og "hvordan skal jeg tjekke om det her passer" fløj gennem mit hovede. Men efter et par dage begyndte koden helt automatisk at få en logisk opdeling, og resten var relativt nemt, selv om det tog noget tid at implementere alle de indbyggede funktioner.

Jeg brugte Emacs til at redigere mine filer.



Her ses et screenshot af Emacs in action.

Brugervejledning

Det er nemt at bruge Enigma . Men det vil måske afskrække nogle at der ikke er noget grafisk. Dette betyder, at folk må lære om kommandolinjen.

Forbedringer

Som Enigma -interpreteren er lige nu, går det ret langsomt med at parse Enigma -programmer. Det burde være relativt nemt at forbedre Enigma på dette punkt. Det er samtidig heller ikke helt umuligt at der er nogle få bugs. De skal selvfølgelig sprøjtes.

Perspektivering

En fordel ved Enigma er at det har potentiale til at ændre på hvordan folk opfatter programmeringssprog. Hvis en person ser et stykke Enigma -kode, vil den person måske blive inspireret til at skrive sit eget programmeringssprog. Egentlig kan det også have endnu bredere positive konsekvenser, idet den person måske ligefrem sætter sig ned for at lave et rigtigt menneskesprog efter at være blevet inspireret af Enigma .

Mens Enigma bare er et programmeringssprog, så kan hele tanken om at lave noget anderledes spores til også at findes i mange andre områder.

Konklusion

Jeg kan konkludere, at Enigma kan skabe adspredelse inden for programmeringssprogssyntax. Det viste sig at være muligt at lave et simpelt sprog, og det har vist mig at programmering tillader en del.

At lave et sprog var sandsynligvis den bedste og mest ligetil løsning.

Bilag

Tidsplan

Når man kun er en person, er det nemmere at planlægge ens tid. Gennem hele projektet har jeg generelt fulgt den meget overordnede plan om, at jeg først lavede programmet, så testede det, så skrev rapporten og derefter formulerede synopsen. Mere præcis er min tidsplan ikke.

Programinformation

```
Enigma 0.1
Copyright (C) 2010  Niels Serup <ns@metanohi.org>
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Litteratur

GNU. GNU Coding Standards. http://www.gnu.org/prep/standards/standards.html#Command_002dLine-Interfaces, April 2010.

Langpop. Programming Language Popularity. http://www.langpop.com/, April 2010.

Sun. BigDecimal (Java Platform SE 7). http://java.sun.com/javase/7/docs/api/java/math/BigDecimal.html, Maj 2010a.

Sun. Trail: The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html, April 2010b.