1DV610 - Laboration 2 - Parser

Syfte

- Skriva kod som bygger på laboration 1
- Skriva kod för att lösa en uppgift
- Förbättra kod enligt CC.
- Reflektera över och arbeta med CC kapitel 2-11 (främst de första kapitlen)

En parser

I laborationen kommer ni använda er tokeniserare från laboration 1 för att skriva en parser. En parser omvandlar en serie tokens till en högre struktur genom så kallad syntaxanalys.

> *"**Syntaxanalys**, syntaktisk analys eller parsning (av latinets pars, del), är den process som givet en följd av symboler från något språk konstruerar eller härleder någon slags syntaktisk struktur för symbolföljden, så att symbolernas inbördes relationer reds ut. Detta är en viktig del i mekanismer för att förstå språk och i någon mening antas att mänsklig språkförståelse delvis inbegriper någon form av syntaktisk analys."*
>
> -Wikipedia

Om vi utgår ifrån WordAndDotGrammar ifrån Laboration 1 för att parsa ett dokument så ger tokenisern dokumentets lexikala delar (tokenmatchningar).

Dokument som skall parsas:

- String: "I love parsers. They are fun."

Lexikalanalys (tokenisering, Laboration 1):
Lexikal grammatik: WORD(/^[\w|åäöÅÄÖ]+/) och DOT(/^\./)

- WORD("I")
- WORD("love")
- WORD("parsers")
- DOT(".")
- WORD("They")
- WORD("are")
- WORD("fun")
- DOT(".")

Nästa steg i parsning är syntaxanalysen som sätter samman ett antal tokens till en högre struktur.

En syntaktisk grammatik för dokument

Syntaxanalysen styrs också av en egen grammatik. I det här fallet skulle vi kunna skriva regler som beskriver vad ett dokument är och vad en mening är. Ett DOCUMENT i den här laborationen består av en samling meningar (SENTENCES) följt utav en

END-tokenmatchning, varje mening (SENTENCE) består av ett antal WORD och sedan en DOT.

Om vi sätter samman detta till en liten grammatik kan de se ut så här:

- **SENTENCE** = (WORD)+ DOT
- **SENTENCES** = (**SENTENCE**)*
- **DOCUMENT** = **SENTENCES** END

Den här grammatiken har tre regler men är inte skriven i reg-exp även om vi lånar plustecken (+ ett eller flera) och stjärna (* noll eller flera) därifrån.

Regeln **SENTENCE** = (WORD)+ DOT, skall tolkas som att ett eller flera ord följt av en DOT blir en **SENTENCE**.

Stjärnan efter **SENTENCES** betyder att samlingen kan vara tom. **SENTENCES** är noll eller flera meningar som följer varandra. Ett **DOCUMENT** är **SENTENCES** följt utav END-token.

Om vi applicerar syntaxanalys på de tokens i tidigare exemplet så får vi följande datastruktur:

DOCUMENT
- **SENTENCES**
  - **SENTENCE** (*WORD("I"), WORD("love"), WORD("parsers"), DOT(".")*)
  - **SENTENCE** (*WORD("They"), WORD("are"), WORD("fun"), DOT(".")*)
- END

Om vi pratar parsers i form av kod istället för grammatik så kan vi låta varje del av syntaxgrammatiken bli egna klasser. Vi kan ha en klass för Document vars objekt vi kan be om samlingen(array, lista eller liknande) meningar och får då ett objekt av klassen "Sentences". Varje mening är ett object som vi sin tur kan få samlingar av Word.

För att få skriva ut alla meningarna i "dokumentet" men där varje mening har ett enda mellanslag mellan orden och en punkt på slutet.

```
Document d = DocumentParser.parse("I love parsers. They are fun.");
Sentences twoSentences = d.getSentences();

for (Sentence s : twoSentences) {
  for (Word w : s.getWords()) {
    System.out.print( w.getMatchedText() );
    System.out.print( " " );
  }
  System.out.println(s.getEndType()); // Prints the matched end type (Eg. DOT)
}
```

Uppgift (för alla betygsnivåer)

Implementera en parser som kan hitta individuella meningar. Parsern är en separat modul. Börja med WordAndDot-grammatiken samt den korta syntaktiska grammatiken i ovanstående exempel. men utöka sedan båda grammatikerna (den lexikala-grammatiken och den syntaktiska-grammatiken) samt implementationen så att den även har stöd för utrop och frågor. SENTENCES blir då en samling meningar som kan antingen vara “en vanlig mening” som avslutas med punkt”, en fråga som avslutas med frågetecken, eller ett utrop som avslutas med “!”.

En parsning skall leda till en trädstruktur av objekt som motsvarar strukturen på dokumentet.

Du bör ha klasser såsom Document, Sentences och Sentence. Ni kan och får ha andra klasser också. Det är upp till er att designa detta i klasser och metoder men arv eller interface kan vara lämpligt. Exempelvis kan Sentence bli basklass åt tre subklasser.

Klassen Document skall kunna

- Vi kan be om samtliga meningar oavsett typ. Vi får dem i samma följd som dokumentet
- Vi kan be om alla vanliga meningar (som avslutas med punkt).
- Vi kan be om alla frågor (som avslutas med frågetecken).
- Vi kan be om alla utrop (som avslutas med utropstecken).

Klasserna för “en vanlig mening”, uttrop och frågor skall alla ha funktionalitet för att:

- stega igenom WORD-objekt som meningen består av.
- Få hela meningen som en enda sträng med “rätt” avslutande tecken(.?!) de enskilda orden separeras med ett enda mellanslag oavsett vilka whitespace som användes i orginaltexten.

Skapa sedan en tredje modul som listar innehållet i ett dokument med formatering. Ni väljer själva hur detta sker. Ex HTML+CSS, Console

- Index på meningen presenteras innan varje mening.
- Olika färger eller format för frågor, utrop och “vanliga meningar”. Så att det syns tydlig skillnad mellan olika typer.
    a. **Exempelvis skulle utrop kunna vara fetmarkerade!**
    b. Frågor kanske är röda?
    c. Vanliga meningar är kursiva.

Till slut så skall ni ha tre moduler

De tre modulerna skall hanteras som individuella projekt och beroendena mellan dem skall vara som följer.

1. Tokenizer beror inte på någon kod och uppfyller fortfarande krav från laboration 1.
2. Parser som beror på Tokenizer
3. PrettyPrinter som beror på er Parser.

Regler

- Skriv **all** kod själv, skriv inte tillsammans med någon (sida vid sida-programmering).
- Kopiera inte kod från någon annan. Skriv inte av kod.
- Använd inte bibliotek eller färdiga metoder för att lösa huvudproblemet att skapa tokeniseraren. Om ni tänker if-satser och for-loopar så gör ni rätt. Ni får använda reguljära uttryck för att beskriva olika token-typer. Ni får använda metoder som måste användas.
- Använd dig av din egen tokenizer från laboration 1. Tokenizerns kod får skrivas om men skall hållas som ett separat projekt som även skall stödja kraven och testerna från laboration 1.
- Använd inte färdiga klasser och bibliotek för att lösa problemet. Ni får använda färdiga klasser för att testa er implementation.
- Objektorienterad kod med klasser och metoder i klasser. Du kan ha kod utanför klasser men bara om den behövs för att starta upp koden (ex. nodejs “server.listen(port…”). Inga metoder i dina klasser får vara statiska mer än om det behövs för att starta upp koden eller om du behöver speciella namngivna konstruktorer (se CC).
- Koden dokumenteras på lämpligt sätt på git
- Ni får ändra tokenizer men den skall uppfylla kraven för laboration 1 och får inte vara specifik för laboration 2.

Validering och verifiering

Fundera ut vilka testfall som behövs för att täcka in de olika reglerna. Skapa en tabell med testfall som alla har beskrivande namn, indata samt förväntat utfall. Testa olika saker i varje testfall. Försök bli komplett för meningar upp till tre ord. Ta med sådant som skall kasta undantag. För högsta betyg skapa även en överblick genom att dela in tester i olika kategorier.

Exempel på testfall:

Testfall namn	**Indata**	**Förväntat beteende**
Hittar varje mening	“a. b.”	getSentences() ger två objekt ifrån sig
Korrekt antal ord senare mening	“a! b c?”	två WORD objekt i andra meningen
Mening ett är ett utrop	“a!”	Första meningen är av typen EXCLAMATION.
Senare ord har rätt bokstäver	“a bc.”	andra ordet i första meningen är “bc”
...		

Kodkvalitetskrav för betyg E-C

Gå igenom all kod inklusive kod från laboration 1 och uppdatera enligt bokens clean code kapitel 2-11 och det vi diskuterat på föreläsningar och workshops. Skriv en kort (4-6 meningar) reflektion för varje kapitel om hur just det kapitlet har påverkat eller inte påverkat din kod. **Använd bokens termer**. Ge exempel med läsbara screenshots från er kod till varje reflektion.

Fokusera på tydlighet, variation, ärlighet och vad som är intressant. Exempelvis om du har icke självklara överväganden med olika kvalitetsregler som står i konflikt med varandra så är dessa extra intressanta.

Jag kommer även titta på och bedöma er kod. Den skall därför i största mån vara skriven för att kunna fortsätta utvecklas av andra programmerare.

Betyg sätts på:

- funktionalitet, (hur väl ni uppfyller kraven)
- projektens framställning, (hur väl ni presenterar projektet via dess artefakter, ex README.md)
- testning, (hur väl ni testat kraven)
- reflektion, (hur väl ni kan diskutera bokens innehåll)
- kodkvalitet, (hur väl jag kan förstå er kod utifrån Clean Code)

För högre betyg (A och B) behöver ni även skapa en till parser alternativt en rejält utökad parser. Jag ger tre förslag (nja det tredje är väldigt fritt).

Alternativ 1 av 3, Extrauppgift för högre betyg (Betyg B+)

Skapa även en parser-modul för en enklare dokumentsyntax som kan parsa enkla dokument med titlar, stycketexter, listor och länkar och kan bryta upp det i olika delar för att skapa exempelvis HTML utifrån dessa. Utgå från den första uppgiften men lägg till funktionalitet för att hantera exempelvis:

- Textstycken exempelvis blir dubbla radbrytningar en styckebrytning, ett dokument består av en samling stycken som i sin tur består av en samling meningar.
- Titlar. Exempelvis: en mening som inleds med “# “ blir en titel. En titel följs alltid av ett stycke.
- Bullet list, Exempelvis: om meningar i ett stycke inleds med * blir dessa bulletpoints.
- Länkar, kan bli en egen meningstyp. Dessa kan ingå i stycken, titlar och i bullet lists.

Exempel på syntax, med tre stycken.

“# Titel text
Första meningen i första stycket. Vad menar han? Titeln hör till detta stycke!

Första meningen i andra stycket. Detta stycke saknar titel.

 # Titeltext till bulletstycke
** en stjärna ger en bulletlist.*
** nästa bullet i samma stycke.”*

Utöka grammatiken från första uppgiften med nya regler för att åstakomma detta. Fundera igenom en bra datastruktur för dokument, stycken, meningar av olika typer.

Validering och verifiering

Skriv testfall för denna som utvärderar varje regel i grammatikerna. Redovisa dessa med namn, indata samt förväntat beteende.

Kodkvalitetskrav för högre betyg

Samma som för för lägre betyg men baka in de 10 reflektionerna från de olika kapitlen till en sammanhängande text som spänner över båda uppgifterna. Du har alltså en enda reflektion fast för både koden i uppgift 1 och uppgift 2. Ca två sidor max. Använd varierade uttryck från boken.

Alternativ 2 av 3, Extrauppgift för högre betyg (Betyg B+)

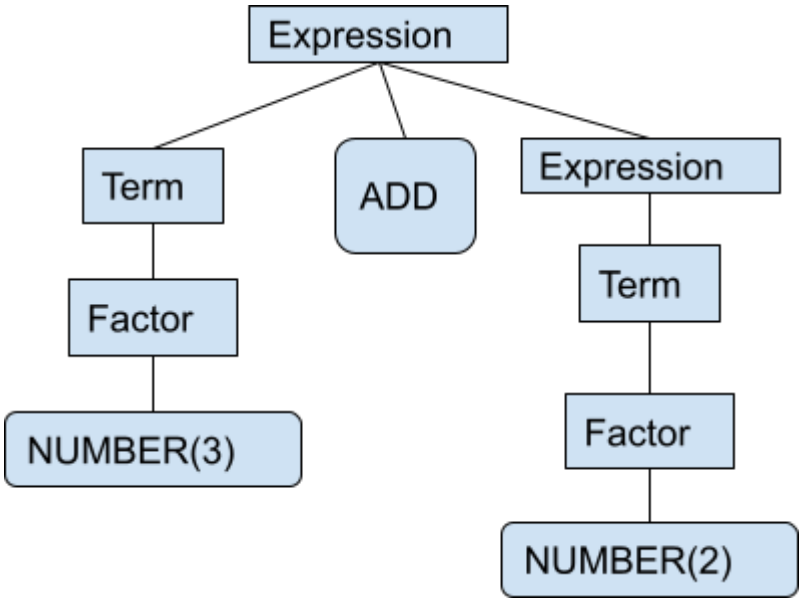
Skapa även en parser för aritmetiska uttryck som kan parsa men också beräkna svaret på uttryck av typen “(2 + 5) * 2.4”.

En syntaktisk grammatik för aritmetik

- Factor = NUMBER
- Factor = “(“ + Expression + “)”
- Factor = SUB Factor
- Term = Factor MUL Term
- Term = Factor DIV Term
- Term = Factor
- Expression = Term ADD Expression
- Expression = Term SUB Expression
- Expression = Term

En Factor kan se ut på tre olika sätt, antingen som ett ensam tokenmatching för NUMBER, eller som en parantes, ett expression och en slutparantes, eller som SUB (ett minustecken) och sedan en faktor. Beskrivningen av syntaxen är rekursiv så “--3” börjar med att matcha en SUB Factor där faktordelen återigen matchas av SUB Factor och i den matchar faktordelen NUMBER. Så Factor(SUB Factor(SUB Factor(3))).

På samma sätt kan reglerna sättas ihop med varandra rekursivt så en enda siffra “3” är ett Expression(Term(Factor(NUMBER(“3”)))). Den högsta nivån i den här grammatiken är Expression som i sig kan representera alla matematiska uttryck. Ett Expression kan rekursivt utvidgas till en Term, ADD eller SUB och sedan ett till Expression för att kunna hantera uttryck som “3+2”.



Kodexempel

```
const grammar = new ArithmeticGrammar();
const stringToMatch = '( 2 +  5 ) * 2.4’;
const t = new Tokenizer(grammar, stringToMatch);
```

```
const p = new MathParser(t);
console.log(p.compute);
```

Guide

Skapa klasser för Expression, Term, Factor, eventuellt fler för de olika reglerna. En parsning av strängen kommer leda till en trädstruktur av objekt från dessa klasser.

Låt uträkningen vara en del av klasserna för Expression, Term och Faktor. Jag har en metod i mina som heter evaluate som returnerar del-uttryckets resultat.

Validering och verifiering

Skriv testfall för denna som utvärderar varje regel i grammatikerna. Redovisa dessa med namn, indata samt förväntat beteende.

Kodkvalitetskrav för högre betyg

Samma som för för lägre betyg men baka in de 10 reflektionerna från de olika kapitlen till en sammanhängande text som spänner över båda uppgifterna. Du har alltså en enda reflektion fast för både koden i uppgift 1 och uppgift 2. Ca två sidor max. Använd varierade uttryck från boken.

Alternativ 3 av 3, Extrauppgift för högre betyg (Betyg B+)

Skapa en egen parser för någon form av strukturerad data. Diskutera med Daniel när du väl har en idé.

Sammanfattning av det som skall lämnas in.

- Betygsnivå E-C:
    - Uppdaterad Tokenizer
    - Parser för “dokument”
    - PrettyPrinter
    - Testning för all kod
    - Reflektion för kapitel 2-11
- Betygsnivå A-B
    - Allt från E-C
    - Extrauppgift
    - Sammanhängande reflektion.

Förslag på hur man kan börja koda.

Börja genom att parsa endast en enda mening. Så skriv metoden "String parseSentence(Tokeanizer t)" som tar en tokenizer som kollar efter en serie "WORD" följt utav en "DOT".
Den kan kasta undantag om den får något annat (ex end) eller om den får bara en DOT utan WORD.