

Java en Triangle

PRACTICUM


Tijdens dit eerste practicum van Vertalerbouw wordt allereerst de Java-kennis een beetje opgefrist. Daarnaast wordt er geoefend met de programmeertaal Triangle, de taal die centraal staat in het boek van Vertalerbouw. Tenslotte wordt een eerste aanzet gemaakt tot het bouwen van een *symbol table*.

Let op: Alle voorgedefinieerde javafiles zijn in een vaste **package**-structuur geplaatst, die telkens begint met `vb.week?`, in de meeste gevallen gevolgd door nog een verder sub-**package**. Dit maakt het eenvoudiger de opgaven van verschillende weken te scheiden. Houd deze structuur ook aan in uw uitwerking.

1.1 Wordcount in Java

Bij het practicum van Vertalerbouw gaan we ervan uit dat uw kennis van de programmeertaal Java tenminste op het niveau van Programmeren 1 en 2 ligt. U moet in staat zijn om zelfstandig Java programma's te schrijven en u dient te weten hoe de command-line tools (met name: `java`, `javac` en `javadoc`) van de Java SDK gebruikt moeten worden. U wordt verder geacht de Java API documentatie te kunnen raadplegen.

Bij deze eerste opgave werken we aan een applicatie die het aantal regels, woorden en karakters telt van de standaard input of van een file. Onder Unix/Linux staat dit programma bekend onder de naam `wc`, een afkorting van *wordcount*. Op Blackboard staat de file `vb/week1/wc/WordCount.java`. Dit programma is een Java implementatie van *wordcount*.

-  **1.1.1** Bestudeer de file `WordCount.java` en compileer het programma. Experimenteer met het programma om te kijken dat het naar behoren werkt.

In het programma `WordCount.java` worden de *woorden* geteld door in elke regel expliciet naar zogenaamde *whitespace* karakters (nl. spaties en tabs) te zoeken. Inplaats daarvan is het ook mogelijk om een `java.util.Scanner` object te gebruiken.

Een `Scanner` splitst de invoer op in *tokens*, de kleinste bouwstenen die een betekenis hebben in de invoer. Een `Scanner` gebruikt een afbreekpatroon (*delimiter pattern*) om tokens van elkaar te scheiden (default: whitespace karakters). In het geval van *wordcount*, zijn de kleinste bouwstenen de woorden van de file: de woorden worden gescheiden door *whitespace*.¹

- ☞ **1.1.2** Schrijf een klasse `WordCountScanner` die een subklasse is van de klasse `WordCount`. De methode `count` van `WordCountScanner` moet een `java.util.Scanner` object gebruiken om de regels in woorden op te splitsen.

Voeg ook een methode `main` aan de klasse `WordCountScanner` toe opdat ook de applicatie `WordCountScanner` gebruikt kan worden om woorden in een tekstbestand te tellen. Test de klasse `WordCountScanner` en vergewis u ervan dat de programma's `WordCount` en `WordCountScanner` hetzelfde gedrag vertonen.

1.2 Wordcount in Triangle

Gelijk vrijwel alle vertaler leerboeken, behandelen Watt & Brown (2000) de theorie en praktijk het bouwen van vertalers aan de hand van een klein programmeertaaltje. Het taaltje van Watt & Brown (2000) heet Triangle. De auteurs hebben ook een vertaler ontwikkeld voor Triangle die code voor de Triangle Abstract Machine (TAM) genereert. Met behulp van de TAM-Interpreter kunnen Triangle programma's daadwerkelijk uitgevoerd worden.² Bijlage B van Watt & Brown (2000) bevat een informele specificatie van Triangle.

Zowel het boek (met name het voorbeeldprogramma op bladzijde 400) als de source code van Triangle bevatten enkele slordige foutjes. Op Blackboard staat een lijst met errata/bugs vinden, samen met een verbeterde versie van `Triangle.jar`.

Uitvoeren van een Triangle programma. We gaan ervan uit dat u de beschikking heeft over `Triangle.jar`, de archive die zowel de Triangle compiler als de TAM Interpreter en Disassembler bevat.

Compileren van een Triangle programma `foo.tri` gaat als volgt:³

```
java -cp Triangle.jar Triangle.Compiler foo.tri
```

Als het programma `foo.tri` geen fouten bevat, zal de compiler de file `obj.tam` genereren waarin zich de (binary) TAM-code bevindt voor de Triangle Abstract Machine. Met behulp van de TAM-interpreter kan het programma vervolgens uitgevoerd worden:

¹ Java biedt ook een klasse `java.util.StringTokenizer` voor het opsplitsen van de invoer in *tokens*. `Scanner` is echter krachtiger en flexibeler en verdient de voorkeur boven `StringTokenizer`.

² De Java source code van zowel de Triangle-vertaler als de TAM-interpreter is ook beschikbaar (Blackboard). Veel van de code fragmenten uit Watt & Brown (2000) komen rechtstreeks in de Java code voor. Helaas is de Java code zwak becommentarieerd: de code bevat nauwelijks zinvol javadoc commentaar en het overige commentaar klopt vaak niet. De programmatuur haalt derhalve helaas niet de standaard van P1 en P2.

³ De `-cp Triangle.jar` option is nodig om ervoor te zorgen dat de inhoud van `Triangle.jar` aan het Java CLASSPATH toegevoegd wordt. U zou er ook voor kunnen kiezen om de file `Triangle.jar` standaard in uw Java CLASSPATH op te nemen.

```
java -cp Triangle.jar TAM.Interpreter
```

Als u wilt zien welke TAM code door de Triangle compiler gegenereerd is kunt u de *disassembler* gebruiken, die een tekstuele representatie van `obj.tam` op de standaard output laat zien:

```
java -cp Triangle.jar TAM.Disassembler
```

- ☞ **1.2.1** Als kennismaking met Triangle, dient u een Triangle programma te schrijven dat het eindcijfer bepaalt van het vak Vertalerbouw. Omdat Triangle geen reële getallen ondersteunt, dient u te rekenen met cijfers tussen 10 en 100, d.w.z. de oorspronkelijke cijfers vermenigvuldigd met 10.

Het programma dient eerst te vragen naar de cijfers voor de twee opgavenseries. Vervolgens dient het programma te vragen naar het cijfer van het practicum. Met deze gegevens moet het programma het eindcijfer voor het vak bepalen. In de Inleiding van deze practicumhandleiding, onder het kopje “Beoordeling”, staan de regels voor het bepalen van het eindcijfer.

Nota Bene. Als uw Triangle programma fouten bevat zult u merken dat de Triangle compiler niet erg sterk is in het geven van foutmeldingen. We hopen dat de compiler die u in de eindopdracht zult ontwikkelen beter met fouten in de invoer zal omspringen.

Na deze vingeroefening, ontwikkelen we vervolgens een *wordcount* programma in Triangle. Het Triangle programma op pagina 400 van Watt & Brown (2000) kan hierbij als voorbeeld dienen van een programma dat karakters inleest en weer wegschrijft.

Let op: het programma op bladzijde 400 bevat (tenminste) twee storende fouten. De condities van `while eol()` en `while eof()` dienen uiteraard voorafgegaan worden door de operator voor logische negatie: `!`. Daarnaast is de procedure `getline` foutief. Voor een correcte `getline` zie Blackboard.

- ☞ **1.2.2** Schrijf analoog aan het `WordCount` programma in Java van Opgave 1.1 een Triangle programma dat karakters van de standaard input inleest en vervolgens het aantal regels, het aantal woorden en het aantal karakters telt op de input en deze aantallen afdruckt op de standaard output. Zorg ervoor dat uw `Triangle` programma dezelfde uitvoer geeft als de Java programma’s van Opgave 1.1.

1.3 Symbol Table

In deze opgave maken we een begin met een *symbol table*, een structuur waarin gegevens over de *identifiers* van een programma efficient kunnen opgeslagen. De programmatuur van deze opgave zult u ook bij de eindopdracht van Vertalerbouw kunnen gebruiken.

Beschouw de volgende grammatica in BNF-formaat met startsymbol `decluse`.

<code>decluse</code>	<code>::=</code>	<code>"(" serie ")"</code>
<code>serie</code>	<code>::=</code>	<code>unit serie</code>
		<code> </code>
		<code>ε</code>
<code>unit</code>	<code>::=</code>	<code>decl</code>
		<code> </code>
		<code>use</code>
		<code> </code>
		<code>"(" serie ")"</code>
<code>decl</code>	<code>::=</code>	<code>"D:" id</code>
<code>use</code>	<code>::=</code>	<code>"U:" id</code>
<code>id</code>	<code>::=</code>	<code>letter id</code>
		<code> </code>
		<code>letter</code>
<code>letter</code>	<code>::=</code>	<code>"a" "b" "c" "d" "e" "f" "g" </code>
		<code>"h" "i" "j" "k" "l" "m" "n" </code>
		<code>"o" "p" "q" "r" "s" "t" "u" </code>
		<code>"v" "w" "x" "y" "z"</code>

De strings tussen dubbele quotes zijn terminals; de dubbele quotes zelf horen niet bij de terminals. Bij de `serie` is het tweede alternatief ϵ wat correspondeert met de 'lege string'. Een `serie` is dus óf een `unit` gevolgd door een `serie` óf de lege string.

Deze BNF beschrijft een simpele hiërarchische structuur van identifiers, waarbij *identifiers* (nl. `id`) gedeclareerd (`decl`) en gebruikt (`use`) kunnen worden. De haakjes corresponderen met zogenaamde *scope*-levels van programmeertalen. Een voorbeeld van een zin van deze grammatica is:

```
(D:aap (U:aap D:noot D:aap (U:noot) (D:noot U:noot)) U:aap)
```

Merk op dat hoewel dit voorbeeld een zin is in de taal gegenereerd door de grammatica `decluse`, de impliciet gesuggereerde relatie tussen de gedeclareerde en gebruikte identifiers niet klopt.

Merk tevens op dat ten behoeve van de leesbaarheid de voorbeeldzin *whitespace* karakters (nl. spaties) bevat die niet door de BNF gedefinieerd worden.

- ☞ **1.3.1** Schrijf een Java programma dat bovenstaande grammatica kan parsen. Het programma moet controleren of de gevonden identifiers goed geformatteerd zijn (d.w.z. een `D:` of een `U:` prefix hebben). Daarnaast moet het programma controleren of de haakjes goed genest zijn.

Uw programma moet zogenaamde *whitespace*-karakters (spaties, tabs en regelovergangen) wel inlezen maar verder negeren.

Hint: U hoeft geen *recursive descent parser* te schrijven. De grammatica is dermate simpel dat u in de inleeslus met enkele variabelen kunt bijhouden of de invoer nog aan de grammatica voldoet.

We hebben tot dusver nog geen nadere beperkingen aan geldige zinnen van de grammatica `decluse` opgelegd. Hieronder staan de *contextbeperkingen* (Engels: *context constraints*) met betrekking tot de grammatica `decluse`:

- Een `id` mag alleen gebruikt worden (`use`) als het daarvóór gedeclareerd is (`decl`) op hetzelfde level of binnen een omsluitende scope.
- Een `id` mag niet nogmaals gedeclareerd worden in dezelfde scope.


- Een id mag wel nogmaals gedeclareerd worden in een diepere scope. Deze nieuwe declaratie overschrijft (tijdelijk) de declaratie van een buitenste scope.

Om contextbeperkingen van de grammatica te controleren moeten de gedeclareerde identifiers in een tabel worden opgeslagen. Een dergelijke tabel wordt *symbol table* of *identification table* genoemd. Voor elke gedeclareerde identifier wordt belangrijke informatie opgeslagen; te denken valt aan het level van de declaratie, de positie in de file (regelnummer en kolomnummer) en het type van de identifier (bij een programma), etc.

Voor deze opgave is het voldoende om van elke identifier alleen het level bij te houden. We definiëren daartoe de klasse `vb.week1.symtab.IdEntry` (zie ook Blackboard).

```
public class IdEntry {
    private int    level = -1;

    public int     getLevel()           { return level;          }
    public void    setLevel(int level)  { this.level = level;    }
}
```

-  **1.3.2** Schrijf een klasse `SymbolTable` waarin informatie over identifiers kan worden opgeslagen. Deze informatie dient binnen de `SymbolTable` als `IdEntry`-objecten te worden opgeslagen. Het raamwerk van de klasse `SymbolTable` en haar methoden ziet er als volgt uit.

```
package vb.week1.symtab;

public class SymbolTable<Entry extends IdEntry> {

    /**
     * Constructor.
     * @ensures this.currentLevel() == -1
     */
    public SymbolTable() {
        // body nog toe te voegen
    }

    /**
     * Opens a new scope.
     * @ensures this.currentLevel() == old.currentLevel()+1;
     */
    public void openScope() {
        // body nog toe te voegen
    }

    /**
     * Closes the current scope. All identifiers in
     * the current scope will be removed from the SymbolTable.
     * @requires old.currentLevel() > -1;
     * @ensures this.currentLevel() == old.currentLevel()-1;
     */
    public void closeScope() {
        // body nog toe te voegen
    }
}
```

```

    /** Returns the current scope level. */
    public int currentLevel() {
        return 0; // body nog toe te voegen
    }

    /**
     * Enters an id together with an entry into this SymbolTable
     * using the current scope level. The entry's level is set to
     * currentLevel().
     * @requires id != null && id.length() > 0 && entry != null;
     * @ensures this.retrieve(id).getLevel() == currentLevel();
     * @throws SymbolTableException when there is no valid
     *     current scope level, or when the id is already declared
     *     on the current level.
     */
    public void enter(String id, Entry entry)
        throws SymbolTableException {
        // body nog toe te voegen
    }

    /**
     * Get the Entry corresponding with id whose level is
     * the highest; in other words, that is defined last.
     * @return Entry of this id on the highest level
     *     null if this SymbolTable does not contain id
     */
    public Entry retrieve(String id) {
        return null; // body nog toe te voegen
    }
}

/** Exception class to signal problems with the SymbolTable */
class SymbolTableException extends Exception {
    /** {@link serialVersionUID} is required for Serializable */
    public static final long serialVersionUID = 24362462L;
    public SymbolTableException(String msg) { super(msg); }
}

```

De klasse `SymbolTable` is geparameteriseerd met de klasse `IdEntry`: een `SymbolTable` kan alleen maar objecten opslaan die een (subklasse van) `IdEntry` zijn. De `IdEntry`-objecten worden geïdentificeerd door een `String`-representatie van de identifier (zie de methoden `enter` en `retrieve`). De methode `enter` gooit een `SymbolTableException` op het moment dat er iets mis is met de identifier die aan de `SymbolTable` moet worden toegevoegd. Het raamwerk van de klasse `SymbolTable` is beschikbaar als `vb/week1/symtab/SymbolTable.java` op Blackboard.

Hint: Bij de implementatie van de klasse `SymbolTable` kunnen de klassen uit Java's *collectie hiërarchie* handig zijn. Met name de interfaces `java.util.Map` en `java.util.List` en de klasse `java.util.Stack` zijn wellicht nuttig.

- ☞ **1.3.3** Breid tenslotte het programma van Vraag 1.3.1 zodanig uit dat de gedeclareerde identifiers in een `SymbolTable`-object worden opgeslagen. Voor elk id dat gebruikt wordt moet gecontroleerd worden of het gedeclareerd is. Houd daarbij rekening met het volgende.

- Als een id twee keer (of meer) gedeclareerd wordt in dezelfde scope, dan moet het programma dit melden, maar wel doorgaan met het verwerken van de invoer.
- Het programma moet tenminste alle woorden afdrukken die gebruikt worden. Voor elk gebruikt id moet het afdrukken op welk scope-level het id *gebruikt* wordt en op welk scope-level het id gedeclareerd is. Het scope-level van de eerste serie (die van declare) is 0.
- Als een woord gebruikt wordt dat niet gedeclareerd is moet het programma dit melden. De rest van het invoer moet echter wel verder verwerkt worden.

Voorbeeld. Beschouw de volgende structuur. Net als het eerdere voorbeeld, voldoet dit voorbeeld aan de grammatica *declare*.

```
(D:x D:y D:z
  U:y U:z
  (D:a D:z
    U:x U:y U:z (D:p D:q U:p U:y) (U:z D:z U:z) U:a
  )
  (D:x D:p D:x
    U:x U:q U:y U:z
  )
)
```

Dit voorbeeld is als `vb/week1/sample-2.txt` op Blackboard te vinden.

Het voorbeeld houdt zich echter niet aan de contextbeperkingen van de grammatica *declare*. Met dit voorbeeld als invoer zou een programma van Vraag 1.3.3 de volgende output kunnen genereren:

```
D:x on level 0
D:y on level 0
D:z on level 0
U:y on level 0, declared on level 0
U:z on level 0, declared on level 0
D:a on level 1
D:z on level 1
U:x on level 1, declared on level 0
U:y on level 1, declared on level 0
U:z on level 1, declared on level 1
D:p on level 2
D:q on level 2
U:p on level 2, declared on level 2
U:y on level 2, declared on level 0
U:z on level 2, declared on level 1
D:z on level 2
U:z on level 2, declared on level 2
U:a on level 1, declared on level 1
D:x on level 1
D:p on level 1
x' already declared on the current level
U:x on level 1, declared on level 1
U:q on level 1, *undeclared*
U:y on level 1, declared on level 0
U:z on level 1, declared on level 0
```