# SHORE

## 1.1.0

# Das Parser-Kochbuch

### Dokumentversion 0.2

Klaus Mayr, Tammo Schnieder

sd&m AG
software design & management
Thomas-Dehler-Strasse 27
81773 München
(089) 6 38 12-0
(089) 6 38 12-150

www.sdm.de

SHORE Parser-Kochbuch

Dieses Handbuch wird vom sd&m-Werkzeugteam gepflegt (mailto: V_P_SHORE@muc.sdm.de)



Das Verfahren der Verarbeitung von SHORE ist von der sd&m AG zum Patent angemeldet.

Der Name SHORE ist geschützt.

Java ist eine Marke von Sun Microsystems, Inc, USA.

Microsoft, Windows und Windows NT sind Marken oder eingetragene Marken der Microsoft Corporation in den USA und/oder anderen Ländern.

ObjectStore ist eine eingetragene Marke der eXcelon Corporation.

verity, search97 und Information Server sind Marken von verity, Inc., USA und können in verschiedenen Staaten eingetragen sein.

Das LIM-Readme ist versehen mit Copyright 1997 Digital Equipment Corporation. Ansonsten erscheinen keine rechtlichen Hinweise.

Andere in diesem Dokument aufgeführte Produkt- oder Firmennamen sind möglicherweise Marken der jeweiligen Eigentümer.

# Historie

| Version | Status | Datum | Autor(en) | Erläuterung |
|---|---|---|---|---|
| 0.2 | in Arbeit | 24.10.2001 | Tammo Schnieder | Konsolidierung Frame-maker- und Word-Teile |
| 0.1 | in Arbeit | 14.08.2001 | Klaus Mayr, Tammo Schnieder | Initiale Version für SHORE 1.1 |

# Inhaltsverzeichnis

# 1 Einleitung

An wen richtet sich das Handbuch?

Dieses Handbuch richtet sich an Projektmitarbeiter, die für eine SHORE-Instanz Parser auf Grundlage von Pattern-Matching neu erstellen oder erweitern wollen. Sie kennen SHORE und wissen, wie Ihre Projektdokumente strukturiert sind.

Hier finden Sie "Kochrezepte" aus denen Sie sich Ihre eigene Parserumgebung kochen können, um Ihre Projektdokumente nach SHORE zu bringen. Die Installation und Einrichtung von SHORE ist nicht Bestandteil dieses Handbuches, dafür existiert ein eigenes Administratorhandbuch.

Wie kann ich das Handbuch verwenden?

Jeder geht auf seine ihm eigene Art und Weise mit einem Handbuch um. Abhängig von Ihren Vorkenntnissen und Ihrem Informationsbedürfnis können Sie dieses Handbuch auf verschiedene Weise nutzen. Um Ihnen den Zugang zu erleichtern, stellen wir Ihnen kurz vor, was wir uns bei der Strukturierung dieses Handbuchs gedacht haben.

# 2 Übersicht

## 2.1 Ihre Anforderungen

Sie haben folgende Anforderungen:

- Sie wollen Ihre Dokumente und Sourcecode im Browser sehen und darin navigieren.
- Sie wollen zusätzliche Zusammenhänge erkennen, die Sie nicht durch einfaches ansehen bekommen.
- Wenn Sie Ihre Dokumente ständig aktualisieren und diese in einem KM-System ablegen, soll auch SHORE ständig auf dem aktuellsten Stand sein.
- Neben Ascii-Quelltexten wollen Sie auch das analysieren, was in Tools wie Rose, Word oder Excel abgelegt ist.
- Dabei wollen Sie Informationen, aus mehreren Quellen zusammenführen und Dokument-übergreifend Querverweise einrichten
- Sie wünschen Genauigkeit bei der Analyse syntaktischer Elemente und zum Beispiel Strings in Kommentaren und Kommentare in Strings nicht als solche erkennen.
- Sie wünschen Fehlertoleranz bei der Analyse syntaktischer Elemente. Insbesondere benötigen Sie Pattern-Matching über Zeilenumbrüche hinweg.
- Sie benötigen ferner auch Fehlertoleranz bei der Zuordnung dieser Elementen zueinander. Insbesondere möchten Sie bei der Verwendung von fachlichen Begriffen, Klassen- oder Methodennamen auch dann eine Zuordnung zu ihren Definitionsstellen vornehmen, wenn die entsprechenden Namen bezüglich Groß/Klein-, Zusammen- oder Getrenntschreibung, Umlauten oder Bindestrichen voneinander abweichen.
- Ideal wäre auch, wenn Spezifikation von Metamodell und Syntaxanalyse in einem möglich wären.

Wenn Sie einen oder mehrere Punkte erfüllt haben möchten, lesen Sie in diesem Buch, welche Lösungen SHORE Ihnen bietet.

# 2.2 SHORE-Eigenschaften

SHORE ist ein Repository für Projektdokumente beliebigen Typs. Es verknüpft in einem Hypertextsystem die unterschiedlichsten Dokumente mit den unterschiedlichsten Datenformaten und Informationsinhalten

Um Ihre Dokumente an SHORE zu übergeben, müssen diese durch spezielle Parser nach XML konvertiert werden. In der XML-Struktur, die die Parser liefern, sind Objekte und Beziehungen zwischen diesen Objekten enthalten, die von den Parsern erkannt wurden.

SHORE speichert die importierten Projektdokumente und es speichert daneben in einer Datenbank auch die Objekte und Beziehungen, die von den Parsern gefunden wurden

## 2.2.1 Was bietet SHORE?

SHORE gibt Ihnen Übersicht über alle Ihre Projektdokumente. Sie können mit einem Browsers in diesen Dokumenten surfen. Darüberhinaus können Sie Anfragen an die Datenbank formulieren und komplexe Auswertungen fahren. Die Anfragen sind über das Anklicken von Hyperlinks im Hypertext ausführbar.

Die einfachsten Anfragen sind Navigationsanfragen, über die Sie die Beziehungen, die von einem Objekt ausgehen, direkt verfolgen. Über komplexere Anfragen extrahieren (und visualisieren) Sie einen Teilgraphen der Beziehungen, die in der SHORE-Datenbank gespeichert sind.

Anfragen, die Projekte häufig benötigen, sind:

- wie sehen unsere Aufrufhierarchien aus?
- wie sehen unsere Vererbungshierarchien aus?
- wo gibt es nicht verwendete Objekte?
- wo gibt es Objekte, die mehrfach definiert sind?
- wie sieht die Nachbarschaft eines Objekts aus?

## 2.2.2 Was erwartet SHORE?

SHORE erwartet XML-Dokumente. Außerdem benötigt es ein Metamodell, in dem die Typen der Objekte und Beziehungen festgelegt werden, die in diesen Dokumenten enthaltenen sind. Auf die Struktur der Metamodelle und XML-Dokumente wird im folgenden genauer eingegangen

### Wie müssen SHORE-Metamodelle aussehen?

Zunächst ein kleines Beispiel eines Metamodells:

```
Dokumenttyp Quelltext


Dokumenttyp ProgrammQuelltext
  ist ein Quelltext


Dokumenttyp UseCaseBeschreibung
```

```
Objekttyp Programm
   ist definiert in ProgrammQuelltext


Objekttyp UseCase
   ist definiert in UseCaseBeschreibung


Beziehungstyp Quelltext_inkludiert_Quelltext alias inkludiert
    von 0 bis * Quelltext
    nach 0 bis * Quelltext
    ist definiert in Quelltext


Beziehungstyp Programm_implementiert_UseCase alias implementiert
   von 0 bis * Programm
   nach 0 bis * UseCase
   ist definiert in Quelltext
```

Das Metamodell ist frei konfigurierbar und erweiterbar und nicht durch SHORE fest vorgegeben. Sie schreiben es selbst und importieren es von einem SHORE Client aus mit dem Kommando

```
shore_imp_mm my.meta
```

Damit kann Ihr SHORE-System dynamisch mit Ihren Anforderungen wachsen.

Im Metamodell können Sie für Ihre Dokumenttypen Objekttypen und Beziehungstypen sowie Vererbungs-Beziehungen definieren:

Vererbung bei Objekttypen oder Dokumenttypen

```
Typ1 ist ein Typ2
```

Vererbung bei Beziehungstypen

```
Typ1 ist Teilmenge von Typ2
```

So können Sie leicht Auswertungen bekommen - sowohl auf spezielle als auch auf allgemeine Typen.

## Wie müssen SHORE-XML-Dokumente aussehen?

XML trennt Inhalt, Struktur und Layout. Inhalt ist Text, der zwischen den Tags steht. Die Struktur wird durch die Tags (`Objekttyp`) (`Beziehungstyp`) und deren Attribute (`Name`) (`Quelltyp Quellname Zieltyp Zielname`) festgelegt. Das Layout ergibt sich über eine durch Stylesheets konfigurierbare Generierung von Html-Seiten.

Der Text der zwischen den Tags steht, ist mit dem Text der geparsten Dokumente identisch. Die Tagstruktur wird von den Parsern ergänzt: sie generieren Markup für Dokumente, Objekte und Beziehungen.

Markup für Dokumente:

```
<Quelltext Name ="q"> .... text ...</Quelltext>
```

Markup für Objekte:

```
<Programm Name="p"> .... text ...</Programm Name>
```

Markup für Beziehungen:

```
<Programm_implementiert_UseCase
   Quelltyp="Programm"
   Quellname="p"
   Zieltyp="UseCase"
   Zielname="u"> ... text ....
</ Programm_implementiert_UseCase >
```

Ein weiteres Markup-Element sind Hotspots:

```
<Hotspots> ... text ...</Hotspot>
```

Hotspots markieren für den SHORE-Browser die Stellen, an denen man das umfassende Objekt anklicken kann. Jedes Objekt und jede Beziehung dürfen höchstens einen Hotspot enthalten.

Sollen mehrere ineinandergeschachtelte Objekte denselben Hotspot haben, so ist bei diesen umfassenden Objekten das Attribut Hotspot="multiple" zu setzen.

Wenn man auf den Hotspot einer Beziehung klickt, navigiert SHORE zum Zielobjekt der Beziehung. Wenn man auf den Hotspot eines Objekts klickt, zeigt SHORE Ihnen die ein und ausgehenden Beziehungen zu diesem Objekt.

Hotspots von Beziehungen sind (wie Hyperlinks) im Browser blau markiert. Hotspots von Objekten dagegen rot.

Alle XML-Dokumente müssen mit einem XML-Header beginnen. So sieht dieser XML-Header aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

# 2.3 Ihr Lösungsansatz mit SHORE

Es ist Ihr Ziel, mit SHORE Wissen über ein System (oder sonstige komplexe Strukturen) zu gewinnen.

In den folgenden Abschnitten zeigen wir Ihnen, wie Sie Ihre Dokumente für SHORE idealerweise aufbereiten. Weiterhin erfahren Sie, wie Sie einen Pattern-Matcher selbst erstellen und nach Bedarf weiterentwickeln können.

Danach sollten Sie ein Gefühl besitzen, was Sie eine solche Parserentwicklung kostet bzw. welcher Aufwand darin steckt.

## 2.3.1 So erstellen Sie einen Pattern-Matcher

Der Bau eines Pattern-Matchers bedeutet, dass Sie sich ein neues SHORE-System aufbauen oder ein bestehendes System erweitern. Der Weg folgt einem Muster und ist iterativ/zyklisch.

Wenn Sie sich entschieden haben, einen neuen Pattern-Matcher einzusetzen, fangen Sie klein an, erkennen Sie ersteinmal wenig und steigern Sie dann die Fähigkeiten Ihres Pattern-Matchers.

Dies sind die Schritte auf dem Weg zu einem Pattern-Matcher:

1. Sie überlegen sich, welche Aussage Sie erwarten oder welche Fragestellung sie beantworten möchten. Werden Sie sich darüber klar, welche Objekte und Beziehungen dafür interessant sind und wie man diese erkennt.
2. Sie prüfen, ob Sie dafür alle notwendigen Informationen besitzen.
3. Wenn nicht, definieren Sie, welche Informationen Sie für SHORE benötigen und in welchen Projektdokumenten diese Informationen enthalten sind.
4. Sie definieren die Struktur der Informationen in Form des Metamodells
5. Sie schreiben oder erweitern einen passenden Parser, der die Informationen in strukturierten Dokumenten erkennt und für SHORE aufbereitet (in XML)
6. Sie importieren die Projektdokumente nach SHORE
7. Sie navigieren in SHORE und stellen Anfragen. Gegebenenfalls formulieren Sie neue Anfragen.
8. Dabei entstehen eventuell weitere Fragestellungen, die Sie beantwortet wissen möchten und der Kreis schließt sich.

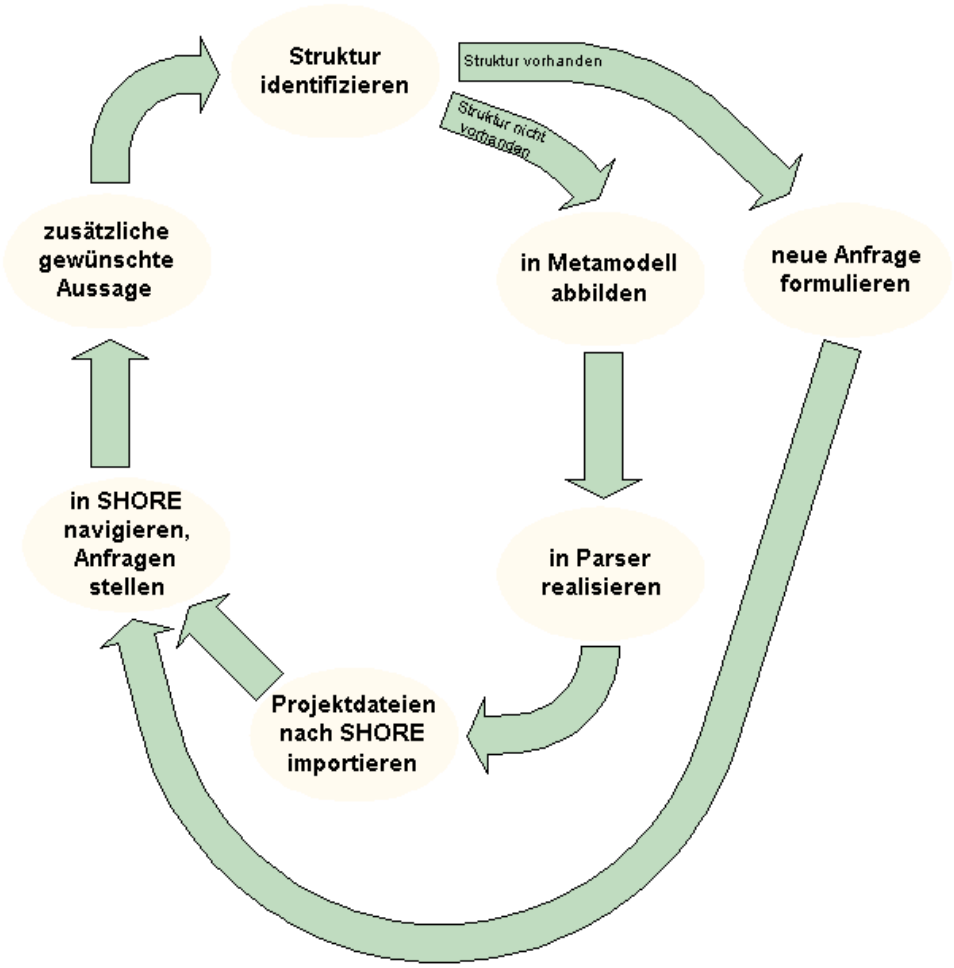



**Abbildung 1:** Entwicklungszyklus einer SHORE-Instanz

## Was Sie bereits nutzen können

### Parser für Programmiersprachen

Gerade bei Quelltexten kann die zur Berechnung von Querbeziehungen (Verwendungsnachweisen von Symbolen) erforderliche semantische Analyse beliebig schwierig sein. Sie in selbst zu entwickeln, würden wir daher den wenigsten unserer SW-Projekte empfehlen. Die Aufwände hierfür sind zu hoch. Das sTm Werkzeugteam ist hier in Vorleistung gegangen und hat für einige wichtige Programmiersprachen solche Parser selbst entwickelt.

Derzeit stehen solche Parser für Java, Cobol, VisualBasic- und C-Quelltexte fertig einsetzbar zur Verfügung. Die ersten drei werden automatisch mit installiert, wenn Sie SHORE auf NT installiert haben. Der C-Parser läuft unter UNIX und verlangt zur Installation eine Anpassung der Makefiles.

### Pattern-Matcher für Programmiersprachen

Bei Quelltexten exotischer Programmiersprachen lohnt sich eine auf einer sauberen syntaktischen und semantischen Analyse basierende Parserentwicklung oft nicht. Wenn man Namenskonflikte schon durch Präfixing (Voranstellen der Namen des umfassenden Namensraumes) weitgehend elimninieren kann, dann genügt in den meisten Fällen auch ein einfacherer Ansatz: das Parsen auf Grundlage von Mustern (Pattern-Matching).

Mit diesem Ansatz konnten die Quellen einiger exotischer Programmiersprachen (RPG3, Natural usw.) erfolgreich bearbeitet werden. Für die meisten unserer Projekte, die mit individuellen Umgebungen dürfte dieser Ansatz eine sehr gute Wahl sein. Mit relativ wenig Aufwand erzielt man immer noch eine Qualität, die sich sehen lassen kann.

### Skriptumgebung zum Import von Projektdokumenten.

Wie wir im Kapitel “Import nach SHORE” auf Seite 66 noch sehen werden, ist es mit dem Erzeugen von XML-Dokumenten nicht getan. Sie sollten Ihre Quelltexte möglichst in SHORE integrierten Parsern zur Verfügung stellen. Damit können Sie die generierten XML-Dokumente automatisch nach SHORE importieren. Die SHORE-Installation stellt Ihnen eine Skriptumgebung bereit, mit der Sie

- Parser über eine einheitliche Schnittstelle ansteuern,
- einen inkrementellen Import mit berücksichtigten Abhängigkeiten durchführen können,
- Teil- oder Vollimporte durchführen können,
- Abhängig von Dateinamen und -endungen unterschiedliche Parser aufrufen können (Auswahl mit Hilfe regulärer Ausdrücke)
- neue Parser leicht hinzuzufügen

Um Ihre eigenen Parser oder Pattern-Matcher in diese Skriptumgebung zu integrieren, brauchen Sie nur einige wenige Schnittstellen zu erweitern.

# 2.4 Grundlagen: Struktur, Inhalt und Layout

Für SHORE sollten sie gedanklich immer Struktur, Inhalt und Layout trennen.

Inhalt

Der Inhalt, den Sie betrachten wollen, sind die Projektdokumente, die in SHORE enthalten sind. Also alle Informationen, aus denen Sie sich Ihr Systemverständnis zusammenstellen und anhand derer Sie Aussagen über das System ableiten können. Wenn Ihnen Informationen fehlen, müssen Sie überlegen, wie Sie diese zu SHORE hinzufügen. Sehen Sie sich beispielsweise die Middlewarebeschreibung im Taschenrechnerbeispiel an (siehe im SHORE-Anwenderhandbuch). Solche recht einfach strukturierten Dokumente, die sie leicht per Hand schreiben können, dienen hier als "Bindemittel". Auch auf diese Weise können Sie mit relativ geringem Aufwand zusätzliche Informationen einfügen.

Struktur

Die Struktur sind die Objekte und Beziehungen in Ihren Projektdokumenten, die die Parser erkennen. Die Struktur wird durch ein Metamodell definiert. Je nachdem wie Sie verschiedene Aspekte modellieren, erhalten Sie später Aussagen über Ihr System. In SHORE ist es recht einfach, erstmal mit einer einfachen Struktur zu beginnen und Sie später zu verfeinern.

Layout

Die nach SHORE importierten Dokumente können unterschiedlich dargestellt werden. Dafür nutzen Sie Cascading Stylesheets, die pro Dokumenttyp unterschiedlich sein können. Dafür können Sie durch die Parser "nachwürzen" indem Sie zusätzliche Strukturen erkennen, für die Sie dann in den Stylesheets die entsprechende Geschmacksnote festlegen.

## 2.4.1 Metamodell

Im Metamodell definieren Sie, welche Projektdokumente es gibt, und welche Strukturen darin enthalten sein können (Objekte und Beziehungen). SHORE erkennt beim Import von Dateien die so definierten Strukturen anhand von entsprechend benannten XML-Tags.

### Metamodell - kurzes Beispiel

Zunächst ein kleines Beispiel eines Metamodells:

```
Dokumenttyp Quelltext

Dokumenttyp UseCaseBeschreibung

Objekttyp Programm
   ist definiert in ProgrammSource

Objekttyp UseCase
   ist definiert in UseCaseBeschreibung

Beziehungstyp Quelltext_inkludiert_Quelltext alias inkludiert
    von 0 bis * Quelltext
    nach 0 bis * Quelltext
    ist definiert in Quelltext
```

```
Beziehungstyp Programm_implementiert_UseCase alias implementiert
   von 0 bis * Programm
   nach 0 bis * UseCase
   ist definiert in Quelltext
```

In diesem Beispiel haben wir zwei Dokumenttypen definiert, `Quelltext` und `UseCaseBeschreibung`. Unter `Quelltext` verstehen wir zunächst Dateien in denen irgendwelche `Programme` enthalten sind. In Dokumenten des Typs `UseCaseBeschreibung` sind `UseCases` beschrieben. Zunächst interessieren uns hier zwei Informationen, für die wir Beziehungstypen definiert haben:

1. Welche Quelltexte enthalten welche Quelltexte?
2. Welche Programme implementieren welche UseCases?

## 2.4.2 XML-Auszeichnungen

Dateien, die in SHORE importiert sind, liegen intern als XML vor. Ziel eines Parsers ist es, XML zu erzeugen, dass SHORE-konform ist. SHORE-konform bedeutet, dass die XML-Tags

- den Deklarationen des Metamodells entsprechen,
- Attribute besitzen und natürlich
- wohlgeformt sind.

Ein `<Hotspot>` markiert einen Bereich, den Sie im Browser später als Hyperlink anklicken können. <Hotspot>

Jedes in XML umgewandelte Projektdokument beginnt mit einer einleitenden Zeile, die es als xml-Dokument klassifiziert. Direkt danach folgt ein Dokumenttag in das der gesamte Dokumentinhalt eingeschlossen ist. Dieses Dokumenttag entspricht einem Eintrag des Metamodells. <Dokument>

Das XML-Tag im konkreten Dokument bekommt als Attribut den Namen des Dokuments. Dieser Name muss pro SHORE-Instanz eindeutig sein.

**Syntax**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<DokumenttypName Name="Dokumentname">
...
</DokumenttypName>
```

**Beispiel**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PLIQuelltext Name="printReport">
...
</PLIQuelltext>
```

Innerhalb eines Dokumentes können Objekte existieren. Ein Objekt schließt einen Bereich ein, der eine bestimmte Bedeutung besitzt. Ein Objekt muss immer einen Hotspot besitzen (stimmt das?##). <Objekt>

Die XML-Auszeichnung sieht so aus:

**Syntax** ...

```
<ObjekttypName Name="Objektname">
...<Hotspot>...</Hotspot>...
</ObjekttypName>...
```

**Beispiel**

```
...
<PLIProgramm Name="PRINTLIST"><Hotspot>PRINTLIST</Hotspot>: proce-
dure(parm) options(main);
...
end PRINTLIST;
</PLIProgramm>...
```

<Beziehung>

Eine Beziehung verknüpft Objekte/Dokumente miteinander. Beziehungen können ein Hotspot haben - müssen aber nicht. Beziehungen ohne Hotspot sind 'implizit', Beziehungen mit Hotspot heißen 'explizit'. Beispielsweise ist eine Verwendungsbeziehung explizit: "wer verwendet was <Hotspot>wo</Hotspot>" und eine strukturelle Beziehungen implizit: "wer gehört zu wem".

**Syntax** ...

```
<BeziehungstypName Quelltyp="Typ" Quellname="Name" Zieltyp="Typ"
Zielname="Name">
...<Hotspot>...</Hotspot>...
</BeziehungstypName>...
```

**Beispiel** (hübsch formatiert. Das ganze muss normalerweise in einer Zeile stehen)

```
...
<Block_verwendet_Variable
   Quelltyp="PLIProgramm"
   Quellname="PRINTLIST"
   Zieltyp="Variable"
   Zielname="PRINTLIST.score">
  <Hotspot>score</Hotspot>
</Block_verwendet_Variable> = 0.
...
</PLIProgramm>...
```

## Nicht extra programmiert werden müssen Beziehungen zu verschachtelten Elementen, da sie implizit verknüpft sind (Ausnahmen?)

Navigationsrichtung

Die Beziehungsrichtung für das auszuzeichnende Element läuft immer zum Zielobjekt. Wenn die Navigation andersherum sein soll, können Sie als zusätzliches Attribut `Richtung="Quelle"` angeben. Dass heißt, dass der Hotspot uin diesem Fall das Quellelement der Beziehung umschließt. Beispiel: Sie haben eine Variable eines bestimmten Typs. Dummerweise heißt die Beziehung im Metamodell `Datenelement_hat_Typ`. Der Hotspot würde also einen Typen umschließen. Kein Problem: ...

```
<Datenelement_hat_Typ
   Quelltyp="Variable"
   Quellname="score"
   Zieltyp="Typ"
   Zielname="integer"
   Richtung="Quelle">
  <Hotspot>score</Hotspot>
</Datenelement_hat_Typ>
...
```

multipler Hotspot

Ineinander geschachtelte Objekte oder Beziehungen dürfen sich einen Hotspot teilen, wenn bei ihnen das Attribut Hotspot="Multiple" gesetzt ist.
Beispiel: im Programm ABRE_Load wird die Variablendeklaration

```
  BOOLEAN ($ok;$match)
```

in XML wie folgt ausgezeichnet (zur besseren Übersichtlichkeit ist diese Zeile hübsch dargestellt. Normalerweise ist dies eine Zeile):

```
<Datenelement_hat_Typ
   Quelltyp="LokaleVariable"
   Quellname="ABRE_Load.$ok"
   Zieltyp="Typ"
   Zielname="BOOLEAN"
   Hotspot="multiple">
   <Datenelement_hat_Typ
      Quelltyp="LokaleVariable"
      Quellname="ABRE_Load.$match"
      Zieltyp="Typ"
      Zielname="BOOLEAN"
      Hotspot="multiple">
         <Hotspot>BOOLEAN</Hotspot>
   </Datenelement_hat_Typ>
</Datenelement_hat_Typ>
(<LokaleVariable Name="ABRE_Load.$ok">
   <Hotspot>$ok</Hotspot>
 </LokaleVariable>;
 <LokaleVariable Name="ABRE_Load.$match">
   <Hotspot>$match</Hotspot>
 </LokaleVariable>)
```

## 2.4.3 Musterlösungen zum Modellieren von Strukturen

aus mail von Ralf bayer, Abbildung von Sequenzen (z.B. Anfrage auf n-tes Element eines Arrays),

# 2.5 Welcher Parseransatz eignet sich für mein Projekt?

Eine laufende SHORE-Instanz verknüpft in der Regel Dokumente unterschiedlicher Typen miteinander. Pro Dokumenttyp gibt es typischerweise einen eigenen Parser, dessen Architektur der Strukturierung des Dokumenttyps angepasst ist. Je nach Struktur eines Dokumenttyps ist es sinnvoll einen bestimmten Parseransatz auszuwählen.

Entscheidungsfindung

Folgende Möglichkeiten gibt es:

- vollständiger Parser (Compilerbautools)
- PatternMatcher (LIM, PAM, eigenes Programm)
- direkt XML schreiben oder umwandeln (XSLT)

Im folgenden wollen wir Kriterien sammeln und diskutieren, die zu der Entscheidung des geeigneten Parseransatzes führt.

| | kompletter Parser | LIM | PAM | eigenes Programm | XSLT |
|---|---|---|---|---|---|
| **Eignung für Dokumenttypen** | | | | | |
| formal strukturiert ohne Namensräume | ++ | ++ | ++ | ++ | o |
| formal strukturiert mit Namensräumen | ++ | o | o | + bis - | |
| lose strukturiert (siehe Text) | o | + | + | + | - |
| Markup-Sprache | + | + | + | + | ++ |
| **Entwicklung, Erweiterbarkeit** | | | | | |
| Erlernbarkeit der Programmierung | o | + | + | | + |
| Entwicklungszeit | o | + | + | + bis - | + |
| Risiko, sich zu verzetteln | mittel | gering | gering | hoch | gering |
| Wartbarkeit | o | + | + | + bis - | + |
| Änderungsfreundlichkeit | - | + | + | + bis - | + |
| Verkettung mehrerer Parser | + | - | + | + bis - | |
| **Sonstiges** | | | | | |
| Geschwindigkeit | ++ | + bis o | + (?) | + bis - | - |

| | kompletter Parser | LIM | PAM | eigenes Programm | XSLT |
|---|---|---|---|---|---|
| zusätzliche Kosten durch Zukauf von Komponenten | eventuell | nein | nein | eventuell | eventuell |

Eignung für Dokumenttyp

formal strukturiert, mit Namensräumen, lose formatiert, Markupsprache

Enbtwicklungskriterien

# 3 Beispielaufgaben "SQL nach SHORE"

Angenommen, Sie kommen neu in ein Projekt und sollen sich ein Bild über die Nutzung einer Datenbank machen, da diese Probleme bereitet. Dazu bekommen Sie von Ihrer Projektleiterin verschiedene Sourcen und Skriptdateien in denen unter anderem SQL-Anweisungen stehen. Gut dass Sie wissen, dass SHORE Ihnen helfen kann, sich in fremden Sourcen zurechtzufinden.

Sie schauen erstmal, wass Sie so haben: ein Skript zum Aufbau der Datenbank, aus denen Sie die Tabellenstruktur entnehmen können (zum Glück nur drei kleine Tabellen), einige Programme verschiedener Sprachen mit SQL-Statements, die zum Glück alle mit `EXEC SQL` beginnen und mit einem `";"` enden.

In der Datenbankdokumentation finden Sie die Syntax von SQL in Bachus-Naur-Notation.

Im folgenden übernehmen wir die Rolle des Projektleiters und stellen Ihnen einige konkrete Aufgaben. Die dazu nötigen Dateien liefern wir ebenfalls mit - sie sind Bestandteil einer SHORE-Installation (ab Version 1.1.0)

Für einige Aufgaben haben wir auch Musterlösungen parat. Diese finden Sie im Anhang.

## 3.1 Vorbereitung

Sie benötigen eine lauffähige Installation von SHORE ab Version 1.1.0 am besten auf Ihrem lokalen PC. Wie Sie SHORE installieren, steht im SHORE Administrator-Handbuch.

Kopieren Sie sich den Ordner `shore\projects\tutorial` mit allen Unterverzeichnissen in ein eigenes Verzeichnis, in dem Sie dann Ihre Übungen machen und aus dem Sie Ihre Resultate nach SHORE importieren (z.B. nach `D:\shore-localdata\projects\tutorial`).

Richten Sie sich pro Aufgabe eine eigene Instanz ein, indem Sie wie folgt vorgehen:

1. Öffnen Sie eine Eingabeaufforderung und wechseln Sie in das SHORE-Verzeichnis in dem die ausführbaren Dateien liegen (z.B. `D:\Programme\shore\bin`).
2. Starten Sie einen SHORE-Server

**Syntax** `shore.exe <dbname.db> <port>`
**Beispiel** `shore.exe Aufgabe1.db 8801`
**Beispiel** `shore.exe Aufgabe2.db 8802`
usw.

3. Öffnen Sie eine weitere Eingabeaufforderung und wechseln Sie in das Verzeichnis, in dem die jeweilige Aufgabe steht.

4. Stellen Sie den Kommandozeilen-Client ein.

**Syntax** `shore admin -server <Servername> -port <port> -wf <Arbeitsverzeichnis>`

**Beispiel** `shore admin -server localhost -port 8801 -wf D:\shore-localdata\projects`

Dabei muss das Arbeitsverzeichnis auf dem Ordner `projects` stehen, unter den Sie anfangs die Übungen hineinkopiert haben.

5. Importieren Sie das Metamodell der jeweiligen Aufgabe.

**Syntax** `shore imp_mm <Metamodell-Datei>`

**Beispiel** `shore imp_mm Aufgabe1.meta`

Falls Sie etwas am Metamodell ändern und das Metamodell neu importieren wollen, benutzen Sie

`shore imp_mm_rescan <Metamodell-Datei>`

6. Starten Sie SHORE im Browser (z.B. `http://localhost:8801/shore/`) und öffnen Sie über das SHORE-Menü Administration > Dokumentenspezifische Parser... den Dialog „Dokumentenspezifische Parser anmelden“. Melden Sie als Parser an: `Dispatcher.exe` für die Dateiendungen `batch; sql; c; pli; lim; xslt`

Nun können Sie anfangen die Aufgaben durchzuarbeiten.

Nachdem Sie ein Skript für eine Aufgabe geschrieben haben, können Sie Ihre Daten nach SHORE importieren.

**Syntax** shore import <Dateien>

**Beispiel** für Aufgabe 1 (Sie stehen in der Eingabeaufforderung im Verzeichnis ist `D:\shore-localdata\projects\tutorial\Aufgabe1`):

```
shore import src\*.sql lim\sql2xsql.lim lim\xSQL2xml.xslt lim\SQL.batch
```

**Beispiel** für Aufgabe 1, Erweiterung 1 (alles in einer Zeile eingeben):

```
shore import src\*.sql lim\sql2xsql-extended1.lim lim\xSQL2xml-extended1.xslt lim\SQL-extended1.batch
```

**Beispiel** für Aufgabe 2:

```
shore import src\* lim\*.lim lim\*.xslt lim\*.batch
```

Kontrollieren Sie anschließend, ob der Parser und der Import erfolgreich waren. Schauen Sie dazu in die folgenden Logfiles:

1. `...\shore\log\parser.log`
2. `...\shore\docstore\<AufgabeNr>\doc\tutorial\<AufgabeNr>\<LIM oder PM>\<importierteDatei>.log`

Falls Fehler im Logfile gemeldet werden, beheben Sie diese und importieren Sie erneut.

Nach einem erfolgreichen import können Sie sich das Ergebnis im Browser ansehen. Dazu geben Sie die URL des Rechners ein, auf dem Sie den SHORE-Server gestartet haben und zu dem Sie importiert haben. Dies ist die gleiche Server-Adresse und Portnummer, die Sie auch im Kommandozeilen-Client eingestellt haben. Die URL für den Browser lautet wie folgt.

**Syntax** http://<servername>:<port>/shore/[<Startseite>]

**Beispiel** `http://localhost:8801/shore/`

**Beispiel** `http://bajuwarort.muc.sdm.de:8802/shore/plugin.html`

# 3.2 Die Dateien

Zu den Aufgaben werden Ihnen Dateien mitgeliefert, die Sie im Verzeichnis `\Aufgaben` unterhalb des Parserkochbuches finden bzw. unterhalb von `...\shore\projects\tutorial`.

| Verzeichnis\Dateiname | Erläuterung |
|---|---|
| Aufgabe1\Aufgabe1.meta | Metamodell zur Aufgabe 1 |
| Aufgabe1\src\defineTables.sql | Enthält SQL-Statements zur Definition der Tabellen, die in den Aufgaben verwendet werden |
| Aufgabe2\Aufgabe2.meta | Metamodell zur Aufgabe 2 |
| Aufgabe2\src\defineTables.sql | Identisch zu Aufgabe1 |
| Aufgabe2\src\getSmallBooks.C | Beispieldatei in C mit einem SQL-Statement |
| Aufgabe3\src\defineTables.sql | Identisch zu Aufgabe1 |
| Aufgabe3\src\getSmallBooks.C | Identisch zu Aufgabe2 |
| Aufgabe3\src*.PLI | Beispieldateien in PLI mit SQL-Statements |

##hier die restlichen Dateien noch eintragen##

# 3.3 Die Aufgaben

Bearbeiten Sie die Aufgaben nacheinander, da Sie aufeinander aufbauen.

In den Verzeichnissen finden Sie in der Regel ein Metamodell und im darunterliegenden `src`-Verzeichnis die für die jeweilige Aufgabe relevanten Source-Dateien, die durch die Parser bearbeitet werden sollen.

## 3.3.1 Aufgabe 1

In dieser Aufgabe bekommen Sie ein bestehendes Metamodell und einige Dateien, in denen SQL-Statements stehen. Ihr Ziel ist es, einen Pattern-Matcher zu schreiben, der in den Dateien die SQL-Tabellen- und -Spalten-Objekte erkennt und mit XML erweitert.

### Dateien

Hier das vorgegebene Metamodell (Datei `Aufgabe1.meta`): Metamodell

```
Dokumenttyp Quelltext


Dokumenttyp SQLQuelltext
   ist ein Quelltext


Objekttyp Typ
   ist definiert in Quelltext


Objekttyp Datenelement
   ist definiert in Quelltext


Objekttyp Tabelle
   ist definiert in Quelltext


Objekttyp Spalte
   ist ein Datenelement
   ist definiert in Quelltext


Beziehungstyp Spalte_ist_definiert_in_Tabelle alias
ist_definiert_in
   von 0 bis * Spalte
   nach 0 bis * Tabelle
```

```
   ist definiert in Quelltext


Beziehungstyp Datenelement_hat_Typ alias hat_Typ
   von 0 bis * Datenelement
   nach 0 bis * Typ
   ist definiert in Quelltext
```

Quelltext

Hier die Quelltextdatei (`defineTables.sql`):

```
EXEC SQL CREATE TABLE BOOK
   (ISBN     CHAR(14)   NOT NULL,
    PAGES    SMALLINT   NOT NULL);
EXEC SQL CREATE TABLE PERSON
   (PERSID     CHAR(10) NOT NULL,
    NAME       CHAR(25) NOT NULL);
EXEC SQL CREATE TABLE READS
   (PERSID     CHAR(10) NOT NULL,
    ISBN       CHAR(14) NOT NULL,
    DATEFROM   DATE     NOT NULL);
```

## Aufgabenstellung

Gesucht: Muster + Generierungsaktionen für Datei `defineTables.sql` sodass folgende Objekte erkannt und mit XML-Markup ausgezeichnet werden:`Tabelle BOOK`

```
Spalte BOOK.ISBN
Spalte BOOK.PAGES

Tabelle PERSON
Spalte PERSON.PERSID
Spalte PERSON.NAME

Tabelle READS
Spalte READS.PERSID
Spalte READS.ISBN
Spalte READS.DATE
```

## Mögliche Erweiterungen

Mögliche Erweiterungen zur Übung:

- das Markup für die Beziehung Spalte_ist_definiert_in_Tabelle wird erzeugt.
- das Markup für die Beziehung Datenelement_hat_Typ wird erzeugt.

## 3.3.2 Aufgabe 2

Zusätzlich zu den in Aufgabe 1 enthaltenen Dateien mit SQL-Statements bekommen Sie C-Sourcen, in denen SQL-Statements eingebettet sind.

Ihre Aufgabe besteht darin, in `SELECT`-Statements eine Verknüpfung der verwendeten Tabelle zu ihrer Definitionsstelle einzufügen.

### Dateien

Das Metamodell aus Aufgabe 1 wird um folgende Elemente erweitert: Metamodell

```
Objekttyp Element
   ist definiert in Quelltext


Objekttyp Block
   ist ein Element
   ist definiert in Quelltext


Objekttyp Datenelement
   ist ein Element
   ist definiert in Quelltext


Dokumenttyp CQuelltext
   ist ein Quelltext


Objekttyp CProgramm
   ist definiert in CQuelltext


Objekttyp CFunktion
   ist ein Block
   ist definiert in CQuelltext


Beziehungstyp Block_verwendet_Tabelle alias verwendet
   von 0 bis * Block
   nach 0 bis * Tabelle
   ist definiert in Quelltext
```

### Aufgabenstellung

Gesucht: Muster + Generierungsaktionen für Datei getSmallBooks.C sodass folgende Objekte und Beziehungen generiert werden:

```
CQuelltext getSmallBooks.C


CProgramm getSmallBooks


Block_verwendet_Tabelle
Quelle:CProgramm getSmallBooks
Ziel:Tabelle   BOOK
```

### Mögliche Erweiterungen

Mögliche Erweiterungen zur Übung:

- die Definitionsstelle der Hostvariablen :isbn wird erkannt und mit XML ausgezeichnet.

  Objekte:

  ```
  Typchar
  DatenelementgetSmallBooks.isbn
  Datenelement_hat_Typ
  Quelle:DatenelementgetSmallBooks.isbn
  Ziel:Typchar
  ```

- die Verwendung der Hostvariablen wird mit einer Beziehung zur Definitionsstelle ausgezeichnet.

  Erweiterung des Metamodells mit der Beziehung

  ```
  Beziehungstyp Block_verwendet_Datenelement alias verwendet
     von 0 bis * Block
     nach 0 bis * Datenelement
     ist definiert in Quelltext
  ```

  erkannte Beziehung:

  ```
  Block_verwendet_Datenelement
  Quelle:CProgrammgetSmallBooks
  Ziel:DatenelementgetSmallBooks.isbn
  ```

.

## 3.3.3 Aufgabe 3

Zusätzlich zu den in Aufgabe 2 enthaltenen Dateien kommen PLI-Sourcen, in denen weitere SQL-Statements eingebettet sind.

Ihre Aufgabe besteht darin, in `INSERT`- und `UPDATE`-Statements eine Verknüpfung der verwendeten Tabelle zu ihrer Definitionsstelle einzufügen.

### Dateien

Das Metamodell aus Aufgabe 2 wird um folgende Elemente erweitert:Dokument-typ PLIQuelltext

Metamodell

```
   ist ein Quelltext


Objekttyp PLIProgramm
   ist definiert in PLIQuelltext
```

### Aufgabenstellung

Gesucht: Muster + Generierungsaktionen für Datei renameBibUser.PLI sodass folgende Objekte und Beziehungen generiert werden:

```
PLIQuelltext renameBibUser.PLI


PLIProgramm RENBUSR


Block_verwendet_Tabelle
Quelle:PLIProgrammRENBUSR
Ziel:Tabelle   PERSON
```

Gesucht: Muster + Generierungsaktionen für Datei newBibUser.PLI sodass folgende Objekte und Beziehungen generiert werden:

```
PLIQuelltext newBibUser.PLI


PLIProgramm NEWBUSR


Block_verwendet_Tabelle
Quelle:PLIProgrammNEWBUSR
Ziel:Tabelle   PERSON
```

## 3.3.4 Aufgabe 4

Aufbauend auf den in Aufgabe 3 eingeführten Sourcen ergibt sich eine weitere Fragestellung. Es soll die Verwendung der einzelnen Spalten verfolgt werden können  (Erkennen der Verwendung von Spalten anhand von Namensgleichheit).

### Aufgabenstellung

Ihre Aufgabe besteht darin, das Metamodell entsprechend zu erweitern und zusätzliche Generierungsaktionen zu implementieren, damit man von der Verwendung der Attribute in der Datei renameBibUser.PLI zu Ihrer Definitionsstelle kommt und umgekehrt.

## Mögliche Erweiterungen

Mögliche Erweiterungen zur Übung:

- Überlegen Sie sich, was Sie tun müssen, um in `newBibUser.PLI` die Spalten zu Ihrer Definitionsstelle zuzuordnen (anhand Ihrer Position bei der `VALUES`-Zuweisung).

# 4 Das erste Metamodell

Sie haben alle Daten zusammen (siehe Kapitel “Beispielaufgaben "SQL nach SHORE"” auf Seite 17) und können nun loslegen.

Die Zutaten: man nehme ein paar Dateien...

Zuerst überlegen Sie sich, welche Informationen für Sie interessant sind und welche Sie schon haben. Betrachten Sie zunächst nur die Tabellendefinition, wie sie in `createDB.sql` vorliegt. Da gibt es zunächst **Tabellen** mit **Feldern** und irgendwelche **SQL-Statements**, die in unterschiedlichen **Dateien** stehen. Diese Dateien sind Programmquelltexte der Sprachen C und PL\I sowie Textdateien, in denen nur SQL-Statements stehen.

Damit können Sie schon ein kleines Metamodell definieren:

```
Dokumenttyp Quelltext

Dokumenttyp SQLQuelltext
   ist ein Quelltext

Objekttyp Typ
   ist definiert in Quelltext

Objekttyp Datenelement
   ist definiert in Quelltext

Objekttyp SQLTable
   ist definiert in Quelltext

Objekttyp SQLColumn
   ist ein Datenelement
   ist definiert in Quelltext

Beziehungstyp Datenelement_hat_Typ alias hat_Typ
   von 0 bis * Datenelement
   nach 0 bis * Typ
   ist definiert in Quelltext
```

Als Dokumenttyp ist hier zunächst nur `SQLQuelltext` definiert, später kommen noch weitere Typen hinzu, die in SHORE importiert werden sollen. Der Dokumenttyp `Quelltext` ist abstrakt und kann bei der Metamodellierung angegeben werden, wenn irgendein beliebiger Dokumenttyp angesprochen werden soll. Diese Abstraktion wird hier in weiser Voraussicht schon einmal definiert ;-).

Die Modellierung der Objekttypen ist an das Standardmetamodell für Programmiersprachen angelehnt, dass mit SHORE ausgeliefert wird (`general.meta`). Der Objekttyp `SQLColumn` leitet sich hier von `Datenelement` ab, da wir aus dem Standardmodell die Beziehung `Datenelement_hat_Typ` nutzen.

Zielformat

Ziel eines Parsers ist es, Quelldateien mit XML anzureichern oder umzuformen, die zum Metamodell passen. Die Beispieldatei `createDB.sql` soll beispielsweise wie folgt aussehen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<SQLQuelltext Name="CREATEDB">

EXEC SQL <SQLTable Name="BOOK">CREATE TABLE <Hotspot>BOOK</Hotspot>

   (<SQLColumn Name="BOOK.ISBN"><Hotspot>ISBN</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="BOOK.ISBN"
   Zieltyp="Typ"
   Zielname="CHAR"><Hotspot>CHAR</Hotspot>(14)</
Datenelement_hat_Typ>   NOT NULL</SQLColumn>,

    <SQLColumn Name="BOOK.PAGES"><Hotspot>PAGES</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="BOOK.PAGES"
   Zieltyp="Typ"
   Zielname="SMALLINT"><Hotspot>SMALLINT</Hotspot></
Datenelement_hat_Typ>   NOT NULL</SQLColumn>)</SQLTable>;


EXEC SQL <SQLTable Name="PERSON">CREATE TABLE <Hotspot>PERSON</
Hotspot>

   (<SQLColumn Name="PERSON.PERSID"><Hotspot>PERSID</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="PERSON.PERSID"
   Zieltyp="Typ"
   Zielname="CHAR"><Hotspot>CHAR</Hotspot>(10)</
Datenelement_hat_Typ> NOT NULL</SQLColumn>,

    <SQLColumn Name="PERSON.NAME"><Hotspot>NAME</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="PERSON.NAME"
   Zieltyp="Typ"
   Zielname="CHAR"><Hotspot>CHAR</Hotspot>(25)</
Datenelement_hat_Typ> NOT NULL</SQLColumn>)</SQLTable>;


EXEC SQL <SQLTable Name="LEND">CREATE TABLE <Hotspot>LEND</Hotspot>

   (<SQLColumn Name="LEND.PERSID"><Hotspot>PERSID</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="LEND.PERSID"
   Zieltyp="Typ"
   Zielname="CHAR"><Hotspot>CHAR</Hotspot>(10)</
Datenelement_hat_Typ> NOT NULL</SQLColumn>,
```

```
    <SQLColumn Name="LEND.ISBN"><Hotspot>ISBN</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="LEND.ISBN"
   Zieltyp="Typ"
   Zielname="CHAR"><Hotspot>CHAR</Hotspot>(14)</
Datenelement_hat_Typ> NOT NULL</SQLColumn>,

    <SQLColumn Name="LEND.DATEFROM"><Hotspot>DATEFROM</Hotspot>
<Datenelement_hat_Typ
   Quelltyp="SQLColumn"
   Quellname="LEND.DATEFROM"
   Zieltyp="Typ"
  Zielname="DATE"><Hotspot>DATE</Hotspot></Datenelement_hat_Typ>
NOT NULL</SQLColumn>)</SQLTable>;


</SQLQuelltext>
```

Im folgenden Kapitel sehen Sie, wie Sie mit einem Pattern-Matcher Strukturen erkennen können und in XML umwandeln.

Pattern-Matching

# 5 Pattern-Matching

Wir möchten Sie gerne mit drei Tools vertraut machen, mit denen Sie auf einfache Weise Dokumente erzeugen können, die Sie nach SHORE importieren können.

- PM
- LIM
- XSLT

Sie möchten beispielsweise hinschreiben:

Beispiel 1 (PM)

```
Objekttyp Funktion
    immer vor /<RundeKlammerAuf>.*<RundeKlammerZu><GeschweifteKlam-
merAuf>/
Beziehungstyp Funktion_ruft_Funktion
   immer vor /<RundeKlammerAuf>.*<RundeKlammerZu><Strichpunkt>/
```

Beispiel 2 (PM)

```
(<Name>)<RundeKlammerAuf>.*<RundeKlammerZu><GeschweifteKlammer-
Auf>/:
    generateObject("Funktion",$1);


    /(<Name>)<RundeKlammerAuf>.*<RundeKlammerZu><Strichpunkt>/:
    generateRelation("Funktion_ruft_Funktion",$1);
```

Oder:

Beispiel 3 (LIM)

```
Wr("<Funktion>");
TIL RundeKlammerAuf() DO
   IdHotspot()
END;
TIL RundeKlammerZu(); GeschweifteKlammerAuf() DO
   FunktionsArgumente()
END;
TIL GeschweifteKlammerZu() DO
   Statement()
END;
Wr("</Funktion>")
```

Tatsächlich erfüllen die Pattern-Matcher, die wir mit SHORE mitliefern, die meisten dieser Anforderungen. Was Rose, Word oder Excel angeht, so können diese Tools den Input zwar nicht direkt lesen, doch ist es durchaus möglich, auf einem Html (XML oder XMI) Export dieser Tools aufzusetzen

# 5.1 Pattern-Matching mit PM

**Achtung:** Dieses Kapitel ist sehr unvollständig und wird auch bis auf weiteres nicht überarbeitet.
Trotzdem ist der SHORE-Pattern-Matcher PM ein sehr wertvolles Tool. Erweiterungen der Beschreibung, Fehlerkorrekturen etc. sind - wie auch für alle anderen Kapitel - sehr willkommen.

PM stellt Ihnen eine Skriptsprache zur Verfügung, in der all das konfigurierbar ist, was Sie zur Dokumentanalyse an Kontrolle benötigen. Die PM-Skriptsprache ist an Perl-Syntax angelehnt und bestimmte Perl-Routinen können Sie in ihr auch direkt aufrufen. Wenn Sie sich mit der Sprache Perl etwas auskennen und wissen, was mit Regulären Ausdrücken so alles möglich ist, dann sollte es Ihnen auch nicht schwer fallen, sich vorzustellen, was mit den angeführten PM-Statements in den obigen Beispielen gemeint ist.

Um PM wirklich verstehen und mit PM arbeiten zu können sind Perl-Kenntnisse erforderlich. Zwar werden auch Nicht-Perl-Programmierer durchaus mit PM erfolgreich arbeiten, doch nach unserer Einschätzung wird PM vor allem für Perl-Programmierer nützlich sein. Der Einstieg ist problemlos.

Im Folgenden wollen wir die Möglichkeiten und Grenzen dieser PM-Skriptsprache darstellen, und Sie in die Lage versetzen, mit dieser Skriptsprache effektiv zu programmieren. Dazu werden wir

a) ganz allgemein Syntax und Semantik dieser Skriptsprache beschreiben.

b) Beispiele geben wie die Skriptsprache konkret verwendet werden kann.

Beides soll im folgenden geschehen. Wenn Sie wollen, können Sie schon jetzt einen kurzen Blick auf die Definition der Skript-Sprachen-Syntax werfen, doch zunächst folgt hier, wie PM arbeitet und welche Möglichkeiten zur Syntax-Analyse und XML-Generierung PM Ihnen bietet.

Wir stellen das PM-Maschinenmodell und die PM-Skriptsprache kurz vor und gehen dann im Detail auf die Funktionsweise von PM-Tokenizers und PM-Scanner ein. Wir zeigen, wie in diesen beiden Verarbeitungsstufen die Statements der Skriptsprache ausgeführt werden. Zuletzt geben wir eine formale Spezifikation der Syntax und Beispiele.

## 5.1.1 Das PM-Maschinenmodell

Das PM-Maschinenmodell ist sehr einfach. Sie spezifizieren

- Patterns
- Aktionen

und teilen damit dem System mit, wo in der Eingabe was zu tun ist. Die Stellen, an denen reagiert werden muss, werden durch die Patterns definiert - die Aktionen, mit denen reagiert wird, implementieren Sie selbst oder Sie greift auf eine Reihe von Standard-Aktionen zurück, die bereits vorhanden sind.

PM

- übernimmt die Steuerung der Suche nach den Patterns, sorgt für die Parameterübergabe an die Aktionen und

- ruft die Aktionen auf.

Die Spezifikation der Patterns und der Aufruf der zugehörigen Aktionen sind in einer einfachen Skriptsprache aufgeschrieben, die wir im Folgenden erläutern.

## 5.1.2 Übersicht über die PM-Skriptsprache

Die PM-Skriptsprache kennt vier Arten von Statements:

- split-Statements
- state-Statements
- read-Statements
- scan-Statements

Die split-, state- und read-Statements (im Folgenden Tokenizer-Statements genannt) werden vom PM-Tokenizer interpretiert. Ihnen folgt in jedem PM-Skript eine Folge von scan-Statement,,deren Abarbeitung der PM-Scanner übernimmt.

Die Folge der Tokenizer-Statements bildet eine Folge von Generationen. Die Folge der Scanner Statements dagegen unterteilt sich in eine Folge von Phasen.

## 5.1.3 PM-Tokenizer und Tokenizer Statements

Jede Generation von Tokenizer Statements beginnt mit einem Split Statment, auf das die zugehörigen Reset Statements und danach die zugehörigen Read Statements folgen:

```
<tokenizer stmt> ::= <split stmt> <reset stmts> <read stmts>
```

Durch die Split-Statements wird der Eingabestrom zerlegt. Die Zerlegung erfolgt hierarchisch anhand von Trennzeichen oder entsprechender Patterns.

Jeder Abschnitt der Zerlegung wird dann durch die Read-Statements sequentiell gelesen. Dabei werden (von vorne beginnend) Tokens abgespalten, die Sie mit push-Aktionen an den nachgelagerten Scanner übergeben können. Nach jedem Abspalten eines Tokens können Sie den Zustand (welche Tokens Sie nun erwarten) wechseln und so die Verarbeitung Ihres Eingabestroms sehr fein steuern.

Ist anhand der spezifizierten Patterns kein Abspalten eines Tokens mehr möglich, so wird der Rest des aktuellen Abschnitts anhand des Split Statements der nächsten Generation weiter zerlegt und auf die daraus resultierenden Unterabschnitte der Zerlegung die nächste Generation von Read-Statements angewandt.

Die Auswahl der Read-Statements einer jeden Generation ist immer auch noch vom aktuellen Zustand abhängig. In den Aktionen der Read-Statements können Sie ihn verändern. Da die Eingabe (und auch der für sie konzipierte Zustandsautomat) fehlerhaft sein kann, kann Ihr Tokenizer in Sackgassen geraten. Sie suchen deshalb vielleicht nach einer Möglichkeit, Ihren Zustand in irgend einer Form wieder zurücksetzen. Es gibt hierfür mehrere Strategien. Die von Ihnen bevorzugte Strategie können Sie in den Reset Statements festlegen.

Die Tokenizer Statements verändern einen Zustand, der im wesentlichen die folgenden Variablen umfasst:

```
$input // die aktuelle Eingabe oder ein Abschnitt davon
```

```
$state  // der aktuelle Zustand
```
Anfangs, das heißt vor dem ersten Split Statement, beinhaltet `$state` einen undefinierten Wert und `$input` den gesamte Eingabe-Strom. Die Zerlegung des Eingabestroms erfolgt rekursiv:`tokenize($input,<split stmt><reset stmts><read stmts><tokenizer stmts>)` **{**

```
my $old;
@input_sections = apply<split stmt> ($input);
($state,$old) = apply<reset stmts>();
while possible {
($section,$state) = apply<read stmts> ($section,$state);
}
for each my $section (@input_sections) {
tokenize($section, <tokenizer stmts>);
}
($state,$old) = apply<reset stmts>();
}
```

Die Wirkung der Statements wird im Folgenden beschrieben. Folgende Tokenizer-Statements gibt es:

- Split Statements:
  ```
  'split' 'on' <pattern>
  ```
- Reset Statements:
  ```
  'save' 'state'
  'start' 'with' 'state' <string>
  'end' 'with' 'state' <string>
  ```
- Read Statements:
  ```
  'if' '(' 'state' <string> 'matches' <pattern>')' '{'
  <actions>
  '}'
  ```

## Split-Statements

Mit einem Split Statement zerlegen Sie die Eingabe zur weiteren Verarbeitung durch die nächste Generation von Tokenizer-Statements. Das angegebene Pattern muss so geklammert sein, dass der Perl-Interpreter die Variablen `$1`, `$2` ... entsprechend belegen kann.

Die Wirkung des Split Statements ist genau so wie man intuitiv erwartet:

```
  @input_sections = apply 'split''on'<pattern> ($input);
<=> @input_sections = split(/<pattern>/,$input);
```

**Hinweis:** Die Zeichen <=> sind nicht Teil der Statements.

## Reset-Statements

Die Wirkung des Reset Statements ist genau so wie man intuitiv erwartet. Durch die Start und End-Statements wird der Zustand gesetzt. Und durch das Save-Statement wird ein Zustand den man sich zuvor gemerkt wiederhergestellt:

Vor dem rekusiven Aufruf

```
 ($state,$old) = apply 'save' 'state';
<=> ($state,$old) = ($state,$state);
 ($state,$old) = apply 'start' 'with' 'state' <string>();
<=> ($state,$old) = (<string>,$old);
```
Nach dem rekusiven Aufruf
```
 ($state,$old) = apply 'save' 'state';
<=> ($state,$old) = ($old,$old);
 ($state,$old) = apply 'end' 'with' 'state' <string>();
<=>($state,$old) = (<string>,$old);
```

## Read-Statements

Wenn das angegebene `<pattern>` auf den Anfang des aktuellen Abschitts der Eingabe `$section` passt, dann verkürzt ein Read Statement den in `$section` gespeicherten String um den innerhalb des `<pattern>` als `$1` identifizierten Teilstring. Danach werden nacheinander die (vom Benutzer definierten) Aktionen `<actions>` aufgerufen und ausgeführt. `($section, $state) =`

```
apply 'if' '(' 'state' <string> 'matches' <pattern> ')' '{'
<actions>;
 '}'
<=>if ($state eq <string> && $section =~ /^<pattern>/) {
        $section = apply 'skipToken'($1);
        $state = apply <actions>;
  }
```

In den Aktionen können Zuweisungen an globale Variablen erfolgen oder es können bereits vorhandene oder benutzerdefinierte Perl-Funktionen aufgerufen werden. Außerdem können hier Zuweisungen an Skript-globale Variablen erfolgen.

```
<action> ::=
  <assignment>
| <function call>
<assignment> ::=
<variable> '=' <expression>;
<expression> ::=
  <function call>
| <value>
```

Den Funktionen können Argumente übergeben werden. Als Argumente zulässig sind:

- globale Variablen `$a` `$b` ... oder `$1` `$2` ...
- oder hart kodierte Strings: `"..."`
- oder Strings in denen globale Variablen dynamisch expandiert werden.

Die globalen Variablen `$a` `$b` ... enthalten Werte, die man durch Zuweisungen verändern kann. Die Werte der Variablen `$1` `$2` ... werden beim Überprüfen der Eingangsbedingung durch das Pattern-Matching mit Werten belegt. `$1` wird immer automatisch als letztes Argument mit übergeben und kann weggelassen werden.

So ist zum Beispiel:

- `skipToken ($1)` gleichbedeutend mit `skipToken()`
- `pushToken($1)` gleichbedeutend mit `pushToken()`
- und statt eines Funktions-Aufrufs `generateObject("Funktion",$1)` können Sie abgekürzt auch schreiben: `generateObject("Funktion")`

## 5.1.4 Tokenizer Actions

Die Tokenizer Aktionen verändern einen Zustand, der die folgende Variablen beinhaltet:
`($section, $state, $file, $line, $column, @tokens)`

Die Aktionen, die Sie im Tokenizer aufrufen können, sind unten aufgelistet.

XML-Generierungsaktionen sollten beim Tokenizing noch nicht verwendet werden. Sie werden im Abschnitt PM-Scanner-Actions beschrieben.

### _setState(<string>)

setzt den Zustand `$state` auf den aktuellen Wert `<string>`:

```
apply '_setState(<string>)
<=> $state = <string>
```

### _pushToken(<type>)

speichert das aktuelle Token von Typ `<type> $1` in ein Array das vom nachfolgenden Scanner ggf. mehrfach durchlaufen wird. `$line` und `$column` werden auf die Postion hinter dem Token gesetzt.

```
apply '_pushToken(<type>)
<=> push @tokens, {
line => $line,
column => $column,
type => <type>,
value => $1,
}
```

### _skipToken()

Immer wenn die Aktionen eines Read-Statement ausgeführt werden, weil das zugehörige Pattern passt, dann wird nach diesen Aktionen der mit `$1` identifizierte Teilstring automatisch abgeschnitten. `_skipToken($1)` wird immer automatisch ausgeführt.

Will man in einem Aktionsblock nicht nur `$1` sondern auch weitere Teilstrings `$2` und `$3` abschneiden, so müssen in den Aktionen sämtliche skip-Statements hingeschrieben und ihnen `$1, $2, $3` als Argumente übergeben werden.

Durch dieses Überspringen von Tokens wird nicht nur die Eingabe $input verkürzt, sondern auch die Werte der Variablen $line und $column werden automatisch angepasst:

```
apply '_skipToken()
<=> my $string = quotemeta($1);
$input =~ s/^$string/;
$line = _adjustLine($string);
$column = adjustColumn($string);
```

## _getFileNr()

liefert den Wert von $file:

```
apply '_getFile();
<=> $file;
```

## _getLineNr()

liefert den Wert von $line:

```
apply '_getLine();
<=> $line;
```

## _getColumnNr()

liefert den Wert von $column:

```
apply '_getColumn();
<=> $column;
```

## _setFile(<value>)

setzt den Wert von $file:

```
apply '_setFile();
<=> $file = <value>;
```

## _setLineNr(<value>)

setzt den Wert von $line:

```
apply '_setLineNr();
<=> $line = <value>;
```

## _setColumnNr(<value>)

setzt den Wert von $column:

```
apply '_setColumnNr();
<=> $column = <value>;
```

## 5.1.5 PM-Scanner und Scan-Statements

Der PM-Scanner arbeitet mit einem Zustand, der die folgenden Variablen enthält
`($phase, $tokenTypes, @tokens)`

Bei `@tokens` handelt es sich um das vom PM-Tokenizer generierte Array von Tokens. Jedes dieser Tokens hat einen Typ und eine Position.

`$tokenTypes` ist String von Tokentypen, den der PM-Scanner beim Pattern-Matching verarbeitet. Von den Positionen in diesem String kann der PM-Scanner genau auf die Positionen im Token-Array zurückschließen.

Das von Tokenizer generierte Array der Tokentypen kann in mehreren Phasen durchlaufen werden. So ist es beispielsweise möglich, in einem ersten Durchlauf sämtliche Definitionsstellen von SHORE-Objekten herauszusuchen um dann in einem zweiten Durchlauf die Verwendungsstellen zu erkennen.

Die Scan Statements haben eine den Read Statements ganz analoge Form:

```
  'if' '(' 'phase' <number> 'matches' <pattern> ')' '{'
<actions>
  '}
```

Die Wirkung dieser Statements ist intuitiv einsichtig.

## 5.1.6 PM-Scan Actions

Die Scan Statements arbeiten mit einem Zustand, der die folgenden Variablen enthält:`(@tokens,@objectStack)`

### openTag (TYPE,TOKEN)

### closeTag (TYPE,TOKEN)

### openObject (TYPE,TOKEN)

### addHotspot (TOKEN)

### closeObject (TYPE,TOKEN)

### addRelation (TYPE,TGTTYPE,TOKEN)

### addObject

## 5.1.7 Syntax der PM Skriptsprache

```
<script> ::= <tokenizer stmts> <scan stmts>

<tokenizer stmt> ::=
  <split stmt> <reset stmts> <read stmts>

<split stmt> ::=
  'split' 'on' <pattern>

<pattern> ::=
'/' <quoted pattern> '/'
'\' <unquoted pattern> '\'

<reset stmt> ::=
| 'save' 'state'
| 'start' 'with' 'state' <string>
| 'end' 'with' 'state'  <string>

<read stmt> ::=
'if' '(' 'state' <string> 'matches' <pattern> ['&&' <filter>] ')'
'{'
<actions>
'}'

<scan stmt> ::=
'if' '(' 'phase' <number> 'matches' <pattern> ['&&' <filter>')'  '{'
<actions>
'}'

<filter> ::=
<expression> <op> <value>

<op> :=
  '=='
| '!='
| '<='
| '>='
| '=~'
| '!~'
| 'eq'
```

```
| 'neq'


<action> ::=
  <assignment>
| <function call>

<assignment> ::=
<variable> '=' <expression>;

<expression> ::=
  <function call>
| <value>

<function call> ::=
<identifier> '(' <values> ')'

<values> ::=
  <value> , <values>
| <value>

<value> ::=
  <variable>
| <string>

<variable> ::=
  '$'<identifier>;
| '$'<number>;

<string> ::= ".*"

<number> ::= [1-9]
<identifier> ::= [_a-zA-Z][_a-zA-Z0-9]*
```

# 5.2 Die Teile eines Menüs

Hier finden Sie eine Beschreibung, wie Sie Pattern-Matching für SHORE nutzen. Anhand des Beispiels lernen Sie Stück für Stück kennen, wie Sie einen Pattern-Matcher in PAM und in LIM schreiben. Dabei gehen wir wie folgt vor:

1. Verarbeitung der unterschiedlichen Dokumenttypen.
2. Erkennen von SQL-Statements generell.
3. Erkennen von Tabellendefinitionen (CREATE TABLE)
4. Erkennen von INSERT

## 5.2.1 Pattern-Matching mit XSLT

# 5.3 Pattern-Matching mit LIM

Mit LIM bekommen Sie eine einfache Möglichkeit an die Hand, Pattern-Matching zu programmieren. Was LIM überhaupt ist, erfahren Sie im Kapitel “Übersicht über die Sprache”. Wie Sie LIM unabhängig von SHORE nutzen können, erfahren Sie im Kapitel “Aufruf von LIM”. Die Syntax von LIM beschreibt Kapitel “Sprachelemente von LIM” und wie Sie LIM für SHORE einsetzen steht in Kapitel “LIM in SHORE”.

## 5.3.1 Übersicht über die Sprache

Da es für LIM kein Lehrbuch gibt und die mitgelieferte Dokumentation recht dürftig ist, folgt hier die Beschreibung der LIM-Sprachelemente

LIM ist eine Sprache um Texte zu bearbeiten und arbeitet mit Mustererkennung und backtracking. LIM ist generell in der Ecke lex, yacc, sed, awk oder trans anzusiedeln und basiert auf Dijkstras Kalkül der "guarded commands". Der Name ist ein Akronym für "Language of the Included Miracle", da LIM Programme angeblich das sogenannte Gesetz "Law of the Excluded Miracle" verletzen können (Details dazu in "A Discipline of Programming", Edsger W. Dijkstra, Prentice-Hall, 1976, und "A generalization of Dijkstra's calculus", Greg Nelson, Trans. on Programming Languages and Systems, October 1989, or SRC Research Report 16 bzw. auf der LIM-Homepage `http://www.research.compaq.com/SRC/lim/` ).

LIM wandelt anhand eines LIM-Programms eine Eingabe sequentiell in eine Ausgabe um. Das LIM-Programm besteht im wesentlichen aus Anweisungen zur Mustererkennung (Lesen) und entsprechenden Schreibaktionen. Die Mustererkennung wird kann annähernd wie eine Grammatikbeschreibung programmiert werden. LIM sucht sich bei der Verarbeitung des Eingabestromes automatisch die passenden Zweige heraus.

Einfache Sprachelemente

Wichtig ist bei LIM dass Anweisungen entweder erfolgreich ausgeführt werden oder nicht ausgeführt werden können und einen Status `FAIL` zurückliefern. Hier einige Beispiele für grundlegende LIM-Sprachelemente:

| LIM-Beispiel | Beschreibung |
|---|---|
| `Wr(s)` | Schreibe `s`. Wobei `s` beispielsweise ein String sein kann. Ist immer erfolgreich. |
| `Rd(s)` | Lies `s`. Schlägt fehl (`FAIL`), wenn `s` nicht das nächste Zeichen in der Eingabe ist. |
| `A ; B` | Lies: "`A`" dann "`B`".<br>Führe `A` aus und bei Erfolg anschließend `B` |
| `A \| B` | Lies: "`A`" oder "`B`".<br>Führe `A` aus. Falls `A` fehlschlägt, führe `B` aus. Wenn `A` und `B` fehlschlagen ist der gesamte Ausdruck fehlgeschlagen. |

| LIM-Beispiel | Beschreibung |
|---|---|
| `Rd("0"); Wr("0") \| Rd("1"); Wr("1")` | Kopiert eine einzelne Null oder Eins von der Ein- zur Ausgabe. Das ";" bindet stärker als das "\|" |

Wenn in LIM ein Befehl fehlschlägt, so hat dies keine Seiteneffekte. D.h. dann wird an der Stelle weitergemacht, an der zuletzt ein Kommando erfolgreich war.

Wenn eine lange Berechnung Seiteneffekte hat und dann fehlschlägt, werden auch die Seiteneffekte rückgängig gemacht. Es ist dann so, als ob diese Berechnung niemals durchgeführt wurde. Beispiel:

```
Rd("0"); Wr("0")
```

und

```
Wr("0"); Rd("0")
```

machen exakt dasselbe. Sie kopieren eine Null von der Ein- zur Ausgabe, wenn möglich. Ansonsten schlagen Sie fehl ohne irgendetwas zu schreiben.

## 5.3.2 Aufruf von LIM

LIM kann von der Windows-Kommandozeile aufgerufen werden und verlangt als Parameter ein LIM-Skript. Seine Eingabe erwartet es dann von STDIN, schreibt seine Ausgabe nach STDOUT und Fehler nach STDERR.

Sequentielle Verarbeitung

LIM funktioniert so, dass in einem LIM-Skript zu lesende Zeichenfolgen und Muster definiert werden und diese verändert oder unverändert wieder ausgegeben werden können. Eine Eingabedatei wird dann anhand dieses LIM-Skripts sequentiell abgearbeitet und das Ergebnis in eine Ausgabedatei geschrieben. Dabei kann man sich ein LIM-Skript wie einen vernetzten Baum von möglichen Mustern vorstellen, in dem ein Blatt für jeweils eine bestimmte Zeichenfolge oder ein Zeichen steht. Dabei existiert bei gültigen Eingabedateien kein unendlicher Zyklus, da LIM ansonsten kein Ergebnis liefern kann. Bei der Verarbeitung versucht LIM nun, die in der Eingabedatei enthaltenen Zeichen anhand des LIM-Skripts zu erkennen. Stellt LIM fest, dass in einem Zweig kein passendes Muster vorhanden ist, so geht es zurück zur letzten passenden Astgabel (bzw. Knoten) und versucht dann den nächsten Zweig abzuarbeiten. Dabei erfolgt ein backtracking, so dass keine Seiteneffekte auftreten z.B. durch Schreibaktionen.

## 5.3.3 Sprachelemente von LIM

### Bestandteile eines LIM-Skriptes

Ein LIM-Skript ist eine Datei, in der Anweisungen für lim.exe stehen. Anhand dieser Anweisungen verarbeitet lim.exe eine Eingabe und schreibt eine Ausgabe.

Ein LIM-Skript besteht aus Kommentaren, globalen Variablen-Definitionen und Prozeduren. In BNF:

**Syntax**

```
LIM-Skript ::= LIM-Element...
LIM-Element ::= Kommentar | globale Variable | Prozedur-Definition
```

**Beispiel**

```
(* LIM-Skript, kopiert alle Zeichen und setzt dabei "<", ">" und "&"
in Ihre HTML-Codes um *)
PROC Main() IS
   TIL Eof() DO
      Rd('<');Wr("&lt;")
      |  Rd('>');Wr("&gt;")
      | Rd('&'); Wr("&amp;")
      | Wr(Rd())
   END;
   Err("Fertig...\n")
END;
```

Dabei muss es immer genau eine Main()-Prozedur geben.

### Kommentar

Kommentare stehen innherhalb von `(*` und `*)`.

### globale Variable

Globale Variablen-Definitionen werden mit `VAR` eingeleitet und mit ";" abgeschlossen.

**Syntax**

```
globale Variable ::= VAR VarList;

VarList ::= VarElement [, VarElement]...

VarElement ::= Identifier := <Expression>
```

**Beispiel**

```
VAR  level := 0, column := 1;
```

### Prozedur-Definition

In LIM können Sie eigene Prozeduren definieren und bereits eingebaute Prozeduren nutzen. Eine Prozedur läuft entweder erfolgreich durch oder schlägt fehl (FAIL). Eigene Prozeduren werden wie folgt deklariert:

**Syntax**

```
Prozedur-Definition ::= PROC [outs  :=]  [inouts:]P([ins])  IS Pro-
zedur-Inhalt END;

outs ::= Identifier [, Identifier]...

inouts ::= (Identifier [, Identifier]...)

ins ::= Identifier  [, Identifier]...

Prozedur-Inhalt ::= LIM-Kommando...
```

**Beispiel**

```
PROC Echo(c) IS
   Rd(c);
   Wr(c)
END;
```

`P` ist der eindeutige Name der Prozedur, `outs`, `inouts`, und `ins` sind Parameter unterschiedlichen Typs. B ist ein Ausdruck, der Inhalt der Prozedur.

## LIM-Kommandos

Die Verarbeitung wird über die LIM-Kommandos gesteuert. In der folgenden Beschreibung stehen A und B jeweils für einen Ausdruck oder Block von Ausdrücken.

**Syntax**

```
LIM-Kommando ::=
   <lokale Variable>
   | <anschließend> LIM-Kommando
   | <oder> LIM-Kommando
   | <berechne> LIM-Kommando
   | <weise zu>
   | <DO>
   | <TIL>
   | <EVAL>
   | <SKIP>
   | <FAIL>
   | <ABORT>
   | <LIM-Statement>
   | <Expression-Operator> <Expression>
   | <Expression>
   | {LIM-Kommando...}
```

### lokale Variable (VAR..IN..END)

Definiert innerhalb einer Prozedur eine lokale Variable. Die Variablen haben ab Ihrem Zeitpunkt der Definition für den Rest der Prozedur Gültigkeit, auch wenn die LIM-Kommandos zwischen `IN` und `END` fehlschlagen. Der Wert einer gleichnamigen globalen Variablen wird nach Verlassen der Prozedur wieder hergestellt.

**Syntax**

```
<lokale Variable> ::= VAR VarList IN LIM-Kommando... END

VarList ::= VarElement [, VarElement]...

VarElement ::= Identifier := <Expression>
```

**Beispiel** Zur Verdeutlichung hier ein kleines LIM-Skript.

```
PROC ThrowInt(n) IS
   (*
    * writes a number as a string to STDERR
    *)
   n = 0 -> Err('0')
 | n < 0 -> Err('-'); TI2(-n)
 | TI2(n)
END;

PROC TI2(n) IS
   n = 0 -> SKIP
 | TI2(n DIV 10); Err('0' + (n MOD 10))
END;

VAR a := 0,
    b := ' ',
    c := -1;
PROC A() IS
    Err("\na (A) = "); ThrowInt(a);
    a := 2;
    Err("\na (A) = "); ThrowInt(a); FAIL;
    {
    VAR a := 3 IN
      Err("\na (in VAR) = "); ThrowInt(a);
      FAIL
    END
    | SKIP
    };
    Err("\na (A) = "); ThrowInt(a)
END;

PROC C() IS
    Err("\na (C) = "); ThrowInt(a);
    a := 2;
    Err("\na (C) = "); ThrowInt(a);
    {
    VAR a := 3 IN
      Err("\na (in VAR) = "); ThrowInt(a);
      FAIL
    END
    | SKIP
```

```
    };
    Err("\na (C) = "); ThrowInt(a)
END;


PROC B() IS
    Err("\na (B) = "); ThrowInt(a)
END;


PROC Main() IS
   {A() | SKIP}
   ; B();
   {C() | SKIP}
   ; B()

END;
```

Die Ausgabe sieht dann so aus:

```
a (A) = 0
a (A) = 2
a (B) = 0
a (C) = 0
a (C) = 2
a (in VAR) = 3
a (C) = 3
a (B) = 2
```

### anschließend (;)

Verkettet Ausdrücke miteinander. Der nächste Ausdruck wird verarbeitet, wenn der aktuelle Ausdruck erfolgreich ist. Ist ein Ausdruck nicht erfolgreich, so schlägt die gesamte Kette fehl.

**Syntax**

```
<anschließend> ::=   ;
```

**Beispiel**

```
A ; B
```

Führe A aus, danach B. Gibt FAIL zurück, wenn A oder B fehlschlagen.

### oder (|)

Definiert alternative Ausdrücke. Der nächste Ausdruck wird dann ausgeführt, wenn der aktuelle Ausdruck fehlschlägt.

**Syntax**

```
<oder> ::=   |
```

**Beispiel**

```
A | B
```

Führe A aus. Wenn A fehlschlägt, führe B aus. Ist erfolgreich, wenn entweder A oder B erfolgreich sind, ansonsten FAIL.

### berechne (->)

Führt einen Ausdruck abhängig vom Ergebnis des vorherigen Ausdrucks aus.

**Syntax**

```
<berechne> ::=   ->
```

**Beispiel**

```
A -> B
```

Führe A aus. Wenn A erfolgreich ist und als Ergebnis einen Wert ungleich "0" zurückgibt, führe B aus. Ist erfolgreich, wenn A und B erfolgreich sind.

### weise zu (:=)

Führt den Ausdruck auf der rechten Seite aus und weist das Ergebnisder linken Seite zu.

**Syntax**

```
<weise zu> ::=  VarElement := Lim-Kommando ...
```

**Beispiel**

```
A -> B
```

Führe A aus. Wenn A erfolgreich ist und als Ergebnis einen Wert ungleich "0" zurückgibt, führe B aus. Ist erfolgreich, wenn A und B erfolgreich sind.

### DO

Führt Ausdrücke solange aus, bis FAIL auftritt. DO selbst ist immer erfolgreich und liefert niemals FAIL zurück.

**Syntax**

```
<DO> ::= DO LIM-Kommando... OD
```

**Beispiel**

```
DO Wr(Rd(' ')) OD
```

Liest und schreibt alle verfügbaren Leerzeichen. Wenn das nächste Zeichen kein Leerzeichen ist, ist DO..OD trotzdem erfolgreich.

### TIL

Führt Ausdrücke solange aus, bis TIL erfolgreich ist. TIL selbst ist immer erfolgreich und liefert niemals FAIL zurück.

**Syntax**

```
<TIL> ::= TIL LIM-Kommando... DO LIM-Kommando... END
```

**Beispiel**

```
TIL Rd('\n') DO Wr(Rd()) END
```

Liest und schreibt alle verfügbaren Zeichen bis zum Zeilenende.

### EVAL

Führt die folgenden Anweisungen aus, wobei ein zurückgegebenes Ergebnis ignoriert wird.

**Syntax**

```
<EVAL> ::= EVAL LIM-Kommando...
```

Dabei müssen die LIM-Kommandos ein Ergebnis zurückgeben.

**Beispiel**

```
TIL Rd('\n') DO Wr(Rd()) END
```

Liest und schreibt alle verfügbaren Zeichen bis zum Zeilenende.

### SKIP

Mache nichts - immer erfolgreich.

**Syntax**

```
<SKIP> ::= SKIP
```

**Beispiel**

```
VAR On := 1, Off := 0, print := Off;
...
VAR ch := '';
DO
  ch := Rd();
  { print -> Wr(ch)
  | SKIP
OD
```

Schreibt alle vorkommenden Zeichen, wenn `print` wahr ist.

### FAIL

Mache nichts - gib FAIL zurück.

**Syntax**

```
<FAIL> ::= FAIL
```

**Beispiel**

```
PROC assert(arg) IS
   arg -> SKIP (* ungleich 0 ist ok *)
   | Err("Fehler\n"); FAIL
END;
```

### ABORT

Schreibt eine Fehlermeldung und bricht das Programm ab. Die Fehlermeldung hat das Muster `<LIM-Skript-Name> aborted at line <lineNo>, read <x> chars, wrote <y> chars`

**Syntax**

```
<ABORT> ::= ABORT
```

**Beispiel**

```
PROC die(rc) IS
   Err("Fatal Error, Return Code: \n");
   ThrowInt(rc);
   Err('\n');
   ABORT
END;
```

## eingebaute Prozeduren (LIM-Statement)

LIM-Statements sind LIM-Kommandos, die schon in LIM enthalten sind.

**Syntax**

```
<LIM-Statement> ::=
   Rd
   | Wr
   | At
   | Eof
   | Err
```

### Rd

Liest das nächste Zeichen vom Eingabestrom. Falls kein weiteres Zeichen verfügbar ist (entspricht End of File), wird FAIL zurückgegeben.

**Syntax**

```
Rd ::= Rd([<StringLiteral> | <CharLiteral> | <CharVariable>])
```

**Beispiel**

```
Rd()
```

Liest das nächste verfügbare Zeichen und gibt dessen Integer-Code zurück. Der Integer-Code wird nur bei dieser Form des Aufrufs zurückgegeben.

**Beispiel**

```
Rd('a')
```

Liest den Buchstaben "a", wenn dieser als nächster Buchstabe verfügbar ist.

**Beispiel**

```
Rd("Hallo")
```

Liest den String "Hallo", wenn dieser als nächstes im Eingabestrom verfügbar ist.

**Beispiel**

```
VAR a := 32;
...
Rd(a)
...
```

Liest als nächstes das Zeichen mit dem Integer-Code 32 (also beispielsweise ein Leerzeichen, wenn nach ASCII codiert ist), wenn dies als nächstes Zeichen verfügbar ist.

### Wr

Schreibt auf die Standardausgabe. Wr ist immer erfolgreich und gibt niemals FAIL zurück.

**Syntax**

```
Wr ::= Wr(<StringLiteral> | <CharLiteral> | <Expression>)
```

**Beispiel**

```
Wr('a')
```

Schreibt den Buchstaben "a".

**Beispiel**
```
Wr("Hallo")
```
Schreibt den String "Hallo".

**Beispiel**
```
VAR a := 32;
...
Wr(a)
...
```
Schreibt als nächstes das Zeichen mit dem Integer-Code 32 (also beispielsweise ein Leerzeichen, wenn nach ASCII codiert ist).

**Beispiel**
```
Wr(Rd())
```
Liest das nächste vorhandene Zeichen (falls verfügbar) und schreibt es sogleich.

### At

Ist erfolgreich, wenn als nächstes in der Eingabe ein zu prüfendes Zeichen bzw. Zeichenfolge steht. Anderenfalls wird FAIL zurückgegeben. Das oder die Zeichen werden nicht "verbraucht", d.h. ein anschließendes `Rd()` gibt das erste durch `At` geprüfte Zeichen zurück..

**Syntax**
```
At ::= At(<StringLiteral> | <CharLiteral> | <Expression>)
```

**Beispiel**
```
At('a')
```
Ist erfolgreich, wenn der nächste Buchstabe in der Eingabe ein "a" ist, sonst FAIL.

**Beispiel**
```
At("Hallo")
```
Gibt FAIL, wenn die nächsten verfügbaren Zeichen nicht "Hallo" sind.

**Beispiel**
```
VAR a := 32;
...
At(a)
...
```
Wenn das nächste verfügbare Zeichen der Eingabe den Integer-Code 32 hat(also beispielsweise ein Leerzeichen, wenn nach ASCII codiert ist) ist At erfolgreich, ansonsten FAIL.

**Beispiel**
```
At(Rd())
```
Ist erfolgreich, wenn ein doppeltes Zeichen als nächstes in der Eingabe verfügbar ist. Dabei würde ein anschließendes `Rd()` den zweiten doppelten Buchstaben lessen.

### Err

Schreibt auf STDERR. Schlägt niemals fehl und wird auch nicht rückgängig gemacht.

**Syntax**
```
Err ::= Err(<StringLiteral> | <CharLiteral> | <Expression>)
```

**Beispiel**

```
Err('a')
```

Schreibt den Buchstaben "a" nach STDERR.

**Beispiel**

```
Err("Hallo")
```

Schreibt den String "Hallo" nach STDERR.

**Beispiel**

```
PROC Dump() IS
   {  Err("rest of line: ");
      TIL Rd('\n'); Err("\n") DO
         Err(Rd())
      END;
      FAIL    (* Undo Rd()s *)
   }
   | SKIP     (* Nie FAIL liefern *)
END;
```

Schreibt den Rest der aktuellen Zeile nach STDERR. Das FAIL läßt den gesamten ersten Block ungültig werden. Durch das SKIP wird die Prozedur immer als erfolgreich verlassen.

**Beispiel**

```
Wr(Rd())
```

Liest das nächste vorhandene Zeichen (falls verfügbar) und schreibt es sogleich.

## Eof

Gibt wahr zurück, wenn das End of File erkannt wird, ansonsten FAIL.

**Syntax**

Eof ::= Eof()

**Beispiel**

```
TIL Eof() DO
   Wr(Rd())
END
```

In dem Beispiel werden alle zeichen bis zum Ende der Datei kopiert.

## Operatoren von Ausdrücken (Expression-Operator)

**Syntax**

```
<Expression-Operator> ::=
   <AND>
   | <OR>
   | <NOT>
   | <gleich>
   | <ungleich>
   | <kleiner als>
   | <größer als>
   | <kleiner gleich>
   | <größer gleich>
   | <summiere>
   | <multipliziere>
   | <DIV>
   | <MOD>
   | <unary minus>
```

### AND

Logische Konjunktion. Der Ausdruck auf der rechten Seite wird nicht mehr ausgewertet, wenn der linke Ausdruck fehlschlägt (d.h. das Ergebnis Null ist).

**Syntax**

<AND> ::= AND

**Beispiel**

```
a AND c
```

### OR

Logische Disjunktion. Der Ausdruck auf der rechten Seite wird nicht mehr ausgewertet, wenn der linke Ausdruck erfolgreich ist (größer Null).

**Syntax**

<OR> ::= OR

**Beispiel**

```
a OR c
```

### NOT

Logische Negierung. Dabei steht 0 für "falsch" und 1 für "wahr". Allerdings werden alle Werte ungleich 0 als "wahr" interpretiert.

**Syntax**

<NOT> ::= NOT

**Beispiel**

```
NOT 0
```

liefert als Ergebnis 1

**Beispiel**

```
NOT 1
```

liefert als Ergebnis 0

**Beispiel**
```
NOT 99
```
liefert als Ergebnis 0

### gleich (=)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn Sie den selben (Integer-)Wert haben.

### ungleich (#)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn Sie unterschiedliche (Integer-)Werte haben.

### kleiner als (<)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn der linke (Integer-)Wert kleiner als der rechte Wert ist.

### größer als (>)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn der linke (Integer-)Wert größer als der rechte Wert ist.

### kleiner gleich (<=)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn der linke (Integer-)Wert gleich oder kleiner als der rechte Wert ist.

### größer gleich (>=)

Vergleicht die Werte zweier Ausdrücke. Gibt wahr zurück, wenn der linke (Integer-)Wert gleich oder größer als der rechte Wert ist.

### summiere (+)

Summiert die Werte zweier Ausdrücke und gibt die Summe als Ergebnis zurück.

### multipliziere (*)

Multipliziert die Werte zweier Ausdrücke und gibt das Produkt als Ergebnis zurück.

### Ganzzahldivision (DIV)

Teilt den Wert des linken Ausdrucks durch den Wert des rechten Ausdrucks und gibt das abgerundete Ergebnis zurück. DIV funktioniert wie in Oberon und Modula-3.

**Beispiel**
```
3 DIV 2
```
liefert 1 als Ergebnis

**Beispiel**
```
3 DIV 4
```
liefert 0 als Ergebnis

**Beispiel**
`11 DIV 4`
liefert 2 als Ergebnis

### Modulo (MOD)

Teilt den Wert des linken Ausdrucks durch den Wert des rechten Ausdrucks und gibt das Ergebnis zurück. Entspricht `e - f * (e DIV f)` und funktioniert wie in Oberon und Modula-3.

**Beispiel**
`3 MOD 2`
liefert 1 als Ergebnis

**Beispiel**
`3 MOD 4`
liefert 3 als Ergebnis

**Beispiel**
`11 MOD 4`
liefert 3 als Ergebnis

### unary minus (-)

Gibt den negierten Wert des Ausdrucks zurück.

**Beispiel**
```
VAR a := 0,
    b := ' ',
    c := -1;
```

Dann liefert `-a` 0 als Ergebnis,
`-b` liefert -32 als Ergebnis
und `-c` liefert 1 als Ergebnis.

## Ausdrücke (Expressions)

**Syntax**
`<Expression> ::= Procedure-Call | Variable | Literal | (<Expression>)`

### Procedure-Call

Aufruf einer LIM-Prozedur.

**Syntax**
`inouts:P(ins)`

**Beispiel**

```
VAR ascii = 1, ebcdic = 2;

PROC char := Encode(char, code) IS
   code = ascii -> ...
 | code = ebcdic -> ...
...
END;

PROC WriteEncoded() IS
...
Wr(Encode(char, ascii))
...
END;
...
```

### Variable

Der Wert einer global oder lokal definierten Variable.

### Literal

Ein Literal ist eine ganze Zahl, ein String oder ein CharLiteral.

**Syntax**

```
Literal ::= Integer | <StringLiteral> | <CharLiteral> | (<Expressi-
on>)
```

### Identifier

Identifier fangen immer mit einem Buchstaben oder einem Unterstrich an. Danach folgen Buchstaben, Zahlen oder Unterstriche.

### Character-Literal

Ein Character-Literal ist ein einzelnes Zeichen oder eine Escape-Sequenz in einfachen Anführungszeichen.

**Syntax**

```
<CharLiteral> ::= '<Character>'

<Character> ::= <normalCharacter> | <EscapeSequence>

<normalCharacter> ::= ' ', 0, 1, ...

<EscapeSequence> ::=
          \\        backslash (\)
  |       \t        tab
  |       \n        newline
  |       \f        form feed
  |       \r        return
  |       \s        blank space
  |       \b        backspace
  |       \v        vertical tab
  |       \e        escape
  |       \'        single quote
  |       \"        double quote
  |       \ddd      char with octal code ddd
  |       \xhh      char with hex code hh
```

### String-Literal

Ein String-Literal ist eine Folge von Character-Literalen in doppelten Anführungszeichen. String-Literale sind als Paramter nur in den eingebauten Funktionen "Rd", "Wr", "At", und "Err" erlaubt.

### Boolsche Ausdrücke

Das Ergebnis von boolschen Operationen entspricht einer 1 für "wahr" und 0 für "falsch". Ansonsten wird alles was nicht 0 ist als wahr interpretiert.

# 5.3.4 LIM in SHORE

## So ist ein LIM-Skript für SHORE aufgebaut

Die Prozeduren eines LIM-Skriptes für SHORE können in mehrere Abschnitte unterteilt werden:

- Dokumenttyp- und Sprachunabhängige Prozeduren
- Sprachabhängige Prozeduren
- Dokumenttypabhängige Prozeduren

Tatsächlich sind die bestehenden LIM-Skripten auch nach dieser Unterteilung organisiert. Dabei haben wir für SHORE die unterschiedlichen Teile in entsprechende Dateien aufgeteilt, die über `#include` eingebunden werden können. Die Auflösung der include-Statements ist nicht LIM-Standard und wird über einen C-Präprozessor vor Aufruf des LIM-Interpreters erledigt. Dies erledigt das Parserframework automatisch.

Unterhalb des SHORE-Verzeichnisses finden sich die Verzeichnisse `parser\lib\lim\include` für unabhängige Prozeduren und `parser\lib\lim\lang` für Prozeduren, die eine Programmiersprache parsen.

Die dokumenttypabhängigen Prozeduren betrachten wir gleichzeitig projektabhängig und stehen daher nicht unter dem SHORE-Verzeichnis sondern sollten oberhalb des Verzeichnisses stehen, wo Ihre Sourcen liegen und wo das xml erzeugt werden.

Ein LIM-Skript für SHORE ist nun konkret so aufgebaut (schematisch):

```
include der Basisfunktionen
include der Programmiersprachen-Funktionen
Ablaufsteuerung:
   1. Einfügen des xml-Headers und Dokumenttyp-Tags
   2. Aufruf der programmiersprachenabhängigen Prozeduren
```

So sieht das Programm für unser SQL-Beispiel in LIM so aus (gekürzt):

```
1.  (*
2.   * SQL2xSQL.lim - LIM script for parsing SQL-Files
3.   *)
4.  #include "basics.lim"
5.  #include "SQL922xSQL92.lim"
6.
7.  PROC SQLProgram() IS
8.    {  Rd("EXEC");
9.       Wr("EXEC");
10.      SQL92_WhiteSpaces();
11.      Rd("SQL");
12.      Wr("SQL");
13.      SQL92_WhiteSpaces();
14.      TIL Rd(";"); Wr(";") DO
15.         {
16.         SQLStatement(); { SQL92_WhiteSpaces() | SKIP}
17.         | SQL92_WhiteSpaces()
18.         | CharCopy()
19.         }
20.      END
21.   | CharCopy()
22.   }
23. END;
24.
25. PROC Start () IS
26.    Echo('#'); TIL Echo ('\n') DO CharCopy() END
27. END;
28.
29. PROC Main() IS
```

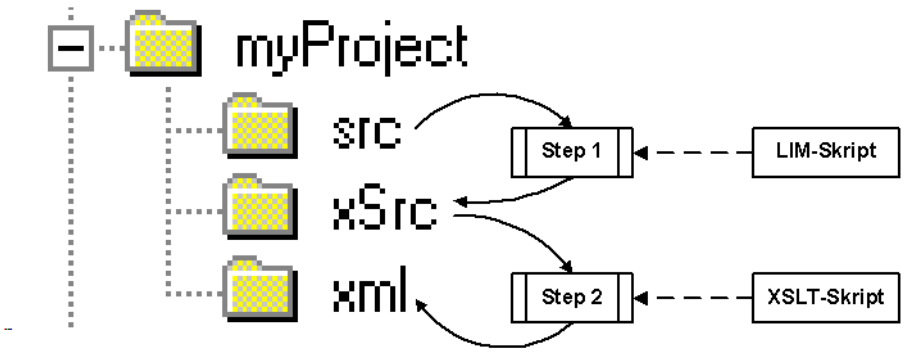
```
30.    Wr("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n");
31.    Wr("\n<SQL92Quelltext>\n");
32.
33.    (* SQL Parsen *)
34.    { Start () | SKIP };
35.    DO SQLProgram() OD;
36.    { Eof() | Wr("***Error***"); DO CharCopy() OD };
37.
38.    Wr("\n</SQL92Quelltext>\n")
39. END;
```

Die Zeilen 1 bis 3 sind Kommentare, in Zeile 4 werden die Basisfunktionen eingebunden, in Zeile 5 wird eine Datei mit SQL-92-Erkennung inkludiert.

Die Prozedur `SQLProgram()` ist für die Erkennung von SQL-Statements zuständig. Ein SQL-Statement steht immer innerhalb von `EXEC SQL` und `";"`

## So arbeitet ein Pattern-Matcher in LIM

Der auf LIM basierte Pattern-Matcher arbeitet zweistufig. In der ersten Stufe wird der Quelltext in einfaches XML umgewandelt, in der zweiten Stufe wird das XML mittels XSLT mit Attributen angereichert, damit es nach SHORE importiert werden kann.




### Die erste Stufe: LIM erzeugt einfaches XML

Mit LIM können Sie einfaches XML erzeugen. Dazu muss ein zum Quelltext passendes LIM-Skript existieren, in dem Anweisungen stehen um das zum Metamodell passende XML zu erzeugen.

### Die zweite Stufe: XSLT erzeugt SHORE-konformes XML

Nachdem LIM eine Datei in XML umgewandelt hat, müssen noch Attribute eingefügt werden, damit SHORE das XML verarbeiten kann. LIM kann bei seiner Verarbeitung zwar entscheiden, ob es ein bestimmtes Objekt oder eine bestimmte Beziehung gefunden hat und schreibt dann die entsprechenden XML-Tags, allerdings kann LIM aufgrund seiner beschränkten Fähigkeit nicht auf einfache Weise die für SHORE notwendigen Attribute einsetzen. Dafür bietet sich aber eine XSL-Transformation an.

## Aufrufoptionen

LIM und XSLT sind in das Parser-Framework eingebunden. Das Framework führt selbstständig den Aufruf beider Stufen aus. Beim Aufruf mit ImportBatch muss die Batchdatei für die LIM/XSLT-Verarbeitung folgender Syntax genügen:

**Syntax**

**Syntax** `limInput2xml`

```
<project path>
<suffix>
<lim script path>
<xslt script path>
<intermediate path>
```

**Beispiel**

```
limInput2xml
   D:/shore-more/testdaten/ParserKochbuch/src
   sql
   D:/shore-more/testdaten/ParserKochbuch/sql2xsql.lim
   D:/shore-more/testdaten/ParserKochbuch/xSQL922xml.xslt
   D:/shore-more/testdaten/ParserKochbuch/xSrc;
```

Für den Aufruf geben Sie folgende Parameter an:

- `project path`
  Der volle Pfad zum Verzeichnis, in dem Ihre Projektdokumente stehen.
- `suffix`
  Die Dateiendung, für die geparst werden soll.
- `lim script path`
  Der Name des LIM-Skripts (mit voller Pfadangabe), in dem die Anweisungen zur XML-Erzeugung für diesen Dokumenttyp stehen.
- `xslt script path`
  Der Name des passenden XSL-Skripts (mit voller Pfadangabe), in dem die Anweisungen zur XSL-Transformation stehen.
- `intermediate path`
  Ein Verzeichnisname, in dem die Zwischenergebnisse gespeichert werden sollen.

## Testmöglichkeiten

## zu ergänzen ##

## Musterlösung in LIM: SQL nach SHORE

Die Dateien für unser Beispiel finden Sie im Ordner `LimExample`. Darunter befinden sich die Ordner `Src`, `xSrc` und `xml`. Im Ordner `Src` finden Sie die Sourcen, die nach SHORE importiert werden sollen, `xSrc` nimmt die Zwischendateien auf, die von LIM erzeugt werden und im Ordner `xml` stehen dann die Dateien im SHORE-konformem xml.

## So verarbeiten Sie unterschiedliche Dokumenttypen

Pro Dokumenttyp setzen Sie üblicherweise ein eigenes LIM-Skript ein. Es ist allerdings auch denkbar ein Skript für mehrere Dokumenttypen einzusetzen, dies betrachten wir in diesem Beispiel aber nicht weiter.

sql2xsql.lim, C2xC.lim und PLI2xPLI.lim

In unserem Beispiel haben wir drei unterschiedliche Dokumenttypen, die wir jeweils einzeln Parsen. Dafür schreiben wir uns jeweils ein LIM-Skript (siehe auch “So ist ein LIM-Skript für SHORE aufgebaut” auf Seite 57) . Für `*.sql`-Dateien verwenden wir dieses Skript (`sql2xsql.lim`):

```
#include "basics.lim"
#include "SQL922xSQL92.lim"

PROC SQLProgram() IS
  {  Rd("EXEC");
     Wr("EXEC");
     SQL92_WhiteSpaces();
     Rd("SQL");
     Wr("SQL");
     SQL92_WhiteSpaces();
     TIL Rd(";"); Wr(";") DO
        {
        SQLStatement(); { SQL92_WhiteSpaces() | SKIP}
        | SQL92_WhiteSpaces()
        | CharCopy()
        }
     END
  | CharCopy()
  }
END;

PROC Start () IS
    Echo('#'); TIL Echo ('\n') DO CharCopy() END
END;

PROC Main() IS
  Wr("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n");
  Wr("\n<SQL92Quelltext>\n");

  (* SQL Parsen *)
  { Start () | SKIP };
  DO SQLProgram() OD;
  { Eof() | Wr("***Error***"); DO CharCopy() OD };

```

```
35.    Wr("\n</SQL92Quelltext>\n")
36. END;
```

Die LIM-Skripten für C und für PL/I sehen identisch aus, nur dass Sie in der Main-Prozedur statt `SQL92Quelltext` die zum Metamodell passenden Typen schreiben. Diese sind für das Skript `C2xC.LIM` der Dokumenttyp `CQuelltext` und für das Skript `PLI2xPLI.lim` der Dokumenttyp `PLIQuelltext`.

sql92.meta

Diese Typen müssen auch im Metamodell deklariert sein (siehe `sql92.meta`):

```
...
Dokumenttyp CQuellText
   ist ein Quelltext

Dokumenttyp PLIQuelltext
   ist ein Quelltext
...
```

sql92.batch

Für den Test nutzen wir aus dem Parserframework die ImportBatch-Variante. Dafür schreiben wir uns eine Batchdatei (`sql92.batch`):

```
limInput2xml
   D:/shore-more/testdaten/ParserKochbuch/src
   sql
   D:/shore-more/testdaten/ParserKochbuch/sql2xsql.lim
   D:/shore-more/testdaten/ParserKochbuch/xSQL922xml.xslt
   D:/shore-more/testdaten/ParserKochbuch/xSrc
   clean;


limInput2xml
   D:/shore-more/testdaten/ParserKochbuch/src
   C
   D:/shore-more/testdaten/ParserKochbuch/C2xC.lim
   D:/shore-more/testdaten/ParserKochbuch/xSQL922xml.xslt
   D:/shore-more/testdaten/ParserKochbuch/xSrc
   clean;


limInput2xml
   D:/shore-more/testdaten/ParserKochbuch/src
   PLI
   D:/shore-more/testdaten/ParserKochbuch/PLI2xPLI.lim
   D:/shore-more/testdaten/ParserKochbuch/xSQL922xml.xslt
   D:/shore-more/testdaten/ParserKochbuch/xSrc
   clean;
```

Da LIM nur einfaches XML ohne Attribute erzeugt, benötigen wir noch einen Nachbrenner in Form eines XSLT-Skripts. xSQL2xml.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">

<xsl:output method="xml" encoding="ISO-8859-1"/>

<!-- ************** Stylesheet Parameter -->

<xsl:param name="filename" select="system-property('INPUT')"/>

<xsl:template match="SQL92Quelltext">
   <SQL92Quelltext Name="{$filename}">
      <xsl:apply-templates/>
   </SQL92Quelltext>
</xsl:template>

<xsl:template match="CQuelltext">
   <CQuelltext Name="{$filename}">
      <xsl:apply-templates/>
   </CQuelltext>
</xsl:template>

<xsl:template match="PLIQuelltext">
   <PLIQuelltext Name="{$filename}">
      <xsl:apply-templates/>
   </PLIQuelltext>
</xsl:template>

</xsl:stylesheet>
```

Damit werden Dokumenttypen erkannt.

## Erkennen von SQL-Statements generell

Erkennen von Tabellendefinitionen (CREATE TABLE)

Erkennen von INSERT

# 5.4 So stellt man sich sein Menü zusammen

Zusammenspiel mehrerer Parser für einen Dokumenttyp, Verknüpfung der Dokumente untereinander. ## zu ergänzen ##

# 5.5 Import nach SHORE

SHORE sieht vor, dass für bestimmte Dateiendungen bestimmte Parser angemeldet sind. Dieses Konzept trägt aber nur, solange die Dateiendungen eindeutig auf bestimmte Parser (oder Dokumenttypen) hindeuten, und auch nur solange alle Dokumente einzeln geparst werden und etwaige Abhängigkeiten zwischen den Eingabedokumenten beim SHORE-Update ignoriert werden können.

In der Praxis kommen oft kompliziertere Fälle vor. Wenn Sie beispielsweise den Wunsch haben, Ihre Worddokumente feiner typisieren zu wollen (beispielsweise Fachkonzepte und DV-Konzepte unterscheiden), werden Sie eine solche Unterscheidung anhand des vollen Dateinamens treffen und nicht nur anhand der Dateiendung.

Darüber hinaus wird es insbesondere bei Quelltexten sehr häufig vorkommen dass Änderungen in einem Dokument ein erneutes Parsen eines anderen Dokuments erforderlich machen. Oder Sie finden den Fall vor, dass zwei verschiedene Java-Dateien zu zwei unterschiedlichen Java-Projekten gehören, die beide in SHORE abgelegt sind und denen sie beide durch die entsprechenden Parser richtig zugeordnet werden müssen.

Die Regel:
Pro Dateiendung ein Parser und pro Eingabe-Dokument ein Ausgabedokument gilt nicht mehr, sobald Sie bei der Auswahl Ihrer Parser selektiver sein und dabei auch noch Abhängigkeiten zwischen Dokumenten berücksichtigen wollen. In diesen Fällen empfehlen wir Ihnen zum Import Ihrer Dokumente auf zwei Perl-Module (Dispatcher und Import-Batch) zurückzugreifen, die im Folgenden näher beschrieben werden:

Der Dispatcher erkennt selektiv anhand seiner Konfiguration welche Parser für eine zu importierende Datei zuständig sind. Import-Batches sind in ähnlicher Weise konfigurierbar, jedoch nicht für einzelne Dateien sondern für Projektverzeichnisse.

## 5.5.1 Direkter import von XML

## 5.5.2 Import mit dem Dispatcher

Der Dispatcher erkennt anhand seiner Konfiguration aus Dateinamen und Dateiendung die zugehörigen SHORE-Dokumenttypen und ruft die entsprechenden Parser auf. Der Dispatcher wird von einem Perl-Skript angesteuert, das Sie anhand einer Vorlage erstellen bzw. anpassen. Die Konfiguration erfolgt, indem Sie in dem Skript eine sogenannte Frontend- und eine Backend-Tabellen einstellen. Der Dispatcher, dessen run-methode danach aufgerufen wird, arbeitet dann zweistufig:

Vorverarbeitung
Um Dokumente nach SHORE zu importieren, wird auf diese Dokumente zunächst ein frei wählbares Frontend-Modul angewandt, das meist die Aufgabe hat, ein technisches Format nach Html oder XML zu konvertieren.

Parser-Nachbrenner
Nach der Vorverarbeitung, reichert ggf. ein Backend-Modul diese XML-Dokumente mit zusätzlichem fachlichen Markup an.

Wie wir noch sehen werden können Sie diese Frontend / Backend-Architektur auf vielfältige Weise und für vielfältige Zwecke einsetzen und nutzen.

Wenn Sie sich mit den Sourcen der Perl-Skriptumgebung vertraut machen, können Sie mit maximaler Flexibiltät den Import von SHORE-Dokumenten beliebiger Art anstoßen.

## So binden Sie Ihren Dispatcher ein

Ihren Dispatcher können Sie als SHORE-Administrator auf folgende Weise in Betrieb nehmen:

1. Sie konfigurieren Ihren Dispatcher (den existierenden Dispatcher können Sie dazu nach Ihren Vorstellungen anpassen) und
2. Sie melden Ihren Dispatcher bei SHORE als Parser für Ihre unterschiedlichen Dokumenttypen an
3. Sie implementieren ggf. weitere (Frontend- oder Backend-) Module mit Funktionen, die von Ihrem Dispatcher-Modul gerufen werden.

Bei der Installation von SHORE oder in Absprache mit dem SHORE-Team erhalten Sie Module, die von Ihrem Dispatcher angesteuert werden können. Es existieren hier beispielsweise anpassbare Parser für Worddokumente und Exceltabellen.

## So konfigurieren Sie den Dispatcher

Der Dispatcher besteht aus einer ausführbaren Datei und einer zugehörigen Konfigurationsdatei. In der Auslieferung sind dies im Verzeichnis `...shore\parser\bin` die Dateien `Dispatcher.exe` und `Dispatcher.pl`

Sie können je nach Bedarf einen oder mehrere Dispatcher einsetzen. Im folgenden beschreiben wie die Konfiguration eines Dispatchers. Wenn Sie mehrere Dispatcher einsetzen wollen, so kopieren Sie `Dispatcher.exe` und `Dispatcher.pl` unter einem gleichen Namen. `Dispatcher.exe` erwartet seine Konfiguration in einer gleichnamigen `*.pl`-Datei

Damit Ihr Dispatcher richtig arbeitet spezifizieren Sie in `Dispatcher.pl` folgende Dinge:

- Ihre Frontend-Tabelle
- Ihre Backend-Tabelle

Die Belegung der Tabellen-Inhalte erfolgt in einem kleinen Perl-Skript. Dieses Skript initialisiert das Dispatcher-Modul und ruft dann dessen run-Methode auf. Die Frontend-Tabelle und die Dokumenttyp-Tabelle müssen Sie als Perl-Arrays, die Backend-Tabelle dagegen als Hash realisieren. Diese Implementierungsdetails sind aus den mitgelieferten Vorlagen sofort ersichtlich, im folgenden sprechen wir weiterhin von Tabellen.

## Die Frontend-Tabelle

Die Zeilen dieser Tabelle haben folgende Form:

```
[Muster-Dateinamen , Dokumenttyp , Frontend-Modul , Argumente ]
```

Durch Zeilen dieser Form legen Sie fest,

- für welchen Mustern von Dateien (Spalte 1)
- welche Dokumenttypen zu generieren sind (Spalte 2)
- welche Frontend-Parser (Spalte 3)
- mit welchen Argumenten aufgerufen werden (Spalte 4)

### Die Backend-Tabelle

Die Frontend-Tabelle muss so beschaffen sein, dass der Dispatcher in der Lage ist, für jede Datei den gewünschten Dokumenttyp zu berechnen. Um Dokumente für den SHORE-Import zu erzeugen, müssen die Dateien, die von den Frontends erzeugt werden, ggf. durch einen Parser noch weiter transformiert (und mit Markup angereichert) werden. Dies wird in der Backend-Tabelle definiert.

Ihre Zeilen sind von der Form:

```
[Dokumenttyp , Backend-Modul, Argumente]
```

Durch die Tabelle legen Sie fest,

- für welchen Dokumenttyp (Spalte 1)
- welches Backend-Modul (Spalte 2)
- mit welchen Argumenten (Spalte 3)

aufgerufen wird.

An der Schnittstelle zwischen Frontend und Backend beachten Sie bitte folgendes:

- wenn Sie kein Frontend-Modul für ein Dokument spezifizieren, so nimmt das Backend-Modul das Original-Dokument als Eingabe

### Schnittstelle zu den Frontends und Backends

Von den Frontend-Modulen, die vom Dispatcher aufgerufen werden können, wird erwartet dass sie eine Funktion `convert` zur Verfügung stellen, die die folgende Signatur hat:

```
convert($p, $doctype, \%args);
```

Von den Backend-Modulen, die vom Dispatcher aufgerufen werden können, wird erwartet dass sie eine Funktion `markup` zur Verfügung stellen, die die folgende Signatur hat:

```
markup($tmpfile, $outputpath, $type, \%args );
```

Als letztes Argument wird in beiden Fällen eine Referenz auf einen anoymen Hash erwartet, den man in Perl nach folgendem Schema initialisert:

```
{ argName1 => $value1, ... argNameN => $valueN,}
```

`$p` ist eine Instanz auf ein `Shore::Interface` Objekt, über dessen Methoden man auf Dinge wie den Namen des zu dispatchenden Files herankommt.

## 5.5.3 Import mit ImportBatch

Beim Import-Batch handelt es sich um ein Perl-Modul, das auf dem SHORE-Server läuft. Es kann zu jeder Zeit unabhängig davon, ob neue Dateien auf den Server gestellt wurden, den Import dieser Dateien durchführen.

Insbesondere organisiert es den inkrementellen Import, sodass sich unterschiedliche SHORE Instanzen mit den Dokumenten aus verschiedenen Projektverzeichnissen nach Wunsch bestücken lassen und redundante Arbeit dabei vermieden wird.

Beim inkrementellen Import stößt es einen Make-Mechanismus an, der feststellt, welche Dateien zu parsen und in die entsprechende Instanz zu importieren sind. Letzeres geschieht immer dann, wenn Dateien nach dem letzten erfolgreichen Import neu dazu gekommen oder von solchen Dateien abhängig sind.

Auf den ersten Blick sieht dies natürlich sehr nach einer reinen Server-Lösung aus. Doch läßt sich dieser Vorgang ohne weiteres auch vom Client aus bedienen, es werden jeweils nur die gerade geänderten Dateien (mit den zugehörigen Batches) zum Server übertragen.

## So konfigurieren Sie Ihre Import-Batches

Das Import-Batch-Modul ist ein Interpreter für eine Skriptsprache. In den Kommandos dieser Skriptsprache werden die Aufträge an die Parser-Module spezifiziert.

### Die Syntax Ihrer Import-Batch-Kommandos

Die Kommandos haben die Form: ...

**Syntax** `<input>2xml <inputfolder> <suffixes> ... [<target>]`

**Beispiel**
```
cobol2xml   # COBOL-Parser
   $SHORE/projects/myPrj/quellen/Cobol/src
   cob
   $SHORE/projects/myPrj/quellen/Cobol/xmlm
   all;
```

Alle Kommandos sind mit einem ";" abzuschließen. Innerhalb der Kommandos sind Zeilenumbrüche und Kommentare möglich, welche stets mit dem Zeichen "#" beginnen und mit dem Ende der Zeile abgeschlossen sind.

Jedes `<input>2xml`-Kommando ist in der Skriptumgebung durch ein eigenes Perl-Modul implementiert, dass den Aufruf eines Parsers kapselt. Jedes `<input>2xml`-Modul erwartet optional weitere Argumente, wie Sie entsprechend innerhalb des Moduls in der Parameterleiste der run-Methode implementiert sind. Sehr häufig erwarten die Parser den Namen von Verzeichnissen, in denen sie ihre Ergebnisse oder Zwischenergebnisse ablegen. Wenn sich ein Modul auf das Make-Modul der Perl-Skript-Umgebung von SHORE abstützt, so müssen Sie das Target für den Make-Mechanismus als letztes Argument übergeben.

### Die Schnittstelle Ihrer Import-Batch-Kommandos

Die einzelnen Kommandos übergeben ihre Argumente immer den run-Methoden der entsprechenden Module.

So steuern Sie Beispielsweise mit dem Kommando `java2xml` die `run`-Methode des Moduls `java2xml.pm` im Paket Parser an. In diesem Modul können Sie somit auch genauer nachsehen, welche Argumente diese Methode erwartet:

```
java2xml <project path> <suffixes> <classpath of project> <intermediate path> clean|delta
```

Bei den übrigen Modulen sieht die Aufrufschnittstelle der Kommandos folgendermaßen aus.

- `cobol2xml <project path> <suffixes> <intermediate path> clean|delta`
- `xcobol2xml <project path> <suffixes> <intermediate path> clean|delta`
- `vb2xml <project path> <suffixes> <intermediate path> clean|delta`
- `limInput2xml <project path><suffixes> <lim script path> <xslt script path> <intermediate path> xml`
- `pmInput2xml <project path><suffixes> <pm script path> <keywords> <intermediate path> <target>`

Alle Pfade sind relativ anzugeben. Bei sämtlichen Pfad-Angaben können Sie sich (wie in Beispiel 3.1 skizziert) auf die Umgebungsvariable SHORE abstützen. Diese Variable ist automatisch immer richtig gesetzt ist, wenn Sie SHORE installiert haben. Weitere Umgebungsvariablen stehen Ihnen nicht zur Verfügung.

### So rufen Sie auf dem SHORE-Server einen Import-Batch auf

Um diese Batches auf dem SHORE-Server lokal zu testen, können Sie den Batchfile-Interpreter folgendermassen aufrufen:

```
perl Import-Batch.pl batchfile.bat <outputfolder>
```

Wichtig ist, dass Sie in diesem Fall neben dem Batchfile auch das von Ihnen gewünschte Ausgabe-Verzeichnis mit angeben.

## So triggeren Sie Import-Batches vom Client aus

Normalerweise haben Sie als Benutzer keinen Zugriff auf die Serverkonsole, wenn sie eine Datei importieren wollen. Das heißt, dass Sie üblicherweise Ihren Import-Batch vom Client aus anstoßen. Dazu melden Sie als Parser `Import-Batch.exe` an. `Import-Batch.exe` ruft `Import-Batch.pl` auf und reicht alle Parameter durch. Dieser Umweg ist notwendig, da SHORE nur direkt ausführbare Dateien als Parser akzeptiert. `Import-Batch.pl` interpretiert die übergebene Batchdateien und stellt die durch ihn generierten xml-Dateien in ein SHORE-internes temporäres Verzeichnis.

Das heißt: Sie können nun Batches Ihrer Wahl zusammenstellen, und von einem beliebigen SHORE-Client aus über das Kommando "shore import" zum Ablauf bringen.

Die Dateien müssen vor dem Import bereits auf dem SHORE-Server im Eingabeverzeichnis des Parsers vorhanden sein. Entweder kopieren Sie dazu die Dateien manuell oder per Skript auf den Server. Sie können sich auch einen Trigger-Mechanismus in Ihrem Konfigurations-Management-System installieren. Es besteht auch die Möglichkeit, dass die `<input>2xml`-Module sich selbst die benötigten Dateien aus dem Konfig-Management holen, wenn sie darauf Zugriff haben. Letzteres kann unter NT allerdings im konkreten Projekt daran scheitern, dass den SHORE Servern die entsprechenden Netzzugriffe verweigert werden, wenn sie als Dienst gestartet wurden. Dies wäre vorher zu prüfen.

Nachdem die Parser XML erzeugt haben, holt sich SHORE die von den Parsern erzeugten Dokumente aus dem Parser-Ausgabeverzeichnis ab und stellt sie in die SHORE-Datenbank ein.

## So triggern Sie Import-Batches über Ihren Dispatcher

Wenn Sie vom Client aus eine oder mehrere Dateien zum Server erst übertragen, danach dort Analyse von Abhängigkeiten durchführen wollen, und dann parsen lassen wollen, so gibt es dazu nur einen Weg. Sie melden den Dispatcher als Parser für sämtliche Dateiendungen an. Wenn Sie dann Ihre Dateien importieren senden Sie als allerletztes eine entsprechende batch-Datei. Damit wird der Dispatcher Ihre Dateien

1. erst sammeln,
2. sie dann in ein Verzeichnis kopieren, wo bereits andere Dateien des zu parsenden Projekts stehen,
3. und am Ende einen Import-Batch für dieses Verzeichnis aufrufen.

Das Parsen der Dateien (mit einer bestimmten Dateiendung) wird damit so lange wie möglich verzögert. Eine Analyse der Abhängigkeiten ist nämlich erst dann sinnvoll möglich wenn alle Dateien, die über den SHORE Import transportiert werden, in den Verzeichnissen angekommen sind, in denen die Parser sie erwarten.

Um den Dispatcher für diese Art der Verarbeitung zu konfigurieren, gehen Sie wie folgt vor (so ist es auch schon in der ausgelieferten `Dispatcher.pl` eingestellt):

- Tragen Sie bei den `frontends` für diejenigen Dateien, die Sie über einen Import-Batch importieren wollen, das Copy-Modul ein.
- Melden Sie für diese Dateien bei den `backends` das Skip-Modul an, welches die ihm übergebenen Dateien ignoriert.
- Bei den `frontends` müssen die `*.batch`-Dateien für den Dokumenttyp ImportBatch eingetragen sein (ist schon vorkonfiguriert).
- Dadurch, das bei den `backends` für ImportBatch das gleichnamige Modul (ImportBatch) eingetragen ist, startet der Dispatcher beim Import von *.batch das Modul `ImportBatch.pm`

Haben Sie Ihren Dispatcher und Import-Batches konfiguriert und als Parser für ihre Dateien angemeldet, so können Sie die zu parsenden Dateien (inclusive der zugehörigen Batches) mit dem Kommando `shore import` an den SHORE Server übergeben.

## Inhaltsverzeichnisse

Wenn Sie die Quelltexte umfangreicher Software-Projekte nach SHORE importieren so hat eine sehr große Zahl von Dokumenten den gleichen Dokumenttyp. Damit werden die Listen, die SHORE anzeigt, wenn Sie sich einen Dokumenttyp ausgewählt haben, sehr lang. Entsprechend mühsam ist dann die Suche nach Dateien in flachen Verzeichnisstrukturen.

Um nach Ihren Dateien zu suchen, können Sie alternativ den Objekttyp "Projekt" auswählen, und von dort aus in die Inhaltsverzeichnisse für die Unterverzeichnisse ihres Projekts navigieren.

### So generieren Sie Projekt-Inhaltsverzeichnisse

Die Inhaltsverzeichnisse werden von ImportBatch-Modul automatisch generiert. Und zwar wird für jedes Kommando `<input>2xml` in einem Batchfile `<projekt>.batch` ein Projekt mit dem Namen `<projekt>-<input>` angelegt.

Im Ausgabeverzeichnis befindet sich ein gleichnamiges Unterverzeichnis, dessen Dateien die Endung `.toc` aufweisen. (für "table of contents").

Um Namenskonflikte bei den Projekt-Inhaltsverzeichnissen auszuschließen, achtet der Import-Batch-Interpreter darauf, dass kein `<input>2xml`-Modul innerhalb eines Batches Kommando mehr als einmal aufgerufen wird.

Generell können Namenskonflikte immer dann entstehen, wenn Sie versuchen Dokumente mit gleichem SHORE-Dokumentnamen von unterschiedlicher Stelle aus zu importieren (relativ zu der Stelle an der der SHORE-Import gerade steht oder einmal gestanden hat).

Grundsätzlich sollten Sie sich überlegen, welche Projektverzeichnisse Sie in einer SHORE-Instanz zusammenwerfen und wie Sie die Eindeutigkeit der Dokumentnamen herstellen wollen (Insbesondere müssen Sie sich klar darüber werden, welches Präfix Sie vor die Dokumentnamen stellen, bzw. welches Präfix Sie weggelassen).

### So importieren Sie Projekt-Inhaltsverzeichnisse

SHORE lässt alle XML-Dateien, die von den Parsern über Import-Batch-Interpreter oder Dispatcher-Aufrufen erzeugt wurden, in einem Verzeichnis ablegen, in dem es diese Dateien zum Import erwartet.

Damit SHORE die erzeugten Dateien auch entgegennimmt, müssen Sie Ihren speziellen Parser, Dispatcher oder Import-Batch-Interperter als Parser nicht nur für alle Dateiendungen anmelden, die geparst werden, sondern auch für die erzeugt werden, also beispielsweise auch `batch` und `toc`

Beispiel:

Wenn Sie Ihr Java-Projekt "jp" über einen Import-Batch `jp.batch` nach SHORE importieren wollen, so sollten Sie `Dispatcher.exe` oder `ImportBatch.exe` für die Dateieindungen `batch`, `toc` und `java` bei SHORE als Parser anmelden.

Falls Sie Ihre Import-Batches direkt auf dem Server gestartet haben und direkt XML importieren wollen, so melden Sie `xml2shore.exe` als Parser an.

## 5.5.4 Kopplung mit einem KM-System

## 5.5.5 kompletter import

# Anhang

# A SHORE Grundbegriffe

Hier finden Sie bei SHORE häufig verwendete Begriffe und ihre Bedeutung.

Die Hauptzutaten: Projektdokumente

SHORE ist offen für Projektdokumente beliebigen Typs, die typischerweise in einem Software-Entwicklungsprojekt anfallen. Es verknüpft in einem Hypertextsystem die unterschiedlichsten Dokumente mit den unterschiedlichsten Datenformaten und Informationsinhalten. Ein Projektdokument ist jede beliebige Datei, die in SHORE aufgenommen werden kann.

Jedes für SHORE interessante Projektdokument enthält strukturierte Informationen, die in Objekte und Beziehungen eingeteilt werden. Objekte sind Teile eines Dokuments, und zwischen den Objekten (oder auch Dokumenten) können Beziehungen definiert werden.

Die Speisekammer: das SHORE-Repository

SHORE arbeitet als Repository für diese Projektdokumente. Das bedeutet, dass die Projektdokumente an SHORE übergeben - oder importiert - werden und in SHORE gespeichert werden. Für jedes importierte Projektdokument speichert SHORE weitere Informationen, die sich aus seiner Struktur ergeben. Die Dokument- und Strukturinformationen - also die gefundenen Objekte und Beziehungen - werden zusätzlich in einer Datenbank gespeichert. Dadurch können komplexe Auswertungen erfolgen. Diese werden mit SHORE verwaltet und stehen für Navigation und Recherchen zu Verfügung. Unterschiedlichste Dokumente können auf diese Weise miteinander verknüpft werden.

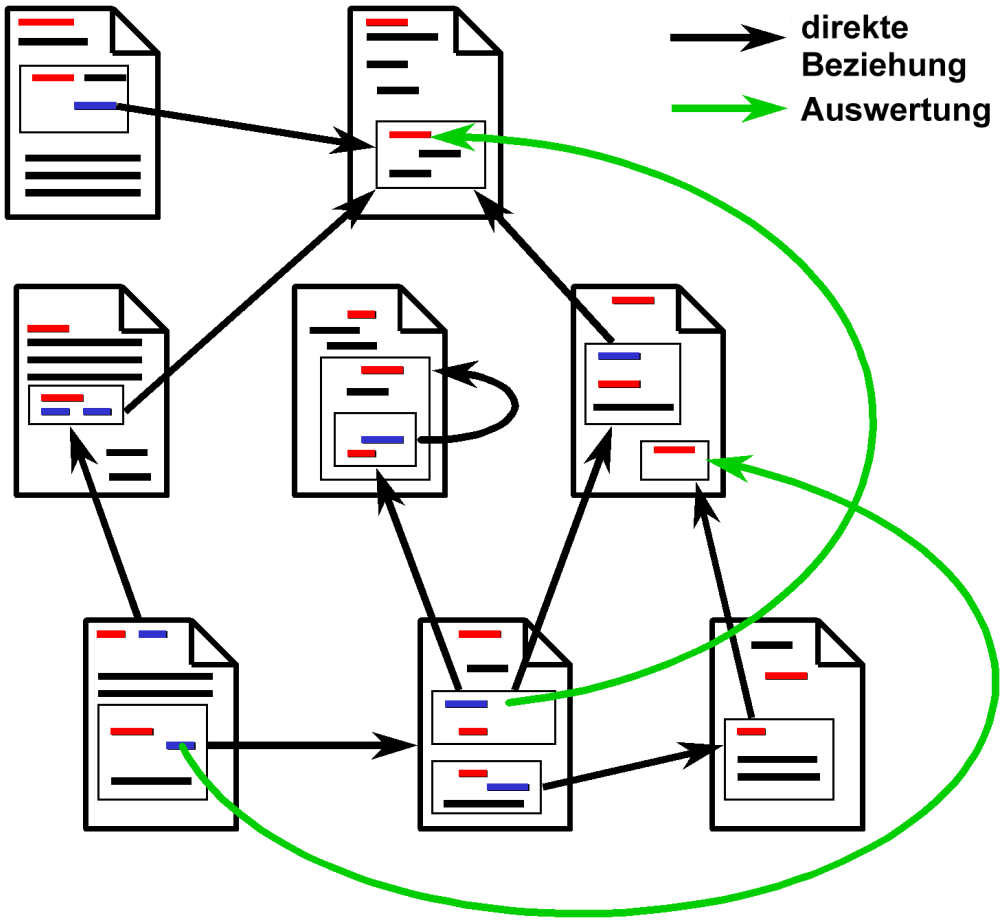



**Abbildung 2:** Beziehungen zwischen Dokumenten

Die Zubereitung: Parsen

Die Dokumente werden von Parsern in das universelle Dokumentformat XML (eXtensible Markup Language) konvertiert. Ein Parser analysiert ein Dokument und erkennt Objekte und Beziehungen, die für eine Hypertext-Navigation relevant sind. Dabei können unterschiedliche Parser eingesetzt werden, die jeweils für unterschiedliche Dokumenttypen eingesetzt werden. Dokumente, die schon in SHORE-konformem XML vorliegen, können auch direkt importiert werden, da SHORE einen XML-Parser beinhaltet.

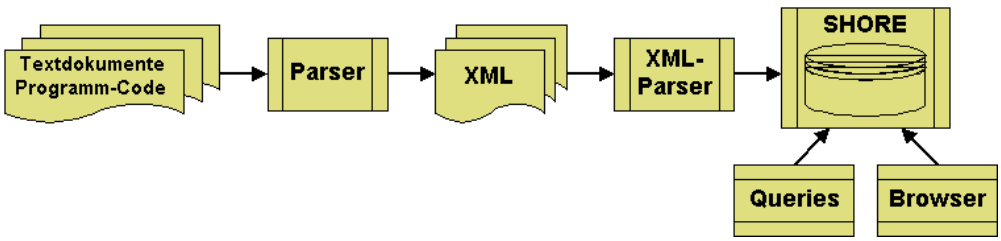



**Abbildung 3:** Schematischer Weg der Dokumente nach SHORE und Zugriffe

Die Gewürze: XML-Tags

Die Objekte und Beziehungen, die von den Parsern in einem Dokument erkannt wurden, werden mit XML-Tags markiert. Beim Import extrahiert SHORE die XML-Tags und speichert Sie in einer Datenbank.

Das Rezept: Meta-modell

Das Datenbankschema ist bei SHORE im Gegensatz zu anderen Repositories frei definierbar. Der Anwender definiert in einem Metamodell, welche Objekte und Beziehungen er in SHORE sehen will. Jedes Projekt kann die Struktur seines Metamodells frei gestalten und passende Parser nach seinen eigenen Bedürfnissen entwickeln.

Modell und Meta-Metamodell

In einem Softwareentwicklungsprojekt bilden die Projektdokumente ein Modell des zu untersuchenden Systems ab.

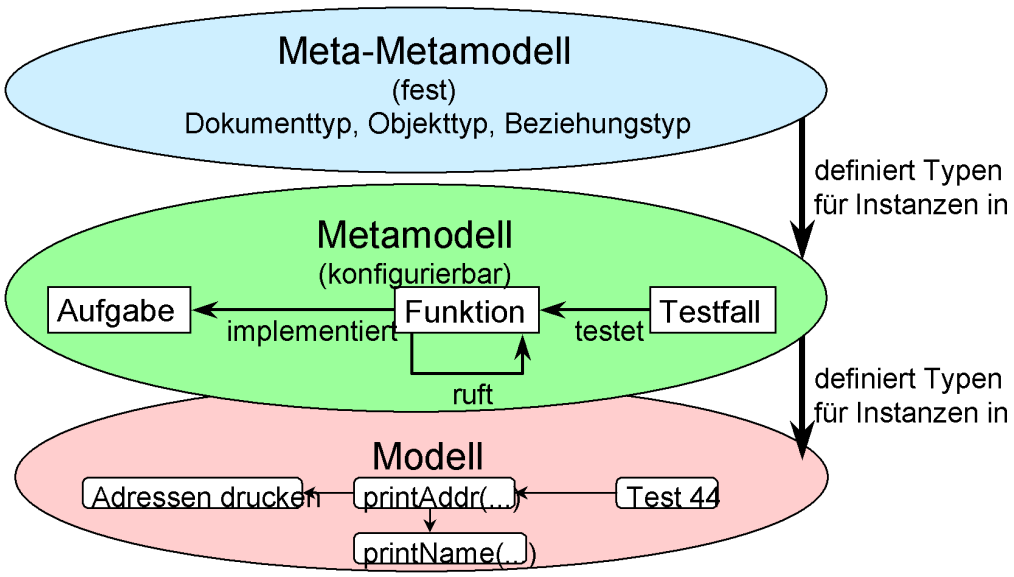



**Abbildung 4:** Modellebenen in SHORE

Dabei wird die Struktur des Modells durch das sogenannte Metamodell definiert. Dieses Metamodell kann pro Projekt individuell definiert werden. Die Typen des Metamodells und die dazwischenliegenden Beziehungen definiert das Meta-Metamodell. Das Meta-Metamodell ist die Definition von Dokument-, Objekt- und Beziehungstyp und ist in SHORE unveränderlich.

Ein konkretes Beispiel dazu: In einem Projekt wird eine Software entwickelt. Es werden unter anderem Konzeptpapiere, Schnittstellenbeschreibungen sowie Java- und Cobolprogramme erstellt. Dies sind die Dokumenttypen.

Objekttypen können sein: in einem Konzeptpapier einzelne Anwendungsfälle, in jedem Anwendungsfall die beteiligten Personen (Aktoren) oder Abläufe. Die Objekttypen von Schnittstellenbeschreibungen sind beispielsweise der Name einer Schnittstelle, ein Zielsystem oder verwendete Datentypen. In den Programmen sind unter anderem folgende Objekttypen enthalten: Programm- und Klassennamen, Methoden, Parameter, Variablen, Datentypen und so weiter.

Hier nur eine Auswahl an denkbaren Beziehungstypen:

- "Programm implementiert Anwendungsfall": der Name des Anwendungsfalles findet sich im Programmkommentar.
- "implementiert Schnittstelle": ein identifizierender Name in einem Programm weist auf die Schnittstellenbeschreibung.
- "ruft": ein Unterprogramm- oder Methodenaufruf
- "ruft": ein eingebetteter Datenbankzugriff (z.B. SQL), der auf die Datenbank-( oder Tabellen-)definition verweist

Das Auftischen: import

Informationen, die nach SHORE gelangen werden als Import bezeichnet. Importiert werden zum Beispiel das Metamodell, Projektdokumente, Stylesheets oder Parser.

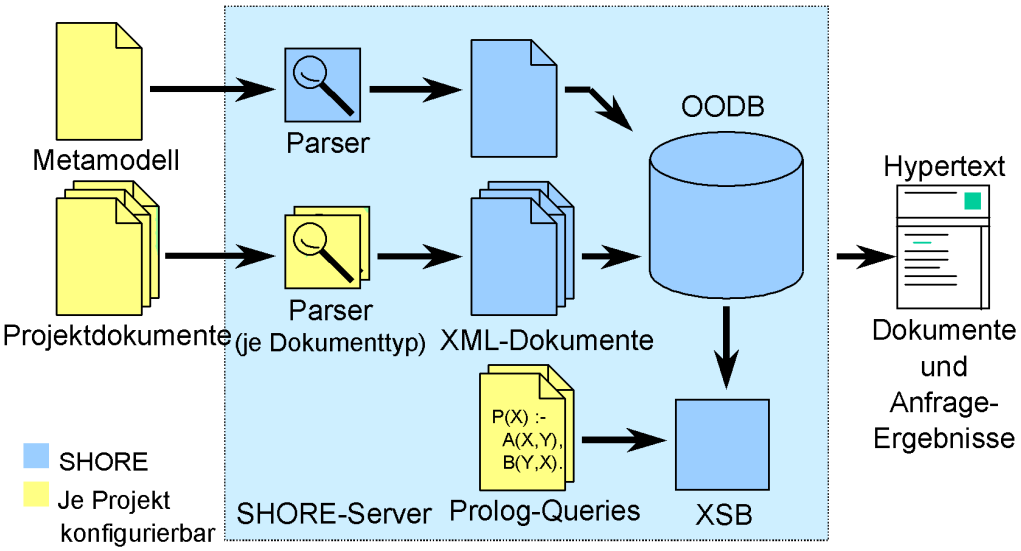



**Abbildung 5:** Import nach SHORE

gepflegt Essen: surfen in SHORE

Alle Dokumente, die nach SHORE importiert sind, kann der Anwender direkt als Hypertext im Browser sehen. Der HTML-Hypertext wird dynamisch aus den XML-Dokumenten generiert. Bei dieser Umwandlung werden die Einträge aus der Datenbank farblich ausgezeichnet und miteinander verlinkt, damit eine Hypertext-Navigation über sie möglich ist (Objekte rot und Beziehungen blau). Weitere Elemente können durch Cascading Stylesheets (CSS) ebenfalls unterschiedlich dargestellt werden.

genau hinschmekken: SHORE-Anfragen

Mit jedem Klick auf eine Hypertext-Beziehung wird eine Anfrage an die SHORE-Datenbank gerichtet, die das passende Dokument heraussucht und im Browser präsentiert. Aufgrund der Tatsache, dass eine Datenbank verwendet wird, ist damit sogar die Navigation in beide Richtungen möglich.

Durch die Anbindung der logischen Programmiersprache Prolog (XSB-Prolog) haben Sie außerdem die Möglichkeit, über Navigations-Anfragen hinaus weitere komplexere Abfragen an die Datenbank zu stellen. Die Ergebnisse dieser Anfragen werden ebenfalls als dynamische HTML-Seiten anzeigt.

Neben der Möglichkeit der Hypertext-Navigation bietet SHORE auch die Möglichkeit, komplexe Abfragen zu stellen. Besonders hervorzuheben ist, dass SHORE auch rekursive Abfragen erlaubt. Damit können Sie zum Beispiel auch baumartige Strukturen aus dem Beziehungsnetz extrahieren.

Typische komplexe Abfragen, die Projekte häufig wünschen, sind:

- wie sehen unsere Aufrufhierarchien aus?
- wie sehen unsere Vererbungshierarchien aus?
- wo gibt es nicht verwendete Objekte?
- wo gibt es Objekte, die mehrfach definiert sind?
- wie sieht die Nachbarschaft eines Objekts aus ... eingeschränkt auf ...?
- usw.

Derjenige, der Anfragen erstellt, benötigt eine genaue Kenntnis des Metamodells. Denn er muss dem System sagen, von welchen Objekten welchen Objekttyps SHORE über welche Beziehungen welchen Beziehungstyps bei der Suche navigieren soll. Je konkreter / allgemeiner (in der Vererbungshierarchie) hierbei die angegebenen Typen sind, desto präziser / umfassender sind auch die Antworten, die SHORE bei diesen Abfragen zurückliefert.

# F Metamodell Syntax

Im Metamodell definieren Sie Strukturen anhand von Dokument-, Objekt- und Beziehungstypen. Grundsätzlich können Sie bei allen Typen Vererbungsmechanismen nutzen.

Dokumenttyp

Dokumenttypen deklarieren Sie im Metamodell mit optionaler Angabe von Dokumenttyp-Supertypen und der Information, ob Dokumente dieses Typs für die Volltextsuche indiziert werden, in der Form

**Syntax**

```
Dokumenttyp Dokumenttypname
        ( ist (ein|eine) Dokumenttypname )*
        [ ist nicht volltextindizierbar ]
```

**Beispiel**

```
Dokumenttyp PLIQuelltext
        ist ein Quelltext
```

Objekttyp

Im Metamodell wird ein Objekttyp wie folgt deklariert mit optionaler Angabe der Dokumenttypen, in denen sie definiert werden und optionaler Angabe von Objekttyp-Supertypen:

**Syntax**

```
Objekttyp Objekttypname
        ( ist (ein|eine) Objekttypname )*
        ( ist definiert in Dokumenttypname )*
```

**Beispiel**

```
Objekttyp PLIProgramm
 ist ein Namensraum
 ist definiert in PLIQuelltext
```

Beziehungstyp

Im Metamodell werden Beziehungstypen deklariert mit Angabe der Quell- und Zielresource und deren Kardinalitäten , mit optionaler Angabe der Dokumenttypen, in denen sie definiert werden können und optionaler Angabe von Beziehungstyp-Supertypen:

**Syntax**

```
Beziehungstyp Beziehungstypname [ alias Aliasname ]
        von n1 bis n2|* Resourcetypname
        nach m1 bis m2|* Resourcetypname
        ( ist Teilmenge von Beziehungstypname )*
        ( ist definiert in Dokumenttypname )*
```

**Beispiel**

```
Beziehungstyp Block_verwendet_Variable alias verwendet
        von 0 bis * Block
        nach 0 bis * Variable
        ist Teilmenge von Block_verwendet_Datenelement
        ist definiert in Quelltext
```

# G Musterlösungen

## G.1 Lösungen mit LIM

Den vom Werkzeugteam geschriebenen Code zur Generierung von Objekten + Muster mit LIM/XSLT finden Sie unterhalb des jeweiligen Aufgabenverzeichnisses im Ordner `lim`.

**Tipp:** Schauen Sie sich als erstes die Bibliothek `basics.lim` an (in `...shore\parser\lib\lim`).
Daraus können Sie einige grundlegende Prozeduren nutzen.
Die Bibliothek binden Sie wie in C mit einem `include`-Statement ein (es wird dann auch tatsächlich durch einen C-Präprozessor ausgeführt, bevor der LIM-Interpreter läuft).

Beispiel:

```
#include "basics.lim"
```

### G.1.1 Musterlösung zu Aufgabe 1

Es gibt jeweils eine Lösung zu der gestellten Aufgabe sowie jeweils zu der vorgeschlagenen Erweiterung.

#### Übersicht der Dateien:

Dateien der Musterlösung zur Grundaufgabe:

`SQL.batch`

`SQL2xSQL.lim`

`xSQL2xml.xslt`

Dateien der Musterlösung zur ersten Erweiterung:

`SQL-extended1.batch`
`SQL2xSQL-extended1.lim`
`xSQL2xml-extended1.xslt`

Dateien der Musterlösung zur zweiten Erweiterung:

`SQL-extended2.batch`
`SQL2xSQL-extended2.lim`
`xSQL2xml-extended2.xslt`

### Tipps zur Musterlösung der Grundaufgabe

- In `SQL2xSQL.lim` ist die Main-Prozedur ganz am Ende der Datei. Die SQL-Statements werden zentral in der Prozedur SQLProgram erkannt, da sie jeweils mit `EXEC SQL` eingeleitet werden und mit einem Semikolon enden.
- In `xSQL2xml.xslt` werden die in LIM erkannten Tags mit SHORE-Attributen angereichert.

## G.1.2 Musterlösung zu Aufgabe 2

Es gibt hier zur Zeit nur die Lösung zur Grundaufgabe. Diese sind jedoch getrennt in die Erkennung von `*.sql` und `*.C`

### Übersicht der Dateien:

| | |
|---|---|
| SQL_lib.lim | Bibliotheksdatei, in der für die Aufgabe SQL-Statements erkannt werden. |
| SQL.batch<br>SQL2xSQL.lim<br>xSQL2xml.xslt | Dokumenttypspezifische Erkennung für *.sql-Dateien. |
| C.batch<br>C2xC.lim<br>xC2xml.xslt | Dokumenttypspezifische Erkennung für *.C-Dateien |
| ImportAufgabe2.bat | Datei mit SHORE-Import-Anweisungen. |

.

### Tipps zur Musterlösung

Während in Aufgabe1 sämtliche LIM-Statements in einer Datei stehen, sind in der Aufgabe2 die SQL-spezifischen Prozeduren herausgelöst und stehen in einer eigenen Datei (`SQL_lib.lim`). Diese wird per `#include` von den dokumenttypspezifischen LIM-Dateien eingebunden.

Dadurch können die Konstrukte auch von anderen Dateitypen genutzt werden.

Das Eintippen der shore import-Befehle, ist in eine Batchdatei ausgelagert. Damit können Sie recht bequem Importieren.

Hier stehen zwar drei import-Befehle, aber Sie können auch mit einem Befehl importieren. Dann wird die Parameterliste etwas länger. Ein tatsächlicher import in die SHORE-Datenbank wird erst dann gestartet, wenn Sie eine `*.batch`-Datei importieren.

# H LIM Command-Reference

**Hinweis:** Die Quelle aller Informationen zur Sprache ist die mitgelieferte Info-Seite die im Sourcecode von LIM enthalten ist. Daraus entstanden ist die folgende Zusammenfassung der Kommandos.

Built-In Procedures

LIM provides the following built-in procedures:

`Rd(str)` Read "`str`" (a string literal), or fail if it is not next in the input.

`Rd(c)` Read the character "`c`" (an expression), or fail.

`Rd()` Read and return the next input character, or fail.

`Wr(str)` Write "`str`" (a string literal). It never fails.

`Wr(c)` Write the character "`c`" (an expression). It never fails.

`At(str)` A noop if "`str`" (a string literal) is next in the input; otherwise fails.

`At(c)` A noop is the next character is "`c`"; otherwise fails.

`Eof()` A noop if at end of file; else fails.

`Err(str)` Write "`str`" to standard error. Never fails, is not undone.

`Err(c)` Write the character "`c`" (an expression) to standard error. Never fails, is not undone.

`A ; B` Do A, then do B. The command "A;B" fails if either "A" or "B" fail Control Flow

`P -> A` Evaluate "P"; if it is non-zero, execute "A", else fail.

`A | B` Do A, else do B if A fails. The command A|B fails only if both "A" and "B" fail.

`DO A OD` repeatedly execute "A" until it fails

`TIL A DO B END` Repeatedly execute "B" until executing "A" succeeds.

`v := E` Evaluate the expression "E" and assign its value to "v". Fail if any function called by E fails.

`EVAL E` Evaluate the expression "E" and discard the result. Fail if any function called by E fails.

`SKIP` No-op

`FAIL` This command always fails.

`ABORT` Print an error message and abort the program.

`outs := inouts:P(ins)` Evaluate the expressions correponding to in parameters "ins", then call procedure "P" with the "out", "inout", and the values of the "ins" parameters. Fails if any function called by the "ins" expressions fails, or if "P" fails.

`{ A }` Execute "A"; braces are just statement parentheses.

`VAR vlist;` where "vlist" is a comma-separated list of expressions Global Variables of the form

`v := E` where "v" is an identifier (the name of the global variable being declared) and "E" is an expression (the initial value for the variable).

`VAR vl IN A END` Introduce local variables "vl" and execute "A". Local Varaibles

`PROC outs := inouts:P(ins) IS B END;` Procedures

where "P" is an identifier, the name of the procedure, outs, inouts, and ins specify the parameters of various types, and "B" is a command, the body of the procedure.

If there are no out parameters, "outs :=" must be omitted.

If there are no inout parameters, "inouts :"" must be omitted.

If there are no in parameters, "ins" must be empty, but the parentheses are still required.

Both "ins" and "inouts" are comma-separated lists of variables, but "inouts" is either a single variable, or a parenthesized comma-separated list of at least two variables.

Expressions

Here are the operators allowed in LIM expressions, listed in groups. Operators within a any group bind more tightly than those in the groups above them; operators in the same group have the same binding power and associate to the left.

`e OR f` logical disjunction; "f" is not evaluated if "e" is non-zero

`e AND f` logical conjunction; "f" is not evaluated if "e" is zero

`NOT e` logical negation

`e = f` equals

`e # f` differs

`e < f` less than

`e > f` greater than

`e <= f` at most

`e >= f` at least

`e + f` sum

`e * f` product

`e DIV f` the floor of the real quotient of "`e`" and "`f`"

`e MOD f` `e - f * (e DIV f)`

`-e` unary minus

The boolean operations produce 1 for true, 0 for false.

The "`DIV`" and "`MOD`" operations are the same as those of Modula-3 and Oberon.

Expressions can also have the forms:

`inouts:P(ins)` procedure call

`v` The value of the variable "v".

`lit` An integer, string, or character literal.

`(e)` round parentheses for grouping

A procedure can be called in an expression if it has exactly one out parameter. The syntactic rule for inouts and for ins are the same as those previously-described for a procedure call used as a command, as are the rules for binding the in and inout parameters. However, after executing the body of the procedure, the value of its out parameter becomes the value of the expression.

Integer literal

An integer literal is a non-empty sequence of decimal digits; it is interpreted base ten.

Character literal

A character literal is a single printing character or escape sequence enclosed in single quotes. The escape sequences allowed are

`\\` backslash (\)

`\t` tab

`\n` newline

| | |
|---|---|
| `\f` | form feed |
| `\r` | return |
| `\s` | blank space |
| `\b` | backspace |
| `\v` | vertical tab |
| `\e` | escape |
| `\'` | single quote |
| `\"` | double quote |
| `\ddd` | char with octal code `ddd` |
| `\xhh` | char with hex code `hh` |

# Index