

# ANTLR

## PRACTICUM

---

In dit practicum maken we kennis met ANTLR, de parser generator die ook bij de eindopdracht van Vertalerbouw gebruikt zal worden. We gebruiken ANTLR om de vertaler voor een eenvoudig expressie-taaltje uit te breiden.

### 3.1 Beginnen met Antlr

**Inleiding.** De officiële website van ANTLR is <http://www.antlr.org>. Deze website heeft onder meer een pagina om te beginnen met ANTLR: <http://www.antlr.org/wiki/display/ANTLR3/FAQ++Getting+Started>. Bij het vak Vertalerbouw wordt verder geen extra materiaal over ANTLR beschikbaar gesteld; via de website van ANTLR is alles te vinden.

**Installatie.** Van de website van ANTLR dient u de laatste (stabiele) versie van ANTLR te downloaden: versie 3.4. Er zijn verschillende distributies, maar het is het eenvoudigst om de complete JAR archive te downloaden: `antlr-3.4-complete.jar` (grootte: 2.2 Mb; zie ook Blackboard). Deze file dient in Java's `CLASSPATH` te worden opgenomen.

ANTLR is een command-line compiler generator. De methode `main` binnen de klasse `org.antlr.Tool` is het startpunt van ANTLR. Om te controleren of de bestanden goed geïnstalleerd zijn (en derhalve in Java's `CLASSPATH` staan), kunt u ANTLR als volgt aanroepen:

```
java org.antlr.Tool
```

ANTLR zal nu 'usage' informatie geven hoe de tool gebruikt dient te worden. Als de jar-bestanden niet allemaal in het `CLASSPATH` staan zal Java daarentegen een exceptie gooien.

Uit de 'usage' informatie kunt u opmaken dat ANTLR tenminste een bestand met extensie `.g` nodig heeft met daarin de definitie van grammatica. Gegeven een grammatica file `foo.g` kunt u ANTLR als volgt aanroepen om een vertaler te genereren:

```
java org.antlr.Tool foo.g
```

Het is overigens ook toegestaan om ANTLRWorks — ANTLR's ontwikkelomgeving voor grammatica's — of de Eclipse plugin voor ANTLR te gebruiken (zie <http://antlr3ide.sourceforge.net/>). De distributie van ANTLRWorks bevat alle benodigde ANTLR software en het is dan ook niet nodig om de source distributie van ANTLR ook nog op te halen. De voorbeelden in deze handleiding zullen er echter steeds vanuit gaan dat u de command-line versie van ANTLR gebruikt.

**Calc.** Op het inleidende hoorcollege over ANTLR is een eenvoudige, rekenmachine-achtige programmeertaal voor numerieke berekeningen behandeld: *Calc*. Tijdens het practicum van deze week gaan we deze taal en de bijbehorende vertaler verder uitbreiden. Aangeraden wordt om de slides van dit hoorcollege over ANTLR (nogmaals) te raadplegen. Deze slides geven extra uitleg over de opbouw en ANTLR-constructies van de *Calc*-vertaler.

Op Blackboard kunt u drie *.g* files vinden:

- *Calc.g*: de grammatica voor de lexer en parser van *Calc*,
- *CalcChecker.g*: de grammatica van de tree parser die de contextanalyse verzorgd, en
- *CalcInterpreter.g*: de grammatica van de tree parser die de *Calc* AST interpreteert.

Deze drie grammatica-bestanden definiëren gezamenlijk de vertaler en interpreter voor *Calc*. Op Blackboard staat ook de Java file *Calc.java* die de vier *Calc*-recognizers (nl. lexer, parser en de twee tree parsers) aan elkaar knoopt. Tenslotte is er een Java file *CalcException.java* die gebruikt wordt door *CalcChecker.g* bij het constateren van *Calc*-specifieke fouten in de invoer.

**Kenmerken van *Calc*.** Een programma in de taal *Calc* bestaat uit nul-of-meer declaraties gevolgd door één-of-meer statements. In de declaratie-sectie kunnen variabelen van het type *integer* gedeclareerd worden. Een variabele mag maar één keer gedeclareerd worden. De taal *Calc* heeft een monolitische blokstructuur.

In het statementgedeelte kunnen assignments en *print*-statements elkaar afwisselen. Bij een assignment krijgt een variabele de waarde van een expressie. Bij een *print*-statement wordt een expressie op het beeldscherm getoond. De operatoren van een expressie zijn de binaire optelling en aftrekking. Als operanden kunnen naast de variabelen ook getalnotaties in expressies voorkomen. Variabelen mogen alleen in statements gebruikt worden als ze daarvoor gedeclareerd zijn. Een voorbeeld-programma in *Calc* is het volgende:

```
// ex1.calc -- valid Calc program
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

Bij het executeren van het programma zal de waarde 15 op het beeldscherm afgedrukt worden.

- ☞ **3.1.1** Haal alle bronbestanden van *Calc* (d.w.z. de drie *.g* bestanden en de twee Java bestanden) van Blackboard. Gebruik ANTLR om een werkende vertaler voor *Calc* te genereren. Schrijf een paar *Calc* programma's en controleer dat de compiler naar behoren werkt.

Het Java programma *Calc.java* ondersteunt de optie *-ast*, om de AST van een *Calc* programma als een *String* af te drukken. Bekijk de file *Calc.java* en experimenteer met deze optie.

Zoals u wellicht gezien heeft, ondersteunt `Calc.java` ook een optie `-dot`. Hiermee is het mogelijk om een `.dot` bestand van de AST van het `Calc` programma te genereren. Een `.dot` bestand kan gevisualiseerd worden met het graaftekenprogramma `GraphViz` (<http://www.graphviz.org/>). Het is voor dit practicum echter niet nodig om dit programma te installeren.

Zoals gezegd breiden we in de rest van deze opgave de taal `Calc` en haar vertaler uit.

- ☞ **3.1.2** Voeg operators voor vermenigvuldiging en deling toe aan de taal `Calc`. Daarvoor dienen de *lexer*, de *parser* en beide *tree parsers* van `Calc` aangepast te worden. Let daarbij op het volgende.

- De gebruikelijke prioriteits-volgorde van operatoren dient in acht te worden genomen (d.w.z. vermenigvuldiging en deling gaan voor optellen en aftrekken). Het volgende `Calc` programma

```
print (3+4*5);
```

zal dus 23 op de standaard uitvoer moeten afdrukken.

- Binnen `CalcInterpreter` dient gecontroleerd te worden dat er niet door nul gedeeld wordt.

*Hint:* In Exercise 4.14 van het boek van Watt & Brown staat beschreven hoe een grammatica aangepast kan worden zodat het verschil in prioriteit tot uitdrukking komt in de parse boom. Ook de slides van het hoorcollege over ANTLR stippen de prioriteit van operatoren (precedence) aan.

- ☞ **3.1.3** Voeg aan `Calc`, analoog aan `print`, een procedure `swap` toe. Een aanroep van `swap(x, y)` – waarbij `x` en `y` twee gedeclareerde variabelen zijn – heeft als gevolg dat de inhoud van de twee variabelen `x` en `y` verwisseld wordt.

Daarvoor dienen weer alle ANTLR specificaties aangepast te worden.

- ☞ **3.1.4** Aan `Calc` zal nu een *if-then-else* *expressie* toegevoegd worden. Beschouw de expressie `if C then E1 else E2`. Als de expressie `C` niet gelijk is aan 0, levert de expressie `if C then E1 else E2` de waarde van `E1` op. Als de expressie `C` wel gelijk is aan 0, levert de expressie `if C then E1 else E2` de waarde van `E2` op.

Breid de grammatica specificaties van `Calc` zodanig uit dat nu ook een *if-then-else* *expressie* ondersteund wordt.

- ☞ **3.1.5** Voeg aan de taal `Calc` de *relationele operatoren* toe. Het gaat om de gebruikelijke binaire operatoren `<`, `<=`, `>`, `>=`, `==` en `!=`. Deze operatoren leveren een integer-resultaat op: 1 voor *true*, en 0 voor *false*. De relationele operatoren hebben een lagere prioriteit dan `PLUS` en `MINUS`, maar hoger dan de *if-then-else* operator.

In de oorspronkelijke taaldefinitie van `Calc` worden declaraties en statements strict gescheiden. Het is uiteraard gebruikersvriendelijker als declaraties en statements elkaar kunnen afwisselen.

- ☞ **3.1.6** Verander de grammaticaspecificaties van `Calc` dusdanig dat declaraties en statements elkaar kunnen afwisselen. De scope-regels blijven uiteraard wel hetzelfde: een variabele mag maar één keer gedeclareerd worden en een variabele mag alleen gebruikt worden als hij daarvoor gedeclareerd is.

Tenslotte moet gelden dat een `Calc` programma niet met een declaratie mag eindigen.

- ☞ **3.1.7** Verander het assignment statement van `Calc` nu in een (rechts-associatief) *multiple-assignment*-statement. Het volgende statement wordt dan een geldig `Calc` statement:

```
x := y := z := 27;
```

Eerst krijgt `z` hier de waarde `27`, die vervolgens wordt toegekend aan `y`, en tenslotte aan `x`.

Het is bij deze opgave toegestaan om de *lookahead*-constante `k` van `CalcParser` van `1` naar `2` te verhogen.

Het probleem bij *multiple-assignment* is dat zowel een assignment zelf als een expressie beide met een identifier kunnen beginnen. Op grond van slechts één lookahead-symbool kan de parser dan niet beslissen welke van de twee alternatieven gekozen moet worden. Het is echter wel degelijk mogelijk om multiple-assignment in een LL(1)-grammatica op te lossen.

- ☞ **3.1.8** Bij deze opgave dient u de grammatica in `Calc.g` zodanig aan te passen dat de *lookahead*-constante op `1` kan blijven staan. U dient daarbij de regels van `assignment` en `expr` samen te voegen.

### Meerdere keren over een AST lopen

ANTLR bevat een rijke collectie van (run-time) klassen en bijbehorende methoden. Het verdient aanbeveling om deze klassen eens aandachtig te bekijken; zie bijvoorbeeld de documentatie op <http://www.antlr.org/api/Java/>. Voor de volgende opgave zijn met name van belang:

- `org.antlr.runtime.tree.CommonTreeNodeStream`
- `org.antlr.runtime.tree.TreeParser`

Voor de volgende taalfeature van `Calc` (n.l. het `do-while` statement) dient de `CalcInterpreter` meerdere malen over eenzelfde deel van de AST te lopen. Dit kan op (tenminste) twee manieren die hieronder kort worden uitgelegd. Een soortgelijke aanpak wordt ook besproken op: <http://www.antlr.org/wiki/display/ANTLR3/Simple+tree-based+interpeter>

**Rewind.** Alle door ANTLR gegenereerde vertalers werken min of meer op dezelfde manier: ze herkennen een zin (sentence) in een stroom (stream) van objecten. Voor een lexer is dit een stroom van karakters, voor een parser een stroom van tokens, en voor een tree parser een stroom van `Tree nodes`. Een ANTLR tree parser loopt dan ook niet over een echte boom, maar over een ‘platgeslagen’ één-dimensionele stroom van `Tree` objecten: een `TreeNodeStream`.

In het geval van een `Calc` tree parser is deze stroom van `Tree` objecten een `CommonTreeNodeStream`. De `(Common)TreeNodeStream` van een tree parser is beschikbaar via de protected instantievariabele `input`. Twee handige methoden van de klasse `CommonTreeNodeStream` zijn `index` en `rewind`. Gegeven de variabele `input`, levert de aanroep `input.index()` een `int`-waarde `ix` op die correspondeert met de index van het huidige `TreeNode` in `input`. Deze index kan vervolgens gebruikt worden om later weer terug te keren naar de positie `ix` in de boom middels een aanroep `input.rewind(ix);`.

Ter illustratie een klein voorbeeld. Zij gegeven een tree parser `FooWalker` met de volgende rule `foo`:

```
foo : ^ (FOO bar)
    ;
```

We gaan het gedrag van `FooWalker` nu wijzigen. Als we nu een `FOO` node in de AST tegenkomen, willen we `foo` hier steeds `n`-keer overheen laten lopen. Dit kan als volgt:

```
foo[int n]
@init { int ix = input.index(); }
: ^(FOO bar)
    {   if (n > 0) {
        input.rewind(ix);
        foo(n-1);
      }
    }
;
```

Overal waar de non-terminal `foo` voorkomt in de grammatica van `FooWalker` zal nu `foo[n]` moeten worden gebruikt, waarbij `n` een `int`-waarde is. Merk op dat we hier ook gebruiken maken van het feit dat elke rule (zoals `foo`) vertaald wordt naar een methode.

**Nieuwe tree parser.** Een andere mogelijkheid om meerdere keren over een deel van de AST te wandelen is de volgende. We maken een *nieuwe* tree parser aan met een `(Common)TreeNodeStream` die start bij de node in de AST waar begonnen moet worden met wandelen. Vervolgens wordt de parse methode aangeroepen (d.w.z. de *rule*) die moet proberen over dit deel van de AST te lopen.

Het aanmaken van de `CommonTreeNodeStream` is eenvoudig. De klasse `CommonTreeNodeStream` heeft namelijk een constructor die als parameter een `CommonTreeNode` object meekrijgt (zie bijvoorbeeld `Calc.java`). Dit betekent dat van elke node in de AST een `CommonTreeNodeStream` gemaakt kan worden.

Het eerdere voorbeeld kan nu als volgt worden uitgewerkt.

```
foo[int n]
: ^(f=FOO bar)
    {   if (n > 0) {
        FooWalker fw = new FooWalker(new CommonTreeNodeStream(f));
        fw.foo(n-1);
      }
    }
;
```

- ☞ **3.1.9** Voeg aan de `Calc`-taal een `do-while`-statement toe. Het `do-while`-statement heeft de volgende syntax (gebruikmakend van ANTLR-notatie):

```
dowhileStatement : DO statements WHILE expression ;
statements       : (statement SEMICOLON!)+ ;
```

Het `dowhileStatement` voert de `statements` minstens één keer uit. Na afloop wordt de `expression` getest. Als de waarde van deze `expression` niet-nul is, dan worden de `statements` nogmaals uitgevoerd. Dit gaat door totdat de `expression` de waarde nul oplevert; dan wordt het `dowhileStatement` beëindigd.


Hieronder volgt een compleet `Calc`-programma met daarin een `do-while`-statement.

```
// dowhile.calc
var n: integer;
```

```
n := 10;
do
    print(n);
    n := n-1;
while n>0;
```

Op Blackboard kunt u een `Calc` programma vinden: `easter.calc`. Dit programma berekent de dag waarop Pasen valt voor de jaren 2012-2021. Het programma `easter.calc` zal de volgende output genereren.

```
2012
4
8
2013
3
31
2014
4
20
2015
4
5
2016
3
27
2017
4
16
2018
4
1
2019
4
21
2020
4
12
2021
4
4
```

-  **3.1.10** Gebruik het programma `easter.calc` om te controleren of uw complete `Calc`-compiler naar behoren werkt.