Week 5

Triangle Extensies

PRACTICUM

Tijdens dit laatste practicum van het eerste deel van het practicum van Vertalerbouw wordt de programmeertaal `Triangle` van Watt & Brown uitgebreid met twee nieuwe taalelementen: een **`repeat-until`**-statement en een **`case`**-statement.

Eindopdracht Vertalerbouw. U kunt alleen deelnemen aan de afsluitende eindopdracht van Vertalerbouw als alle practica van de eerste vijf weken uiterlijk aan het *begin* van het practicum van dinsdag 4 juni 2013 afgetekend zijn. Het is dus niet de toegestaan dat er nog op het practicum van 4 juni aan de practica van het eerste deel gewerkt wordt: vanaf die datum staat het practicum van Vertalerbouw in het teken van de eindopdracht.

5.1 Uitbreiden van Triangle

Voor de opdrachten van dit practicum is het nodig kennis te nemen van de Java-broncode in Watt & Brown, zoals ook in het hoorcollege behandeld. Appendix D bevat de klassendiagrammen van de Triangle vertaler. Met name diagram D.1 geeft een goed overzicht van de implementatie van de compiler.

Laatste versie van Triangle. Zorg dat u begint met de verbeterde versie van de Triangle broncode op Blackboard (en dus *niet* de oorspronkelijke versie van Watt & Brown).

Stappenplan. Voordat u begint met het aanpassen van de Triangle compiler is het zaak om eerst goed na te denken over de aanpassingen aan de Triangle *taal*:

1. Bedenk hoe de (concrete en abstracte) *syntax* uitgebreid moet worden voor de betreffende uitbreiding van Triangle.

2. Geef de *context-beperkingen* van de nieuwe uitbreiding aan.
3. Beschrijf de *semantiek* van de nieuwe uitbreiding.

Als u de aanpassingen aan de Triangle *taal* duidelijk geformuleerd hebt, volgen de aanpassingen aan de Triangle *compiler* (in de package directory `Triangle`) hier haast vanzelf uit.

4. Voeg nieuwe *tokens* toe door de klasse `SyntacticAnalyzer/Token.java` aan te passen.
5. Pas *eventueel* de scanner `SyntacticAnalyzer/Scanner.java` aan opdat dat de nieuwe tokens herkend worden.
6. Voeg nieuwe `AST`-klassen toe in de directory `AbstractSyntaxTrees`. Zorg er hierbij voor dat de nieuwe `AST`-klassen een subklasse worden van de meest geschikte `AST`-subklasse.
7. Pas de klasse `SyntacticAnalyzer/Parser.java` aan zodat de nieuwe taaluitbreiding (d.w.z. de tokens) herkend worden en de nieuwe `AST`-objecten gegenereerd worden.
8. Zorg dat voor elke nieuwe `AST`-klasse `XYZ` er een `visitXYZ` methode aan de interface `AbstractSyntaxTrees/Visitor.java` toegevoegd wordt.
9. Implementeer de context-beperkingen van de taaluitbreiding door nieuwe `visitXYZ`-methoden aan de klasse `ContextualAnalyzer/Checker.java` toe te voegen.
10. Implementeer de semantiek van de taaluitbreiding door de nieuwe `visitXYZ`-methoden toe te voegen aan de code generator klasse `CodeGenerator/Encoder.java`.
11. (Optioneel) Pas `TreeDrawer/LayoutVisitor.java` zodanig aan dat ook de nieuwe taalconstructies op de juiste manier 'getekend' worden door de klasse `TreeDrawer/Drawer.java`.
12. Test tenslotte de uitbreiding met enkele Triangle programma's. Test vooral op randgevallen en op constructies die niet geldig zijn volgens de contextregels. Bij het practicummateriaal op Blackboard staan enkele voorbeeldprogramma's.

Aangeraden wordt om eerst de *scanner* en *parser* helemaal correct te krijgen voordat begonnen wordt met de `Checker` en `Encoder`. Hiertoe moeten de nieuwe `visit`-methoden van deze twee `Visitor`-klassen in eerste instantie leeg gemaakt worden (d.w.z. de body bevat alleen **`return null;`**). De `Triangle.Compiler` zal dan weliswaar de AST opbouwen en ook de AST twee keer aflopen, maar niets doen voor nieuwe AST klassen.

☞ **5.1.1** *(Watt & Brown, Exercise 9.6a)* Voeg een nieuw iteratie-statement toe aan Triangle:

`repeat` C **`until`** E

Het **`repeat-until`**-commando wordt als volgt uitgevoerd. Eerst wordt het commando C uitgevoerd, waarna de expressie E wordt geëvalueerd. Als de waarde van de expressie E gelijk is aan *true*, wordt de iteratie beëindigd, anders wordt de lus weer uitgevoerd. De iteratie blijft net zolang doorgaan totdat E de waarde *true* oplevert. Merk op dat het commando C tenminste één keer uitgevoerd wordt. Het type van E moet `Boolean` zijn.

De volgende opgave betreft het **`case`**-statement waarnaar ook in de eerste huiswerkopgavenserie gevraagd werd.

☞ **5.1.2** *(Watt & Brown, Exercise 9.7)* Voeg aan de Triangle-taal een **`case`**-statement toe:

```
case E of
    IL1 : C1;
    IL2 : C2;
    ...
    ILm : Cm;
else: C0;
```

Het **`case`**-commando wordt als volgt uitgevoerd. Eerst wordt E geëvalueerd; als de waarde van E overeenkomt met een 'Integer-literal' IL_i, dan wordt het commando C_i uitgevoerd. Als de waarde van E niet overeenkomt met één van de Integer-literals, dan wordt het commando C_0 uitgevoerd. De expressie E moet van het type `Integer` zijn en de verschillende Integer-literals moeten verschillend zijn. Er zal altijd minstens één IL_i-C_i-paar aanwezig moeten zijn; m.a.w. $m \geq 1$.