## Opgave 2. Verzamelingen.

Voor deze opgave moet er een interpreter geschreven worden voor de verderop gedefinieerde "commandotaal". Alle "zinnen" in deze commandotaal zijn opdrachten voor bewerkingen met verzamelingen van natuurlijke getallen.
Ga als volgt te werk :

1- Bedenk welke objecten in deze opgave nodig zijn, en specificeer ieder van deze objecten in een interface.
2- Implementeer de class Lijst volgens de specificatie in de gegeven interface LijstInterface. De hiervoor benodigde class Knoop is eveneens gegeven. Alletwee zijn deze te vinden in de directory /usr/prac/ds/opgave2.
   Het is bij deze opgave verplicht om alle objecten die je zelf gespecificeerd hebt en waarin willekeurig veel andere objecten moeten kunnen worden opgeslagen, te implementeren met de class Lijst.
   Specificeer objecten die maar 1 elementtype bevatten generiek m.b.v. een template.
3- Kopieer uit de directory /usr/prac/ds/opgave2 ook de interface Data en de class DSException. Gebruik deze interfaces om bij het specificeren van de eigen objecten de juiste eigenschappen aan die objecten op te leggen. Gebruik instanties van de class DSException indien je een exception wilt gooien.
4- Maak, gebruik makend van de objecten, een ontwerp van de interpreter. Gebruik de methode van recursive descent (zie voorbeeld op blad 4).
5- Laat de zelf gemaakte interfaces en het ontwerp van de interpreter bij de ontwerpbespreking goedkeuren.
6- Als je ontwerp en de interfaces goedgekeurd zijn en je de interfaces geimplementeerd hebt, kan er worden begonnen met programmeren. Schrijf eerst het parser (herkenner) gedeelte van je ontwerp helemaal uit. D.w.z. schrijf een programma dat regels invoer inleest en bij een correcte opdracht niets doet en bij een foute opdracht één duidelijke foutmelding geeft.
7- Pas als de parser werkt, moet deze worden uitgebreid tot een interpreter. De interpreter moet opdrachten, geschreven in de verderop gedefinieerde taal, herkennen **en** uitvoeren.

Met behulp van de verderop beschreven commandotaal kunnen we zowel verzamelingen van natuurlijke getallen als variabelen die zo'n verzameling bevatten, manipuleren. De natuurlijke getallen kunnen uit willekeurig veel cijfers bestaan. Als namen voor een variabelen zijn alleen identifiers toegestaan.
Er zijn vier operatoren beschikbaar op verzamelingen in de taal :

| | | | | | |
|---|---|---|---|---|---|
| + | : | A + B | = | $\{ x \mid x \in A \vee x \in B \}$ | vereniging |
| * | : | A * B | = | $\{ x \mid x \in A \wedge x \in B \}$ | doorsnede |
| – | : | A – B | = | $\{ x \mid x \in A \wedge x \notin B \}$ | verschil |
| \| | : | A \| B | = | $\{ x \mid x \in A + B \wedge x \notin A * B \}$ | symmetrisch verschil |
| | | | = | $\{ x \mid x \in (A + B) - (A * B) \}$ | |

De operator '*' heeft een hogere prioriteit dan '+', ' | ' en '–', die dezelfde prioriteit hebben. De operatoren zijn links-associatief.
Voorbeeld: A – B * Set1 + D moet geëvalueerd worden als (A – (B * Set1)) + D.

Er zijn twee soorten commando's beschikbaar. Van elk een voorbeeld :

1. Set1 = A + B – Set1 * (D + Set2)
   Bereken de verzameling die het resultaat is van de expressie rechts van het '='-teken en associeer de variabele met de identifier Set1 met deze verzameling.

2. ? Set1 + Set2
   Bereken de verzameling die het resultaat is van de expressie, en print de elementen van deze verzameling, gescheiden door spatie's, op één regel, op de standaard output.
   **Print verder geen tekst of komma's o.i.d. i.v.m. de automatische test.**

In de gebruikte grammatica wordt voor de beschrijving van de syntax van de commandotaal de EBNF-notatie gebruikt. Verder worden de tekens "<" en ">" worden gebruikt voor omschrijvingen (b.v <eof>).

**SYNTAX COMMANDOTAAL**

program = { statement } <eof> .

Een programma is een een willekeurig aantal statements (commando's) afgesloten door het einde van de file.

statement = assignment | print_statement | commentaar .

Een statement is een assignment statement, een print statement of commentaar.

assignment = identifier '=' expressie <eoln> .

Een assignment statement is een identifier, gevolgd door het '='-teken, waarachter een expressie staat afgesloten door een regelovergang.

print_statement = '?' expressie <eoln> .

Een print statement is een '?' gevolgd door een expressie en afgesloten door een regelovergang.

commentaar = '/' <een regel tekst> <eoln> .

Commentaar is een regel tekst die begint met het '/'-teken en afgesloten wordt door een regelovergang.

identifier = letter { letter | cijfer } .

Een identifier begint met een letter en bestaat alleen uit letters en cijfers.

expressie = term { additieve_operator term } .

Een expressie is een term, gevolgd door nul of meer termen, alle termen gescheiden door een additieve operator.

term = factor { multiplicatieve_operator factor } .

Een term is een factor, gevolgd door nul of meer factoren, alle factoren gescheiden door een multiplicatieve operator.

factor = identifier | complexe_factor | verzameling .

Een factor is een identifier, een complexe factor of een verzameling.

complexe_factor = '(' expressie ')' .

Een complexe factor is een expressie tussen ronde haakjes.

verzameling = '{' rij_natuurlijke_getallen '}' .

Een verzameling is een rij natuurlijke getallen tussen accolades.

rij_natuurlijke_getallen = [ natuurlijk_getal { ',' natuurlijk_getal } ] .

Een rij natuurlijke getallen is of leeg of een opsomming van één of meer natuurlijke getallen gescheiden door komma's.

additieve_operator = ’+’ | ’|’ | ’–’ .
Een additieve operator is een ’+’-, een ’|’- of een ’ – ’-teken.

multiplicatieve_operator = ’*’ .
Een multiplicatieve operator is een ’*’-teken.

natuurlijk_getal = positief_getal | nul .
Een natuurlijk getal is een positief getal of nul.

positief_getal = niet_nul { cijfer } .
Een positief getal begint niet met de nul, heeft geen teken en bevat 1 of meer cijfers.

cijfer = nul | niet_nul .
Een cijfer is wel of niet een nul.

nul = ’0’ .
Nul is het getal 0.

niet_nul = ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’ .
Niet_nul is een getal uit het bereik 1 t/m 9.

letter = ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’ |
’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’ | ’Y’ | ’Z’ |
’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ |
’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’ .

Een letter is een hoofdletter of een kleine letter.

**OPMERKINGEN**

1) Spaties in natuurlijke getallen en identifiers zijn niet toegestaan. Verder hebben spaties geen functie en kunnen ze overal op een regel geplaatst worden om de leesbaarheid te vergroten.
2) Als er een fout in een statement ontdekt wordt, druk dan één duidelijke foutmelding af gevolgd door een regelovergang. Verklarende foutmeldingen zijn ook noodzakelijk om in de testfase van het programma te ontdekken wat het programma precies doet (dit is meestal iets anders dan wat het zou moeten doen).
3) Het programma moet van standaard input lezen en op standaard output schrijven. De uitvoer moet bestaan uit één-regelige foutmeldingen of één regel met getallen, gescheiden door spaties en afgesloten met een end-of-line achter het laatste getal.
4) Commentaar moet worden genegeerd.

Als voorbeeld van hoe een ontwerp tot stand komt, kunnen we ons afvragen hoe m.b.v. de methode 'recursive descent' een factor herkend kan worden. Bij 'recursive descent' roepen de procedures die de invoer parseren elkaar recursief aan.
Zo kunnen we b.v. een boolean methode `factor()` ontwerpen die
- een verzameling van natuurlijke getallen als functieresultaat retourneert

als er een grammaticaal juiste factor op de invoer staat, en
- een foutmelding geeft,
- een DSException gooit

als er *niet* een grammaticaal juiste factor op de invoer staat.
Wanneer de methode `factor()` een complexe factor parseert, moet daarvoor een expressie geparseerd moet worden, waarvoor een term geparseerd moet worden, waarvoor weer een factor geparseerd moet worden. Na 3 tussenaanroepen wordt `factor()` dus weer aangeroepen (m.a.w. recursie).

Een eerste schets van de methode `factor()` is

```
Verzameling factor () throws DSException {
/* factor() leest zo mogelijk een correcte factor van de invoer.
   Is dit gelukt dan is deze factor geëvalueerd en de resulterende
   verzameling geretourneerd.
   Is dit niet gelukt, dan is een foutmelding gegeven en is er een
   DSException gegooid.
*/

    if (het volgende character is een letter) {
        lees een identifier
        haal de bij de identifier horende verzameling op
    } else
    if (het volgende character is '{') {
        lees een verzameling
    } else
    if (het volgende character is '(') {
        bepaal de verzameling die de uitkomst is van de complexe factor
    } else
        Geef een duidelijke foutmelding
        gooi DSEXception
    }
    RETURN de waarde van de verzameling
}
```

N.B. Let op de overeenkomst tussen de syntax-definitie van een factor en het ontwerp van `factor()`. De syntax-regels zijn een zeer grondige analyse/beschrijving van de invoer en hoeven bij het ontwerp alleen nog maar in een Java beschrijving te worden omgezet.