Vogon

Paul Lenz

Das komplette Referenzhandbuch

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Sprache Vogon ist eine einfache, streng typisierte Sprache die sich einer an C angelehnten Syntax bedient. Sie wurde entworfen um im Rahmen von Vorlagen für Emails und Rechnungen innerhalb der Software Afterbuy ein Höchstmaß an Flexibilität zu ermöglichen. Sie erweitert damit das Spektrum an Möglichkeiten das derzeit bereits mit dem Variablenparser geboten wird.
Ihre Einfachheit zeichnet sich durch eine feste und eng begrenzte Menge an Datentypen und wenige Kontrollstrukturen aus. Obwohl Methoden zur Verfügung stehen, ist es nicht möglich eigene zu definieren und der Skriptcode muss nicht in einer speziellen Methode hinterlegt werden um ausgeführt werden zu können. Ein Skript ist demnach eine Abfolge von Ausdrücken und Anweisungen. Dieser Ansatz betrachtet eine Vorlage somit als abgeschlossene Einheit und verringert die Komplexität, so dass der Umgang leichter fällt.
Im ersten Teil des vorliegenden Dokumentes wird eine Einführung im Umgang mit Vogon zum Schreiben von Templates vermittelt. Vogon selbst ist jedoch nicht allein auf Templates beschränkt. Der an Lua angelehnte Ansatz ermöglicht eine unendliche[1] Erweiterung des Funktionsumfanges, da aufrufbare Methoden durch den ausführenden Prozess zur Verfügung gestellt werden.
Im zweiten Teil wird zusätzlich noch der AST-Interpreter *Xpomul* vorgestellt und im Detail betrachtet. Dieser ist die Standardklasse zum Ausführen von Vogon-Skripten. Da dieses Thema sehr technisch ist, setzt es einiges an Hintergrundwissen voraus. Hier wird auch erläutert wie man selbst Methoden, zur Ausführung innerhalb eines Skriptes, definieren und einbinden kann.

[1] Zumindest theoretisch, da dies unendliche Ressourcen zum Speichern der Methoden erfordern würde. Wenn man jedoch annimmt, dass aus dem unbegrenzten Vorrat zu jedem beliebigen Zeitpunkt nur eine endliche Teilmenge zur Verfügung steht, so bleibt die Aussage auch in der Praxis wahr.

Teil I

Benutzerhandbuch

Kapitel 2

Scriptsprache für Templates

Templates können derzeit mit Variablen versehen werden, um sie dynamisch zu halten. Wenngleich das schon eine mächtige Möglichkeit für Templates ist, gibt es Fälle in denen die damit gebotene Funktionalität nicht hinreichend ist. An diesem Punkt erweitert Vogon die Möglichkeiten mit einem deutlich höheren Maß an Flexibilität in der Gestaltung der Templates.

2.1 Ausdrücke und Anweisungen

Ein Script in der Sprache Vogon ist eine Folge von Anweisungen und Ausdrücken. Eine Anweisung kann dabei das Definieren einer Variablen, das Aufrufen einer Methode oder das Zuweisen eines Wertes zu einer Variablen sein. Ausdrücke ergeben immer Werte die dann in einer Anweisung verwendet werden können, um sie einer Variablen zuzuweisen oder an eine Methode zu übergeben.
So ist ein Ausdruck zum Beispiel das Addieren zweier Zahlen (`25 + 17`), oder der Vergleich zweier Zeichenketten (`"Birne" == "Apfel"`).
Die einfachste Form der Anweisung ist der Aufruf einer Methode. Dazu schreibt man den Namen der Methode gefolgt von Parametern, eingeschlossen in runde Klammern. Erwartet eine Methode keine Parameter so werden die Klammern leer gelassen. Dies wird in einem einfachen Hallo-Welt-Beispiel verdeutlicht.

```
1 Print("Hallo Welt.");
```

Beachten Sie, dass Anweisungen immer mit einem Semikolon (`;`) beendet werden müssen.
Dieser einfache Quelltext stellt schon ein vollständiges Vogon-Skript dar und ruft die Methode `Print()` auf, um die Zeichenkette `"Hallo Welt."` auszugeben. Diese Methode erwartet mindestens einen Parameter, jedoch werden die zusätzlichen Varianten später erläutert. Als Beispiel einer Methode ohne Parameter eignet sich `Pi()`. Diese gibt einen Wert für die Kreiszahl π zurück.

```
1 Pi();
```

Der gezeigte Aufruf ist jedoch weitesgehend sinnlos, da der von der Methode zurückgegebene Wert nicht verwendet wird. Hier könnte man die Methoden `Print()` und `Pi()` verbinden.

```
1 Print(Pi());
```

Erwartet eine Methode mehrere Parameter, so werden diese durch Kommata getrennt.

2.2 Datentypen und Variablen

Um Ausdrücke noch besser nutzen zu können bietet Vogon die Möglichkeit innerhalb eines Skriptes eigene Variablen zu definieren. Dafür stehen insgesamt sechs Datentypen zur Verfügung. Die Tabelle 2.2 listet diese sowie deren Wertebereiche auf.
Variablen werden nun in der Form `<Datentyp> <Variablenname>;` geschrieben. Dabei ist für den Variablennamen zu beachten, dass dieser nur aus Klein- und Großbuchstaben bestehen darf. Zahlen und Unterstriche (`_`) sind nicht zulässig.[1] Im folgenden Beispiel wird eine Ganzzahlvariable mit dem Namen `zahl` definiert.

```
1 int zahl;
```

[1] Da Zahlen und Unterstriche in Variablennamen zu fehleranfälligen Bezeichnungen führen können, wurden diese nicht zugelassen. Das dient in erster Linie der Eingrenzung möglicher Quellen für logische Fehler, die ihrer Natur entsprechend oft nur schwer zu finden sind.

Name	Bedeutung	Kleinster Wert	Größter Wert
int	Ganzzahlen	-9223372036854775808 (-2^{63})	9223372036854775807 ($2^{63}-1$)
float	Fließkommazahlen	-1,79769313486232E+308	1,79769313486232E+308
bool	Wahrheitswerte	false	true
date	Datum und Uhrzeit	01.01.0001 00:00:00	31.12.9999 23:59:59
string	Zeichenketten	Beliebig groß	
structured	Strukturiert	-	

Tabelle 2.2: Verfügbare Datentypen

Es ist auch möglich mehrere Variablen gleichen Typs in einer Zeile zu definieren, in dem die Namen durch Kommata getrennt angegeben werden.

```
1 float summe, versandkosten, mehrwertsteuer;
```

Natürlich können diesen Variablen nun Werte zugewiesen werden. Der Tabelle kann entnommen werden in welchen Bereichen sich gültige Werte für jeden Datentyp bewegen müssen. Im folgenden Beispiel werden den eben definierten Variablen Werte zugewiesen.

```
1 summe = 42.0;
2 versandkosten = 3.5;
3 mehrwertsteuer = 7.98;
```

Beachten Sie bei der Zuweisung von Fließkommazahlen, dass diese im englischen Format mit Punkt als Dezimaltrenner geschrieben werden müssen, wie im Beispiel eben gezeigt. Bei der Zuweisung von Werten zu Variablen ist zu beachten, dass der zuzuweisende Wert auch zum Datentyp der Variable passen muss. So würde die Zuweisung im folgenden Beispiel einen Fehler verursachen, da eine Zeichenkette nicht einer Ganzzahlvariablen zugewiesen werden kann.

```
1 int zahl;
2 zahl = "42";
```

Generell können folgende Zuweisungen ausgeführt werden[2]:

- Ganzzahl zu Ganzzahl
- Ganzzahl zu Fließkommazahl
- Fließkommazahl zu Fließkommazahl
- Wahrheitswert zu Wahrheitswert
- Datum zu Datum
- Zeichenkette zu Zeichenkette
- Strukturiert zu Strukturiert

Es ist also nur möglich Werte des gleichen oder eines ähnlichen (im Falle von Ganz- und Fließkommazahlen) Typs zuzuweisen. In Abschnitt 2.4.4 werden Konvertierungsmethoden gezeigt, mit deren Hilfe sich Werte von einem Typ in einen anderen konvertieren lassen.
Ein spezieller Datentyp aus der Tabelle ist der Typ `Structured`. Dieser verbindet mehrere einzelne Variablen zu einer einzigen, so dass redundante Informationen nicht im Variablennamen selbst geführt werden müssen. Wichtig ist hierbei, dass solche Variablen nur vom System erzeugt werden können. Ihr Zweck ist es Informationen aus dem System im Script zur Verfügung zu stellen. In Abschnitt 2.9 werden diese noch genauer behandelt.
Es ist jedoch möglich Variablen dieses Typs anzulegen und ihnen den Wert von systemgestellten Variablen zuzuweisen. Dies dient der einfacheren Verwendung.

[2]Die Zuweisung von Fließkommazahlen zu Ganzzahlvariablen wurde bewusst weggelassen, da hier ein Datenverlust auftreten kann, wenn Nachkommstellen abgeschnitten werden. Hier können sich schwer zu findende logische Fehler einschleichen.

2.3 Operatoren

Operatoren werden verwendet um Werte zu verändern und Ausdrücke zu erzeugen. In Vogon werden die Operatoren in drei Kategorien unterteilt die in den folgenden Abschnitten erläutert werden.
Grundsätzlich kennt Vogon ein- und zweistellige Operatoren. Die Anzahl der Stellen gibt die Anzahl der Operanden an. Für eine allgemeine einstellige Operation $\circ$ gilt die Schreibweise $\circ a$, wobei a der Operand ist. Eine zweistellige Operation hat die Schreibweise $a \circ b$, wobei a als linker und b als rechter Operand bezeichnet werden. Ob eine Operation für die Datentypen der gegebenen Operanden definiert ist, kann den jeweiligen Tabellen in den folgenden Abschnitten entnommen werden.

2.3.1 Arithmetische Operatoren

Arithmetische Operatoren ermöglichen es Zahlwerte, also Variablen oder Werte vom Typ `int` oder `float` zu Addieren ($+$), zu Subtrahieren($-$), zu Multiplizieren ($*$) und zu Dividieren ($/$). Zusätzlich gibt es noch einen Modulo-Operator ($\%$) welcher den ganzzahligen Rest ermittelt.
Zusätzlich zu `int` und `float` ist die Addition zusammen mit der Subtraktion auch für Werte vom Typ `date` definiert.

```
float summe, versandkosten, mehrwertsteuer, gesamtsumme;
summe = 42.0;
versandkosten = 3.5;
mehrwertsteuer = summe * 0.19;
gesamtsumme = summe + mehrwertsteuer + versandkosten;
```

Operator	Linker Operand	Rechter Operand	Rückgabetyp
$+$	Int	Int	Int
$+$	Int	Float	Float
$+$	Float	Int	Float
$+$	Float	Float	Float
$+$	Date	Date	Date
$-$	Int	Int	Int
$-$	Int	Float	Float
$-$	Float	Int	Float
$-$	Float	Float	Float
$-$	Date	Date	Date
$*$	Int	Int	Int
$*$	Int	Float	Float
$*$	Float	Int	Float
$*$	Float	Float	Float
$/$	Int	Int	Int
$/$	Int	Float	Float
$/$	Float	Int	Float
$/$	Float	Float	Float
$\%$	Int	Int	Int
$\%$	Int	Float	Float
$\%$	Float	Int	Float
$\%$	Float	Float	Float

2.3.2 Relationsoperatoren

Relationsoperatoren stellen zwei Werte in ein Verhältnis zueinander und geben einen Wert vom Typ `bool` zurück.

Operator	Linker Operand	Rechter Operand	Rückgabetyp
$>$	Int	Int	Bool
$>$	Int	Float	Bool
$>$	Float	Int	Bool
$>$	Float	Float	Bool
$>$	Date	Date	Bool
$>$	String	String	Bool
$<$	Int	Int	Bool
$<$	Int	Float	Bool

<	Float	Int	Bool
<	Float	Float	Bool
<	Date	Date	Bool
<	String	String	Bool
>=	Int	Int	Bool
>=	Int	Float	Bool
>=	Float	Int	Bool
>=	Float	Float	Bool
>=	Date	Date	Bool
>=	String	String	Bool
<=	Int	Int	Bool
<=	Int	Float	Bool
<=	Float	Int	Bool
<=	Float	Float	Bool
<=	Date	Date	Bool
<=	String	String	Bool
==	Int	Int	Bool
==	Int	Float	Bool
==	Float	Int	Bool
==	Float	Float	Bool
==	Bool	Bool	Bool
==	Date	Date	Bool
==	String	String	Bool
!=	Int	Int	Bool
!=	Int	Float	Bool
!=	Float	Int	Bool
!=	Float	Float	Bool
!=	Bool	Bool	Bool
!=	Date	Date	Bool
!=	String	String	Bool

2.3.3 Logische Operatoren

Logische Operatoren können verwendet werden um Werte vom Typ `bool` zu verknüpfen und logische Ausdrücke zu erzeugen. In den meisten Fällen werden die Werte von Relationsoperatoren verwendet werden.

```
bool a;
a = 5 > 3 && 3 > 1;
```

In diesem Beispiel wird der Variablen `a` der Wert `true` zugewiesen, da 5 größer ist als 3 (`5 > 3`) und (`&&`) 3 größer ist als 1 (`3 > 1`).
Die logischen Operatoren enthalten mit dem Nicht-Operator (`!`) den einzigen einstelligen Operator von Vogon.

Operator	Linker Operand	Rechter Operand	Rückgabetyp	Bemerkung
&&	Bool	Bool	Bool	Und
||	Bool	Bool	Bool	Oder
^	Bool	Bool	Bool	Exklusives Oder
!	Bool	Keiner	Bool	Nicht

2.3.4 Der dreistellige Operator `?:`

Oftmals hat man Situationen in denen man abhängig von einem Wahrheitswert den einen oder anderen Wert einer Variablen zuweisen muss. Die einfachste Lösung wäre hier mit der einfachen Verzweigung durch `if` zu arbeiten.

```
string text;
if (a < b)
  text = "a ist kleiner als b.";
else
  text = "b ist kleiner oder gleich a.";
```

Diese Lösung erfordert vier Zeilen für die Verzweigung und lässt sich mit dem dreistelligen Operator auf eine Zeile verkürzen[3].

```
string text;
text = a < b ? "a ist kleiner als b." : "b ist kleiner oder gleich a.";
```

Das allgemeine Format dieses Operators lautet `<Bedingung> ? <Wahrwert> : <Falschwert>` und bedeutet sinngemäß: "Wenn die Bedingung erfüllt ist, so gib `<Wahrwert>` zurück, sonst `<Falschwert>`." Die Bedingung muss ein Ausdruck sein, der einen Wert vom Typ `bool` erzeugt. Wenn der Wert der Bedingung `true` ist, so wird der Wert von `<Wahrwert>`, sonst der Wert von `<Falschwert>` zurückgegeben. Für `<Wahrwert>` und `<Falschwert>` können beliebige Ausdrücke angegeben werden. Es können also auch Methoden aufgerufen werden die einen Wert zurückgeben.
Wichtig ist dabei, dass die Datentypen von `<Wahrwert>` und `<Falschwert>` übereinstimmen müssen. Die einzige Ausnahme dieser Regel sind die Datentypen `int` und `float`.
So ist die folgende Anweisung durchaus möglich, obwohl `<Wahrwert>` (`1`) den Typ `int` und `<Falschwert>` (`2.0`) den Typ `float` hat.

```
Print(1 >= 2 ? 1 : 2.0);
```

Vorsicht ist hier nur geboten, wenn dieser Operator bei einer Zuweisung verwendet wird.

```
float zahl1;
int zahl2;

zahl1 = 1 >= 2 ? 1 : 2.0;

zahl2 = 1 >= 2 ? 1 : 2.0;
```

In obigem Beispiel wird die Zuweisung in Zeile vier ohne Probleme funktionieren. Die Zuweisung in Zeile sechs wird jedoch einen Fehler auslösen, da der Wert des Ausdrucks `1 >= 2 ? 1 : 2.0` den Typ `float` hat und einer Variablen vom Typ `int` zugewiesen werden soll. Der Wert des dreistelligen Operators nimmt immer dann den Typ `float` an, wenn mindestens einer der Werte den Typ `float` hat. Ist der zweite Wert nicht vom Typ `float`, so kann er nur den Typ `int` haben.
Die folgende Anweisung wäre also nicht möglich, da `<Wahrwert>` den Typ `int` und `<Falschwert>` den Typ `string` hat.

```
Print(1 >= 2 ? 1 : "2.0");
```

Eine mögliche Korrektur dessen wäre das folgende Beispiel.

```
Print(1 >= 2 ? 1 : ToFloat("2.0"));
```

Die Methode `ToFloat` konvertiert eine Zeichenkette in einen Wert vom Typ `float`, sofern möglich. Diese Methode wird später noch genauer betrachtet. Mit ihr würde das eben gezeigte Beispiel jedoch funktionieren.

2.4 Methoden

In Vogon sind Methoden die Hauptarbeitstiere. Sie stellen die gesamte Funktionalität zur Verfügung. Welche Methoden verwendet werden können, wird im Folgenden in Form einer Referenz vorgestellt und näher beschrieben.

2.4.1 Methoden für Zeichenketten

Es gibt eine Reihe von Methoden um mit Zeichenketten zu arbeiten. Wichtig ist, dass keine dieser Methoden den Wert der ihr übergebenen Zeichenkette verändert.

Print
Parameter: String
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Zeichenkette aus.
Beispiel: Print("Hallo");

Print
Parameter: Int
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Ganzzahl aus.
Beispiel: Print(42);

[3]Die Einsparung an Zeilen geht dabei jedoch unter Umständen zu Lasten der Lesbarkeit.

Print
Parameter: Float
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Fließkommazahl aus.
Beispiel: Print(3.5);

Print
Parameter: Bool
Rückgabetyp: Void
Beschreibung: Gibt die angegebenen logischen Wert aus.
Beispiel: Print(true);

Print
Parameter: Date
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Datum aus.
Beispiel: Print(GetDate());

PrintLine
Parameter: String
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Zeichenkette aus und bewegt die Ausgabe in eine neue Zeile.
Beispiel: PrintLine("Hallo");

PrintLine
Parameter: Int
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Ganzzahl aus und bewegt die Ausgabe in eine neue Zeile.
Beispiel: PrintLine(42);

PrintLine
Parameter: Float
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Fließkommazahl aus und bewegt die Ausgabe in eine neue Zeile.
Beispiel: PrintLine(3.5);

PrintLine
Parameter: Bool
Rückgabetyp: Void
Beschreibung: Gibt die angegebenen logischen Wert aus und bewegt die Ausgabe in eine neue Zeile.
Beispiel: PrintLine(true);

PrintLine
Parameter: Date
Rückgabetyp: Void
Beschreibung: Gibt die angegebene Datum aus und bewegt die Ausgabe in eine neue Zeile.
Beispiel: PrintLine(GetDate());

Format
Parameter: String, Array
Rückgabetyp: String
Beschreibung: Formatiert die Zeichenkette mit den angegebenen Werten.
Beispiel: Format("Der Artikel wiegt 0 Kg.", 3.5);

Replace
Parameter: String, String, String
Rückgabetyp: String
Beschreibung: Ersetzt in der angegebenen Zeichenkette jedes Vorkommen der zweiten Zeichenkette durch die dritte und gibt die neue Zeichenkette zurück.
Beispiel: Replace("Hallo Welt.", "Welt", Üniversum");

Contains
Parameter: String, String
Rückgabetyp: Bool
Beschreibung: Ermittelt, ob die zweite Zeichenkette in der ersten vorkommt.
Beispiel: Contains("Hallo Welt.", "Hallo");

ContainsAny
Parameter: String, Array
Rückgabetyp: Bool
Beschreibung: Ermittelt, ob eine der Zeichenketten in der ersten vorkommt.
Beispiel: ContainsAny("Hallo Welt.", "Hallo", "Welt", "World.");

ContainsAll
Parameter: String, Array
Rückgabetyp: Bool
Beschreibung: Ermittelt, ob alle angegebenen Zeichenketten in der ersten vorkommen.
Beispiel: ContainsAll("Hallo Welt.", "Hallo", "Welt", "World.");

StartsWith
Parameter: String, String
Rückgabetyp: Bool
Beschreibung: Ermittelt ob die zweite Zeichenkette am Anfang der ersten Zeichenkette vorkommt.
Beispiel: StartsWith("Hallo Welt.", "Hallo");

EndsWith
Parameter: String, String
Rückgabetyp: Boolean
Beschreibung: Ermittelt ob die zweite Zeichenkette am Ende der ersten Zeichenkette vorkommt.
Beispiel: EndsWith("Hallo Welt.", "Welt.");

Length
Parameter: String
Rückgabetyp: Int
Beschreibung: Ermittelt die Länge einer Zeichenkette.
Beispiel: Length("Hallo Welt.");

2.4.2 Mathematische Methoden

Abs
Parameter: Int
Rückgabetyp: Int
Beschreibung: Ermittelt den Betrag einer Ganzzahl.
Beispiel: Abs(-5);

Abs
Parameter: Float
Rückgabetyp: Float
Beschreibung: Ermittelt den Betrag einer Fließkommazahl.
Beispiel: Abs(-3.5);

Round
Parameter: Float
Rückgabetyp: Float
Beschreibung: Rundet die angegebene Fließkommazahl.
Beispiel: Round(3.5);

Round
Parameter: Float, Int
Rückgabetyp: Float
Beschreibung: Rundet die angegebene Fließkommazahl mit der angegebenen Genauigkeit.
Beispiel: Round(3.1415925, 3);

Floor
Parameter: Float
Rückgabetyp: Float
Beschreibung: Ermittelt die größte Ganzzahl die kleiner oder gleich der angegebenen Fließkommazahl ist.
Beispiel: Floor(3.5);

Ceiling
Parameter: Float
Rückgabetyp: Float
Beschreibung: Ermittelt die kleinste Ganzzahl die größer oder gleich der angegebenen Fließkommazahl ist.
Beispiel: Ceiling(3.5);

Truncate
Parameter: Float
Rückgabetyp: Float
Beschreibung: Berechnet den ganzzahligen Teil einer angegebenen Fließkommazahl mit doppelter Genauigkeit.
Beispiel: Truncate(3.5);

2.4.3 Methoden für Datumswerte

Now
Parameter: -
Rückgabetyp: Date
Beschreibung: Gibt das aktuelle Datum und Zeit zurück.
Beispiel: Now();

UtcNow[4]
Parameter: -
Rückgabetyp: Date
Beschreibung: Gibt das aktuelle Datum und Zeit in UTC zurück.
Beispiel: UtcNow();

AddYears
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Jahren zu dem angegebenen Datum dazu.
Beispiel: AddYears(Now(), 2);

AddMonths
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Monaten zu dem angegebenen Datum dazu.
Beispiel: AddMonths(Now(), 2);

AddDays
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Tagen zu dem angegebenen Datum dazu.
Beispiel: AddDays(Now(), 2);

AddDays
Parameter: Date, Float
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Tagen zu dem angegebenen Datum dazu.
Beispiel: AddDays(Now(), 2.3);

AddHours
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Stunden zu dem angegebenen Datum dazu.
Beispiel: AddHours(Now(), 2);

AddHours
Parameter: Date, Float
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Stunden zu dem angegebenen Datum dazu.
Beispiel: AddHours(Now(), 2.3);

AddMinutes
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Minuten zu dem angegebenen Datum dazu.
Beispiel: AddMinutes(Now(), 2);

AddMinutes
Parameter: Date, Float
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Minuten zu dem angegebenen Datum dazu.
Beispiel: AddMinutes(Now(), 2.3);

AddSeconds
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Sekunden zu dem angegebenen Datum dazu.
Beispiel: AddSeconds(Now(), 2);

AddSeconds
Parameter: Date, Float
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Sekunden zu dem angegebenen Datum dazu.
Beispiel: AddSeconds(Now(), 2.3);

AddMilliseconds
Parameter: Date, Int
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Millisekunden zu dem angegebenen Datum dazu.
Beispiel: AddMilliseconds(Now(), 2);

AddMilliseconds
Parameter: Date, Float
Rückgabetyp: Date
Beschreibung: Addiert die angegebene Anzahl an Millisekunden zu dem angegebenen Datum dazu.
Beispiel: AddMilliseconds(Now(), 2.3);

Date
Parameter: Date
Rückgabetyp: Date
Beschreibung: Gibt den Datumsteil des angegebenen Datums zurück.
Beispiel: Date(Now());

IsLeapYear
Parameter: Date
Rückgabetyp: Bool
Beschreibung: Gibt zurück ob das angegebene Jahr ein Schaltjahr ist.
Beispiel: IsLeapYear(Now());

Year
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt die Jahreszahl des angegebenen Datums zurück.
Beispiel: Year(Now());

Month
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt den Monat des angegebenen Datums zurück.
Beispiel: Month(Now());

Day
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt den Tag des angegebenen Datums zurück.
Beispiel: Day(Now());

Hour
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt die Stunde des angegebenen Datums zurück.
Beispiel: Hour(Now());

Minute
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt die Minute des angegebenen Datums zurück.
Beispiel: Minute(Now());

Second
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt die Sekunde des angegebenen Datums zurück.
Beispiel: Second(Now());

Millisecond
Parameter: Date
Rückgabetyp: Int
Beschreibung: Gibt die Millisekunde des angegebenen Datums zurück.
Beispiel: Millisecond(Now());

2.4.4 Konvertierungsmethoden

Einige der folgenden Konvertierungsmethoden nehmen als zusätzlichen Parameter eine Formatangabe an. Welche Formatangaben möglich sind, wird in Abschnitt 2.8 erläutert.

ToString
Parameter: Int
Rückgabetyp: String
Beschreibung: Konvertiert eine Ganzzahl in eine Zeichenkette.
Beispiel: ToString(5);

ToString
Parameter: Float
Rückgabetyp: String
Beschreibung: Konvertiert eine Fließkommazahl in eine Zeichenkette.
Beispiel: ToString(5);

ToString
Parameter: Bool
Rückgabetyp: String
Beschreibung: Konvertiert einen logischen Wert in eine Zeichenkette.
Beispiel: ToString(5);

ToString
Parameter: Date
Rückgabetyp: String
Beschreibung: Konvertiert ein Datum in eine Zeichenkette.
Beispiel: ToString(5);

ToString
Parameter: Int, String
Rückgabetyp: String
Beschreibung: Konvertiert eine Ganzzahl in eine Zeichenkette unter Berücksichtigung der Formatangabe.
Beispiel: ToString(129, "x");

ToString
Parameter: Float, String
Rückgabetyp: String
Beschreibung: Konvertiert eine Fließkommazahl in eine Zeichenkette unter Berücksichtigung der Formatangabe.
Beispiel: ToString(3.5, "x");

ToString
Parameter: Date, String
Rückgabetyp: String
Beschreibung: Konvertiert ein Datum in eine Zeichenkette unter Berücksichtigung der Formatangabe.
Beispiel: ToString(Now(), "yyyy-MM-dd");

ToInt
Parameter: String
Rückgabetyp: Int
Beschreibung: Konvertiert eine Zeichenkette in eine Ganzzahl.
Beispiel: ToInt("5");

ToFloat
Parameter: String
Rückgabetyp: Float
Beschreibung: Konvertiert eine Zeichenkette in eine Fließkommazahl.
Beispiel: ToFloat("5");

ToDate
Parameter: String
Rückgabetyp: Date
Beschreibung: Konvertiert eine Zeichenkette in ein Datum.
Beispiel: ToDate("13.04.1984");

ToBool
Parameter: String
Rückgabetyp: Bool
Beschreibung: Konvertiert eine Zeichenkette in einen logischen Wert.
Beispiel: ToBool(true");

2.5 Verzweigung mit if - else

Mit den bisher aufgezeigten Möglichkeiten lassen sich nur überaus einfache Templates erzeugen. Gerade die Relationsoperatoren lassen noch keinen Nutzen erkennen. Dieser wird erst offenbar, wenn man die Möglichkeit erhält Anweisungen in Abhängigkeit von Ausdrücken auszuführen. Um das zu erreichen gibt es die so genannte Verzweigung.
In Vogon wird die Verzweigung durch `if` und `else` ermöglicht. Das folgende Beispiel zeigt eine einfache Verzweigung.

```
1 if (1 < 2)
2   Print("Eins ist kleiner als zwei.");
```

Die zu prüfende Bedingung wird in ein Paar runder Klammern eingeschlossen und muss einen Wert vom Typ `bool` ergeben. Somit sind alle Ausdrücke die sich aus relationalen oder logischen Operationen zusammensetzen an dieser Stelle verwendbar. Aber auch Methoden, die einen Wert vom Typ `bool` zurückgeben können hier verwendet werden.

```
1 if (Contains("Hallo Welt.", "Hallo"))
2   Print("In \"Hallo Welt\" kommt \"Hallo\" vor.");
```

Es ist natürlich auch möglich Anweisungen für den Fall, dass die Bedingung nicht erfüllt ist, anzugeben. Dazu wird das Schlüsselwort `else` verwendet.

```
1  string text;
2  if (produkteGekauft == 1)
3    text = "ein Produkt";
4  else
5    text = Format("{0} Produkte", ToString(produkteGekauft));
6
7  Print(Format("Sie haben {0} gekauft", text));
8
9  // Ausgabe bei produkteGekauft == 1
10 // "Sie haben ein Produkt gekauft."
11
12 // Ausgabe bei produkteGekauft == 5
13 // "Sie haben 5 Produkte gekauft."
```

Müssen mehrere Anweisungen ausgeführt werden, so müssen diese in einen Block geschrieben werden. Dazu dienen geschweifte Klammern wie das folgende Beispiel zeigt.

2.6 Wiederholungen mit foreach

Als letztes fehlen noch Schleifen um eine mächtige Möglichkeit für Templates zu haben. Erst die Schleifen ermöglichen es sinnvoll mit einer Bestellung von mehreren Artikeln umzugehen. Vogon unterstützt nur einen Schleifentyp, welcher mit einer Sammlung von Objekten arbeitet. Das folgende Beispiel zeigt, wie die Schleife verwendet wird.

```
1 forach(item in Order.Items)
2 {
3   PrintLine(item.Name);
4 }
```

Im Beispiel ist `Order` eine strukturierte Variable, die vom System zur Verfügung gestellt wird. Dies wird später genauer behandelt. Die Variable `Order.Items` stellt alle Artikel einer Bestellung zur Verfügung. Diese können nun in der Schleife nacheinander verarbeitet werden. Die Variable `item` stellt für jeden Schleifendurchlauf den aktuellen Artikel zur Verfügung. Der Name der Variablen kann frei gewählt werden, darf jedoch keinem Namen einer bereits vorhandenen Variablen entsprechen.

In Abschnitt 2.10 werden Beispiele aufgeführt in denen die Verwendung der Schleife genauer demonstriert wird.

2.7 Kommentare

Auch in Vogon ist es möglich Kommentare zu schreiben. Diese werden unterteilt in Ein- und Mehrzeilenkommentare. Sie helfen den Quellcode leichter zu verstehen oder Teile zu Testzwecken auszukommentieren, ohne sie entfernen zu müssen. Das folgende Beispiel zeigt die Verwendung beider Möglichkeiten.

```
1 /*
2   Rechnungsvorlage
3 */
4
5 // Die folgende Zeile ist auskommentiert
6 // Print("Testausgabe");
```

2.8 Formatierung von Zeichenketten

Die Formatierung von Zeichenketten ist eine nützliche Möglichkeit dynamisch erstellte Texte in einer lesbaren Form im Code unterzubringen. Da in Vogon eine Verkettung von Zeichenketten durch einen Operator[5] nicht erlaubt ist, entfällt schon eine Möglichkeit Zeichenketten dynamisch zu erzeugen. Das liegt in erster Linie daran, dass hier ein großes Potenzial für schwer lesbaren Code existiert.

Vogon stellt statt dessen die Methode `Format` zur Verfügung. Diese ist allgemein wie folgt definiert:

```
1 Format(String formatString, Array values)
```

Der erste Parameter ist also eine Zeichenkette, welche Formatangaben enthält. Das folgende Beispiel zeigt, wie die Methode verwendet wird.

```
1 Format("Sie haben {0} Produkte gekauft.", Order.ItemCount);
```

Im ersten Parameter ist ein Platzhalter (`{0}`) angegeben, welcher durch den ersten folgenden Parameter, in diesem Beispiel `Order.ItemCount`, ersetzt wird. In jeder Formatangabe können bis zu 14 verschiedene Platzhalter verwendet werden, welche von `0` bis `13` durchnummeriert sind und immer bei `0` beginnen.

```
1 Format("Sie haben {0} Produkte mit einem Gesamtwert von {1} {2} gekauft.", Order.ItemCount,
2 Order.Currency, Order.TotalPrice);
```

In diesem Beispiel werden an die Methode `Format` vier Parameter übergeben. Es sind neben der Formatangabe noch drei Werte zur Ersetzung im Text angegeben. Es können also die Platzhalter `{0}`, `{1}` und `{2}` verwendet werden.

Parameter	Platzhalter	Beispielwert
`"Sie haben {0} Produkte mit einem Gesamtwert von {1} {2} gekauft."`	-	-
`Order.ItemCount`	`{0}`	3
`Order.Currency`	`{1}`	EUR
`Order.TotalPrice`	`{2}`	33,50

[5] & in VB/VB.Net, . in PHP oder + in den meisten anderen Sprachen.

Mit den Beispielwerten aus der Tabelle wäre die Ausgabe der Methode `Format` wie im folgenden Beispiel angegeben.

```
Format("Sie haben {0} Produkte mit einem Gesamtwert von {1} {2} gekauft.", Order.ItemCount,
Order.Currency, Order.TotalPrice);

// Ausgabe:
// Sie haben 3 Produkte mit einem Gesamtwert von EUR 33,50 gekauft.
```

2.8.1 Zahlwerte formatieren

Wie bereits erwähnt unterscheidet Vogon zwischen Fließ- und Ganzkommazahlen. Beide können durch Konvertierungsmethoden (siehe Abschnitt 2.4.4) in Zeichenketten konvertiert werden, wobei die Darstellung je nach Situation anders sein kann. Dafür gibt es die jeweiligen Überladungen der Methode `ToString`, die neben dem zu konvertierenden Wert noch eine Formatangabe akzeptieren.
In der folgenden Tabelle werden die Methoden `ToString(Int, String)` und `ToString(Float, String)` betrachtet, wobei jeweils mögliche Werte zur Formatierung angegeben und erläutert werden[6].

Symbol	Bedeutung	Int	Float	Beispiel	Ausgabe
C	Währung	×[7]		`ToString(-123, "C");`	$-123,00$ €
D	Dezimal	×	×	`ToString(-123, "D");`	-123
E	Wissenschaftlich	×	×	`ToString(-123.45, "E");`	$-1,234500E+002$
F	Fixed	×	×	`ToString(-123.45, "F");`	$-123,45$
G	Allgemein (Standard)	×	×	`ToString(-123, "G");`	-123
N	Zahl	×	×	`ToString(-123, "N");`	$-123,00$
P	Prozent	×	×	`ToString(-123.45, "P");`	$12.345,00\%$

Tabelle 2.7: Formatierungssymbole für Zahlwerte

Diese Formatierungssymbole lassen sich auch mit der Methode `Format` kombinieren. Dafür können den Platzhaltern die entsprechenden Symbole beigefügt werden, wie im folgenden Beispiel.

```
Format("Sie haben Waren im Wert von {0:C} gekauft.", 123);
// Sie haben Waren im Wert von 123,00 € gekauft.
```

In diesem Beispiel wird die Zahl 123 als Währung formatiert und mit dem Währungssymbol für Euro versehen. Es ist zu beachten, dass das Währungssymbol von der Sprache der Vorlage abhängt.

2.8.2 Datumswerte formatieren

Mit dem Typ `date` stehen Datum und Uhrzeit in einer Variable zur Verfügung. Je nach Fall ist eine unterschiedliche Darstellung wünschenswert. Die möglichen Formatierungssymbole werden in der nachfolgenden Tabelle aufgeführt[8].

Symbol	Bedeutung	Beispiel	Ausgabe
d	Kurzes Datum	`ToString(Now(), "d");`	08.04.2013
D	Langes Datum	`ToString(Now(), "D");`	Montag, 8. April 2013
t	Kurze Zeit	`ToString(Now(), "t");`	17:03
T	Lange Zeit	`ToString(Now(), "T");`	17:03:24
f	Volles Datum, kurze Zeit	`ToString(Now(), "f");`	Montag, 8. April 2013 17:03
F	Volles Datum, lange Zeit	`ToString(Now(), "F");`	Montag, 8. April 2013 17:03:24
g	Allgemeines Datum, kurze Zeit	`ToString(Now(), "g");`	08.04.2013 17:03
G	Allgemeines Datum, lange Zeit	`ToString(Now(), "G");`	08.04.2013 17:03:24
M	Monat	`ToString(Now(), "M");`	08 April
Y	Year	`ToString(Now(), "Y");`	April 2013

Tabelle 2.8: Formatierungssymbole für Datumswerte

Wie bei der Formatierung von Zahlwerten können auch Datumswerte direkt mit der Methode `Format` formatiert werden. Das Vorgehen ist hierbei das gleiche.

[6] Eine vollständige Erläuterung findet sich unter http://msdn.microsoft.com/de-de/library/dwhawy9k%28v=vs.80%29.aspx.
[7] Ein × gibt an, dass das Formatierungssymbol für den Typ gültig ist.
[8] Eine vollständige Erläuterung findet sich unter http://msdn.microsoft.com/de-de/library/az4se3k1%28v=vs.80%29.aspx.

```
Format("Zum aktuellen Zeitpunkt ({0:F}) ist der Bestand des Produktes hinreichend.", Now());
// Zum aktuellen Zeitpunkt (Montag, 8. April 2013 17:03:24) ist der
// Bestand des Produktes hinreichend.
```

2.8.3 Weitere Formatierungsmöglichkeiten

Für reine Textausgaben, gibt es noch eine weitere Möglichkeit auf die Formatierung Einfluss zu nehmen.

Listing 2.1: Erweiterte Formatierung mit `Format`

```
PrintLine(Format("{0,-15}|{1,-10}", "Name", "Preis"));
PrintLine("--------------------------");
PrintLine(Format("{0,-15}|{1,10:C}", "Ladekabel", 9.99));
PrintLine(Format("{0,-15}|{1,10:C}", "Handy", 489.99));

// Name           |Preis
// --------------------------
// Ladekabel      |    9,99 €
// Handy          |  489,99 €
```

In diesem Beispiel wird eine Tabelle ausgegeben. Um auf die unterschiedlichen Breiten der einzelnen Wörter reagieren zu können und dennoch gleich breite Spalten zu erhalten, können dem Platzhalter Spaltenbreiten angegeben werden. Der Wert `Name` wird mit dem Platzhalter `"{0,-15}"` formatiert. Dabei gibt die `0` an, welcher Parameter an dieser Stelle ausgegeben werden soll. Danach folgt getrennt durch ein Komma eine Breitenangabe. In diesem Fall, soll der Wert auf einer Breite von 15 Zeichen ausgegeben werden. Das passiert unabhängig davon wie lang das Wort ist, solange es nicht länger als 15 Zeichen ist. Das Minuszeichen gibt an, dass der Wert linksbündig ausgerichtet werden soll. Ließe man es weg, so würde der Wert rechtsbündig ausgegeben werden.

```
PrintLine(Format("|{0,15}|", "Name"));
PrintLine(Format("|{0,-15}|", "Name"));

// |           Name|
// |Name           |
```

Diese Art der Formatierung funktioniert jedoch nur mit Schriftarten die jedes Schriftzeichen gleich breit darstellen, was bei normalen Templates selten der Fall. Sie wurde nur der Vollständigkeit halber erwähnt.

In Listing 2.1 werden Währungswerte mit dem Platzhalter `"{1,10:C}"` formatiert. Dabei wurde das Formatierungszeichen für Währung (`C`) wie in Abschnitt 2.8.1 beschrieben verwendet, um die Zahl als Währung zu formatieren. Das Zeichen wird dabei durch einen Doppelpunkt vom ersten Teil der Formatierung getrennt. An dieser Stelle können beliebige andere Formatierungszeichen verwendet werden, solange sie zulässig sind.

2.9 Durch das System gestellte Variablen

Neben den selbst definierbaren Variablen, stellt das System je nach Art des Templates selbst Variablen zur Verfügung. Diese sind immer daran zu erkennen, dass sie aus zwei Teilen bestehen, welche durch einen Punkt getrennt werden. Ein Beispiel, das in vorhergehenden Abschnitten schon oft eingesetzt wurde, ist `Order.Items`. Dabei bezieht `Order` sich auf den aktuellen Verkaufsvorgang, wenn das Template zum Beispiel für eine Rechnung verwendet wird. `Order` enthält dann mehrere untergeordnete Elemente wie `Items`, die einzelne Werte des Verkaufs enthalten. Diese Variablen werden daher als strukturiert bezeichnet.
In der folgenden Tabelle sind einige Variablen und Werte Beispielhaft aufgelistet.

Name	Bedeutung	Typ	Wert
Order.ItemCount	Anzahl der Produkte im Vorgang	Int	2
Order.TotalPrice	Gesamtpreis des Vorgangs	Float	499,98
Order.Currency	Währung des Vorgangs	String	EUR
Order.ValueAddedTax	Mehrwertsteuersatz	Float	19
Order.Buyer	Käuferinformationen	Structured	-
Order.ShippingAddress	Lieferadresse	Structured	-

Tabelle 2.9: Beispiel für eine strukturierte Systemariable

Im Beispiel sind zwei strukturierte Variablen `Order.Buyer` und `Order.ShippingAddress` enthalten. Da es nicht möglich ist mehr

als eine Ebene tief in eine strukturierte Variable vorzudringen, müssen untergeordnete strukturierte Variablen zunächst in einer benutzerdefinierten Variable gespeichert werden, bevor auf ihre Unterelemente zugegriffen werden kann.

```
// der folgende Code verursacht einen Fehler
PrintLine(Format("Sehr geehrte/r {0} {1} {2},", Order.Buyer.Title, Order.Buyer.Prename,
Order.Buyer.Name));

// daher eine Variable vom Typ structured
structured buyer;

// Wert von Order.Byuer zuweisen
buyer = Order.Buyer;

// Und mit der Variable arbeiten
PrintLine(Format("Sehr geehrte/r {0} {1} {2},", buyer.Title, buyer.Prename, buyer.Name));
```

Die in Tabelle 2.9 gemachten Angaben sind nur beispielhaft und können von tatsächlichen Variablen abweichen. Welche Variablen zur Verfügung stehen ist den jeweiligen Anwendungsdokumentationen zu entnehmen.

2.10 Beispiele

Teil II

Technisches Handbuch

Kapitel 3

Die Grammatik und Antlr

Die Sprache Vogon wird durch eine Grammatik beschrieben, welche für AntlrV3 geschrieben wurde. Sie besteht aus Tokens sowie Lexer- und Parserregeln. Mit hilfe der Grammatik wird aus einem Codelisting ein Baum erstellt, der den Code komplett abbildet. Ein solcher Baum wird auch als abstrakter Syntaxbaum (Abstract Syntax Tree oder auch *AST*) bezeichnet.

In den folgenden Abschnitten werden die meisten Teile der Grammatik genauer betrachtet. Dabei werden Listings und Syntaxdiagramme verwendet.

3.1 Token

Die Token sind wie in Tabelle 3.1 beschrieben.

Token	Wert	Token	Wert	Token	Wert	Token	Wert
SLIST	-	OPMULT	'*'	RELNEQ	'!='	TRUE	'true'
CALL	-	OPDIV	'/'	RELGT	'>'	FALSE	'false'
VARDEC	-	OPMOD	'%'	RELLT	'<'		
DOT	-	OPAND	'&&'	RELGTOE	'>='		
QUESTION	'?'	OPOR	'||'	RELLTOE	'<='		
COLON	','	OPXOR	'^'	KEYIF	'if'		
OPPLUS	'+'	OPNOT	'!'	KEYELSE	'else'		
OPMINUS	'-'	RELEQ	'=='	KEYFOREACH	'foreach'		

Tabelle 3.1: Token der Sprache

Diese Token werden verwendet um den Typ eines Knotens im *AST* anzugeben. Somit kann man bei der Traversierung des *AST* schnell erkennen für welches Konstrukt der Syntax er steht und ihn entsprechend verarbeiten.

3.2 Lexerregeln

Die Aufgabe des Lexers ist es den Eingabestrom in Token zu zerlegen, welche an den Parser weitergereicht werden, damit dieser Muster erkennen kann. In Antlr können Regeln für den Lexer beschrieben werden, die Token definieren. Die Lexerregel hilft dem Lexer also den Eingabestrom in Token zu zerlegen die nicht allein durch die Token aus Tabelle 3.1 beschrieben werden.
Im Folgenden werden die Lexerregeln beschrieben und sowohl in ihrer Definition als auch als Syntaxdiagramm dargestellt.

3.2.1 ESC_SEQ

Diese Regel beschreibt Escape-Sequenzen in Zeichenketten und stellt kein eigenständiges Token dar. Das wird durch das Wort *fragment* in der Definition angezeigt.

Listing 3.1: Definition der Lexerregel ESC_SEQ

```
1 fragment
2 ESC_SEQ [StringBuilder buf]
3     :   '\\'
```

```
        ('b' {buf.Append("\b");}
        |'t' {buf.Append("\t");}
        |'n' {buf.Append("\n");}
        |'f' {buf.Append("\f");}
        |'r' {buf.Append("\r");}
        |'\"' {buf.Append("\"");}
        |'\'' {buf.Append("\'");}
        |'\\' {buf.Append("\\");}
        )
    ;
```

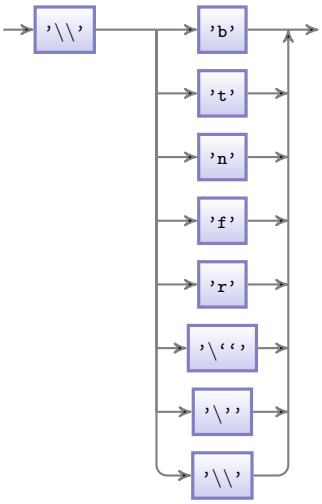

Abbildung 3.1: Syntaxdiagramm der Lexerregel *ESC_SEQ*

3.2.2 EXPONENT

Diese Regel beschreibt einen Exponenten für Fließkommazahlen. Auch sie erstellt kein eigenständiges Token, da sie als Fragment markiert ist.

Listing 3.2: Definition der Lexerregel EXPONENT

```
fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+;
```

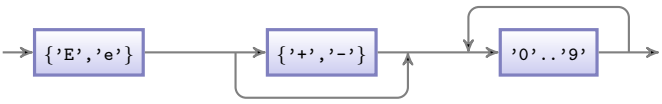

Abbildung 3.2: Syntaxdiagramm der Lexerregel *EXPONENT*

3.2.3 STRING

Diese Lexerregel beschreibt Zeichenketten. In Vogon müssen Zeichenketten zwischen doppelten Anführungszeichen stehen und können Escape-Sequenzen enthalten.

Listing 3.3: Definition der Lexerregel STRING

```
STRING
@init { StringBuilder buf = new StringBuilder(); }
    :  '"' ( ESC_SEQ[buf] | s=~('\\'|'"') {buf.Append(Convert.ToChar(s));} )* '"'
    {this.Text = buf.ToString();}
    ;
```

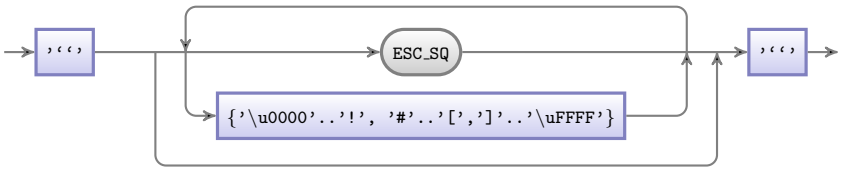

Abbildung 3.3: Syntaxdiagramm die Lexerregel *STRING*

3.2.4 COMMENT

Diese Lexerregel steht für Ein- und Mehrzeilenkommentare. Diese werden durch die Angabe `{$channel=HIDDEN;}` in einem separaten Kanal an den Parser weitergereicht.

Listing 3.4: Definition der Lexerregel COMMENT

```
1 COMMENT
2     :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
3     |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
4     ;
```

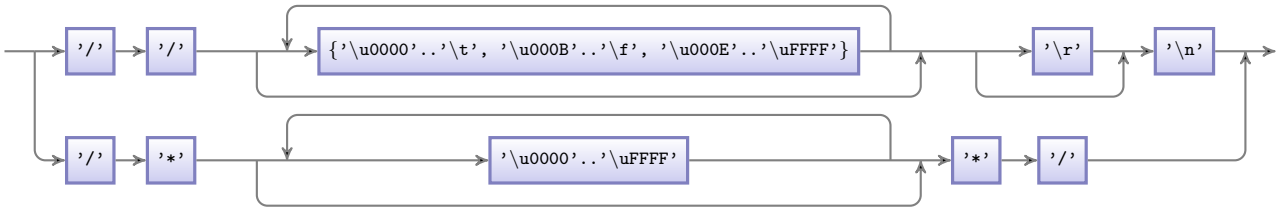

Abbildung 3.4: Syntaxdiagramm der Lexerregel *COMMENT*

3.2.5 FLOAT

Diese Lexerregel beschreibt Fließkommazahlen. Sie beschreibt die möglichen Arten im Quelltext eine Fließkommazahl anzugeben. Ferner wird in dieser Regel die zuvor definierte Regel *EXPONENT* verwendet.

Listing 3.5: Definition der Lexerregel FLOAT

```
1 FLOAT
2     :   '-'? ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
3     |   '-'? '.' ('0'..'9')+ EXPONENT?
4     |   '-'? ('0'..'9')+ EXPONENT
5     ;
```

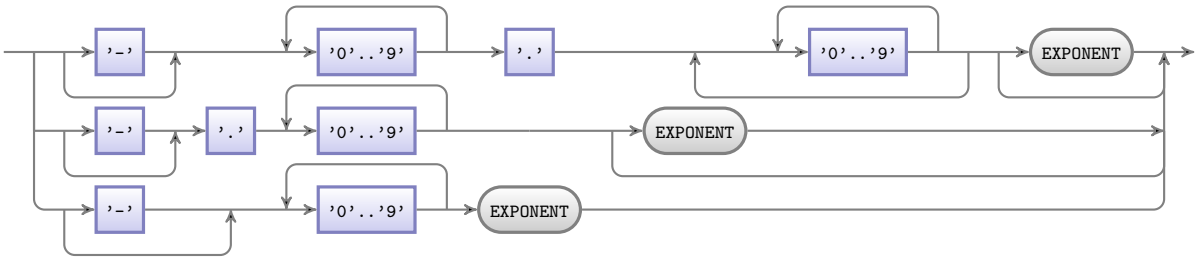

Abbildung 3.5: Syntaxdiagramm der Lexerregel *FLOAT*

3.2.6 INT

Diese Lexerregel beschreibt Ganzzahlen.

Listing 3.6: Definition der Lexerregel INT

```
1 INT :   '-'? '0'..'9'+
2     ;
```

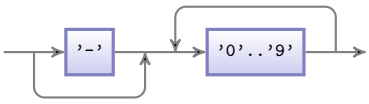

Abbildung 3.6: Syntaxdiagramm der Lexerregel *INT*

3.2.7 ID

Diese Lexerregel beschreibt Bezeichner wie sie für Variablen und Methoden verwendet werden können. Sie besagt, dass Bezeichner nur mit einem großen oder kleinen Buchstaben beginnen können, welcher von beliebig vielen Groß- oder Kleinbuchstaben und Zahlen gefolgt werden kann.

Listing 3.7: Definition der Lexerregel ID

```
1 ID  :   ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*
2     ;
```

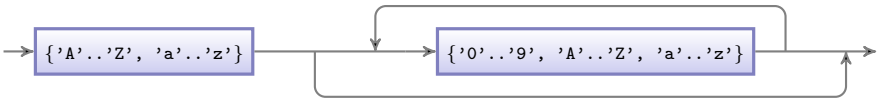

Abbildung 3.7: Syntaxdiagramm der Lexerregel *ID*

3.3 Parserregeln

Parserregeln werden direkt vom Parser verarbeitet und setzen sich aus Token die vom Lexer geliefert werden oder anderen Parserregeln zusammen.

3.3.1 mid

Diese Parserregel beschreibt einen zweiteiligen Bezeichner wie er für Systemvariablen verwendet wird.

Listing 3.8: Definition der Parserregel mid

```
1 mid
2         :       ID '.' ID
3                 -> ^(DOT ID ID)
4         ;
```

Abbildung 3.8: Syntaxdiagramm der Parserregel *mid*

3.3.2 atom

Listing 3.9: Definition der Parserregel atom

```
1 atom
2         :       ID
3         |       mid
4         |       STRING
5         |       FLOAT
6         |       INT
7         |       call
8         |       '(' expr ')' -> expr
9         ;
```

3.3.3 mexpr

Listing 3.10: Definition der Parserregel mexpr

```
1 mexpr
2         :       atom (('*'^|'/'^|'%'^) atom)*
3         ;
```

3.3.4 aexpr

Listing 3.11: Definition der Parserregel aexpr

```
1 aexpr
2         :       mexpr (('+'^|'-'^) mexpr)*
3         ;
```

3.3.5 pcondExpr

Listing 3.12: Definition der Parserregel pcondExpr

```
1 pcondExpr
2         :       aexpr
3         |       TRUE
4         |       FALSE
5         ;
```

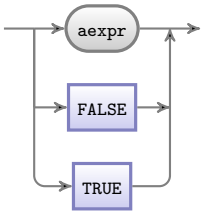

Abbildung 3.9: Syntaxdiagramm der Parserregel *pcondExpr*

3.3.6 condExpr

Listing 3.13: Definition der Parserregel condExpr

```
1 condExpr
2         :       pcondExpr (('=='^|'!='^|'<'^|'>'^|'<='^|'>='^) pcondExpr)*
3         |       '!' condExpr
4                 -> ^('!' condExpr)
5         ;
```

3.3.7 logexpr

Listing 3.14: Definition der Parserregel logexpr

```
1 logexpr
2         :       condExpr (('&&'^|'||'^|'^'^) condExpr)*
3         ;
```

3.3.8 slogexpr

Listing 3.15: Definition der Parserregel slogexpr

```
1 slogexpr
2         :       logexpr (QUESTION^ alt1=expr COLON! alt2=expr)?
3         ;
```

3.3.9 expr

Listing 3.16: Definition der Parserregel expr

```
expr
@init { this.paraphrases.Push("in expression"); }
@after { this.paraphrases.Pop(); }
        :       slogexpr
        ;
```

3.3.10 assign

Listing 3.17: Definition der Parserregel assign

```
assign
@init { this.paraphrases.Push("in assign"); }
@after { this.paraphrases.Pop(); }
        :       ID '=' expr -> ^(ASSIGN ID expr)
        ;
```

3.3.11 parameter

Listing 3.18: Definition der Parserregel parameter

```
parameter
        :       expr
        ;
```

3.3.12 call

Listing 3.19: Definition der Parserregel call

```
call
@init { this.paraphrases.Push("in call"); }
@after { this.paraphrases.Pop(); }
        :       ID '(' (parameter (',' parameter)*)? ')'
                -> ^(CALL ID parameter*)
        ;
```

3.3.13 ifStat

Listing 3.20: Definition der Parserregel ifStat

```
ifStat
@init { this.paraphrases.Push("in if"); }
@after { this.paraphrases.Pop(); }
        :       'if' '(' logexpr ')' s1=stat
                ( 'else' s2=stat -> ^(KEYIF logexpr $s1 $s2)
                |                -> ^(KEYIF logexpr $s1)
                )
        ;
```

3.3.14 foreachStat

Listing 3.21: Definition der Parserregel foreachStat

```
foreachStat
@init { this.paraphrases.Push("in foreach"); }
@after { this.paraphrases.Pop(); }
        :       KEYFOREACH^ '(''! ID 'in''! mid ')''! stat
        ;
```

3.3.15 block

Listing 3.22: Definition der Parserregel block

```
block
        :       lc='{' stat* '}'
                -> ^(SLIST[$lc, "SLIST"] stat*)
        ;
```

3.3.16 stat

Diese Parserregel beschreibt eine Anweisung wie sie im Quelltext auftreten kann. Das Semikolon zum Abschluss einer Anweisung tritt hier nicht in allen Alternativen auf und wird durch das Ausrufungszeichen nicht mit in den AST aufgenommen. Das Semikolon als leere Anweisung, wie in Zeile sieben des Listings gezeigt, ist nicht zulässig.

Listing 3.23: Definition der Parserregel stat

```
1 stat
2         :       expr';'!
3         |       ifStat
4         |       foreachStat
5         |       block
6         |       assign';'!
7 //      |       ';'!
8         ;
```

3.3.17 variablelist

Diese Parserregel beschreibt eine Reihe von Variablennamen die durch ein Komma getrennt werden.

Listing 3.24: Definition der Parserregel variablelist

```
1 variablelist
2         :       ID (','! ID)*
3         ;
```

3.3.18 variable

Diese Parserregel beschreibt eine Variablendefinition. Dabei wird der Typname angegeben, gefolgt von einem oder mehreren Variablennamen. Durch den Tree-Rewrite wird ein Knoten vom Tokentyp *VARDEC* angelegt.

Listing 3.25: Definition der Parserregel variable

```
1 variable
2 @init { this.paraphrases.Push("in declaration"); }
3 @after { this.paraphrases.Pop(); }
4         :       type=ID variablelist ';'
5                 -> ^(VARDEC $type variablelist)
6         ;
```

3.3.19 declaration

Diese Parserregel beschreibt eine Deklaration. In Vogon können nur Variablen deklariert werden.

Listing 3.26: Definition der Parserregel declaration

```
1 declaration
2         :       variable
3         ;
```

3.3.20 element

Diese Parserregel beschreibt ein Element. Das ist entweder eine Deklaration oder eine Anweisung.

Listing 3.27: Definition der Parserregel element

```
1 element
2         :       declaration
3         |       stat
4         ;
```

3.3.21 script

Diese Parserregel beschreibt ein Skript in der Sprache Vogon. Es besteht aus mindestens einem Element. Ein leeres Skript ist nicht zulässig.

Listing 3.28: Definition der Parserregel script

```
1 script
2         :       element+
3         ;
```

Kapitel 4

Lexer und Parser

Um den Quellcode verarbeiten zu können, muss dieser zunächst in eine Form gebracht werden, in der leichter mit ihm umgegangen werden kann. Dazu wird ein Parser benötigt. Dabei wird die Arbeit des Parsers durch einen Lexer erleichtert, da der Parser nicht direkt den Quellcode parst. Der Lexer zerlegt den Quellcode in Token, welche dem Parser das Erkennen von Mustern entsprechend der Grammatik ermöglichen. Der von Vogon verwendete Parser, und damit auch der Lexer, wurden mittels Antlr generiert.
Der Parser erzeugt einen abstrakten Syntaxbaum (AST), welcher den Quellcode als Baum darstellt. Dieser ist sehr leicht zu Verarbeiten und dient als Grundlage aller weiteren Schritte.

Kapitel 5

Der AST-Interpreter Xpomul

In der Standardausführung wird der vom Parser erzeugte AST direkt interpretiert. Dieser Ansatz ist sehr gut geeignet um die Grammatik zu testen, da keine weiteren Zwischenschritte ablaufen, die ihrerseits weitere mögliche Fehlerquellen darstellen. Sie ist außerdem sehr leicht zu implementieren.
Diese Methode wird derzeit von der Klasse *VogonScript* verwendet um Skripte auszuführen. Es wird dabei der Visitor-Pattern verwendet[1], was bedeutet, dass eine rekursive Funktion den AST traversiert und bei jedem Knoten eine zu dem Knoten gehörende Aktion ausführt.
In den folgenden Abschnitten wird gezeigt wie *Xpomul* mit Datentypen, Variablen und Methoden umgeht, um diese zu realisieren. *Xpomul* macht intensiven Gebrauch von dem C#-Schlüsselwort `dynamic`. Dieses ermöglicht eine recht einfache Implementierung des Interpreters geht jedoch zu Lasten der Performance, da der C#-Compiler Code für späte Bindung generiert, der eine Typprüfung erst zur Laufzeit vornimmt. Da die Umsetzung mit jedem anderen Ansatz jedoch mit einem enorm hohen Grad an Komplexität verbunden ist, und bisherige Performance-Untersuchungen eine hinreichende Ausführungsgeschwindigkeit zeigen, wird die späte Bindung in Kauf genommen.

5.1 Datentypen

Der Interpreter *Xpomul* kennt die folgenden acht Datentypen:

Name	Im Skript	Wert	Wert binär
Void	-	0	000
Int	int	1	001
Float	float	2	010
Bool	bool	3	011
Date	date	4	100
String	string	5	101
Structured	structured	6	110
Array	-	7	111

Diese sind in der Aufzählung `VariableType` definiert.

```
1  public enum VariableType
2  {
3          Void,
4          Int,
5          Float,
6          Bool,
7          Date,
8          String,
9          Structured,
10         Array
11 }
```

[1] siehe

5.2 Variablen

Variablen werden innerhalb von *Xpomul* durch die Klasse `VariableDescriptor` dargestellt.

Listing 5.1: Definition der Klasse `VariableDescriptor`

```
internal class VariableDescriptor
{
        public String Name { get; set; }
        public VariableType Type { get; set; }
        public dynamic Value { get; set; }
}
```

Für jede im Script definierte Variable wird eine Instanz dieser Klasse erzeugt und zum `VariableContext` des aktuellen Scripts hinzugefügt, sofern eine Variable mit diesem Namen noch nicht existiert. In diesem Kontext für Variablen werden auch durch das System vorgegebene Variablen gespeichert. Vordefinierte Variablen sollten stets vom Typ `VariableType.Structured` sein, um zu vermeiden, dass eine noch nicht dokumentierte Variable ein vorhandenes Script eines Kunden ungültig macht, weil es eine Variable gleichen Namens definieren möchte. Zudem können systemdefinierte Variablen nicht Ziel einer Zuweisung sein.

Um die Implementierung zu vereinfachen wird für `VariableDescriptor.Value` das Schlüsselwort `dynamic` verwendet. Dieses verhält sich ähnlich die `Object` unter VisualBasic.Net. Der C#-Compiler wird an jeder Stelle an der mit dem Wert von `VariableDescriptor.Value` gearbeitet wird, zusätzlichen Code generieren, der wieder typsicher ist, also eine späte Bindung vornimmt. Dies geht natürlich zu Lasten der Performance, steht hier jedoch einer erheblich einfacheren Implementierung des Interpreters gegenüber, so dass das Opfer in Kauf genommen wurde.

5.3 Methoden

Vogon ermöglicht es integrierte Methoden aufzurufen. Methoden werden dem Skript über die Klasse `MethodContext` zur Verfügung gestellt. Dabei wird die Standardmenge[2] an Methoden durch einen `MasterContext` zur Verfügung gestellt. Das passiert automatisch, sofern nicht anders gewünscht. ListingSowohl die Mächtigkeit als auch die Einsetzbarkeit von Vogon kann in erheblichem Maße durch die zur Verfügung gestellten Methoden beeinflusst werden. So ist es durchaus möglich Methoden zu definieren, die sich direkt auf das System auswirken.

5.3.1 Signaturen und Signaturschlüssel

Signaturen werden von Vogon verwendet, um das Überladen von Methoden zu ermöglichen. Eine Signatur besteht aus dem Namen der Methode, gefolgt von den Namen der Datentypen der einzelnen Parameter. Alle diese Namen werden durch ein Leerzeichen getrennt. So lautet die Signatur einer Methode “Print” die einen Parameter vom Typ “String” erwartet “Print String”. Der Rückgabetyp einer Methode wird für die Signatur nicht berücksichtigt.

Zusätzlich wird einer Methode noch ein Signaturschlüssel zugeordnet. Dieser ist eine 64bit breite, vorzeichenlose Ganzzahl. Sie wird erzeugt indem die Zahlwerte der einzelnen Datentypen der Parameter in umgekehrter Reihenfolge hintereinandergestellt werden. So lautet der Signaturschlüssel einer Methode mit der Signatur “ToString Float String” in Binärform 101010 entsprechend der Tabelle aus . Listing zeigt die Methode zur Berechnung des Schlüssels zu einer Signatur.

Listing 5.2: Methode zum Berechnen des Schlüssels einer Signatur

```
internal static UInt64 CalculateSignatureKey(String signature)
{
    String[] parts = signature.Split(' ');
    UInt64 key = 0;

    for (int index = 1; index < parts.Length; index++)
    {
        VariableType type = VariableType.Void;
        if (!Enum.TryParse<VariableType>(parts[index], out type))
            throw new VogonException(String.Format("Unknown type '{0}'.", parts[index]));
        key = key | (((type == VariableType.Array) ? UInt64.MaxValue : Convert.ToUInt64(type))
            << ((index - 1) * 3));
    }
```

[2] Siehe Methodenreferenz

```
14
15      return key;
16  }
```

Findet *Xpomul* einen Methodenaufruf, so wird zunächst die für den Aufruf erforderliche Signatur ermittelt. Anschliessend wird diese Signatur aufgelöst um die passende Methode im `MethodContext` zu finden. Sollte die Auflösung keine Methode ergeben, so kann der Aufruf nicht ausgeführt werden und das Skript wird mit einer Fehlermeldung abgebrochen.
Die Auflösung der Signatur ist ein recht mathematischer Prozess, der im folgenden erläutert werden soll.

Es seien die folgenden Mengen definiert:

$$\mathbb{M} = \{m | m \text{ ist eine definierte Methode}\}$$

$$\mathbb{S} = \{s | s \text{ ist ein möglicher Signaturschlüssel}\}$$

Dann gilt:

$$\forall m \in \mathbb{M} : \exists! \widetilde{m} \in \mathbb{S} : \widetilde{m} \text{ ist Signaturschlüssel von } m.$$

Das heisst für alle möglichen Methoden gibt es genau einen Signaturschlüssel. Umgekehrt kann es aber für jeden Signaturschlüssel mehrere Methoden geben, da der Signaturschlüssel nicht den Namen der Methode berücksichtigt.

Beispiel: der Signaturschlüssel für die Methode mit der Signatur "Replace String String String" lautet 101101101.

Mit diesem Signaturschlüssel kann jetzt schon geprüft werden, ob ein im Skript vorhandener Aufruf ausführbar ist. Komplizierter wird es bei Methoden die eine variable Anzahl an Parametern unterstützen. Diese verwenden den Datentyp `VariableType.Array`, welcher den Binärwert 111 hat. Um zu Prüfen, ob ein gewünschter Aufruf durch eine Methode mit variabler Parameteranzahl erfüllt werden kann, müssen noch weitere Definitionen ergänzt werden.

Für $n \in \mathbb{N}$ und $s \in \mathbb{S}$ ist die *MostSignificantSignature* (*MSS*) definiert durch:

$$MSS(s,n) := \sum_{i=0}^{n-1} \begin{cases} s \wedge (7 * 2^{3i}) & \text{wenn } (s \wedge (7 * 2^{3i})) \veebar (7 * 2^{3i}) \neq 0 \\ 0 & \text{sonst} \end{cases} \tag{5.1}$$

wobei n der maximalen Anzahl an möglichen Parametern für eine Methode entspricht[3].
Die Methode mit der Signatur "Format String Array" hat den Signaturschlüssel 111101[4]. Die *MSS* dieses Signaturschlüssels ist 101, da alle Parameter des Arrays entfernt werden.

Weiter ist die Kardinalität einer *MSS* gegeben durch:

$$|MSS(s,n)| := \text{die Anzahl der von } 0 \text{ verschiedenen Summanden von } MSS(s,n). \tag{5.2}$$

Es gilt $|MSS(s,n)| \in \mathbb{N}$.
Die Kardinalität der eben im Beispiel genannten *MSS* lautet also 1, da sie nur einen von 0 verschiedenen Summanden hat.

Das folgende Listing zeigt, wie diese mathematischen Definitionen in *Xpomul* umgesetzt wurden.

Listing 5.3: Methode zum Ermitteln von MSS und $|MSS|$

```
1  internal static MostSignificantSignature CalculateMostSignificantSignature(UInt64 sigKey)
2  {
3      UInt64 signature = 0;
4      Byte cardinality = 0;
5
6      for (int n = 0; n < VogonScript.MAX_PARAMETERS; n++)
7      {
8          UInt64 testValue = Convert.ToUInt64(7 * Math.Pow(2, 3 * n));
9          if (((signatureKey & testValue) ^ testValue) != 0)
10         {
11             signature += signatureKey & testValue;
12             cardinality++;
```

[3] Für *Xpomul* gilt $n = 15$.

[4] Eigentlich $111111111111111111111111111111111111111111111111111111111111101$, da es sich um eine 64bit Ganzzahl handelt und ab dem ersten Parameter vom Typ `VariableType.Array` alle weiteren Bit mit 1 gesetzt werden. Hier ist zu beachten, dass die Binärwerte der Datentypen in umgekehrter Reihenfolge zur Signatur verbunden werden.

```
13          }
14          else
15              break;
16      }
17      return new MostSignificantSignature(signature, cardinality);
18  }
```

Für $n \in \mathbb{N}$ und einen ermittelten Signaturschlüssel $s \in \mathbb{S}$ eines Methodenaufrufes ermittelt die Funktion sig gegeben durch:

$$sig(s, n) := s \wedge \sum_{i=0}^{n-1} \left(7 * 2^{3i}\right) \tag{5.3}$$

den signifikanten Teil des Signaturschlüssels s, um zu Prüfen, ob dieser zu einer Arraymethode passt. Dabei ist n die Kardinalität der *MSS* einer für den gewünschten Aufruf in Frage kommenden Methode. Stark vereinfacht werden aus der Signatur des gewünschten Aufrufs also alle Parameter entfernt die noch nicht zum Array-Teil der in Frage kommenden Methode gehören.

Beispiel:
Die Signatur des gewünschten Aufrufs lautet "Func String String Int" (f_1). Es wird eine Methode "Func String String Array" (f_2) im `MethodContext` gefunden, welche zumindest vom Namen zutreffen könnte.

	f_1		f_2	
Signaturschlüssel	001101101	$s \in \mathbb{S}$	111101101	$\widetilde{m} \in \mathbb{S}$
MSS	-		101101	
Kardinalität	-		2	
$sig(s, 2)$	101101		-	

Da die Werte 101101 und 101101 identisch sind, kann f_1 durch f_2 erfüllt werden.

Gegenbeispiel:
Die Signatur des gewünschten Aufrufs lautet "Func String String Int" (f_1). Es wird eine Methode "Func String Int Array" (f_2) im `MethodContext` gefunden, welche zumindest vom Namen zutreffen könnte.

	f_1		f_2	
Signaturschlüssel	001101101	$s \in \mathbb{S}$	111001101	$\widetilde{m} \in \mathbb{S}$
MSS	-		001101	
Kardinalität	-		2	
$sig(s, 2)$	101101		-	

Da die Werte 101101 und 001101 verschieden sind, kann f_1 durch f_2 nicht erfüllt werden.

Mit den soeben definierten Funktionen kann die Möglichkeit einen im Skript vorhandenen Methodenaufruf auszuführen durch die folgende Gleichung ausgedrückt werden:

$$\forall s \in \mathbb{S} : \exists! m \in \mathbb{M}, \exists n \in \mathbb{N} : m \text{ erfüllt } s :\Longleftrightarrow MSS(\widetilde{m}, n) \veebar sig(s, |MSS(\widetilde{m}, n)|) = 0 \tag{5.4}$$

5.4 Methoden definieren

Um den Funktionsumfang von Vogon zu erweitern, können eigene Methoden geschrieben und zum Methodenkontext eines Scripts hinzugefügt werden.

Die Klasse `MethodContext` hält Instanzen der Klasse `MethodDescriptor` in einem `Dictionary<String, MethodDescriptor>`. Der Schlüssel dieses Wörterbuches ist die Signatur einer Methode. Somit kann jede Signatur nur ein mal vorkommen und es ist anhand der Signatur immer eindeutig welche Methode aufgerufen werden soll.
Eine Methode für Vogon ist nichts weiter als eine .Net-Methode, die mit dem Attribut `VogonMethodAttribute` markiert wurde und dem Delegaten aus Listing x entspricht. Es ist jedoch zu beachten, dass eine solche Methode, damit sie verwendet werden kann als öffentlich und statisch definiert und in einer statischen Klasse untergebracht sein muss.

Listing 5.4: Der Delegat `VogonMethodDelegate`

```
1 public delegate dynamic VogonMethodDelegate(VariableDescriptor[] parameters);
```

Dieser gibt an, dass eine Vogon-Methode eine Rückgabe vom Typ `dynamic` hat und ein Array von `VariableDescriptor` als Parameter bekommt. Über dieses Array werden die Übergabeparameter aus dem Skript an die .Net-Methode übergeben. Als Beispiel sei die Definition der Methode `ToString()` gegeben.

Listing 5.5: Definition der Methode `ToString()`

```
1 [VogonMethod("ToString", "ToString Int", VariableType.String)]
2 [VogonMethod("ToString", "ToString Float", VariableType.String)]
3 [VogonMethod("ToString", "ToString Bool", VariableType.String)]
4 [VogonMethod("ToString", "ToString Date", VariableType.String)]
5 public static dynamic ToString(VariableDescriptor[] parameters)
6 {
7         return Convert.ToString(parameters[0].Value);
8 }
```

Die .Net-Methode `ToString()` wurde mehrfach mit dem `VogonMethodAttribute`-Attribut markiert. Der erste Parameter des Attributes gibt den Namen, der zweite die Signatur und der dritte den Rückgabetyp der Methode an. Wie aus dem Beispiel ersichtlich ist, kann eine .Net-Methode auch für mehrere Vogon-Methoden verwendet werden. Die Methode `ToString()` konvertiert einen übergebenen Wert in eine Zeichenkette. Dafür wird einfach die Methode `Convert.ToString()` aus dem .Net-Framework aufgerufen. Das mehrfache Markieren einer .Net-Methode mit dem `VogonMethodAttribute`-Attribut erzeugt also Überladungen von Vogon-Methoden, weshalb die in den Attributen angegebenen Signaturen eindeutig sein müssen. Sollte das nicht der Fall sein, wird die Klasse `MethodContext` eine Ausnahme auslösen, sobald sie eine Method hinzufügen möchte, deren Signatur bereits hinterlegt ist.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings