# XML Plugin 2.0

Parsen von XML-Dokumenten mit Hollywood

**Andreas Falkenhahn**

# Inhaltsverzeichnis

# 1 Allgemeine Informationen

## 1.1 Einführung

Das XML-Plugin ermöglicht es Hollywood-Skripten, XML-Dateien einfach und effizient zu parsen, sodass Sie diese äußerst flexible universelle Auszeichnungssprache nutzen können, die für so viele verschiedene Zwecke verwendet werden kann. Das Plugin bietet eine leistungsstarke Bibliotheksschnittstelle, mit der Sie auf alle Arten von Daten in XML-Dokumenten wie Knoten (Nodes), Attribute, Namensräume (Namespaces), Entitäten (Entities), Attributlisten (Attlists), CDATA-Abschnitte und mehr zugreifen können.

Darüber hinaus unterstützt das XML-Plugin auch die Serialisierungsschnittstelle von Hollywood, was bedeutet, dass Sie Hollywood-Tabellen bequem in XML-Dokumente serialisieren können, indem Sie nur einen einzigen Aufruf von Hollywoods Befehl `SerializeTable()` ausführen. Auf die gleiche Weise können Sie auch ganze XML-Dokumente in Hollywood-Tabellen deserialisieren, indem Sie einfach den Befehl `DeserializeTable()` von Hollywood verwenden. Einfacher geht es nicht!

## 1.2 Lizenz

FÜR EINEN BESTIMMTEN VERWENDUNGSZWECK. DAS VOLLSTÄNDIGE RISIKO DER QUALITÄT UND AUSFÜHRBARKEIT DES PROGRAMMS LIEGT BEIM ANWENDER. SOLLTE SICH DAS PROGRAMM ALS DEFEKT HERAUSSTELLEN, LIEGEN ALLE KOSTEN FÜR SERVICE, INSTANDSETZUNG ODER NACHBESSERUNG BEIM ANWENDER.

KEIN COPYRIGHT-INHABER ODER DRITTER, DER DAS PROGRAMM WIE OBEN ERLAUBT WEITERVERKAUFT, KANN FÜR SCHÄDEN IRGENDWELCHER ART HAFTBAR GEMACHT WERDEN (DIES BEINHALTET, IST ABER NICHT BESCHRÄNKT AUF, DATENVERLUST INFOLGE UNFÄHIGKEIT DES PROGRAMMS, MIT ANDEREN PROGRAMMEN ZUSAMMENZUARBEITEN), SELBST WENN EIN SOLCHER INHABER ODER DRITTER AUF DIE MÖGLICHKEIT EINES SOLCHEN SCHADENS HINGEWIESEN WURDE, AUSSER ES BESTEHT EINE SCHRIFTLICHE EINWILLIGUNG ODER WIRD VOM GESETZ VERLANGT.

## 1.3 Anforderungen

– Hollywood 9.0 oder besser

## 1.4 Installation

Die Installation von xml.hwp ist unkompliziert und einfach: Kopieren Sie einfach die Datei `xml.hwp` für die Plattform Ihrer Wahl in Hollywoods Plugin-Verzeichnis. Auf allen Systemen außer auf AmigaOS und kompatiblen müssen Plugins in einem Verzeichnis mit dem Namen `Plugins` gespeichert werden, das sich im selben Verzeichnis wie das Hauptprogramm von Hollywood befindet. Auf AmigaOS und kompatiblen Systemen müssen Plugins stattdessen in `LIBS:Hollywood` installiert werden. Unter macOS X muss sich das Verzeichnis `Plugins` im Verzeichnis `Resources` des Programmpakets befinden, d.h. im Verzeichnis `HollywoodInterpreter.app/Contents/Resources`. Beachten Sie, dass `HollywoodInterpreter.app` im Programmpaket `Hollywood.app` selbst gespeichert ist, nämlich in `Hollywood.app/Contents/Resources`.

Unter Windows sollten Sie auch die Datei `xml.chm` in das Verzeichnis `Docs` Ihrer Hollywood-Installation kopieren. Wenn sich dann der Cursor über einem xml.hwp-Befehl in der Hollywood-IDE befindet, können Sie die Online-Hilfe aufrufen, indem Sie F1 drücken.

Unter Linux und macOS kopieren Sie das Verzeichnis `xml`, das sich im Verzeichnis `Docs` des xml.hwp-Distributionsarchivs befindet, in das Verzeichnis `Docs` Ihrer Hollywood-Installation. Beachten Sie, dass sich unter macOS das Verzeichnis `Docs` innerhalb des Programmpakets `Hollywood.app` befindet, d.h. in `Hollywood.app/Contents/Resources/Docs`.

# 2 Über das XML-Plugin

## 2.1 Danksagungen

xml.hwp ist eines der ersten Hollywood-Plugins und wurde von Andreas Falkenhahn geschrieben. Es wurde ursprünglich als konzeptioneller Nachweis für die neue Bibliotheks-Plugin-Schnittstelle von Hollywood 5 geschrieben. Später wurde es erweitert, um die neue Serialisierungsschnittstelle von Hollywood 9 zu unterstützen, um XML-Dokumente bequem in Hollywood-Tabellen zu serialisieren und umgekehrt. Bedanken möchte ich mich bei Roberto Ierusalimschy, Andre Carregal und Tomas Guisasola für ihr LuaExpat-Plugin, auf dem die Bibliotheksschnittstelle von xml.hwp basiert.

Ein besonderer Dank geht an Helmut Haake und Dominic Widmer für die Übersetzung des Handbuchs ins Deutsche. Fehler oder Verbesserungsvorschläge bzgl. des deutschen Handbuchs bitte an das Übersetzungsteam richten, welches unter handbuch@gmx.ch oder https://amiga-resistance.info erreicht werden kann.

Wenn Sie mich kontaktieren möchten, senden Sie bitte eine E-Mail an andreas@airsoftsoftwair.de oder nutzen Sie das Kontaktformular unter http://www.hollywood-mal.com.

## 2.2 Häufig gestellte Fragen

In diesem Abschnitt werden einige häufig gestellte Fragen behandelt. Bitte lesen Sie sie zuerst, bevor Sie in einem Forum nachfragen, da Ihr Problem hier möglicherweise behandelt wurde.

**F: Gibt es ein Hollywood-Forum, in dem ich mit anderen Benutzern in Kontakt treten kann?**

A: Ja, bitte besuchen Sie die "Community" oder "Forum"-Sektion des offiziellen Hollywood-Portals unter http://www.hollywood-mal.com.

**F: Wo kann ich um Hilfe bitten?**

A: Es gibt ein lebhaftes englischsprachiges Forum auf http://forums.hollywood-mal.com. Sie können gerne Ihre Frage dort stellen. Ausserdem ist ein deutschsprachiges Forum vorhanden, welches Sie unter https://www.amiga-resistance.info/ erreichen können.

**F: Ich habe einen Fehler gefunden.**

A: Bitte informieren Sie mich darüber in den speziellen Bereichen des Forums.

## 2.3 Bekannte Probleme

Hier ist eine Liste von Punkten, die xlsx.hwp noch nicht unterstützt oder die auf irgendeine Weise verwirrend sein könnten:

- Ist noch offen (tpd)

## 2.4 Zukunft

Hier sind einige Punkte, die auf meiner Aufgabenliste stehen:

– mehr Beispiele hinzufügen

Zögern Sie nicht, mich zu kontaktieren, wenn xml.hwp eine bestimmte Funktion fehlt, die für Ihr Projekt wichtig ist.

## 2.5 Geschichte

Bitte schauen Sie in die auf englisch verfasste Datei `history.txt`. Hier finden Sie ein vollständiges Änderungsprotokoll von xml.hwp.

# 3 Anwendung

## 3.1 Schnittstellenübersicht

Es gibt zwei Möglichkeiten, dieses Plugin zu verwenden: Entweder über die Bibliotheksschnittstelle oder über die Serialisierungsschnittstelle. Die Verwendung des Plugins über die Serialisierungsschnittstelle ist einfacher und sehr bequem, geht aber auf Kosten der Flexibilität. Die Verwendung des Plugins über die Bibliotheksschnittstelle ist etwas schwieriger, bietet aber volle Flexibilität. In den nächsten beiden Kapiteln finden Sie einen kurzen Überblick über die beiden unterschiedlichen Schnittstellen.

## 3.2 Bibliotheksschnittstelle

Die Verwendung der Bibliotheksschnittstelle von xml.hwp verleiht Ihrem Skript die größte Flexibilität und ermöglicht Ihnen den Zugriff auf alle Funktionen von XML-Dokumenten. Die Bibliotheksschnittstelle basiert auf dem SAX-Parsing-Modell, was bedeutet, dass Sie eine Reihe von Callbacks angeben, die der XML-Parser beim Parsen Ihres Dokuments aufruft. Einer dieser Callbacks ist der StartElement()-Callback. Er wird immer dann aufgerufen, wenn der Parser im Dokument auf ein neues XML-Tag stößt. Wie in diesem Handbuch beschrieben, erhält StartElement() drei Argumente: Den Parser-Handle, den Elementnamen und eine Tabelle mit den im Tag angegebenen XML-Attributen. Ein StartElement()-Callback könnte also so aussehen:

```
Function p_StartElement(p, name$, attrs)
   DebugPrint("New tag found:", name$)
   For k,v In Pairs(attrs) Do DebugPrint(k .. "=" .. v)
EndFunction
```

Der obige Code gibt den Namen des XML-Tags aus, das der Parser gerade verarbeitet hat, sowie alle Attribute, die in der XML-Tag-Deklaration angegeben sind. Nachdem wir den Callback definiert haben, müssen wir nun einen XML-Parser erstellen. Dies geschieht mit dem Befehl xml.CreateParser(). Sie müssen alle Callbacks, die Sie verwenden möchten, an diesen Befehl übergeben. Für unser kleines Beispiel wollen wir nur den Callback StartElement() verwenden, also erstellen wir den Parser wie folgt:

```
p = xml.CreateParser({StartElement = p_StartElement})
```

Als nächstes können wir dem Parser XML-Daten zuführen. Dies geschieht durch den Aufruf der Methode parser:Parse():

```
p:Parse([[<plugin name="XML" author="A. Falkenhahn" version="1.0"/>]])
```

Das Ausführen dieses Codes gibt den Namen des Plugins sowie die Attribute `name`, `author` und `version` aus. Sie können parser:Parse() beliebig oft aufrufen. Wenn Sie fertig sind, vergessen Sie nicht, parser:Free() aufzurufen, um den Parser-Handle freizugeben. Hier ist also der vollständige Beispielcode unseres minimalen XML-Parsers von oben, fertig zum Kopieren und zum Einfügen in die Hollywood-IDE:

```
@REQUIRE "xml"

Function p_StartElement(p, name$, attrs)
   DebugPrint("New tag found:", name$)
```

```
   For k,v In Pairs(attrs) Do DebugPrint(k .. "=" .. v)
EndFunction

p = xml.CreateParser({StartElement = p_StartElement})
p:Parse([[<plugin name="XML" author="A. Falkenhahn" version="1.0"/>]])
p:Free()
```

Natürlich gibt es neben StartElement() noch viele andere Callback-Typen. Siehe Abschnitt 4.1 [xmlCreateParser], Seite 13, für Details.

Alternativ können Sie auch die Serialisierungsschnittstelle des Plugins verwenden. Diese ist einfacher, da nur ein einziger Befehlssaufruf erforderlich ist, um Hollywood-Tabellen in XML-Dokumente und umgekehrt zu konvertieren, aber Sie haben nicht die fein abgestimmte Kontrolle über alles, wie Sie es bei der Verwendung der Bibliotheksschnittstelle haben.

Im nächsten Kapitel finden Sie weitere Details zur Serialisierungsschnittstelle des Plugins.

## 3.3 Serialisierungsschnittstelle

Wenn Sie die Bibliotheksschnittstelle von xml.hwp (siehe oben) aus irgendeinem Grund nicht verwenden möchten, können Sie auch die Serialisierungsschnittstelle des Plugins benutzen. Diese ist einfacher zu verwenden, da nur ein einziger Befehlsaufruf erforderlich ist, um Hollywood-Tabellen in XML-Dokumente und umgekehrt zu konvertieren, aber Sie haben nicht die fein abgestimmte Kontrolle über alles, wie Sie es bei der Verwendung der Bibliotheksschnittstelle haben.

Der Zugriff auf die Serialisierungsschnittstelle von xml.hwp erfolgt über die Befehle `SerializeTable()` und `DeserializeTable()` von Hollywood oder alternativ über die Befehle `ReadTable()` und `WriteTable()`. Durch die Verwendung der Serialisierungsschnittstelle können Sie ein XML-Dokument durch nur einen einzigen Befehlsaufruf in eine Hollywood-Tabelle konvertieren:

```
t = DeserializeTable(FileToString("test.xml"), "xml")
```

Der obige Code liest alle Knoten und Attribute aus `test.xml` und speichert sie in der Hollywood-Tabelle `t`.

Die Art und Weise, wie die XML-Daten in der Tabelle gespeichert werden, hängt von dem Serialisierungsmodus ab, den Sie mit xml.SetSerializeMode() eingestellt haben. Das XML-Plugin unterstützt drei verschiedene Serialisierungsmodi:

1. Listenmodus: Dadurch werden alle XML-Knoten als aufeinanderfolgende Listenelemente in der Tabelle gespeichert. Der erste XML-Knoten befindet sich bei Index 0, der zweite bei Index 1 und so weiter. Dies ist der Standardmodus. Siehe Abschnitt 3.4 [Listen-Serialisierung], Seite 7, für Details.
2. Benannter Modus: Dadurch werden alle XML-Knoten als benannte Elemente in der Tabelle gespeichert. Das bedeutet, dass Sie bequem über den Namen auf Knoten zugreifen können, anstatt numerische Indizes verwenden zu müssen. Der Nachteil dieses Modus besteht darin, dass Sie nicht mehrere Knoten mit demselben Namen auf der gleichen Ebene haben können, da die Knoten nach Namen gespeichert werden und jeder Name nur einmal pro Knotenebene verfügbar ist. Ein weiterer Nachteil besteht darin, dass Sie keine Kontrolle über die Reihenfolge der Knoten haben, wenn Sie sie zurück in ein XML-Dokument serialisieren. Siehe Abschnitt 3.5 [Benannte Serialisierung], Seite 8, für Details.

3. Hollywood-Modus: Dies ist ein spezieller Modus, mit dem Sie beliebige Hollywood-Tabellen serialisieren können. Im Gegensatz zu den ersten beiden Modi erfordert der Hollywood-Modus kein bestimmtes Layout der Tabelle. Sie können in diesem Modus jede Tabelle serialisieren, genauso wie Sie es mit den Befehlen `ReadTable()` und `WriteTable()` von Hollywood können. Die Tabelle kann sogar Binärdaten oder Code wie Hollywood-Befehle enthalten. Siehe Abschnitt 3.6 [Hollywood-Serialisierung], Seite 10, für Details.

Nachdem Sie eine XML-Datei in eine Hollywood-Tabelle konvertiert haben, können Sie beliebige Änderungen direkt an der Hollywood-Tabelle vornehmen. Nachdem Sie mit allen Änderungen fertig sind, können Sie Ihre Hollywood-Tabelle einfach in nur einer einzigen Zeile wie folgt wieder in ein XML-Dokument konvertieren:

```
StringToFile(SerializeTable(t, "xml"), "test2.xml")
```

Der obige Code konvertiert die Tabelle `t` mithilfe des Plugins xml.hwp in ein XML-Dokument und speichert das XML-Dokument als `test2.xml`.

Wie Sie sehen können, ist die Serialisierungsschnittstelle sehr einfach zu verwenden, bietet jedoch nicht so viel Flexibilität wie die Bibliotheksschnittstelle, die Ihnen eine fein abgestimmte Kontrolle über viele Funktionen von XML-Dokumenten gibt.

## 3.4 Listen-Serialisierung

Der Listen-(De)Serialisierungs-Modus, der auch der Standardmodus ist, speichert alle XML-Knoten als aufeinanderfolgende Listenelemente in der Tabelle. Der erste XML-Knoten befindet sich bei Index 0, der zweite bei Index 1 und so weiter. Die vom Listen-Deserialisierer erzeugte Tabelle ist eine Tabelle mit Tabellen, die das gesamte XML-Dokument enthält. Die folgenden Tabellenfelder werden für jeden Knoten initialisiert:

`Name` Name des XML-Knotens.

`Text` Zeichendaten des XML-Knotens.

`Attrs` Setzt auf eine Tabelle mit Untertabellen, die alle Attribute als Schlüssel- und Wertepaare enthalten. Für jede Untertabelle in der Tabelle `Attrs` werden die folgenden Felder initialisiert:

- `Key` Der Name des Attributs.
- `Value` Der Wert des Attributs.

`Nodes` Wird auf eine Tabelle gesetzt, die die Elemente des Knotens enthält.

Um alle Einträge der Tabelle deserialisiert aus einem XML-Dokument mit `DeserializeTable()` auszugeben, könnten Sie die folgende rekursive Funktion verwenden:

```
Function p_PrintNodes(t, indent)
    DebugPrint(RepeatStr(" ", indent) .. "+" .. t.name)
    For Local k = 0 To ListItems(t.attrs) - 1
       DebugPrint(RepeatStr(" ", indent + 1) ..
          "attr: " .. t.attrs[k].key .. "=" .. t.attrs[k].value)
    Next
    If Not EmptyStr(t.text)
       DebugPrint(RepeatStr(" ", indent + 1) .. t.text)
```

```
    EndIf
    For Local k = 0 To ListItems(t.nodes) - 1
       p_PrintNodes(t.nodes[k], indent + 4)
    Next
EndFunction

t = DeserializeTable(FileToString("test.xml"), "xml")
p_PrintNodes(t[0], 0)
```

Um einen Knoten zu ändern, können Sie dann einfach den gewünschten Wert in der Hollywood-Tabelle ändern und ihn dann mit Hollywoods Befehl `SerializeTable()` zurück nach XML serialisieren. Nehmen wir zum Beispiel an, dies ist Ihre XML-Datei:

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
</catalog>
```

Um den Preis auf 39,95 zu ändern und die aktualisierte XML-Datei auf der Festplatte zu speichern, könnten Sie wie folgt vorgehen:

```
t = DeserializeTable(FileToString("catalog.xml"), "xml")
t[0].nodes[0].nodes[3].text = "39.95"
StringToFile(SerializeTable(t, "xml"), "catalog_new.xml")
```

Beachten Sie, dass bei der Verwendung der Listen-Serialisierung die Tabelle, die Sie an `SerializeTable()` übergeben, genau den oben beschriebenen Konventionen entsprechen muss, d.h. die Tabelle muss eine Reihe von Untertabellen enthalten, die mit aufeinanderfolgenden numerischen Indizes gespeichert sind, und die oben beschriebenen Felder müssen für alle Knoten initialisiert werden oder es wird ein Fehler auftreten.

Für das oben gezeigte XML-Beispiel könnte es jedoch bequemer sein, die Benannte Serialisierung zu verwenden, da keine Knotennamen zweimal verwendet werden. Siehe Abschnitt 3.5 [Benannte Serialisierung], Seite 8, für Details.

## 3.5 Benannte Serialisierung

Die benannte (De-)Serialisierung speichert alle XML-Knoten als benannte Elemente in der Tabelle. Das bedeutet, dass Sie bequem über den Namen auf Knoten zugreifen können, anstatt numerische Indizes verwenden zu müssen. Der Nachteil dieses Modus besteht darin, dass Sie nicht mehrere Knoten mit demselben Namen auf gleichen Ebene haben können, da die Knoten nach Namen gespeichert werden und jeder Name nur einmal pro Knotenebene verfügbar ist. Ein weiterer Nachteil ist, dass Sie keine Kontrolle über die Reihenfolge der Knoten haben, wenn Sie sie zurück in ein XML-Dokument serialisieren.

Die folgenden Tabellenfelder werden für jeden Knoten initialisiert, wenn der benannte Serialisierungsmodus verwendet wird:

`Text` Die Zeichendaten des XML-Knotens.

`Attrs` Setzt auf eine Tabelle mit Untertabellen, die alle Attribute als Schlüssel- und Wertepaare enthalten. Für jede Untertabelle in der Tabelle `Attrs` werden die folgenden Felder initialisiert:

- `Key` Der Name des Attributs.
- `Value` Der Wert des Attributs.

`Nodes` Wird auf eine Tabelle gesetzt, die die Elemente des Knotens enthält.

Nehmen wir zum Beispiel an, dies sei Ihre XML-Datei:

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
</catalog>
```

Um den Preis auf 39,95 zu ändern und die aktualisierte XML-Datei auf der Festplatte zu speichern, könnten Sie wie folgt vorgehen:

```
xml.SetSerializeMode(#XML_SERIALIZEMODE_NAMED)
t = DeserializeTable(FileToString("catalog.xml"), "xml")
t.catalog.nodes.book.nodes.price.text = "39.95"
StringToFile(SerializeTable(t, "xml"), "catalog_new.xml")
```

Beachten Sie, dass bei der Verwendung der benannten Serialisierung die Tabelle, die Sie an `SerializeTable()` übergeben, genau den oben beschriebenen Konventionen entsprechen muss, d.h. die Tabelle muss eine Reihe von Untertabellen enthalten, die an benannten Indizes gespeichert sind, und die oben beschriebenen Felder müssen für alle Knoten initialisiert werden, sonst tritt ein Fehler auf.

Beachten Sie auch, dass die benannte Serialisierung XML-Knoten als benannte Tabellenfelder speichert und Sie denselben Namen nur einmal pro Baumebene verwenden können. Daher können Sie die benannte Serialisierung nicht verwenden, um XML-Dateien zu deserialisieren, die mehrere Knoten mit demselben Namen haben, z. B. so:

```
<?xml version="1.0"?>
<catalog>
   <item>First book</item>
   <item>Second book</item>
   <item>Third book</item>
</catalog>
```

In der oben gezeigten XML-Datei gibt es drei `<item>`-Tags auf der gleichen Baumebene. Dies kann nicht mit der benannten Serialisierung deserialisiert werden, da in Hollywood-Tabellen jeder Name nur einmal verwendet werden kann. Um solche Tabellen zu deserialisieren, müssen Sie die Listen-Serialisierung verwenden. Siehe Abschnitt 3.4 [Listen-Serialisierung], Seite 7, für Details.

## 3.6 Hollywood-Serialisierung

Die Hollywood-(De-)Serialisierung ist ein spezieller Modus, mit dem Sie beliebige Hollywood-Tabellen serialisieren können. Im Gegensatz zu den ersten beiden Modi erfordert der Hollywood-Modus kein bestimmtes Layout der Tabelle. In diesem Modus können Sie jede Tabelle in XML serialisieren, genau wie Sie es mit den Befehlen `ReadTable()` und `WriteTable()` von Hollywood können. Die Tabelle kann sogar Binärdaten oder Code wie Hollywood-Befehle enthalten.

Der Nachteil von diesem Modus besteht darin, dass beim Deserialisieren von XML-Daten zurück in eine Hollywood-Tabelle die XML-Daten einer bestimmten Konvention folgen muss, da Hollywood den Typ der in XML-Elementen gespeicherten Daten kennen muss. Obwohl es wahrscheinlich in den meisten Fällen funktionieren wird, ist es daher nicht möglich, beliebige XML-Daten mit der Hollywood-Serialisierung allgemein zu deserialisieren. Sie können nur XML-Daten sicher deserialisieren, die zuvor mit der Hollywood-Serialisierung serialisiert wurden. Hier ist ein Beispiel, betrachten Sie diese Hollywood-Tabelle:

```
t = {1, 2, 3, 4, 5, test = "Hello", subtable = {x = 5, y = 6, z = "8"}}
```

Wenn Sie diese Tabelle mit der Hollywood-Serialisierung in XML serialisieren, sieht sie so aus:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
   <item.0>1</item.0>
   <item.1>2</item.1>
   <item.2>3</item.2>
   <item.3>4</item.3>
   <item.4>5</item.4>
   <test>Hello</test>
   <subtable>
      <y>6</y>
      <x>5</x>
      <z type="string">8</z>
   </subtable>
</root>
```

Sie sehen, dass aufeinanderfolgende Tabellenindizes in Form von `<item.n>` gespeichert werden und dass der Tag `<z>` ein zusätzliches Attribut `type` hat, das dem Deserialisierer mitteilt, dass der Wert als Zeichenkette in der Hollywood-Tabelle gespeichert werden soll und nicht als Zahl. All diese Dinge sind spezielle Konventionen von der Hollywood-Serialisierung, weshalb sie nicht verwendet werden kann, um beliebige XML-Dokumente zu deserialisieren. Der Tag `<root>` auf der obersten Ebene kann über den Befehl xml.SetSerializeOptions() in einen anderen Tag geändert werden.

Beachten Sie, dass die Hollywood-Serialisierung auch Binärdaten unterstützt. Falls die Tabelle also Hollywood-Befehle oder Zeichenketten enthält, die Binärdaten enthalten, werden diese Daten als Base64 im XML codiert.

Um eine beliebige Hollywood-Tabelle in ein XML-Dokument zu deserialisieren und sie dann wieder in eine Hollywood-Tabelle zu serialisieren, können Sie folgendes tun:

```
xml.SetSerializeMode(#XML_SERIALIZEMODE_HOLLYWOOD)
StringToFile(SerializeTable(t, "xml"), "test.xml")
copy_of_t = DeserializeTable(FileToString("test.xml"), "xml")
```

# 4 Befehlsreferenz

## 4.1 xml.CreateParser

**BEZEICHNUNG**
xml.CreateParser – erstellt einen neuen Parser

**FRÜHERER NAME**
xmlparser.New (V1.x)

**ÜBERSICHT**
```
p = xml.CreateParser(t[, sep$, merge])
```

**BESCHREIBUNG**
Dieser Befehl erstellt einen neuen Parser, der zum Durchlaufen von XML-Dokumenten verwendet werden kann. Sie müssen eine Tabelle übergeben, die eine Auswahl von Callback-Funktionen enthält, die aufgerufen werden sollen, während das XML-Dokument geparst wird. Die folgenden Callback-Funktionen können derzeit im Tabellenargument übergeben werden:

`AttlistDecl`
: Die Funktion, die Sie hier angeben, wird für die Deklarationen von attlist in der DTD aufgerufen. Siehe Abschnitt 6.1 [AttlistDecl], Seite 23, für Details. (V2.0)

`CharacterData`
: Die hier angegebene Funktion wird für alle Daten aufgerufen, die nicht Teil einer XML-Entität sind. Siehe Abschnitt 6.2 [CharacterData], Seite 24, für Details.

`Comment`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser im XML auf einen Kommentar stößt. Siehe Abschnitt 6.3 [Comment], Seite 24, für Details.

`DefaultExpand`
: Dies ist dasselbe wie `DefaultHandler`, außer dass der Handler die Erweiterung interner Entitätsreferenzen nicht verhindert. Siehe Abschnitt 6.4 [DefaultExpand], Seite 25, für Details.

`DefaultHandler`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser Zeichen im Dokument findet, die sonst nicht verarbeitet würden. Siehe Abschnitt 6.5 [DefaultHandler], Seite 25, für Details.

`ElementDecl`
: Die hier angegebene Funktion wird für Elementdeklarationen in einer DTD aufgerufen. Siehe Abschnitt 6.6 [ElementDecl], Seite 26, für Details. (V2.0)

`EndCDataSection`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser das Ende eines XML-CDATA-Abschnitts erkennt. Siehe Abschnitt 6.7 [EndCDataSection], Seite 27, für Details.

`EndDocTypeDecl`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser das Ende einer DOCTYPE-Deklaration erreicht. Siehe Abschnitt 6.8 [EndDocTypeDecl], Seite 28, für Details. (V2.0)

`EndElement`
: Die Funktion, die Sie hier angeben, wird aufgerufen, wenn ein XML-Knoten geschlossen wird. Siehe Abschnitt 6.9 [EndElement], Seite 28, für Details.

`EndNamespaceDecl`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser das Ende einer XML-Namensräume-Deklaration erkennt. Beachten Sie, dass Sie den optionalen Parameter `sep$` angeben müssen, wenn der Parser Namensräume verarbeiten soll. Siehe Abschnitt 6.10 [EndNamespaceDecl], Seite 29, für Details.

`EntityDecl`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser eine Entity-Deklaration erkennt. Siehe Abschnitt 6.11 [EntityDecl], Seite 29, für Details. (V2.0)

`ExternalEntityRef`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser eine externe Entitätsreferenz erkennt. Siehe Abschnitt 6.12 [ExternalEntityRef], Seite 30, für Details.

`NotationDecl`
: Die Funktion, die Sie hier angeben, wird aufgerufen, wenn der Parser auf eine Notationsdeklaration trifft. Siehe Abschnitt 6.13 [NotationDecl], Seite 30, für Details.

`NotStandalone`
: Die hier angegebene Funktion wird aufgerufen, wenn das Dokument nicht eigenständig ist, ohne dass dies in der XML-Deklaration angegeben wird. Siehe Abschnitt 6.14 [NotStandalone], Seite 31, für Details.

`ProcessingInstruction`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser XML-Verarbeitungsanweisungen erkennt. Siehe Abschnitt 6.15 [ProcessingInstruction], Seite 31, für Details.

`SkippedEntity`
: Die Funktion, die Sie hier angeben, wird aufgerufen, wenn Entitäten übersprungen werden. Siehe Abschnitt 6.16 [SkippedEntity], Seite 32, für Details. (V2.0)

`StartCDataSection`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser den Beginn eines XML-CDATA-Abschnitts erkennt. Siehe Abschnitt 6.17 [StartCDataSection], Seite 32, für Details.

`StartDocTypeDecl`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser die DOCTYPE-Deklaration erreicht. Siehe Abschnitt 6.18 [StartDocTypeDecl], Seite 33, für Details.

`StartElement`
: Die Funktion, die Sie hier angeben, wird aufgerufen, wenn der Parser auf einen neuen XML-Knoten stößt. Siehe Abschnitt 6.19 [StartElement], Seite 33, für Details.

`StartNamespaceDecl`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser eine XML-Namensräume-Deklaration erkennt. Beachten Sie, dass Sie den optionalen Parameter `sep$` angeben müssen, wenn der Parser Namensräume verarbeiten soll. Siehe Abschnitt 6.20 [StartNamespaceDecl], Seite 34, für Details.

`UnparsedEntityDecl (veraltet)`
: Die hier angegebene Funktion wird aufgerufen, wenn der Parser Deklarationen von nicht geparsten Entitäten erkennt. Siehe Abschnitt 6.21 [UnparsedEntityDecl], Seite 35, für Details. Beachten Sie, dass `UnparsedEntityDecl` veraltet ist und Sie stattdessen `EntityDecl` verwenden sollten (siehe oben).

`XMLDecl`
: Die Funktion, die Sie hier angeben, wird aufgerufen, wenn der Parser XML-Deklarationen erkennt. Siehe Abschnitt 6.22 [XMLDecl], Seite 35, für Details. (V2.0)

Das optionale Trennzeichen `sep$` kann verwendet werden, um das Zeichen zu definieren, das in den erweiterten Elementnamen des Namensraums verwendet wird. Wenn `sep$` nicht definiert ist, verarbeitet der Parser keine Namensräume.

Der optionale Parameter `merge` kann verwendet werden, um zu konfigurieren, ob der `CharacterData`-Callback versuchen soll, so viele Daten wie möglich zu einer einzigen Zeichenkette zusammenzufassen oder nicht. Dies ist standardmäßig `True`. Wenn Sie diesen Parameter auf `False` setzen, könnte Ihr `CharacterData`-Callback häufiger aufgerufen werden, da die Zeichendaten nicht zu größeren Textabschnitten zusammengeführt werden.

Diese Funktion gibt ein Parser-Objekt zurück. Sie können parser:Parse() aufrufen, um XML mit Hilfe des Parser-Objekts zu parsen. Wenn Sie mit dem Parsen fertig sind, rufen Sie parser:Free() auf, um das Parser-Objekt freizugeben.

**EINGABEN**

`t` Tabelle mit einer oder mehreren Callback-Funktionen (siehe oben)

`sep$` optional: Trennzeichen

`merge` optional: ob der Parser versuchen soll, Zeichendaten zu größeren Textabschnitten zusammenzufassen (Standardwert `True`)(V2.0)

**RÜCKGABEWERTE**

`p` ein Parser-Objekt

**BEISPIEL**

Siehe Abschnitt 6.19 [StartElement], Seite 33.

## 4.2 xml.SetSerializeMode

**BEZEICHNUNG**

xml.SetSerializeMode – stellt den Serialisierungsmodus ein (V2.0)

**ÜBERSICHT**

```
xml.SetSerializeMode(mode)
```

**BESCHREIBUNG**

Damit kann der gewünschte Serialisierungsmodus eingestellt werden, wenn das Plugin über seine Serialisierungsschnittstelle verwendet wird. Sie müssen den gewünschten Serialisierungsmodus im Argument `mode` übergeben. Die folgenden Serialisierungsmodi werden derzeit unterstützt:

`#XML_SERIALIZEMODE_LIST`
: Dadurch werden alle XML-Knoten als aufeinanderfolgende Listenelemente in der Tabelle gespeichert. Der erste XML-Knoten befindet sich bei Index 0, der zweite bei Index 1 und so weiter. Dies ist der Standardmodus. Siehe Abschnitt 3.4 [Listen-Serialisierung], Seite 7, für Details.

`#XML_SERIALIZEMODE_NAMED`
: Dadurch werden alle XML-Knoten als benannte Elemente in der Tabelle gespeichert. Das bedeutet, dass Sie bequem über den Namen auf Knoten zugreifen können, anstatt numerische Indizes verwenden zu müssen. Der Nachteil dieses Modus besteht darin, dass Sie nicht mehrere Knoten mit demselben Namen auf derselben Ebene haben können, da die Knoten nach Namen gespeichert werden und jeder Name nur einmal pro Knotenebene verfügbar ist. Ein weiterer Nachteil besteht darin, dass Sie keine Kontrolle über die Reihenfolge der Knoten haben, wenn Sie sie zurück in ein XML-Dokument serialisieren. Siehe Abschnitt 3.5 [Benannte Serialisierung], Seite 8, für Details.

`#XML_SERIALIZEMODE_HOLLYWOOD`
: Dies ist ein spezieller Modus, mit dem Sie beliebige Hollywood-Tabellen serialisieren können. Im Gegensatz zu den ersten beiden Modi erfordert der Hollywood-Modus kein bestimmtes Layout der Tabelle. Sie können in diesem Modus jede Tabelle serialisieren, genauso wie Sie es mit den Befehlen `ReadTable()` und `WriteTable()` von Hollywood können. Die Tabelle kann sogar Binärdaten oder Code wie Hollywood-Befehle enthalten. Siehe Abschnitt 3.6 [Hollywood-Serialisierung], Seite 10, für Details.

Weitere Optionen können mit dem Befehl xml.SetSerializeOptions() konfiguriert werden. Siehe Abschnitt 4.3 [xml.SetSerializeOptions()], Seite 16, für Details.

**EINGABEN**

`mode` gewünschter Serialisierungsmodus (siehe oben für mögliche Werte)

## 4.3 xml.SetSerializeOptions

**BEZEICHNUNG**

xml.SetSerializeOptions – legt die Serialisierungsoptionen fest (V2.0)

**ÜBERSICHT**

```
xml.SetSerializeOptions(t)
```

**BESCHREIBUNG**

Mit diesem Befehl können bestimmte Optionen festgelegt werden, wenn das Plugin über seine Serialisierungsschnittstelle verwendet wird. Das einzige Argument, das von diesem Befehl verwendet wird, ist eine Tabelle, die eines oder mehrere der folgenden Elemente enthalten kann:

`RootNode` Dies kann verwendet werden, um den Namen des XML-Wurzelknotens festzulegen. Dies wird nur verwendet, wenn der Serialisierungsmodus `#XML_SERIALIZEMODE_HOLLYWOOD` ist. Der Standardwert ist `"root"`.

`ForceLowerCase`
Bei Verwendung der Listen- oder benannten Serialisierungsmodi konvertiert das XML-Plugin immer alle Knotennamen in Kleinbuchstaben. Wenn Sie das nicht möchten, setzen Sie dieses Tabellenelement auf `False`. Der Standardwert ist `True`. Dieser Tag wird für den Hollywood-Serialisierungsmodus ignoriert.

`IgnoreWhiteSpace`
Bei Verwendung der Listen- oder benannten Serialisierungsmodi setzt das XML-Plugin das Feld `Text` von Knoten, die nur Leerzeichen (z.B. Leerzeichen, Tabulatoren, Zeilenumbrüche usw.) enthalten, auf eine leere Zeichenkette. Wenn Sie das nicht möchten, setzen Sie dieses Tabellenelement auf `False`. Der Standardwert ist `True`. Dieser Tag wird für den Hollywood-Serialisierungsmodus ignoriert.

**EINGABEN**

`t` Tabelle mit den gewünschten Serialisierungsoptionen

# 5 Parser-Methoden

## 5.1 parser:Free

**BEZEICHNUNG**
parser:Free – löscht/schliesst den Parser

**FRÜHERER NAME**
parser:Close (V1.x)

**ÜBERSICHT**
```
parser:Free()
```

**BESCHREIBUNG**
Schließt den Parser und löscht den gesamten von ihm verwendeten Speicher. Ein Aufruf der Methode parser:Free() ohne vorherigen Aufruf von parser:Parse() könnte zu einem Fehler führen.

**EINGABEN**
keine

**BEISPIEL**
Siehe Abschnitt 6.19 [StartElement], Seite 33.

## 5.2 parser:GetBase

**BEZEICHNUNG**
parser:GetBase – ermittelt die Basis für relative URIs

**ÜBERSICHT**
```
s$ = parser:GetBase()
```

**BESCHREIBUNG**
Ermittelt die Basis, die zum Auflösen relativer URIs in System IDs verwendet werden soll. Siehe auch parser:SetBase().

**EINGABEN**
keine

**RÜCKGABEWERTE**
`s$` Basis-Zeichenkette

## 5.3 parser:GetCallbacks

**BEZEICHNUNG**
parser:GetCallbacks – gibt die Parser-Callbacks zurück

**ÜBERSICHT**
```
cb = parser:GetCallbacks()
```

**BESCHREIBUNG**
Dies gibt einfach eine Tabelle zurück, die alle Callbacks enthält, die beim Erstellen des Parsers mit xml.CreateParser() angegeben wurden.

**EINGABEN**
keine

**RÜCKGABEWERTE**

| | |
|---|---|
| `cb` | Tabelle mit Parser-Callbacks |

## 5.4 parser:GetError

**BEZEICHNUNG**
parser:GetError – gibt erweiterte Fehlerinformationen zurück (V2.0)

**ÜBERSICHT**
```
s$, line, col, pos = parser:GetError()
```

**BESCHREIBUNG**
Wenn parser:Parse() einen Fehler zurückgibt, können Sie diese Methode verwenden, um erweiterte Fehlerinformationen zu erhalten. Die Methode gibt eine menschenlesbare Fehlermeldung in `s$`, den Zeilen- und Spaltenindex des Fehlers in `line` und `col` und den Byteindex des Fehlers in `pos` zurück.

**EINGABEN**
keine

**RÜCKGABEWERTE**

| | |
|---|---|
| `s$` | Fehlermeldung |
| `line` | Zeilenindex (ab 1) |
| `col` | Spaltenindex (ab 1) |
| `pos` | Byte Index des Fehlers (ab 1) |

## 5.5 parser:GetPosition

**BEZEICHNUNG**
parser:GetPosition – gibt die aktuelle Parsing-Position zurück

**FRÜHERER NAME**
parser:Pos (V1.x)

**ÜBERSICHT**
```
line, col, pos = parser:GetPosition()
```

**BESCHREIBUNG**
Diese Methode gibt die aktuelle Parser-Position zurück. Die Zeilen- und Spaltenindizes werden in `line` und `col` und der Byteindex in `pos` zurückgegeben.

**EINGABEN**
keine

**RÜCKGABEWERTE**

| | |
|---|---|
| `line` | aktueller Zeilenindex (ab 1) |
| `col` | aktueller Spaltenindex (ab 1) |
| `pos` | aktueller Byte-Index (ab 1) |

## 5.6 parser:Parse

**BEZEICHNUNG**
parser:Parse – parst XML-Code

**ÜBERSICHT**
```
ok = parser:Parse(s$)
```

**BESCHREIBUNG**
Diese Methode parst XML-Code. Die Zeichenkette `s$` enthält einen Teil (oder vielleicht alles) des zu parsenden Dokuments. Die Methode gibt einen booleschen Wert zurück, der Erfolg (`True`) oder Fehler (`False`) anzeigt. Wenn parser:Parse() fehlschlägt, können Sie parser:GetError() verwenden, um erweiterte Fehlerinformationen zu erhalten.

**EINGABEN**

| | |
|---|---|
| `s$` | zu parsender XML-Code |

**RÜCKGABEWERTE**

| | |
|---|---|
| `ok` | boolescher Wert, der Erfolg (`True`) oder Misserfolg (`False`) anzeigt |

**BEISPIEL**
Siehe Abschnitt 6.19 [StartElement], Seite 33.

## 5.7 parser:SetBase

**BEZEICHNUNG**
parser:SetBase – setzt die Basis für relative URIs

**ÜBERSICHT**
```
parser:SetBase(s$)
```

**BESCHREIBUNG**
Setzt die Basis, die für die Auflösung von relativen URIs in System IDs verwendet werden soll, auf die in s$ übergebene Zeichenkette. Siehe auch parser:GetBase().

**EINGABEN**

| | |
|---|---|
| `s$` | gewünschte Basis |

## 5.8 parser:SetEncoding

**BEZEICHNUNG**

parser:SetEncoding – legt die Parser-Codierung fest

**ÜBERSICHT**

```
parser:SetEncoding(e$)
```

**BESCHREIBUNG**

Leget die vom Parser zu verwendende Codierung fest. Es gibt vier integrierte Codierungen, die als Zeichenketten übergeben werden: `"US-ASCII"`, `"UTF-8"`, `"UTF-16"` und `"ISO-8859-1"`. parser:SetEncoding() darf nicht aufgerufen werden, nachdem das Parsing bereits mit parser:Parse() gestartet wurde.

**EINGABEN**

| | |
|---|---|
| `e$` | Codierung, die vom Parser verwendet werden soll |

## 5.9 parser:Stop

**BEZEICHNUNG**

parser:Stop – stoppt den Parser

**ÜBERSICHT**

```
ok = parser:Stop()
```

**BESCHREIBUNG**

Bricht den Parser ab und verhindert, dass er die zuletzt übergebenen Daten weiter parst. Verwenden Sie dies, um das Parsen des Dokuments anzuhalten, wenn beispielsweise ein Fehler in einem Callback entdeckt wird. Das Parser-Objekt kann nach diesem Aufruf keine weiteren Daten annehmen. Die Methode gibt einen booleschen Wert zurück, der Erfolg (`True`) oder Fehler (`False`) anzeigt.

**EINGABEN**

keine

**RÜCKGABEWERTE**

| | |
|---|---|
| `ok` | boolescher Wert, der Erfolg (`True`) oder Misserfolg (`False`) anzeigt |

# 6 Callback-Referenz

## 6.1 AttlistDecl

**BEZEICHNUNG**

AttlistDecl – Handler für attlist-Deklarationen in der DTD (V2.0)

**ÜBERSICHT**

```
AttlistDecl(p, elname$, attname$, atttype$, dflt$, isrequired)
```

**BESCHREIBUNG**

Diese Funktion wird für attlist-Deklarationen in der DTD (Dokumenttypdefinition) aufgerufen. Es wird für jedes Attribut aufgerufen. Eine einzelne Attlist-Deklaration mit mehreren deklarierten Attributen generiert also mehrere Aufrufe an diesen Handler. Der Parameter `elname$` gibt den Namen des Elements zurück, für das das Attribut deklariert wird. Der Attributname befindet sich im Parameter `attname$`. Der Attributtyp befindet sich im Parameter `atttype$`. Es ist die Zeichenkette, die den Typ in der Deklaration darstellt, wobei Leerzeichen entfernt wurden.

Der Parameter `dflt$` enthält den Standardwert. Im Fall von #IMPLIED- oder #REQUIRED-Attributen ist er Null. Sie können diese beiden Fälle unterscheiden, indem Sie den Parameter `isrequired` überprüfen, der im Fall von #REQUIRED-Attributen `True` ist. Attribute, die `#FIXED` sind, haben ebenfalls ein `True isrequired`, aber sie haben den festen Wert ungleich Null im Parameter `dflt`.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `elname$` | Name des Elements, für das das Attribut deklariert wird |
| `attname$` | Attributname |
| `attype$` | Attributtyp |
| `dflt$` | Standardwert |
| `isrequired` | `True` oder `False`, je nachdem, ob das Attribut erforderlich ist |

**BEISPIEL**

```
Function p_AttlistDecl(p, elname$, attname$, atttype$, dflt$, isreq)
   DebugPrint(elname$, attname$, atttype$, dflt$, isreq)
EndFunction

p = xml.CreateParser({AttlistDecl = p_AttlistDecl})
p:Parse([[
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE lab_group [
   <!ELEMENT student_name (#PCDATA)>
   <!ATTLIST student_name student_no ID #REQUIRED>
   <!ATTLIST student_name tutor_1 IDREF #IMPLIED>
   <!ATTLIST student_name tutor_2 IDREF #IMPLIED>
```

```
]>
<root/>
]])
p:Free()
```

Der obige Code zeigt, wie Attlist-Deklarationen behandelt werden.

## 6.2 CharacterData

**BEZEICHNUNG**

CharacterData – Parser hat Zeichendaten gefunden

**ÜBERSICHT**

```
CharacterData(p, s$)
```

**BESCHREIBUNG**

Diese Funktion wird immer dann aufgerufen, wenn der Parser auf Daten stößt, die nicht Teil einer XML-Entität sind. Die Daten werden Ihrem Callback als Zeichenkette im Parameter `s$` übergeben.

Beachten Sie, dass die Zeichendaten für ein einzelnes XML-Element möglicherweise in mehreren separaten Blöcken an Ihren Callback übergeben werden. Stellen Sie also sicher, dass Ihr Code nicht davon abhängt, alle Zeichendaten auf einmal zu erhalten. Standardmäßig versucht das XML-Plugin, Zeichendaten in so wenig Textblöcken wie möglich zusammenzuführen. Sie können die Zusammenführungsfunktion deaktivieren, indem Sie `False` im optionalen dritten Argument in Ihrem Aufruf von xml.CreateParser() übergeben. Beachten Sie, dass selbst bei aktivierter Option zum Zusammenführen von Daten (was die Standardeinstellung ist) nicht garantiert ist, dass alle Daten auf einmal eingehen, seien Sie also darauf vorbereitet, dies korrekt zu handhaben.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `s$` | die Zeichendaten |

**BEISPIEL**

```
Function p_CharacterData(p, s$)
   DebugPrint(s$)
EndFunction

p = xml.CreateParser({CharacterData = p_CharacterData})
p:Parse([[<app>Hollywood</app>]])
p:Free()
```

Der obige Code gibt "Hollywood" aus.

## 6.3 Comment

**BEZEICHNUNG**

Comment – Parser hat einen Kommentar gefunden

**ÜBERSICHT**

```
Comment(p, s$)
```

**BESCHREIBUNG**

Diese Funktion wird immer dann aufgerufen, wenn der Parser auf einen Kommentar im XML stößt. Die Kommentardaten werden Ihrem Callback als Zeichenkette im Parameter `s$` übergeben.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `s$` | die Kommentardaten |

**BEISPIEL**

```
Function p_Comment(p, s$)
   DebugPrint(s$)
EndFunction

p = xml.CreateParser({Comment = p_Comment})
p:Parse([[
    <!--I'm a comment-->
    <app>Hollywood</app>
]])
p:Free()
```

Der obige Code gibt `"Ich bin ein Kommentar"` aus.

## 6.4 DefaultExpand

**BEZEICHNUNG**

DefaultExpand – Standard-Handler mit Erweiterungen

**ÜBERSICHT**

```
DefaultExpand(p, s$)
```

**BESCHREIBUNG**

Dies macht dasselbe wie DefaultHandler(), außer dass dieser Handler die Erweiterung interner Entitätsreferenzen nicht verhindert. Die Entitätsreferenz wird nicht an den Standard-Handler übergeben.

**PARAMETER**

| | |
|---|---|
| `p` | Standard-Handler für andere Zeichen |
| `s$` | die Zeichendaten |

## 6.5 DefaultHandler

**BEZEICHNUNG**

DefaultHandler – Standard-Handler für andere Zeichen

**ÜBERSICHT**

```
DefaultHandler(p, s$)
```

**BESCHREIBUNG**

Diese Funktion wird für alle Zeichen im Dokument aufgerufen, die sonst nicht verarbeitet würden. Dies umfasst sowohl Daten, für die keine Handler festgelegt werden können (wie einige Arten von DTD-Deklarationen), als auch Daten, die gemeldet werden könnten, für die jedoch derzeit kein Handler festgelegt ist. Die Zeichen werden genau so übergeben, wie sie im XML-Dokument vorhanden waren, außer dass sie in UTF-8 codiert werden. Liniengrenzen werden nicht normalisiert. Beachten Sie, dass ein Byte-Order-Markierungszeichen nicht an den Standard-Handler übergeben wird. Es gibt keine Garantie dafür, wie Zeichen zwischen Aufrufen des Standard-Handlers aufgeteilt werden: Beispielsweise kann ein Kommentar auf mehrere Aufrufe aufgeteilt werden. Das Festlegen der Handler mit diesem Aufruf hat den Nebeneffekt, dass die Erweiterung von Verweisen auf intern definierte allgemeine Entitäten deaktiviert wird. Stattdessen werden diese Verweise an den Standard-Handler übergeben.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `s$` | die Zeichendaten |

## 6.6 ElementDecl

**BEZEICHNUNG**

ElementDecl – Handler für Elementdeklarationen in der DTD (V2.0)

**ÜBERSICHT**

```
ElementDecl(p, name$, type, quantifier, children)
```

**BESCHREIBUNG**

Diese Funktion wird für Elementdeklarationen in einer DTD aufgerufen. Der Name des Elements wird im Parameter `name$` übergeben. Wenn `type` gleich `#XML_CTYPE_EMPTY` oder `#XML_CTYPE_ANY` ist, dann ist `quantifier` `#XML_CQUANT_NONE` und die Tabelle `children` ist Null.

Wenn `type` `#XML_CTYPE_MIXED` ist, dann ist `quantifier` `#XML_CQUANT_NONE` oder `#XML_CQUANT_REP` und die Tabelle `children` enthält die Elemente, die gemischt werden dürfen. Alle diese untergeordneten Elemente haben dann den Typ `#XML_CTYPE_NAME` ohne Quantifizierung. Nur der Wurzelknoten kann vom Typ `#XML_CTYPE_EMPTY`, `#XML_CTYPE_ANY` oder `#XML_CTYPE_MIXED` sein.

Für den Typ `#XML_CTYPE_NAME` enthält der Parameter `name$` den Namen und die Tabelle `children` ist Null. Der Parameter `quantifier` gibt alle Quantifier an, die dem Namen hinzugefügt wurden.

Die Typen `#XML_CTYPE_CHOICE` und `#XML_CTYPE_SEQ` geben jeweils eine Auswahl oder Sequenz an. Die Tabelle `children` enthält dann die Knoten in der Auswahl oder Sequenz.

Für die in der Tabelle `children` übergebenen Elemente werden die Felder `name`, `type`, `quantifier` und `children` gesetzt. Sie entsprechen den gleichnamigen Parametern, die an `ElementDecl()` übergeben werden.

Mögliche Werte für den Parameter/das Feld `type`:

```
#XML_CTYPE_EMPTY
#XML_CTYPE_ANY
#XML_CTYPE_MIXED
#XML_CTYPE_NAME
#XML_CTYPE_CHOICE
#XML_CTYPE_SEQ
```

Mögliche Werte für den Parameter/das Feld `quantifier`:

```
#XML_CQUANT_NONE
#XML_CQUANT_OPT
#XML_CQUANT_REP
#XML_CQUANT_PLUS
```

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Name des Elements |
| `type` | Elementtyp |
| `quantifier` | Elementquantifizierer |
| `children` | untergeordnete Knoten oder `Nil` |

**BEISPIEL**

```
Function p_ElementDecl(p, elname$, type, quant, children)
   DebugPrint(elname$, type, quant, children)
EndFunction

p = xml.CreateParser({ElementDecl = p_ElementDecl})
p:Parse([[
<!DOCTYPE student [
   <!ELEMENT student (id|surname)>
   <!ELEMENT id (#PCDATA)>
]>
<student>
   <id>9216735</id>
</student>
]])
p:Free()
```

Der obige Code zeigt, wie Attlist-Deklarationen behandelt werden.

## 6.7 EndCDataSection

**BEZEICHNUNG**

EndCDataSection – Ende des CDATA-Abschnitts wurde erreicht

**ÜBERSICHT**
`EndCDataSection(p)`

**BESCHREIBUNG**
Diese Funktion wird aufgerufen, wenn der Parser das Ende eines XML-CDATA-Abschnitts erkennt. Im Callback können Sie parser:GetPosition() verwenden, um die aktuelle Parser-Position zu erhalten.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |

## 6.8 EndDocTypeDecl

**BEZEICHNUNG**
EndDocTypeDecl – Parser hat das Ende einer DOCTYPE-Deklaration erreicht (V2.0)

**ÜBERSICHT**
`EndDocTypeDecl(p)`

**BESCHREIBUNG**
Diese Funktion wird aufgerufen, wenn der Parser das Ende der DOCTYPE-Deklaration nach dem schließenden > erkennt, jedoch nach der Verarbeitung einer externen Teilmenge.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |

## 6.9 EndElement

**BEZEICHNUNG**
EndElement – XML-Element wurde geschlossen

**ÜBERSICHT**
`EndElement(p, name$)`

**BESCHREIBUNG**
Diese Funktion wird immer dann aufgerufen, wenn ein XML-Element geschlossen wird. Der Name des XML-Elements wird in `name$` übergeben.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Name des XML-Elements |

**BEISPIEL**
Siehe Abschnitt 6.19 [StartElement], Seite 33.

## 6.10 EndNamespaceDecl

**BEZEICHNUNG**

EndNamespaceDecl – Parser hat das Ende einer Namensraum-Deklaration erkannt

**ÜBERSICHT**

```
EndNamespaceDecl(p, name$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Gültigkeitsbereich die durch `name$` angegebene Namensraum-Deklaration verlassen wird. Dieser Handler wird für jede Namensraum-Deklaration nach dem Handler für das End-Tag des Elements aufgerufen, in dem der Namensraum deklariert wurde.

Beachten Sie, dass Sie den optionalen Parameter `sep$` in Ihrem Aufruf von xml.CreateParser() angeben müssen, wenn Sie möchten, dass der Parser Namensräume verarbeitet.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Name des Namensraum |

## 6.11 EntityDecl

**BEZEICHNUNG**

EntityDecl – Parser hat eine Entität-Deklaration erkannt

**ÜBERSICHT**

```
EntityDecl(p, name$, isparm, value$, base$, sysid$, pubid$, notation$)
```

**BESCHREIBUNG**

Diese Funktion wird für Entität-Deklarationen aufgerufen. Das Argument `isparm` ist `True`, wenn die Entität eine Parameter-Entität ist, andernfalls `False`. Für interne Entitäten, z.B:

```
(<!ENTITY foo "bar">)
```

value$ ist ein Zeichenkette und `sysid$`, `pubid$` und `notation$` sind Null. Die Zeichenkette `Wert$` kann Null sein oder eine leere Zeichenkette, die ein gültiger Wert ist. Für externe Entitäten ist `value$` Null und `sysid$` eine Zeichenkette. Das Argument `pubid$` ist null, es sei denn, es wurde ein öffentlicher Identifikator angegeben. Das Argument `notation$` hat nur für nicht geparste Entität-Deklarationen einen Zeichenketten-Wert.

Der Parameter `base$` wird auf das gesetzt, was mit parser:SetBase() gesetzt wurde. Wenn dies nicht gesetzt wurde, ist er Null.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Entitätsname |
| `isparm` | `True`, wenn die Entität eine Parameterentität ist |
| `base$` | Basis, die für relative System IDs zu verwenden ist |

`sysid$` System ID

`pubid$` Öffentliche ID

`notation$`
Notationsname

## 6.12 ExternalEntityRef

**BEZEICHNUNG**

ExternalEntityRef – Parser hat eine externe Entitätsreferenz erkannt

**ÜBERSICHT**

```
ExternalEntityRef(p, subparser, base$, sysid$, pubid$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser eine externe Entitätsreferenz erkennt. Der Parameter `subparser` enthält einen Parser-Handler, der mit denselben Callbacks und demselben Kontext wie der Hauptparser erstellt wurde und zum Parsen der externen Entität verwendet werden sollte. Der Parameter `base$` ist die Basis für relative Systemidentifikatoren. Er wird von parser:SetBase() gesetzt und kann Null sein. Der Parameter `sysid$` ist der Systemidentifikator, der in der Entität-Deklaration angegeben ist, und ist niemals Null. Der Parameter `pubid$` ist die öffentliche ID, die in der Entität-Deklaration angegeben ist, und kann Null sein.

**PARAMETER**

`p` Parser-Handler

`subparser`
Subparser-Handler zum Parsen der externen Entität

`base$` Basis, die für relative System IDs zu verwenden ist

`sysid$` System ID

`pubid$` Öffentliche ID

## 6.13 NotationDecl

**BEZEICHNUNG**

NotationDecl – Parser hat eine Notationsdeklaration erkannt

**ÜBERSICHT**

```
NotationDecl(p, name$, base$, sysid$, pubid$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser eine XML-Notationsdeklaration erkennt. Der Notationsname wird im Parameter `name$` übergeben. Der Parameter `base$` ist die Basis für relative Systemidentifikatoren. Er wird von parser:SetBase() gesetzt und kann Null sein. Der Parameter `sysid$` ist der Systemidentifikator, der in der Entität-Deklaration angegeben ist, und ist niemals Null. Der Parameter `pubid$` ist die öffentliche ID, die in der Entität-Deklaration angegeben ist, und kann Null sein.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Notationsname |
| `base$` | Basis, die für relative System IDs zu verwenden ist |
| `sysid$` | System ID |
| `pubid$` | Öffentliche ID |

## 6.14 NotStandalone

**BEZEICHNUNG**

NotStandalone – Umgang mit nicht "standalone" Dokumenten

**ÜBERSICHT**

```
ok = NotStandalone(p)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser erkennt, dass das Dokument nicht "standalone" ist. Dies geschieht, wenn eine externe Teilmenge oder ein Verweis auf eine Parameterentität vorhanden ist, das Dokument jedoch in einer XML-Deklaration nicht auf "yes" gesetzt ist. Dieser Callback erwartet einen Rückgabewert. Wenn der Callback `False` zurückgibt, wird das Parsing abgebrochen.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |

**RÜCKGABEWERTE**

| | |
|---|---|
| `ok` | gibt `True` zurück, um mit dem Parsen fortzufahren, und `False`, um abzubrechen |

## 6.15 ProcessingInstruction

**BEZEICHNUNG**

ProcessingInstruction – Parser verarbeitet eine Anweisung

**ÜBERSICHT**

```
ProcessingInstruction(p, target$, data$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser XML-Verarbeitungsanweisungen erkennt. Das `target$` ist das erste Wort in der Verarbeitungsanweisung. Die `data$` sind die restlichen Zeichen darin, nachdem alle Leerzeichen nach dem Anfangswort übersprungen wurden.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `target$` | erstes Wort in der Verarbeitungsanweisung |

`data$` Rest der Zeichen nach dem Anfangswort

**BEISPIEL**

```
Function p_ProcessingInstruction(p, target$, data$)
   DebugPrint(target$, data$)
EndFunction

p = xml.CreateParser({ProcessingInstruction = p_ProcessingInstruction})
p:Parse([[<foo><?Hollywood rocks?></foo>]])
p:Free()
```

Der obige Code gibt "Hollywood rocks" aus.

## 6.16 SkippedEntity

**BEZEICHNUNG**

SkippedEntity – Parser hat eine Entität übersprungen (V2.0)

**ÜBERSICHT**

```
SkippedEntity(p, name$, isparm)
```

**BESCHREIBUNG**

Diese Funktion wird in zwei Situationen aufgerufen:

1. Es wird eine Entitätsreferenz gefunden, für die keine Deklaration gelesen wurde, und dies ist kein Fehler.
2. Eine interne Entitätsreferenz wird gelesen, aber nicht expandiert, weil DefaultHandler() aufgerufen wurde.

Das Argument `isparm` ist `True` für eine Parameterentität und `False` für eine allgemeine Entität.

Hinweis: Übersprungene Parameterentitäten in Deklarationen und übersprungene allgemeine Entitäten in Attributwerten können nicht gemeldet werden, da das Ereignis nicht mit der Meldung der Deklarationen oder Attributwerte synchron wäre.

**PARAMETER**

`p` Parser-Handler

`name$` Entitätsname

`isparm` `True` für Parameterentitäten, `False` für allgemeine Entitäten

## 6.17 StartCDataSection

**BEZEICHNUNG**

StartCDataSection – CDATA-Abschnitt wird gleich geparst

**ÜBERSICHT**

```
StartCDataSection(p)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser den Beginn eines XML-CDATA-Abschnitts erkennt. Im Callback können Sie parser:GetPosition() verwenden, um die aktuelle Parser-Position zu erhalten.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |

## 6.18 StartDocTypeDecl

**BEZEICHNUNG**

StartDocTypeDecl – Parser hat die DOCTYPE-Deklaration erreicht

**ÜBERSICHT**

```
StartDocTypeDecl(p, name$, sysid$, pubid$, subset)
```

**BESCHREIBUNG**

Diese Funktion wird zu Beginn einer DOCTYPE-Deklaration aufgerufen, bevor eine externe oder interne Teilmenge geparst wird. Der Callback erhält den DOCTYPE-Namen in `name$`, die System-ID in `sysid$`, die öffentliche ID in `pubid$`. Der Parameter `subset` ist `True`, wenn die DOCTYPE-Deklaration eine interne Teilmenge hat.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | DOCTYPE Name |
| `sysid$` | System ID |
| `pubid$` | Öffentlich ID |
| `subset` | `True`, wenn die DOCTYPE-Deklaration eine interne Teilmenge hat |

## 6.19 StartElement

**BEZEICHNUNG**

StartElement – ein XML-Element wurde gefunden

**ÜBERSICHT**

```
StartElement(p, name$, attrs)
```

**BESCHREIBUNG**

Diese Funktion wird immer dann aufgerufen, wenn ein neues XML-Element geöffnet wird. Der Name des XML-Elements wird in `name$` übergeben. Der Parameter `attrs` ist eine Tabelle mit allen Elementattributnamen und -werten. Die Tabelle enthält einen Eintrag für jedes Attribut im Start-Tag des Elements. Der Name des Attributs wird als Tabellenindex verwendet.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |

| | |
|---|---|
| `name$` | Name des XML-Elements |
| `attrs` | Tabelle mit allen Elementattributen |

**BEISPIEL**

```
Function p_StartElement(p, name$, attrs)
   DebugPrint("Open:", name$, attrs.name, attrs.author)
EndFunction
Function p_EndElement(p, name$)
   DebugPrint("Close:", name$)
EndFunction

p = xml.CreateParser({StartElement = p_StartElement,
  EndElement = p_EndElement})
p:Parse([[<plugin name="XML" author="Andreas Falkenhahn"/>]])
p:Free()
```

Der obige Code gibt `"Open: plugin XML Andreas Falkenhahn"` und dann `"Close: plugin"` aus.

## 6.20 StartNamespaceDecl

**BEZEICHNUNG**

StartNamespaceDecl – Parser hat eine Namensraum-Deklaration erkannt

**ÜBERSICHT**

```
StartNamespaceDecl(p, name$, uri$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn ein Namensraum deklariert wird. Der Callback erhält den Namen und die URI des Namensraums in den Parametern `name$` und `uri$`. Beachten Sie, dass obwohl Namensraum-Deklarationen innerhalb von Start-Tags auftreten, der Start-Handler für Namensraum-Deklarationen vor dem Start-Tag-Handler für jeden in diesem Start-Tag deklarierten Namensraum aufgerufen wird.

Beachten Sie auch, dass Sie den optionalen Parameter `sep$` in Ihrem Aufruf von xml.CreateParser() angeben müssen, wenn Sie möchten, dass der Parser Namensräume verarbeitet.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Name des Namensraum |
| `uri$` | Namensraum-URI |

**BEISPIEL**

```
Function p_StartNamespaceDecl(p, name$, uri$)
   DebugPrint(name$, uri$)
EndFunction

p = xml.CreateParser({StartNamespaceDecl = p_StartNamespaceDecl}, "?")
```

```
p:Parse([[<foo xmlns:space='a/namespace'/>]])
p:Free()
```

Der obige Code gibt "space a/namespace" aus.

## 6.21 UnparsedEntityDecl

**BEZEICHNUNG**

UnparsedEntityDecl – der Parser hat eine nicht geparste Entität-Deklaration erkannt (veraltet)

**ÜBERSICHT**

```
UnparsedEntityDecl(p, name$, base$, sysid$, pubid$, notation$)
```

**BESCHREIBUNG**

Diese Funktion wird aufgerufen, wenn der Parser Deklarationen von nicht analysierten Entitäten empfängt. Dies sind Entitätsdeklarationen, die ein Notationsfeld (NDATA) haben. Als Beispiel

```
<!ENTITY logo SYSTEM "images/logo.gif" NDATA gif>
```

wäre der Parameter `name$` "logo", `sysid$` wäre "images/logo.gif" und `notation$` wäre "gif". Für dieses Beispiel wäre der Parameter `pubid$` gleich Null. Der Parameter `base$` wäre das, was mit parser:SetBase() gesetzt wurde. Wenn nichts gesetzt wurde, wäre es `Nil`.

Beachten Sie, dass `UnparsedEntityDecl()` veraltet ist und Sie stattdessen EntityDecl() verwenden sollten.

**PARAMETER**

| | |
|---|---|
| `p` | Parser-Handler |
| `name$` | Entitätsname |
| `base$` | Basis, die für relative System IDs zu verwenden ist |
| `sysid$` | System ID |
| `pubid$` | Öffentliche ID |
| `notation$` | Notationsname |

## 6.22 XMLDecl

**BEZEICHNUNG**

XMLDecl – Parser hat eine XML-Deklaration erreicht (V2.0)

**ÜBERSICHT**

```
XMLDecl(p, version$, encoding$, standalone)
```

**BESCHREIBUNG**

Diese Funktion wird für XML-Deklarationen und auch für Textdeklarationen aufgerufen, die in externen Entitäten entdeckt werden. Der Unterschied besteht darin, dass der

Parameter `version$` für Textdeklarationen null ist. Der Parameter `encoding$` kann für eine XML-Deklaration Null sein. Das Argument `standalone` enthält -1, 0 oder 1, was jeweils anzeigt, dass es keinen eigenständigen Parameter in der Deklaration gab, dass er als "no" oder als "yes" angegeben wurde.

**PARAMETER**

`p` Parser-Handler

`version$` Version in XML-Deklaration

`encoding$`
Kodierung in XML-Deklaration

`standalone`
eigenständiger Zustand der Deklaration, kann 0, -1 oder 1 sein (siehe oben)

**BEISPIEL**

```
Function p_XMLDecl(p, version$, encoding$, standalone)
   DebugPrint(version$, encoding$, standalone)
EndFunction

p = xml.CreateParser({XMLDecl = p_XMLDecl})
p:Parse([[<?xml version="1.0" encoding="ISO-8859-1"?><body/>]])
p:Free()
```

Der obige Code gibt "1.0 ISO-8859-1 -1" aus.

# Anhang A Lizenzen

## A.1 Expat license



## A.2 LuaExpat license

LuaExpat is free software: it can be used for both academic and commercial purposes at absolutely no cost. There are no royalties or GNU-like "copyleft" restrictions. LuaExpat qualifies as Open Source software. Its licenses are compatible with GPL. LuaExpat is not in the public domain and the Kepler Project keep its copyright. The legal details are below.

The spirit of the license is that you are free to use LuaExpat for any purpose at no cost without having to ask us. The only requirement is that if you do use LuaExpat, then you should give us credit by including the appropriate copyright notice somewhere in your product or its documentation.

The LuaExpat library is designed and implemented by Roberto Ierusalimschy. The implementation is not derived from licensed software.

# Index