Week **2**

Recursive-Descent Parsing

PRACTICUM

Tijdens dit practicum wordt een eenvoudige *one-pass recursive-descent* vertaler ontwikkeld in Java. We volgen daarbij hoofdstuk 4 van Watt & Brown. De te ontwikkelen compiler dient een tabel-representatie in LaTeX te vertalen naar een tabel-representie in HTML.

Opmerking: Het practicum van deze week is niet moeilijk maar wel veel van hetzelfde. De opdracht illustreert dat het redelijk eenvoudig is om een recursive-descent parser te schrijven, maar dat het voor grotere talen sterk aan te bevelen is om dergelijke ontleed-programma's met behulp van een generator automatisch te laten genereren.

2.1 Scanner

TeX is een krachtig tekstopmaaksysteem waarmee professioneel drukwerk gemaakt kan worden. TeX is een wereldwijde standaard voor het opmaken van wetenschappelijke artikelen en boeken. TeX is met name geschikt als de tekst veel wiskundige en formules bevat, maar ook met platte tekst heeft TeX geen enkele moeite. Door de vele commando's van TeX lijkt het 'opmaken' van TeX documenten een beetje op programmeren.

LaTeX is een uitgebreid macropakket voor TeX, dat eenvoudiger te gebruiken en te leren is dan TeX zelf. Zo ondersteunt LaTeX bijvoorbeeld een `tabular`-omgeving, waarmee het eenvoudig is om een *tabel* weer te geven.

Voorbeeld. Beschouw het volgende voorbeeld van een `tabular`-specificatie (ook beschikbaar als `sample-1.tex` op Blackboard).

```
% An example to test the Tabular application.

\begin{tabular}{lcr}
  Aap  & Noot & Mies \\
```

latexTabular	::=	beginTabular colsSpec rows endTabular
colsSpec	::=	LCURLY identifier RCURLY
rows	::=	rows row
	|	ϵ
row	::=	entries DOUBLE_BSLASH
entries	::=	entry otherEntries
	|	ϵ
otherEntries	::=	AMPERSAND entries
	|	ϵ
entry	::=	num | identifier | ϵ
beginTabular	::=	BSLASH BEGIN LCURLY TABULAR RCURLY
endTabular	::=	BSLASH END LCURLY TABULAR RCURLY
num	::=	num digit
	|	digit
identifier	::=	identifier letter
	|	identifier digit
	|	letter
digit	::=	"0" | "1" | ... | "9"
letter	::=	"a" | "b" | ... | "z"
	|	"A" | "B" | ... | "Z"
BSLASH	::=	" \"
DOUBLE_BLASH	::=	" \\"
LCURLY	::=	"{"
RCURLY	::=	"}"
AMPERSAND	::=	"&"
BEGIN	::=	"begin"
END	::=	"end"
TABULAR	::=	"tabular"

Figuur 2.1: BNF grammatica van LaTeX's `tabular` omgeving.

```
  Wim  & Zus  & Jet  \\
  1    & 2    & 3    \\
  Teun & Vuur & Gijs \\
\end{tabular}
```

Het begin (resp. einde) van een `tabular`-omgeving in LaTeX wordt aangegeven door de string `\begin{tabular}` (resp. `\end{tabular}`). Een `tabular`-omgeving verwacht één argument tussen accolades. In het voorbeeld is dat de letter-combinatie `lcr`. Dit argument geeft aan hoe de kolommen van de tabel geformatteerd moeten worden: `l` staat voor *links-uitgevuld*, `c` staat voor *gecentreerd* en `r` staat voor *rechts-uitgevuld*. Het aantal letters in het argument geeft het aantal kolommen weer. Bij dit practicum zal het argument van de `tabular`-omgeving *niet* gebruikt worden: we controleren het aantal kolommen niet en de uitlijning van de kolommen gebruiken we ook niet.

De elementen (i.e. *entries*) van een `tabular` worden rij-gewijs gespecificeerd. Per rij worden de elementen van elkaar gescheiden door een ampersand-teken ("`&`"). Een rij wordt afgesloten door twee backslashes ("`\\`"). Bij dit practicum zijn de tabel-elementen of een num (een getal) of een identifier (een letter gevolgd door nul of meer letters of cijfers).[1]

[1] Bij dit practicum gebruiken we een vereenvoudigde versie van de `tabular`-omgeving. De officiële LaTeX-`tabular` is veel uitgebreider en er zijn geen beperkingen voor de tabel-elementen.

In figuur 2.1 staat een BNF-grammatica voor de `tabular`-omgeving. Enkele opmerkingen t.a.v. deze grammatica:

- De ϵ-tekens in de grammatica (bijvoorbeeld bij rows) staan voor 'leeg', corresponderend met de lege string.
- In colsSpec wordt het argument voor de `tabular`-omgeving als een identifier gespecificeerd. De eis dat deze identifier uit alleen de letters `l`, `c` en `r` mag bestaan komt dus niet tot uitdrukking in de grammatica.

☞ **2.1.1** De grammatica van Fig. 2.1 staat nog niet in de juiste vorm om als basis te dienen voor het algoritme van §4.3.4 van Watt & Brown. Gebruik §4.2.2 en §4.2.3 van Watt & Brown om de grammatica in EBNF-formaat te herschrijven en zorg ervoor dat de grammatica geen links-recursieve productieregels meer bevat.

Op Blackboard kunt u het bestand `Token.java` vinden. Dit bestand definieert de klasse `Token` voor het opslaan van de *tokens* (d.w.z. terminals) van de latexTabular-taal. Zoals u kunt zien worden de strings "`begin`", "`end`" en "`tabular`" beschouwd als aparte tokens.

Ter vergelijking: In §4.1.1 en aan het einde van §4.5 van Watt & Brown wordt de klasse `Token` voor Mini-Triangle besproken.

☞ **2.1.2** Schrijf nu een klasse `Scanner` die een latexTabular-specificatie kan scannen. De klasse dient tenminste de volgende twee methoden te ondersteunen.

```
public class Scanner {
    private InputStream     in;

    /**
     * Constructor.
     * @param in the stream from which the characters will be read
     */
    public Scanner(InputStream in) {
        this.in = in;
    }


    /**
     * Returns the next Token from the input.
     * @return the next Token
     * @throws SyntaxError when an unknown or unexpected character
     *         has been found in the input.
     */
    public Token scan() throws SyntaxError
}
```

Op de website van Vertalerbouw kunt u een 'lege huls' van de de klasse `Scanner.java` vinden.

U dient rekening te houden met het volgende.

- Als de `Scanner` een karakter inleest dat het niet kent dient er een `SyntaxError`-exceptie gegooid te worden. U dient deze (triviale) klasse `SyntaxError` zelf te definiëren.

- Een LaTeX `tabular`-specificatie kan ook TeX commentaar bevatten (zie bijvoorbeeld het eerdere voorbeeld, `sample-1.tex`. Commentaar in TeX begint met een procentteken (‘`%`’) en het commentaar loopt door tot het einde van de regel. (Merk op dat dit niet in de grammatica van figuur 2.1 staat: het is namelijk geen zaak van de parser maar van de scanner.)

Hint: In §4.5 van Watt & Brown wordt een scanner voor Mini-Triangle ontwikkeld.

☞ **2.1.3** Voeg aan de klasse `Scanner` een methode `main` toe, waarmee de `Scanner` getest kan worden. Voor elk token dat de scanner vindt, moet de naam van het token en de representatie van het token afgedrukt worden.

Gegeven het voorbeeld uit de inleiding (`sample-1.tex`), zou bijvoorbeeld de volgende uitvoer gegenereerd kunnen worden:

```
BSLASH              '\'
BEGIN               'begin'
LCURLY              '{'
TABULAR             'tabular'
RCURLY              '}'
LCURLY              '{'
IDENTIFIER          'lcr'
RCURLY              '}'
IDENTIFIER          'Aap'
AMPERSAND           '&'
IDENTIFIER          'Noot'
AMPERSAND           '&'
IDENTIFIER          'Mies'
DOUBLE_BSLASH       '\\'
IDENTIFIER          'Wim'
AMPERSAND           '&'
IDENTIFIER          'Zus'
AMPERSAND           '&'
IDENTIFIER          'Jet'
DOUBLE_BSLASH       '\\'
NUM                 '1'
AMPERSAND           '&'
NUM                 '2'
AMPERSAND           '&'
NUM                 '3'
DOUBLE_BSLASH       '\\'
IDENTIFIER          'Teun'
AMPERSAND           '&'
IDENTIFIER          'Vuur'
AMPERSAND           '&'
IDENTIFIER          'Gijs'
DOUBLE_BSLASH       '\\'
BSLASH              '\'
END                 'end'
LCURLY              '{'
TABULAR             'tabular'
RCURLY              '}'
Scanning OK. Number of lines: 8
```

2.2 Parser

In deze opgave gebruiken we de zojuist ontwikkelde scanner om een *recursive-descent* parser te ontwikkelen voor de latexTabular grammatica. Voor iedere non-terminal XYZ schrijven we een methode `parseXYZ` die ervoor zorgt dat de non-terminal XYZ ontleed wordt. We baseren ons uiteraard niet op de BNF-grammatica van Fig. 2.1, maar op de getransformeerde EBNF-grammatica van Vraag 2.1.1.

☞ **2.2.1** Schrijf een klasse `Parser` die de grammatica van Fig. 2.1 en Vraag 2.1.1 ontleedt. U dient daarvoor §4.3.4 van Watt & Brown te volgen.

Zij opgemerkt dat de `Parser` (nog) *geen* uitvoer mag genereren; de invoer mag alleen geparsed worden. Er wordt verder **geen** *abstracte syntax tree* opgebouwd.

De klasse dient tenminste de volgende twee methoden te implementeren.

```
public class Parser {
    /**
     * @param    scanner the Scanner object to be used for parsing
     * @requires scanner != null;
     */
    public Parser(Scanner scanner) {
        ...
    }


    /**
     * Parses the input as LaTeX tabular specification.
     * @returns {@code true} if parsing was successful
     */
    public boolean parse() {
        ...
    }

}
```

Enkele opmerkingen:

- De methoden `parseXYZ` moeten als `protected` methoden gedefinieerd worden. Hierdoor kan de `Parser` eenvoudig uitgebreid en veranderd worden.
- De `SyntaxError`-excepties die gegooid kunnen worden door de scanner moeten afgevangen worden in de methode `parse`.

☞ **2.2.2** Voeg aan de klasse `Parser` een methode `main` toe, waarmee de `Parser` getest kan worden.

2.3 Codegeneratie

De oorspronkelijke opgave was om LaTeX-tabellen naar HTML-tabellen om te zetten. Nu we een scanner en parser hebben om de latexTabular-grammatica te ontleden moeten we er nog voor

htmlTable	::=	BEGIN_TABLE rows END_TABLE
rows	::=	row∗
row	::=	BEGIN_ROW entries END_ROW
entries	::=	entry∗
entry	::=	BEGIN_ENTRY any-char∗ END_ENTRY
BEGIN_TABLE	::=	“<table border = "1">”
END_TABLE	::=	“</table>”
BEGIN_ROW	::=	“<tr>”
END_ROW	::=	“</tr>”
BEGIN_ENTRY	::=	“<td>”
END_ENTRY	::=	“</td>”

Figuur 2.2: EBNF grammatica van HTML table.

zorgen dat er een HTML-tabel gegenereerd wordt. De structuur van een HTML-tabel is beschreven in de grammatica van Fig. 2.2.

Om een HTML-tabel in een webbrowser te kunnen zien dienen er ook nog HTML-document tags om de tabel gezet te worden. Voor de tabel dient te komen: `<html><body>`, en na de tabel: `</body></html>`.

Voorbeeld. De LaTeX `tabular`-specificatie van het eerdere voorbeeld (`sample-1.tex`) zou vertaald kunnen worden naar het volgende (complete) HTML-document.

```
<html><body>
<table border="1">
<tr>
  <td>Aap</td>
  <td>Noot</td>
  <td>Mies</td>
</tr>
<tr>
  <td>Wim</td>
  <td>Zus</td>
  <td>Jet</td>
</tr>
<tr>
  <td>1</td>
  <td>2</td>
  <td>3</td>
</tr>
<tr>
  <td>Teun</td>
  <td>Vuur</td>
  <td>Gijs</td>
</tr>
</table>
</body></html>
```

☞ **2.3.1** Schrijf een klasse `ParserEmit`, die een subklasse is van de klasse `Parser` van Vraag 2.2.1. Het verschil met `Parser` is dat `ParserEmit` tijdens het parsen van een LaTeX `tabular` meteen HTML ‘code’ wegschrijft.

Hint: Ten opzichte van de superklasse `Parser` is de Java-code van de klasse `ParserEmit` een stuk korter: slechts een paar methoden van `Parser` hoeven overschreven te worden.

☞ **2.3.2** Test tenslotte uw klasse `ParserEmit` op enkele voorbeeld LaTeX `tabular`-bestanden, die u op Blackboard kunt vinden.

Het testen van een compiler op z'n gedrag bij foute invoer is minstens zo lastig als testen bij goede invoer. De volgende opgave vormt dan ook een goede voorbereiding op de dingen die komen gaan.

☞ **2.3.3** Schrijf zelf nog twee test-invoerfile waarin andere fouten zitten dan in `sample-4.tex` hierboven, en laat zien dat de compiler ook die fouten vindt.