SBNF – Vom freien Textformat nach XML

Wandlungskünstler

XML ist das Standardformat für den Austausch von Informationen. Daten werden aber oft und gern in einem einfachen Textformat direkt editiert, gespeichert oder ausgetauscht. Das hier vorgestellte SBNF – Semantic Backus Naur Format – zusammen mit einem passenden Parser, bildet ein mögliches Bindeglied zwischen Textformaten und deren Abbildung und Weiterverarbeitung mit XML oder innerhalb von Java-Applikationen. Im vorliegenden Artikel wird eine Methodik vorgestellt, wie freie syntaktisch definierte Texte nach XML oder in Java-interne Daten konvertiert werden können.

von Hartmut Schorrig

Beispiele für die Erfassung von Daten in freien, syntaktisch definierten Texten sind alle Quellen in Programmiersprachen. Diese werden vom Compiler geparst und geeignet übersetzt. Interessant ist es aber bereits, Datenstrukturen, die in der C- und C++-Programmierung in Headerfiles beschrieben werden, nach XML zu konvertieren. Im Zuge einer Weiterverarbeitung kann daraus beispielsweise eine Dokumentation in XML erstellt oder ein Java-Programm generiert werden, das auf die Daten, präsentiert als Byte-Array, symbolisch zugreifen kann. Ein weiteres Anwendungsgebiet für SBNF ist der Datenaustausch mit Tools, die entweder nicht über einen XML-Output verfügen oder bei denen der XML-Output mit Ballast-Informationen überladen ist, die im konkreten Anwendungsfall nicht interessieren. So lässt sich mit dem SBNF-Parser ein aus Excel erzeugtes CSV-Format leicht nach XML konvertieren. Die Entwicklung des SBNF hat ihren Ursprung in der Verarbeitung textueller Logfiles und Traces, die als Stimuli-Input für Softwaretests verwendet werden sollten.

Historisches

Das Backus Naur Format (BNF) ist ein Ergebnis der Softwareentwicklung Ende der 50er Jahre. Damals wurde die Programmiersprache Algol 60 erstmals mit BNF vollständig syntaktisch beschrieben. Die Bezeichnung dieses Formats ist den Informatikern John Warner Backus und Peter Naur gewidmet. Die Schreibweise des BNF ist jedem Entwickler vertraut, wenn beispielsweise optionale Argumente in eckigen Klammern geschrieben werden:

```
DIR [Laufwerk:][Pfad][Dateiname] [/A[[:]Attribute]]
```

Niklaus Wirth erweiterte das BNF und beschrieb damit die von ihm in den 80er Jahren geschaffene Programmiersprache Pascal. Diese Erweiterung ist als EBNF (Extended BNF) bekannt. Ein wichtiges Merkmal ist die Einführung der geschweiften Klammern {...} für Wiederholungen. In der ursprünglichen BNF wurde stattdessen stark rekursiv definiert.

Der Verfasser selbst benutzte in den 80er Jahren für eigene Arbeiten EBNF, mit der Absicht, einen Parser zu entwickeln, der EBNF original einliest. Auch ein Versuch in den 90er Jahren, einen Parser in C++ zu realisieren, scheiterte an Aufwand und Zeitbudget. Erst mit der Verwendung von Java als Programmierbasis, der vorhandenen Hintergrund-Erfahrung und der Notwendigkeit, textuelle Stimulationsskripte einzulesen, entstand dieser Parser und eine entsprechende SBNF-Definition im Jahr 2006.

Ein kleines Beispiel

Eine Einkaufsliste liegt in einem einfachen, syntaktisch aber fassbaren Format vor:

```
Einkaufszettel
==============
2 Stck Butter
1 x Mehl
5 Eier
```

Die SBNF-Beschreibung der Syntax lautet wie folgt:

```
$setLinemode.

$main=einkaufsliste::=Einkaufszettel \n
<![=]*?> \n
{<position> \n }.

position::=<#?@menge> [<?@einheit>Stck|x|]
                                    <$?text()>.
```

Mit *$setLinemode* wird der Parser in den Zeilenmodus geschaltet. Im Standardfall werden Zeilenendezeichen als Leerraum überlesen, wie in C- oder Java-Quelltexten üblich. Im Zeilenmodus wird ein Zeilenendezeichen in der Syntax als Terminalsymbol *\n* vorgeschrieben. *$main=* leitet die Syntaxbeschreibung des Gesamttextes ein. Das ist im Beispiel eine *einkaufsliste*. Diese muss mit der Zeichenkette *Einkaufszettel* beginnen, gefolgt von einem Zeilenende. Danach muss eine Zeile folgen, die beliebig viele = enthält. *[=]** ist ein regulärer Ausdruck, wie er auch in Perl und anderen Scriptsprachen geläufig ist. In SBNF lassen sich alle regulären Ausdrücke verarbeiten, die in *java.utilRegex* benutzt werden können. Das *<!* leitet reguläre Ausdrücke ein. Nach der Trennzeile folgen beliebig viele *<position>*-Komponenten, jeweils in einer eigenen Zeile. Ausdrücke in *<..>* sind so genannte Syntaxkomponenten, in Algol 60 damals auch Metamorpheme genannt. *position* ist dabei gleichzeitig der Name der Syntaxkomponente in der Syntaxvorschrift und der Semantik. Die *position* ist wie folgt definiert: *<#?@menge>* verlangt

eine Zahl mit der Bedeutung *menge*. Nach dem *?* steht die Semantik-Bezeichnung, die in XML als Element- oder Attributname wieder erscheint. Danach kann in einer *position*-Zeile eine optionale Angabe von *Stck* oder *x* mit der Bedeutung *einheit* folgen. Die Semantik steht hier am Anfang der Optionsklammer *[<?Semantik>...]*. Die Optionsklammer enthält verschiedene Möglichkeiten, getrennt durch |. Mit einer Angabe |] wird explizit die nicht genutzte Option ermöglicht. Im Beispiel-Eingabetext tritt das bei *5 Eier* auf. Anschließend muss in einer *position*-Zeile ein Bezeichner folgen, der im XML-Dokument als TEXT-Element zur *<position>* gespeichert wird. Der Ausdruck *<$* verlangt einen Bezeichner wie er in Programmiersprachen üblich ist. Will man hier mehr, zum Beispiel Umlaute, kann man reguläre Ausdrücke einsetzen oder zusätzliche Zeichen nach dem $ notieren. Mit dem Aufruf des Parsers und Konverters auf der Kommandozeile

```
java -cp %CLASSPATH% vishia.stringScan.SBNF2Xml
 -ishoppinglist.txt -sshoppinglist.sbnf -youtput.xml
```

bekommt man eine XML-Ausgabe der Form

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<einkaufsliste>
 <position menge="2" einheit="Stck">Butter</position>
 <position menge="1" einheit="x">Mehl</position>
 <position menge="5">Eier</position>
</einkaufsliste>
```

Damit lässt sich die Einkaufsliste nun algorithmisch in der XML-Welt verarbeiten und beispielsweise dem Einkaufsroboter als Input anbieten.

Parsen von Headerfiles

In der C- und C++-Programmierung haben Headerfiles häufig die Aufgabe, Datenstrukturen zu definieren, die zwischen verschiedenen Anwendungen ausgetauscht werden. Nicht in jedem Fall ist eine XML-basierte Datenschnittstelle die richtige Wahl bei der Kopplung verschiedener Programmteile. Binär kodiert ist die Information dichter packbar und direkt verwertbar. Deshalb ist es wichtig, die in Headerfiles definierten Datenstrukturen auch außerhalb der C-Umgebung zu verarbeiten. Sei es, dass eine in Java programmierte Komponente Daten aufbereitet und über ein Byte-Array mit interner UDP-Kommunikation einer schnellen Echtzeitsteuerung in C bereitstellt; sei es, dass ausgetauschte oder erzeugte Daten von einer anderen Programmkomponente analysiert werden sollen. Grundsätzlich gibt es dabei drei Möglichkeiten:

- Man schreibt Headerfiles und unabhängig davon den byterichtigen Zugriff auf die Daten in der nicht C(++)-Umgebung.
- Man beschreibt Datenstrukturen außerhalb der C(++)-Programmierung, beispielsweise mit XML, und erzeugt daraus die für die Verarbeitung in C(++) notwendigen Headerfiles.
- Man nimmt die Headerfiles als Source und erzeugt daraus ein XML-Abbild für die Verarbeitung der Datenstrukturen außerhalb von C(++). Aus XML lässt sich alles andere konvertieren.

Die erste Variante ist mangels besserer Mittel recht häufig anzutreffen, jedoch wegen doppelter Datenhaltung und Abstimmungsaufwand nicht zu empfehlen. Die zweite und dritte Variante hat jeweils ihre Berechtigung. Es kommt darauf an, welche Tools verwendet werden und welche Kenntnisse und Gepflogenheiten in einem Softwareteam vorherrschen. In Headerfiles sind Datenstrukturen sehr übersichtlich darstellbar. In diesem Sinne ist die letztere Variante auch dann von Vorteil, wenn XML und/oder UML-Tools in breitem Umfang verwendet werden. SBNF und die Möglichkeit der Konvertierung nach XML bereiten den Weg für die Realisierung dieser Wahl. Letztlich kann mithilfe dieser Konvertierung auch eine Dokumentation der Datenstrukturen abgeleitet werden. Folgender Beispielausschnitt aus einem Headerfile sei gegeben:

```
/** Beispiel-Struktur. */
typedef struct Beispiel_t
{ /** Ein Element ist dokumentiert. Details sind in
  * * Listenpunkten
  * * weiteren Listenpunkten
  * : mit Unterabsätzen
  * angeführt. Es kann auch ‚fett‘ oder ‚kursiv“
  * geschrieben werden.
  *
  * Absatztrennung mit einer Leerzeile.
  */
 struct Beispiel2_t* element1;
}Beispiel;

/** Eine Methode wird definiert. Diese hat
 * @param wert einen Parameter.
 * @return und liefert etwas zurück.
 */
int MethodBsp3(int wert);
```

Mit SBNF und dem oben genannten Parser entsteht das in Listing 1 gezeigte XML-Dokument.

Listing 1

```
<structDefinition typetag="Beispiel_t" name=
                                        "Beispiel">
 <description><brief>
  <p>Beispiel-Struktur.</p>
 </brief></description>
 <attribute name="element1">
  <description>
   <brief>
    <p>Ein Element ist dokumentiert</p>
   </brief>
   <rest>
    <p>Details sind in</p>
    <ul><li><p>Listenpunkten</p>
     </li><li>
      <p>weiteren Listenpunkten</p>
      <p>mit Unterabsätzen angeführt. Es kann auch
       <b>fett</b> oder <i>kursiv</i> geschrieben
                                        werden.</p>
     </li>
    </ul>
    <p>Absatztrennung mit einer Leerzeile.</p>
   </rest>
  </description>
  <type forward="struct" ident="Beispiel2_t"
                                        pointer="true">
 </attribute>
</structDefinition>
<methodDef name="methodBsp3">
 <description><brief>
  <p>Eine Methode wird definiert</p>
 </brief>
 <rest>
  <p>Diese hat</p>
 </rest>
 <paramDescription ident="wert">
  <p>einen Parameter.</p>
 </paramDescription>
 <returnDescription>
  <p>und liefert etwas zurück.</p>
 </returnDescription>
</description>
<type ident="int" />
<typedParameter name="wert">
 <type ident="int" />
</typedParameter>
</methodDef>
```

Im Kommentar des Headerfiles sind Schriftauszeichnungen und eine Liste in der Art notiert, wie sie bei einem Wiki als Eingabeformat gebräuchlich ist. Die Umsetzung dieser Formatierung nach XML ist keine Leistung des SBNF-Parsers selbst, sondern wird bei der XML-Konvertierung mit der Klasse *vishia.Xml.ConverterWikistyleTextToXml* erzeugt. Die damit erzeugbare Formatierung ist im Headerfile besser lesbar als die teils übliche Durchsetzung des Textes mit HTML-Tags. Daraus lässt sich mit weiterer XSL-Konvertierung ein Dokument erzeugen, auch gemischt mit Informationen aus anderen XML-Dateien, beispielsweise einem XMI-Export aus UML-Modellen. Mit den in XML gespeicherten Parserergebnissen ist es möglich, über XSLT eine Wrapper-Java-Klasse zu erzeugen, mit deren Hilfe auf Daten in einem Byte-Array typ- und byterichtig zugegriffen werden kann.

Codebeispiele für Header-SBNF-Parsing

Nachfolgend ist ein Ausschnitt aus einem SBNF für das Parsen eines Headerfiles dargestellt. Der Inhalt sollte für C(++)-Kenner verständlich sein.

```
structDefinition::=struct [<$?@typetag>]
\{ { <structContent?> }
\} <$?@name>[ \[ <#?@arraySize> \] ] ;.

structContent::=<?>
[/**<description?-@>*/]
[ <invalidBlock?+>
| <validBlock?+>
| <unionDefinition?+unionAttribute>
| <structDefinition?+structAttribute>
| <attribute?+?>
| <defineDefinition?+?>
| <structContentInsideCondition?+?>
].

attribute::= <type> <$?@name> [<arraysize>];.
```

Bezüglich der SBNF-Notation ist anzumerken, dass die geschweiften Klammern, die als Terminalsyntax bei einer Strukturdefinition vorkommen, hier mit \{ und \} zu umschreiben sind. Das ist für alle Zeichen notwendig, die für SBNF selbst verwendet werden, also insbesondere für \<, \>, \., \ ?, \|. Mit \n \r \t werden wie in C oder Java üblich Steuerzeichen bezeichnet. Zusätzlich gibt es die Kennzeichnung \e für das Textende.

Interessant ist, wie eine Kommentierung verarbeitet wird. Die <description> steht bei einer Notationskonvention, wie sie dem bekannten Javadoc entspricht, immer über dem entsprechenden Element:

```
/** Beschreibung zu einer Struktur. */
struct Xy
{/** Beschreibung zu einem Element. */
 int dasElement ;
}
```

Mit der Schreibweise <description?-?> wird erreicht, dass die Beschreibung geparst, das Ergebnis aber vorerst nur temporär notiert wird. Danach erfolgt in einer Optionsklammer [..|..|..] die Entscheidung, um welches Element es sich handelt. Die Beschreibung soll aber zum Element zugehörig gespeichert werden. Das wird erreicht, indem beim Element <... ?+...> angegeben wird. Bezüglich der Art, wie die <description> geparst wird, gibt es ebenfalls eine Spezialdefinition:

```
description::=<*{ * }|*/?!test_description>.

test_description::=<*|\.|\.\r\n|\.\n|\e|\@?brief/
                                    p+> [\.] [<*|\e|\@?rest/p+>]
[{ @return <*@\e?returnDescription/p+>
| @param <paramDescription>
| @<*@\e?auxDescription>
}].

paramDescription::=<$?@ident><*@\e?p+>.
```

Der Parser kann grundsätzlich Kommentare überlesen, ohne dass das explizit im SBNF-Script angegeben ist. Die Kommentarzeichen (bei C typisch /*...*/ und //) sind dabei wählbar. Wird aber eine Kommentareinleitung wie /** als Terminalsymbol angegeben, dann erkennt der Parser dies und parst in den Kommentar hinein. Zusätzlich sind der Parser und die darunterliegende Stringverarbeitung in der Lage, die üblichen Einrückungen und Sterne-Spalten auszublenden. Das wird mit der Angabe <*{ * }|*/ erreicht. Der gesamte Text bis ausschließlich dem Kommentarendezeichen */ wird eingelesen, dabei werden alle Leerzeichen und * vor der Anfangs-Spaltenposition überlesen. Das eingelesene Ergebnis wird mit der Angabe <... !test_description> in einen Zwischenpuffer geparkt und dort anschließend mit der Syntaxangabe test_description : :=... näher untersucht. Damit wird das Innere einer Beschreibung im Kommentarblock auseinander genommen, um Schreibweisen wie @param name Beschreibung der Parameter ebenfalls syntaktisch im SBNF zu erfassen.

Parsen für Java-Auswertung

Es muss nicht zwingend XML ausgegeben werden, man kann die Ergebnisse auch direkt mit Java auswerten. Der vorliegende Parser liefert zunächst sein Ergebnis in einem ParserStore ab, bei dem jeweils eine Informationseinheit die Inhaltsinformation mit der zugehörigen Semantik verzeichnet, vergleichbar mit einem Kartenstapel aus Karteikarten. Dabei sind über bestimmte Karten jeweils Gummis gezogen, das sind die Karten pro Syntaxkomponente. Kartenblöcke sind dann wieder zusammengefasst zur darüberliegenden Syntaxkomponente. Die Abfrage dieses Ergebnisses in Java erfolgt mit *Methodenaufrufen Parser.getFirstParseResult(), ParseResultItem.next(parent)* und *ParseResultItem.nextSkipIntoComponent(parent)*. Mittels einer Klasse *vishia.stringScan.SBNFjavaOutput* ist es nun möglich, das gesamte Parser-Ergebnis in Anwenderklassen zu schreiben. Dazu wird das Prinzip der Reflection in Java genutzt. Die Semantik der Parserergebnisse, im SBNF-Skript festgelegt, muss Klassennamen, Attributen oder *set*-Methoden der Anwenderklassen entsprechen. Dann wird der Inhalt typgerecht in Instanzen von Anwenderklassen, deren Namen den Syntaxkomponentennamen in dem benutzten SBNF-Skript entsprechen, eingetragen. SBNF ist zurzeit eine Entwicklung, die in einem kleinen Team genutzt wird. Die Quellen des Java-Parsers liegen entsprechend der LGPL offen.

Dr. Hartmut Schorrig *hat von 1974 bis 1978 an der Technischen Hochschule Ilmenau (jetzt Technische Universität) Informatik studiert und 1986 zum Dr.-Ing. promoviert. Nach anfänglicher Arbeit an industrienaher Forschung ist er seit 1994 in der freien Wirtschaft als Softwareentwickler in der Automatisierungstechnik tätig.*

Links & Literatur

[1] Artikel zum Backus-Naur-Format: www.de.wikipedia.org/wiki/BNF

[2] Beschreibung des SBNF mit Beispielen und Download: www.vishia.de/SBNF