Blatt 03: ANTLR

Carsten Gips, BC George (HSBI)

1 Zusammenfassung

Ziel dieses Aufgabenblattes ist die Erstellung eines einfachen *Pretty Printers* für eine einfache fiktive Sprache mit Expressions und Kontrollstrukturen.

Dazu werden Sie eine passende kontextfreie Grammatik definieren mit Lexer- und Parser-Regeln und dabei auch übliche Vorrangregeln beachten.

Für diese Grammatik erstellen Sie mit Hilfe von ANTLR einen Lexer und einen Parser, die zu einem Eingabeprogramm einen Parse-Tree erzeugen.

Den im Parse-Tree repräsentierten Code des Eingabeprogramms können Sie mit Hilfe einer Traversierung konsistent eingerückt wieder auf der Standardausgabe ausgeben - das ist der *Pretty Printer*.

Sie werden merken, dass viele Strukturen im Parse-Tree für diese Aufgabe nicht relevant sind und den Baum mit einer weiteren Traversierung in einen vereinfachten Baum, den sogenannten Abstract-Syntex-Tree (*AST*), transformieren und diesen erneut als formatierten Code auf der Konsole ausgeben.

2 Methodik

Nutzen Sie das Starter-Projekt in der Vorgabe.

Laden Sie sich das Projekt herunter, binden Sie es in Ihre IDE ein und vergewissern Sie sich, dass alles funktioniert: Führen Sie das enthaltene Programm aus, ändern Sie die mitgelieferten Beispielgrammatik.

Bauen Sie dann Ihre Grammatik für die Aufgabe schrittweise auf. Testen Sie diese mit Hilfe der Beispielprogramme der Zielsprache (s.u.) und überlegen Sie sich selbst weitere Code-Schnipsel, die Sie mit Ihrem Parser einlesen bzw. die Ihr Parser zurückweisen sollte.[1] Es empfiehlt sich, in dieser Phase mit dem ANTLR-Plugin für IntelliJ zu arbeiten.

Erkunden Sie dann die Strukturen Ihres Parse-Trees. Diese sind an die Regeln Ihrer Grammatik gekoppelt und sind deshalb so individuell wie Ihre Grammatik. Mit einer Traversierung des Baumes können Sie die gewünschte Ausgabe programmieren und auch die Erstellung des vereinfachten Baumes (AST).

3 Sprachdefinition

Ein Programm besteht aus einer oder mehreren Anweisungen (*Statements*).

3.1 Anweisungen (*Statements*)

Eine Anweisung ist eine einzeilige Befehlsfolge, beispielsweise eine Zuweisung oder eine Operation. Sie muss immer mit einem Newline abgeschlossen werden. Eine Anweisung hat keinen Wert.

[1] Um den Text lesbar zu halten, wird hier oft nur von “Parser” gesprochen - gemeint ist aber die gesamte auf diesem Blatt zu erstellende Toolchain: Lexer - Parser - AST - Ausgabe.

```
a := 10 - 5  # Zuweisung des Ausdruckes 10-5 (Integer-Wert 5) an die Variable a
b := "foo"   # Zuweisung des Ausdrucks "foo" (String) an die Variable b
```

Kontrollstrukturen (s.u.) zählen ebenfalls als Anweisungen.

3.2 Ausdrücke (*Expressions*)

Die einfachsten Ausdrücke sind Integer- oder String-Literale. Variablen sind ebenfalls Ausdrücke. Komplexere Ausdrücke werden mit Hilfe von Operationen gebildet, dabei sind die Operanden jeweils auch wieder Ausdrücke. Ein Ausdruck hat/ergibt immer einen Wert.

Die Operatoren besitzen eine Rangfolge, um verschachtelte Operationen aufzulösen. Sie dürfen daher nicht einfach von links nach rechts aufgelöst werden. Die Rangfolge der Operatoren entspricht der üblichen Semantik (vgl. Java, C, Python).

Es gibt in unserer Sprache folgende Operationen mit der üblichen Semantik:

3.2.1 Vergleichsoperatoren

Operation	Operator
Gleichheit	==
Ungleichheit	!=
Größer	>
Kleiner	<

3.2.2 Arithmetische Operatoren

Operation	Operator
Addition / String-Literal-Verkettung	+
Subtraktion	-
Multiplikation	*
Division	/

3.2.3 Beispiele für Ausdrücke

```
10 - 5  # Der Integer-Wert 5
"foo"   # Der String "foo"
```

```
a       # Wert der Variablen a
a + b   # Ergebnis der Addition der Variablen a und b
```

3.3 Bezeichner

Werden zur Bezeichnung von Variablen verwendet. Sie bestehen aus einer Zeichenkette der Zeichen `a-z`,`A-Z`, `0-9`, `_`. Bezeichner dürfen nicht mit einer Ziffer `0-9` beginnen.

3.4 Variablen

Variablen bestehen aus einem eindeutigen Bezeichner (Variablennamen). Den Variablen können Werte zugewiesen werden und Variablen können als Werte verwendet werden. Die Zuweisung erfolgt mithilfe des `:=`-Operators. Auf der rechten Seite der Zuweisung können auch Ausdrücke stehen.

```
a := 5      # Zuweisung des Wertes 5 an die Variable a
a := 2 + 3  # Zuweisung des Wertes 5 an die Variable a
```

3.5 Kommentare

Kommentare werden durch das Zeichen # eingeleitet und umfassen sämtliche Zeichen bis zum nächsten Newline.

3.6 Kontrollstrukturen

3.6.1 While-Schleife

While-Schleifen werden mit dem Schlüsselwort **`while`** eingeleitet. Sie bestehen im Weiteren aus einer Bedingung, die durch ein **`do`** abgeschlossen wird, einer Folge von Anweisungen und werden mit dem Schlüsselwort `end` abgeschlossen.

Die Bedingung kann aus einem Vergleichsausdruck bestehen.

```
while <Bedingung> do
    <Anweisung_1>
    <Anweisung_2>
end
```

```
a := 10
b := 0
while a >= 0 do
    a := a - 1
    b := b + 9
end
```

3.6.2 Bedingte Anweisung (If-Else)

Eine bedingte Anweisung besteht immer aus genau einer **`if`**-Anweisung, gefolgt von einer Bedingung, die mit einem **`do`** abgeschlossen wird und einer Folge von Anweisungen.

Danach wird die bedingte Anweisung entweder mit dem Schlüsselwort `end` abgeschlossen oder es folgt genau ein **`else`**-Teil.

Ein **`else`**-Teil wird mit dem Schlüsselwort **`else`** eingeleitet. Darauf folgt ein **`do`** und eine Folge von Anweisungen. Der **`else`**-Teil wird mit dem Schlüsselwort `end` abgeschlossen.

```
if <Bedingung> do
    <Anweisung_1>
    <Anweisung_2>
end
```

```
if <Bedingung> do
    <Anweisung>
else do
    <Anweisung>
end
```

```
a := "abc"
if a < "abc" do
    a := "wuppie"
else do
    a := "nope"
end
```

3.7 Datentypen

Unsere Sprache hat zwei eingebaute Datentypen, für die entsprechende Literale erkannt werden müssen:

Datentyp	Definition der Literale
`int`	eine beliebige Folge der Ziffern `0-9`
`string`	eine beliebige Folge von ASCII-Zeichen, eingeschlossen in `"`

Die Sprache ist dynamisch typisiert, d.h. beim Parsen werden Ihnen keine Typ-Angaben im Code begegnen. Aber Sie müssen die entsprechenden Werte (Literale) parsen können.

3.8 Beispiele

```
a := "wuppie fluppie"
```

```
a := 0
if 10 < 1 do
```

```
    a := 42
else do
    a := 7
end
```

4 Aufgaben

4.1 Grammatik (4P)

Definieren Sie für die obige Sprache eine geeignete ANTLR-Grammatik.

Sie werden sowohl Lexer- als auch (rekursive) Parser-Regeln benötigen. Beachten Sie die üblichen Vorrangregeln für die Operatoren, orientieren Sie sich hier an Sprachen wie Java oder Python oder C.

4.2 Pretty Printer (3P)

Erzeugen Sie mithilfe der Grammatik und ANTLR einen Lexer und Parser. Damit können Sie syntaktisch korrekte Eingabe-Programme in einen Parse-Tree überführen.

Programmieren Sie eine Traversierung Ihres Parse-Trees, in der Sie syntaktisch korrekte Programme konsistent eingerückt ausgeben können.

Jede Anweisung soll auf einer eigenen Zeile stehen. Die Einrückung soll mit Leerzeichen erfolgen und konsistent sein. Sie brauchen keine Begrenzung der Zeilenlänge implementieren.

Demonstrieren Sie die Fähigkeiten an mehreren Beispielen mit unterschiedlicher Komplexität.

Beispiel:

Aus

```
a     := 0
    if    10 < 1
       do
a    :=     42      # Zuweisung des Wertes 42 an die Variable a
else do
        a :=      7
  end
```

soll

```
a := 0
if 10 < 1 do
    a := 42
else do
    a := 7
end
```

werden.

Hinweis: Es geht nur um die Ausgabe syntaktisch korrekter Programme. Sie brauchen sich um die Semantik (z.B. passende Typen wie etwa keine Multiplikation von Strings mit Integern o.ä.) noch keine Gedanken machen! Achten Sie auf die korrekten Einrücktiefen. Die Zeilenlänge spielt hier keine Rolle, es wird einfach direkt nach jedem Statement umgebrochen (bzw. wie bei den Kontrollstrukturen gezeigt).

Hinweis: Das Thema Pretty Printing ist interessant und kann recht schnell ziemlich aufwändig werden. Sie finden im Paper "A prettier printer" von Philip Wadler (Wadler 2003) und im Blog "The Hardest Program I've Ever Written" von Bob Nystrom (Nystrom 2015) gut geschriebene Beiträge, um tiefer in die Materie einzusteigen.

4.3 AST (3P)

Beim Parsen bekommen Sie von ANTLR einen Parse-Tree zurück, der direkt die Struktur Ihrer Grammatik widerspiegelt. Die einzelnen Zweige sind damit in der Regel aber auch viel zu tief verschachtelt.

Überlegen Sie sich, welche Informationen/Knoten Sie für die formatierte Ausgabe wirklich benötigen - das ist Ihr Abstract-Syntex-Tree (*AST*).

Programmieren Sie eine Transformation des Parse-Tree in die von Ihnen hier formulierten AST-Strukturen. Dies können Sie beispielsweise mit einer passenden Traversierung (Visitor-Pattern) erreichen.

Passen Sie den Pretty-Printer so an, dass er auch den AST ausgeben kann. (Alternativ können auch einen zweiten Pretty-Printer für den AST implementieren.)

Quellen

Nystrom, R. 2015. „The Hardest Program I've Ever Written". 2015. https://journal.stuffwithstuff.com/2015/09/08/the-hardest-program-ive-ever-written/.

Wadler, P. 2003. „A Prettier Printer". *The Fun of Programming, Cornerstones of Computing*, 223–43. https://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf.

> **Last modified:** f7ac9d2 (reformat using shorter lines, 2025-08-09)