Week **4**

Codegeneratie

PRACTICUM

Tijdens dit vierde practicum wordt eerst ANTLR gebruikt om een compiler te genereren voor het decluse-taaltje van week 1. Daarna wordt er geoefend met het schrijven van TAM-assembler programma's, gebruikmakend van het programma `vb.TAM.Assembler`. Tenslotte wordt er een *codegenerator* ontwikkeld voor de `Calc`-compiler van week 3 die `TAM`-bytecode zal genereren.

4.1 Nogmaals decluse

In deze opgave gebruiken we ANTLR om een compiler te schrijven voor het decluse taaltje van Opgave 1.3 van week 1. In tegenstelling tot Opgave 1.3 ontwikkelen we nu een *two-pass* compiler in plaats van een *one-pass* compiler. Voor details t.a.v. het decluse-taaltje verwijzen we naar Opgave 1.3. Voor de volledigheid staat in Fig. 4.1 (nogmaals) de BNF grammatica van decluse. Zij opgemerkt dat de specificatie van zogenaamd *whitespace* niet in Fig. 4.1 is opgenomen. Het spreekt voor zich dat *whitespace* (zoals te doen gebruikelijk) genegeerd moet worden.

☞ **4.1.1** Gebruik ANTLR om een lexer en parser te genereren voor de decluse grammatica van Fig. 4.1. Probeer uw parser specificatie zo bondig mogelijk op te zetten.

Het is in de lexer en parser trouwens niet toegestaan om de context beperkingen van decluse te controleren; er mag geen referentie naar een *symbol table* o.i.d. in voor komen.

Als er geen syntaxfouten in een decluse 'programma' zijn aangetroffen, dienen analoog aan Opgave 1.3 de *context constraints* gecontroleerd te worden. Zie Opgave 1.3 voor een beschrijving van deze eisen.

☞ **4.1.2** Gebruik ANTLR om een *tree parser* te genereren die de context constraints van een decluse-programma controleert. U dient hierbij de door u bij Opgave 1.3 geïmplementeerde *symbol table* te gebruiken.

decluse	::=	"(" serie ")"
serie	::=	unit serie
	|	ϵ
unit	::=	decl
	|	use
	|	"(" serie ")"
decl	::=	"D:" id
use	::=	"U:" id
id	::=	letter id
	|	letter
letter	::=	LOWER | UPPER

Figuur 4.1: BNF grammatica van de decluse-taal.

Vergelijkend met de ad-hoc aanpak van 1.3 is de hier gevolgde aanpak snel, gestructureerd, elegant en eenvoudig uitbreidbaar. In het algemeen nemen *compilergeneratoren* zoals ANTLR de taalontwerper veel werk uit handen. En daarbij zijn de gegenereerde vertalers doorgaans maar een fractie minder efficiënt dan met de hand geschreven programmatuur.

4.2 TAM-Assembler

Bij de files voor dir practicum vindt u `vb.Tam.Assembler`, een *assembler* voor de Triangle Abstract Machine (TAM).[1] TAM zelf wordt uitgebreid beschreven in Watt & Brown (2000), met name in Appendix C.

Met behulp van de assembler kan een tekstrepresentatie van een TAM-assembly-programma omgezet worden naar TAM 'bytecode'. Het programma `vb.TAM.Assembler` wordt als volgt gebruikt:

```
java vb.TAM.Assembler foo.tasm foo.tam
```

waarbij `foo.tasm` het invoerbestand is en `foo.tam` het uitvoerbestand voor de TAM-bytecode. De assembler maakt gebruik van het (in Java 1.4 geïntroduceerde) package `java.util.regex`.

Ter illustratie staat hieronder een TAM-programma dat twee getallen inleest en vervolgens controleert of de beide getallen aan elkaar gelijk zijn.

```
; [file: eqtest.tasm, started: 13-Apr-2003, version: 16-Apr-2004]
; TAM Assembler program which reads two numbers and prints 'Y' if
; the two numbers are equal or 'N' if the numbers are not equal.

        PUSH        2           ; reserve space for the 2 numbers
        LOADA       0[SB]       ; address of n0: 0[SB]
        CALL        getint      ; read number into n0
        LOADA       1[SB]       ; address of n1: 1[SB]
        CALL        getint      ; read number into n1
        LOAD(1)     0[SB]       ; load number n0
        LOAD(1)     1[SB]       ; load number n1
```

[1] De TAM-assembler is in 2003 ontwikkeld door Matthijs Bomhoff, studentassistent Vertalerbouw 2002/2003.

```
        LOADL       1           ; size of the arguments is 1
        CALL        eq          ; n0 == n1 ?
        JUMPIF(0)   L1[CB]      ; if !(n0 == n1) then goto L1
        LOADL       89          ; load 'Y' on the stack
        CALL        put         ; print 'Y'
        JUMP        L2[CB]      ; jump over 'N' part.
L1:     LOADL       78          ; load 'N' on the stack
        CALL        put         ; print 'N'
L2:     POP(0)      2           ; pops the 2 numbers
        HALT
```

Dit bestand `eqtest.tasm` maakt deel uit van het practicummateriaal van deze week.

Enkele opmerkingen t.a.v. `vb.TAM.Assembler`:

- Een TAM-assembly-programma kan geannoteerd worden met end-of-line *commentaar*: commentaar begint met `;` en strekt zich uit tot het einde van de regel.
- In een TAM-assembler programma kunnen *symbolische labels* gebruikt worden (zoals `L1` en `L2` in het voorbeeld programma). De assembler zorgt er voor dat de juiste labels worden ingevuld in de TAM-bytecode. De labels dienen te beginnen met een letter, waarna nul of meer letters of cijfers kunnen volgen.

☞ **4.2.1** Schrijf een TAM-assembly-programma dat drie gehele getallen inleest van de standaard invoer en vervolgens het kleinste van deze drie getallen afdrukt op de standaard output.

Voorzie uw programma van zinvol commentaar en zorg ervoor dat uw programma zo efficient mogelijk werkt (en dus geen zinloze instructies bevat).

4.3 Codegenerator voor Calc

Bij deze opgave ontwikkelen we een ANTLR *tree parser* die gegeven een AST van een `Calc`-programma een TAM-assembler programma genereert. Het gegenereerde TAM-assembler programma kan vervolgens met behulp van `vb.TAM.Assembler` omgezet worden naar TAM-bytecode.

Een `Calc`-programma `foo.calc` zou dan bijvoorbeeld als volgt vertaald kunnen worden en vervolgens uitgevoerd te worden:

```
java Calc -code_generator < foo.calc > foo.tasm
java TAM.Assembler foo.tasm foo.tam
java TAM.Interpreter foo.tam
```

We gaan er hierbij vanuit dat de optie `-code_generator` bij `Calc` ervoor zorgt dat de *codegenerator* wordt gebruikt als tree parser (en dus niet `CalcInterpreter`). Merk op de gegenereerde TAM-assembler code hier naar de standaard output wordt geschreven. Voorts gaan we ervan uit dat `Trangle.jar` (met daarin `TAM.Assembler` en `TAM.Interpreter`) zich in Java's `CLASSPATH` bevindt.

☞ **4.3.1** Schrijf een codegenerator voor de volledige `Calc`-taal. De codegenerator dient TAM-assembly-code te genereren die vervolgens met assembler naar TAM-bytecode omgezet kan worden. De codegenerator dient als *tree parser* in ANTLR ontwikkeld te worden.

Pas uw `Calc`-compiler (met name `Calc.java`) zodanig aan dat nu óf de `CalcInterpreter` óf de te ontwikkelen `CalcCodeGenerator` als laatste *pass* over de AST-boom gaat.

Zorg ervoor dat de gegenereerde TAM-code hetzelfde gedrag vertoont als de eerder ontwikkelde interpreter die een `Calc`-programma simuleert over de AST-representatie. Met name het programma `easter.calc` dient nog dezelfde uitvoer te genereren.

Merk op dat we nu dus twee mogelijkheden tot onze beschikking hebben om een `Calc` programma te executeren; direct met de `Interpreter` of indirect (via TAM-code) met de `CalcCodeGenerator`. Interpretatie van de gegenereerde TAM-code (middels `TAM.Interpreter`) is overigens bijna een factor tien sneller dan directe interpretatie van de AST met de `CalcInterpreter`.