Kapitel 14
Datenstrukturen und ASN.1

Wir haben schon mehrfach im Zusammenhang mit anwendungsübergreifendem Datenaustausch auf die Notwendigkeit hingewiesen, Daten in einem Datenstrom durch Aufprägen einer Struktur sicher lesbar zu machen, und dabei als binäres Gegenstück zu der XML-Kodierung, die an die Textform gebunden sind, auf die Struktursprache ASN.1 hingewiesen. ASN.1 (*abstract syntax notation one*) ist eine recht komplexe Sprache, die für unterschiedliche Zwecke eingesetzt werden kann, was rechtfertigt, dass wir ihr hier ein eigenes Kapitel widmen und sie nach als Anhängsel eines anderen Themas abhandeln. Die Sprachsyntax ist in den ITU–Normen X.680 ff. beschrieben,[1] die als Referenz bei aktiver Tätigkeit auf diesem Gebiet – sei es durch die Umsetzung bestehender Protokolle in eigenen Anwendungen oder durch Entwicklung eigener Protokolle – unbedingt herangezogen werden sollten. Wie bei anderen Themen zuvor werden wir hier einen Einblick in die Grundmechanismen schaffen, der für das Verständnis der meisten ASN.1–Anwendungen ausreichend ist.

Was ist oder besser was beinhaltet ASN.1? Etwas salopp ausgedrückt handelt es sich um eine Sprache zur Beschreibung von Datenstrukturen. Im Unterschied zu einer Programmiersprache, in der ja auch festgelegt wird, was mit den Daten zu geschehen hat, hört ASN.1 bei der „Programmierung" der Struktur auf. Diese Beschränkung ist nun nicht so zu verstehen, dass ASN.1 eine Untermenge einer Programmiersprache ist, wie das bei der IDL (*interface definition language*) in CORBA der Fall ist. Obwohl meist in diesem Sinne eingesetzt, besitzt ASN.1 eine vollständige Sprachsyntax, die im die Definition einfacher eigener Datentypen erlaubt, aber wie echte Sprachen auch selbstproduzierend ist, also Regeln besitzt, um aus sich heraus gültige Zeichenfolgen zu erzeugen.[2] Zusätzlich zu Syntax enthält ASN.1 auch eindeutige Regeln, wie die durch die Struktur beschriebenen Daten Bit für Bit auf einen Binärdatenstrom zu übertragen sind. Hier existieren aber, wie wir sehen werden, zwangsweise einige einschränkende verbindliche Regelungen, damit die Daten transparent kodiert werden können.

---

[1] Die Normen sind nicht ganz billig, jedoch besteht die Möglichkeit für private Nutzer, einige Dokumente kostenlos zu beziehen. Ein Blick auf die Internetseiten der ITU lohnt sich daher.

[2] Für tiefere theoretische Hintergründe zu den Begriffen „Syntax" usw. muss ich Sie auf einführende Lehrbücher der Informatik oder Linguistik verweisen.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_14, © Springer-Verlag Berlin Heidelberg 2010

Daraus ergeben sich nun eine Reihe unterschiedlicher Einsatzmöglichkeiten (*wobei wir auch gleich die Nomenklatur festlegen*):

- Mit Hilfe eines ASN.1–Kodes und den Kodierungsvorschriften für Text- oder Binärdaten der X.690 können Anwendungen programmiert werden, die Daten strukturiert auf eine Datenstrom schreiben oder kontrolliert zurücklesen.

  Als ASN.1–Code bezeichnen wir die Strukturbeschreibung in der ASN.1–Sprache, bestehend aus Typvereinbarungen und Variablendeklarationen.[3]
- Ein ASN.1–Code (*eingeschränkt eigentlich daraus nur die Typvereinbarungen*) kann mit Hilfe eines „ASN.1–Compilers“ in ein Klassenmodell einer Programmiersprache übersetzt werden, das die Schreib- und Leseeigenschaften bereits beinhaltet. Der „Rest“ der Anwendung wird „drumherum“ programmiert.
- Ein ASN.1–Code kann mittels einer Klassenbibliothek in eine ASN.1–Variablenliste umgesetzt werden, mit der direkt Datenströme bearbeitet werden können. Unter einer ASN.1–Variablenliste soll ein Programmteil verstanden werden, der in der Lage ist, ASN.1–Textdaten oder ASN.1–Binärdaten zu verstehen, aber noch keine direkte Anwendung darstellt wie das bearbeitete Kompilat des letzten Spiegelstrichs.
- Mittels einer geeigneten Erweiterung einer Variablenliste kann die Struktur eines Datenstroms beziehungsweise ein markanter Teil davon analysiert und mit in einer Datenbank gespeicherten Beschreibungen verglichen werden. Aufgrund dieser Information kann die zur Interpretation der Daten geeignete Anwendung zu laden.
- ... (*weitere Anwendungsfälle finden Sie sicher selbst*)

14.1 Einführung in die Syntax

Es ist nicht ganz einfach, in sehr knapper Form in eine Sprache einzuführen. Die Reihenfolge der Begriffseinführung mag im folgenden etwas eigenwillig erscheinen, sollte Ihnen aber aufgrund der Kenntnisse von Programmiersprachen einen problemlosen Einstieg ermöglichen. ASN.1 verfügt zunächst wie alle Programmiersprachen über einen Vorrat an einfachen Datentypen (*auch primitive Datentypen genannt*), auf die letztlich (*fast*) alles zurückgeführt wird:

```
BIT, BOOLEAN, CHARACTER, INTEGER, OCTET, REAL,
STRING (mit einer Anzahl von Spezialisierungen)
```

Zur Speicherung von Daten werden Variablen bestimmten Typs angelegt und optional kommentiert

```
artNr INTEGER        -- Artikelummer
vorhanden BOOLEAN    -- Artikel vorrätig
```

[3] Das ist genau das gleiche wie in Programmiersprachen: Typbeschreibungen legen fest, was wir theoretisch machen können, aber erst Variablen schaffen den benötigten Speicherplatz.

Damit sind schon einige Sprachregeln offen gelegt:

- `Alle Datentypenbezeichnungen beginnen mit einem Großbuchstaben, der Rest ist bis auf einige Steuerzeichen beliebig,`
- `alle Variablenbezeichnungen beginnen mit einem Kleinbuchstaben,`
- `Kommentare beginnen mit -- und kennzeichnen den Rest der Zeile als Kommentar`

- Die Begriffe müssen in einer festgelegten Reihenfolge auftreten, hier beispielsweise *Variablenbezeichnung – Typbezeichnung*., also einen durch die Syntaxregeln festgelegten Kontext einhalten.

Eigene Datentypen können in Form eines C–ähnlichen `typedef` definiert werden. Die Syntax einer Typvereinbarung ist die gleiche wie für die Zuweisung von Werten zu Variablen. Typen werden dabei andere Typen, Variablen andere Variablen oder Konstante zugewiesen. Das obige Beispiel lässt sich durch Definition eines eigenen Datentyps für eine Artikelnummern und die zusätzliche Vergabe einer Nummer dann auch so formulieren

```
ArtNr ::= INTEGER
artNr ArtNr ::= 4711
```

Beachten Sie, dass die Bedeutung der einzelnen Worte in einem Satz (*als Worte sollen dabei auch Zeichenfolgen wie* ::= *verstanden werden; ein Satz ist wie üblich eine sinnmäßig abgeschlossene Folge von Worten*) ähnlich wie bei einer gesprochenen Sprache stark kontextabhängig ist. In C/C++ sind die Typ- und Wertzuweisungen symbolmäßig sehr deutlich getrennt. Außerdem ist die Reihenfolge der Zeilen im ASN.1–Code im Gegensatz zu den meisten Programmiersprachen egal. Der Typ `ArtNr` darf in einer Variablendeklaration auftreten, bevor er definiert wird, das heißt die beiden Zeilen können ausgetauscht werden, ohne das dies wie bei einem C–Compiler zu einer Fehlermeldung führt (*die Festlegungen müssen natürlich eindeutig sein.* `artNr ArtNr ::= REAL ::= 4711` *anstelle der zweiten Zeile würde dem Typ* `ArtNr` *einen anderen Grundtyp zuweisen als die erste Zeile, was natürlich bei einer Auswertung nicht akzeptiert werden dart*) . Bei der Auswertung von Wortfolgen kommt es nur auf die Reihenfolge an, aber nicht auf eine bestimmte Position in einer Zeile. Trennzeichen wie das Semikolon in C/C++ werden nur an bestimmten Stellen verwendet (*in den hier vorgestellten Sätzen noch nicht*):

```
ArtNr ::= 4711                  -- ungültig, keine Variable
ArtNr ::= INTEGER ::= 4711      -- zur Aufnahme des Wertes

artNr INTEGER    artNr ::= 4711  -- Zuweisung gültig
```

Wie in anderen Programmiersprachen wird man aber von einem professionellen ASN.1–Code eine Kontextbezogene Textstruktur in Form von Zeilenumbrüchen und Einrückungen erwarten, wie wir das von C/C++ gewohnt sind.

Vielfach ist es notwendig, den Wertebereich einer Variablen zu begrenzen. Dies erfolgt durch Begrenzung des Bereiches im Typ

```
ArtNr ::= INTEGER { 4710 .. 4712, 5513 }
```

Die Variable `artNr` darf dann nur die festgelegten Werte annehmen, das heißt die im Beispiel erfolgte Zuweisung ist gültig, eine Zuweisung der Art `artNr ::= 6909` ist unzulässig und bei der Auswertung durch einen Compiler durch eine Fehlermeldung zu ahnden.

- Durch { } wird eine Liste gekennzeichnet,
- durch .. ein Bereich von Werten,
- durch, werden verschiedene diskrete Werte einer Liste getrennt. In Aufzählungen wird also ein Trennzeichen eingesetzt, was kontextmäßig auch Sinn macht, wie bei der Auswertung von ASN.1–Kodes festgestellt werden kann.

Es ist wichtig, hier zwischen möglichen Bereichen für die Werte einer Variablen und der Zuweisung eines konkreten Wertes zu unterscheiden, da im ASN.1–Code beides vermischt werden kann. In

```
artNr ArtNr ::= INTEGER { 4710 .. 4712, 5513 }
```

kann die deklarierte Variable Werte in dem angegebenen Bereich annehmen, der Inhalt der Variablen ist aber nicht definiert (*möglicherweise wird bei dieser Stufe der Auswertung noch gar kein Speicher hierfür zur Verfügung gestellt*). Die Zuweisung (*wieder eine weitere Kontexmöglichkeit*)

```
artNr { 4711 }
```

legt nun definitiv den angegebenen Wert ab (*falls er zulässig ist*). Im Sinne einer geordneten Nomenklatur legen wir die Zuweisungsmöglichkeiten fest:

(a) Zuweisungen der Art `artNr ::= 4711` legen im ASN.1–Code eine Konstante fest oder führen eine Vorbelegung von Variablen für den Zeitpunkt der Deklaration durch.
(b) `artNr{4711}` stellt den Inhalt der Variablen in ASN.1–Textdaten dar, das heißt von einer Variablenliste werden solche Daten gelesen oder geschrieben. Auch sie müssen natürlich innerhalb der festgelegten Bereiche liegen.
(c) Alternativ zu den ASN.1–Textdaten können Variablenlisten auch Binärdaten verarbeiten, die später diskutiert werden. Auch für sie gelten die Bereichsregeln.

Derartige Zeilen sind aber im Grunde bereits das Ergebnis der Anwendung einer Syntax, die abstrakter beschrieben werden kann, indem man in der Typdefinition die Auswahl aus mehreren anderen Typen zulässt. Verteilen wir die Bereiche auf zwei Untertypen, so ist der obigen Definition äquivalent

```
A ::= INTEGER { 4710 .. 4712 }
B ::= INTEGER { 5513 }
ArtNr ::=  A | B
```

Das Symbol | steht für „oder“. Diese mehrstufige Definition eines Begriffs wird als „Produktion“ eines Types bezeichnet und darf auch rekursiv fortgesetzt werden, wie das folgende Beispiel verdeutlicht:

```
A ::= CHARACTER { "A" }
B ::= CHARACTER { "B" }
C ::= D | empty
D ::= A, B | A, B, D    -- rekursive Definition
c C                    -- Variablenvereinbarung
```

Aufgrund dieser „Produktion“ darf die Variable C folgende Werte aufweisen:

```
""
"AB"
"ABAB"
"ABABAB"
...
```

Solch ein Buchstabenbeispiel für die mögliche Belegung einer Variablen ist zwar demonstrativ, aber nicht unbedingt sehr praxisbezogen. Deshalb hier eine andere Möglichkeit, die mehr den von uns behandelten Themen entspricht: Verbirgt sich hinter dem Datentyp A ein ganzzahliger Indexwert, hinter B ein Fließkommawert, so könnte C auch den Inhalt einer schwach besetzten Matrix darstellen. Bei der Datenübertragung werden nur die besetzten Felder mitsamt ihres Inhaltes übertragen.

Konstante können auch durch einen Bezeichner abgekürzt werden. Die Auflösungsregeln besagen dabei, dass rekursiv über die zu den Listen gehörenden Typen eingesetzt und bei Erfolg abgebrochen wird. Das lässt sich am besten durch ein beim ersten Lesen sicher verwirrendes Beispiel erläutern:

```
a INTEGER ::= 1
T ::= INTEGER { a(3), b(a) }
c T ::= a         -- c hat den Wert 3   (Zuweisung 1)
d T ::= b         -- d hat den Wert 1   (Zuweisung 2)
```

Bei der Zuweisung 1 (*Zeile 3*) wird eine Variable vom Typ `T` deklariert und ihr der Wert `a` zugewiesen. Der Compiler sucht dazu zunächst die Definition des Typs `T` und findet dort in der Bereichsliste bereits den Bezeichner `a` . Die Rekursion endet daher bei `a(3)` und wird nicht bis in die erste Zeile fortgesetzt. An dieser Stelle wird dem Platzhalter `a` und damit auch der Variablen `C` der Wert 3 zugewiesen.

Bei der Zuweisung 2 (*Zeile 4*) steht Platzhalter `b` aber stellvertretend für den Bezeichner `a` . Die Deklaration `a(3)` ist aber nun nicht nutzbar, da sie auf der gleichen Stufe mit `b` steht, und die Auswertung wird fortgesetzt. `a` ist als Variable vom richtigen Typ in der ersten Zeile deklariert, so dass die Rekursion bis hier

durchläuft. Der Variablen `d` wird fortan der Wert der Variablen `a` zugewiesen, in diesem Fall der Wert 1 .

Als Zwischenbilanz können wir feststellen, dass uns die Textform von ASN.1 die Möglichkeit gibt, Strukturen (= *Typen und Variablen*), Strukturen und Daten (=*Typen und Variablen mit Wertzuweisungen*) oder nur Daten (= *Variablen mit Wertzuweisungen, für deren Interpretation aber eine Struktur notwendig ist*) zu kodieren. In der Binärkodierung sind aber nur noch Daten vorhanden; Strukturdefinitionen werden nicht in eine äquivalente Binärform transformiert. Nun haben wir bislang einen einfachen Typ durch einen eigenen Bezeichner substituiert und können dadurch auf Textebene ein hohes Maß an Ordnung schaffen. Es ist natürlich wünschenswert, das später auch in der Binärkodierung wiederfinden zu können, das heißt eine Variable des Typs `INTEGER` von einer Variablen des Typs `ArtNr` unterscheiden und beispielsweise die Werte auf Einhaltung der festgelegten Bereiche überprüfen zu können. Das ist durch Zuordnung zu einer von vier Klassen oder Vergabe von speziellen Typkennziffern (*im Neudeutschen „ tag“*) oder einer Kombination aus beidem möglich. Im folgenden Beispiel wird einem ganzzahligen Typ eine andere Klasse zugeordnet (*die Typkennziffer 2 ist dem Typ* `INTEGER` *als Standard zugeordnet*).

```
ArtNr ::= [PRIVATE 2] INTEGER

oder als abstrakte Produktion:

ArtNr ::= TaggedType
TaggedType ::= Tag Type
Type ::= INTEGER | ...[4]

Tag ::= "[" Class ClassNumber "]"
ClassNumber ::= number | DefinedValue
Class ::= UNIVERSAL | APPLICATION | PRIVATE | empty
```

In dieser Produktion ist nur `ArtNr` ein von uns eingeführter Begriff, alles andere sind in X.680 erklärte Begriffe und gehören damit gewissermaßen zum Sprachstandart. Der Begriff `DefinedValue` in `ClassNumber` ist mit `a,b,c,d` im Beispiel davor erklärt und meint einfach eine Konstante. Diese Festlegung ist schon bemerkenswert, wenn man bislang zwar programmiert, sich aber noch nicht mit der Theorie von Sprachen auseinander gesetzt hat: Es wird ein neues Syntaxelement eingeführt, das auch zu speziellen Kodierungsvorschriften im Binärbereich führt, und zwar mit den Sprachmitteln der Sprache selbst. Die Aussage der Produktion ist die von uns gewünschte Spezifizierung auf Binärebene: Ein Integer bleibt ein Integer, die Zuordnung zu einer anderen Klasse oder die Deklaration einer anderen Klassennummer erlaubt aber eine Differenzierung zwischen verschiedenen Integer–Typen.

---

[4] Zu unterscheiden ist „`..`“ von „ `...`“. Zwei Punkte bezeichnen einen Bereich, drei Punkte wie hier eine nicht näher festgelegte, aber meist klare Fortsetzung. Der Unterschied ist wichtig und in X.680 näher beschrieben.

Für die Differenzierung ist aber wie bei Texten mit reinen Datenzuweisungen eine Strukturbeschreibung notwendig.

Wie oben schon angesprochen, sind einige feste Vereinbarungen für die Binärkodierung von Daten notwendig. Wie wir noch darlegen werden, sind die Typkennziffern `0..31` der Klasse `UNIVERSAL` fest mit bestimmten Kodierungsvorschriften liiert. Das genügt, um beliebige Daten bei Vorliegen des zugehörenden ASN.1–Kodes korrekt interpretieren zu können.[5]

Nun sind Daten meist nicht nur eine Aneinanderreihung einfacher Datentypen, sondern in größeren Einheiten strukturiert. Komplexere zusammengesetzte Typen können in ASN.1 auf mehrere Arten gebildet werden. In der ersten Produktion haben wir bereits eine Möglichkeit kennen gelernt; der zusammengesetzte Typ wird aber auf eine recht umständliche Weise gebildet. ASN.1 stellt deswegen einige weitere Standardttypen bereit, die sich in der Praxis besser einsetzen lassen.[6] Der `SEQUENCE` –Typ entspricht etwa dem `struct` in C:

```
Adresse ::= SEQUENCE {
     name     VisibleString,
     strasse  VisibleString,
     ort      VisibleString
}   -- Ende des Adresstyps
```

Innerhalb der Aufzählung der `SEQUENCE` sind nur Variablenname zulässig, keine Typdefinitionen. Es handelt sich im die Attribute der Datenstrukturen, deren Bezeichnungen (*wie üblich*) nur innerhalb der Aufzählung Gültigkeit besitzen und deshalb auch an anderen Stellen im Code wieder benutzt werden dürfen, ohne dass es zu Konflikten kommt. Auch die Dereferenzierung entspricht der C/C++–Notation:

```
adr { name ::= "Fred" }   -- oder
adr.name ::= "Fred"
```

Wesentlich ist die Sortierung der Elemente in einer `SEQUENCE`. Es müssen Werte der angegebenen Typen in der angegebenen Reihenfolge auftreten, wobei die die Typen selbst natürlich ihrerseits wieder komplex sein dürfen. Um eine gewisse Bandbreite bei den Daten zuzulassen, sind zusätzliche Kennungen definiert:

```
PersData ::= SEQUENCE {
    adr   Adresse,
    tel   VisibleString OPTIONAL,
    verh  BOOLEAN DEFAULT TRUE
}   -- Ende
```

---

[5] Natürlich ist auch ein Schlupfloch für Geheimniskrämer vorhanden, auf das wir aber nicht weiter eingehen werden.

[6] Auch sie lassen sich natürlich abstrakt durch Produktionen erklären.

Mit OPTIONAL gekennzeichnete Einträge dürfen in Datendokumenten auch fehlen, bei mit DEFAULT gekennzeichneten Einträgen wird bei Fehlen des Eintrags im Datendokument automatisch der angegebene Wert ergänzt.

Ist die Reihenfolge der Einträge nicht festgelegt, so ist an Stelle des Schlüsselworts SEQUENCE das Schlüsselwort SET zu verwenden. Ein SET–Typ unterscheidet sich vom SEQUENCE–Typ nur dadurch, dass die Einträge im Bitstream eine beliebige Reihenfolge besitzen dürfen. Sie müssen aber mit den gleichen Einschränkungen wie bei SEQUENCE alle auftreten.

Zwischen Feldern und Strukturen wird nicht besonders unterschieden. Felder von Typen werden als SEQUENCE oder SET betrachtet, in der die spezifizierten Attribute mehrfach hintereinander auftritt und die einzelnen Instanzen vereinbarungsgemäß abgezählt werden. Diese Auffassung von einem Feld ist im Vergleich mit der sonstigen strengen Vorgehensweise etwas inkonsequent und wird durch die Erweiterungen

```
Adressliste ::= SEQUENCE OF Adresse
Adresstabelle ::= SEQUENCE (SIZE(1..100)) OF Adresse
PAListe ::= SET OF { PersData, Adresse }
```

realisiert. Die Anzahl der Einträge bei der ersten und der dritten Variante ergibt sich implizit aus der Länge des Bitstreams und kann beliebig sein. In der zweiten Variante muss die Längenangabe eingehalten werden. Der Parametersatz von SIZE lässt so ziemlich wieder alles zu, was wir bisher über Inhalte wissen, also Bereiche, Aufzählungen, Variable (*diesmal echte Variable, also Bezüge auf externe Größen, s.u.*) usw. Kontextmäßig müssen wir somit bei der Auswertung von SEQUENCE oderSET mit mindestens drei Möglichkeiten rechnen.

Ist neben der Reihenfolge der Einträge auch der Typ der Einträge nicht genau festzulegen, so lassen sich die möglichen Belegungen mit dem Typ CHOICE angeben:

```
TelNr ::= CHOICE {
     text  VisibleString,
     int   INTEGER
}    -- Ende
```

Im Datendokument wird hierdurch festgelegt, dass das zu diesem Typ gehörende Datenfeld eine Telefonnummer enthält, jedoch kann diese als ASCII–String oder als ganze Zahl angegeben sein. Die CHOICE–Kodierung in ASN.1–Daten weicht von der Kodierung anderer Daten ab:
Während beiSEQUENCE/SET der Variablenname mit den Attributen als Liste zwingend auftritt (*und das Äquivalent dazu im binären Datensatz*), ist das bei CHOICE–Typen in ASN.1–Textdaten allenfalls noch ein Kann, in ASN.1–Binärdaten gar nicht vorgesehen, d.h dort tritt nur noch das Attribut in Erscheinung. Um die ASN.1–Binärdaten korrekt interpretieren zu können, sind deshalb bei CHOICE–Typen eindeutige Typkennzeichnungen zu vergeben und die Klasse ist auf den Typ CONTEXT festgelegt. Zwei verschiedene INTEGER–Werte sind daher zu spezifizieren durch die Typkennziffern (*die Klasse liegt implizit fest*)

```
ASN.1 - code:
    person Persdata

ASN.1 - Textdaten:
    person{adr
           {".."),...}
```

```
telNr TelNr

telNr{int{15}} -- oder
telNr.int{15} -- oder
int{15} -- falls
-- eindeutig
```

```
A ::= CHOICE {
    i1 [0] INTEGER,
    i2 [1] INTEGER
}   --  End
```

Das zieht unter Umständen noch weitere Kreise: Treten in einem `SET`–Typ unterschiedliche `CHOICE`–Attribute auf, so müssen sich diese in den Typkennziffern unterscheiden, da sonst aufgrund der bei `SET` nicht festgelegten Attributreihenfolge nicht zu entscheiden ist, welches Datum wo kodiert ist. `CHOICE` stellt also einen Sonderfall in der Kodierung dar und ist deswegen deutlich von `SEQUENCE` oder `SET` abzugrenzen. Auch rekursive `CHOICE`–Definitionen verhalten sich anders als die anderen rekursiven Typdefinitionen. Die Festlegungen für den `CHOICE`–Typ sind ähnlich wie die recht lasche Unterscheidung von `SET, SET OF` und `SET (SIZE..` sprachlich nicht sehr konsequent.[7]

Mit Hilfe dieser Sprachelemente sollte eigentlich jede notwendige Datenstruktur zu erzeugen sein. Will ein Programmierer aus irgendeinem Grunde bestimmte Datenstrukturen in seiner ASN.1–Beschreibung nicht offen legen (*oder ist er zu bequem, zu einer bestehenden Datenstruktur ein ASN.1–Korsett zu entwerfen*), so kann er den Datentyp `EXTERNAL` verwenden, das heißt was sich später im Bitstream an dieser Stelle befindet, liegt ausschließlich in seinem Kontrollbereich.

Sehr wichtig für die universelle Einsetzbarkeit ist der spezielle Datentyp `OBJECT IDENTIFIER`, dessen Details im nächsten Teilkapitel eingehender diskutiert werden. Mit seiner Hilfe werden Protokolle und Datensätze verbindlich gekennzeichnet. Für universelle Protokolle, also solche, die nicht nur privat von einem Unternehmen genutzt werden, werden Werte für `OBJECT IDENTIFIER` von einer zentralen Organisation verwaltet. An jeden Eintrag ist eine komplette ASN.1–Kodierung eines bestimmten Protokolls oder Datensatzes gebunden. Er stellt damit die Schnittstelle für die Softwareentwickler unterschiedlicher Unternehmen dar. Will ein Softwareunternehmen an einer bestimmten Form des Datenaustausches mit seinen Produkten teilnehmen, so muss es nur dafür sorgen, dass der beschriebene ASN.1–Code unterstützt wird. Sollen zwischen zwei Anwendungen Daten

[7] Vom rein theoretischen Standpunkt aus wären vermutlich sogar Bezeichnungen wie „missglückt“ nicht ganz ungerechtfertigt. Wie bei natürlichen Sprachen darf man aber auch ASN.1 eine gewisse inhomogene Entwicklungshistorie unterstellen, die zwangsweise ein paar nicht so ganz geglückte Konstrukte mit sich bringen. Die Praxis relativiert das Ganze dadurch weiter, dass problematische Konstrukte dort zu selten auftreten, um allgemeine Probleme zu verursachen.

ausgetauscht werden, so beginnt die Sitzung mit dem Austausch von Objektkennzeichnern, und die gewünschte Kommunikation ist möglich, wenn beide eine Übereinstimmung in den unterstützten Protokollen feststellen.

> **Aufgabe.** Das war nun der Versuch, auf wenigen Seiten in die Syntax von ASN.1 einzuführen. X.680 benötigt dazu immerhin ca. 90 Seiten, Lehrbücher über ASN.1 bringen es auf ähnliche Seitenzahlen wie dieses Buch. Ich habe mich hier bemüht, diejenigen Sprachelemente darzustellen, die für die Bearbeitung der meisten Praxisanwendungen ausreichen. ASN.1 beinhaltet noch eine ganze Reihe anderer Sprachelemente, allerdings muss man auch die entsprechenden Anwendungen haben, um diese Eigenschaften nutzen zu können (*und natürlich auch einen ASN.1–Compiler, der das alles versteht*).

Die direkt definierten Typen und ihr Kontext sind in der X.680 „ab initio“ mit den Sprachmitteln von ASN.1 selbst definiert und nicht wie oben mehr oder weniger an Beispielen und Analogien. Hierdurch entsteht die große Variabilität bei eindeutiger Syntax, allerdings ist hier möglicherweise auch die Inkonsequenz bei der Festlegung einzelner zu Sprachelementen erklärter Produktionen zu suchen, indem sich unterschiedliche Arbeitsgruppen über verschieden Produktionen oder Kodierungen hergemacht haben. Zur Überprüfung, wie tief das Verständnis nach der obigen Einführung bereits ist, vollziehen Sie im Detail die „Produktion“ des allgemeinen `SEQUENCE`–Typs nach:[8]

```
SequenceType ::= SEQUENCE "{""}" |
    SEQUENCE "{" ExtensionAndException
        OptionalExtensionMarker "}" |
    SEQUENCE "{" ComponentTypeLists "}"

ExtensionAndException ::= "..." | "..." ExceptionSpec

OptionalExtensionMarker ::= ","... | empty

ComponentTypeLists ::= RootComponentTypeList |
    RootComponentTypeList ","ExtensionAndException

ExtensionAdditions OptionalExtensionMarker |
    RootComponentTypeList "," ExtensionAndException
ExtensionAdditions
    ExtensionEndMarker "," RootComponentTypeList |
    ExtensionAndException
    ExtensionAdditions ExtensionEndMarker ","
    RootComponentTypeList
```

[8] Anmerkung: Falls Ihnen diese Produktion unvollständig erscheint, da die Berücksichtigung von Feldern fehlt, haben Sie vermutlich Recht. In der X.680 sind Felder und ähnliches aber separat als „Constrained Types“ erklärt und nehmen dann wieder Bezug auf SEQUENCE oder SET. Sinnvoller wäre hier sicher eine separate Bezeichnung wie ARRAY gewesen, aber ich war halt nicht dabei, um so was zu verhindern ;-) Ehrlicherweise muss man allerdings zugestehen, dass sich hinter „Constrained Types“ mehr verbirgt als einfache Felder, wie wir sie aus C/C++ kennen.

```
RootComponentTypeList ::= ComponentTypeList
ExtensionEndMarker ::= "," "..."

ExtensionAdditions ::= "," ExtensionAdditionList |
                           empty

ExtensionAdditionList ::= ExtensionAddition |
    ExtensionAdditionList "," ExtensionAddition

ExtensionAddition ::= ComponentType |
                      ExtensionAdditionGroup

ExtensionAdditionGroup ::=
                  "[[" ComponentTypeList "]]"

ComponentTypeList ::=
    ComponentType |
    ComponentTypeList "," ComponentType

ComponentType ::=
    NamedType |
    NamedType OPTIONAL |
    NamedType DEFAULT Value |
                       COMPONENTS OF Type
```

Anmerkung. `ExceptionSpec` ist nicht weiter aufgeschlüsselt. Zu unterscheiden ist zwischen „`..`“ und „`...`“. Die erste Form kennzeichnet einen Bereich (*2..10*), die zweite Form steht für eine beliebige Zeichenkette ohne Leerzeichen. Zwischen Hochkommas eingeschlossene Zeichen sind direkt in den ASN.1–Code zu übernehmen.

Aufgabe. Besorgen Sie sich aus dem Internet einige RFC–Dokumtene (*Protokollbeschreibungen im Internet, beispielsweise RFC 2459*) sowie PKCS#–Dokumente und interpretieren Sie einige der ASN.1–Beschreibungen zur Übung.

Ähnlich C/C++ bietet ASN.1 konsequenterweise die Möglichkeit, zur weiteren Strukturierung Module mit Strukturbeschreibungen zu definieren und festzulegen, was daraus in andere Module exportierbar ist und was aus anderen Modulen importiert wird. Ich gehe an dieser Stelle nicht weiter darauf ein und verweise auf die X.680.

14.2 Binärkodierung

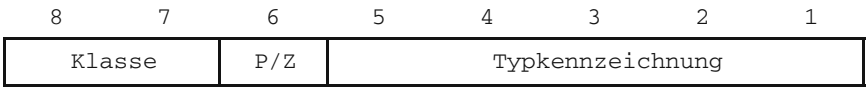
Im letzten Kapitel und der Norm X.680 wird die Syntax von ASN.1 erklärt und verschiedene Produktionen von Begriffen zu Sprachelementen erklärt. In der Norm X.690 wird nun festgelegt, wie diese Standardproduktionen im Binärformat auszusehen haben. Beginnen wir auch dieses Kapitel mit der Praxis: Bei der Kodierung

von Binärdaten ist zwischen Blockkodierung und Stromkodierung zu unterscheiden. Bei der Blockkodierung liegt bereits zu Beginn des Kodierungsprozesses fest, welche Datenmenge umgesetzt wird, so dass die Blockgröße als Steuergröße zu Beginn des Datenblockes angegeben werden kann. Bei der Stromkodierung ist zu dem Zeitpunkt, zu dem der Beginn eines Datensatzes kodiert wird, noch nicht abzusehen ist, wie viele Daten noch folgen, so dass die Länge noch nicht angegeben werden kann. Das Ende eines Datenstroms wird deshalb durch spezielle Steuersequenzen angezeigt.[9]

Blockkodierung:

Stromkodierung:

Beide Kodierungstypen sind in ASN.1 vorgesehen, wobei aber in der Praxis überwiegend die Blockkodierung zum Einsatz kommt. Ein ASN.1–Binärdatensatz besitzt zu Beginn zwei Steuerfelder für die Kennzeichnung des Datentyps und der Satzlänge. Jedes der Steuerfelder kann aus einem oder mehreren Bytes bestehen. Das Datentypfeld enthält drei verschiedene Informationen: die Datenklasse und die Typnummer sowie eine Kennzeichnung, ob es sich um einen einfachen Datentyp oder eine Struktur handelt. Es besteht mindestens aus einem Byte



Die Klassennummer kann einen der vier Werte annehmen, die wir im letzten Kapitel spezifiziert haben und von denen nur drei vom ASN.1–Programmierer eigenständig bedient werden können.

Klassen	*Bitkodierung*
Universal	0 0
Anwendung	0 1
Kontextspezifisch	1 0
Privat	1 1

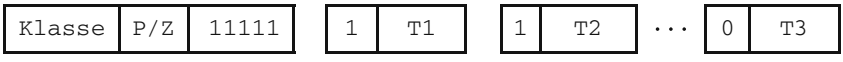
[9] Ein C–String ist eine typische Stromkodierung, da nach dem Abschlußzeichen „\0“ gesucht werden muss. PASCAL–Strings führen die Länge in der ersten Position und sind somit blockkodiert.

Für die als Standard eingesetzte Klasse UNIVERSAL sind die Typkennzeichnungen von 31 Standardklassen mit einem bestimmten Datentyp und einer bestimmten Kodierung festgelegt:

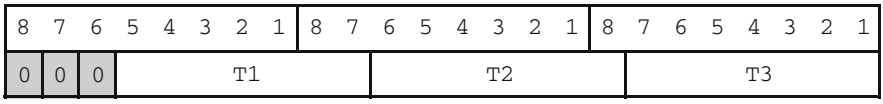
```
UNIVERSAL 0      Reserved for use by encoding rules
UNIVERSAL 1      Boolean type
UNIVERSAL 2      Integer type
UNIVERSAL 3      Bitstring type
UNIVERSAL 4      Octetstring type
UNIVERSAL 5      Null type
UNIVERSAL 6      Object identifier type
UNIVERSAL 7      Object descriptor type
UNIVERSAL 8      External type and Instance-of type
UNIVERSAL 9      Real type
UNIVERSAL 10     Enumerated type
UNIVERSAL 11     Embedded-pdv type
UNIVERSAL 12     UTF8String type
UNIVERSAL 13-15  Reserved for future editions of
                 this Recommendation | International
                 Standard
UNIVERSAL 16     Sequence and Sequence-of types
UNIVERSAL 17     Set and Set-of types
UNIVERSAL 18-22, 25-30 Character string types
UNIVERSAL 23-24 Time types
```

Wie schon oben angesprochen, ist kein Typkennzeichen für den Typ CHOICE vorhanden; CHOICE–Attribute beziehungsweise Variablen werden mittels der hierfür reservierten Klassennummer CONTEXT gekennzeichnet. Wird eine andere Klasse als UNIVERSAL verwendet, so haben die Typkennziffern nichts mehr mit der obigen Liste der Kodierungen zu tun. APPLICATION 2 kann also alles mögliche sein und muss nichts mit einem INTEGER zu tun haben. Was sich hinter dem Typ verbirgt, kann nur mit Hilfe des ursprünglichen ASN.1–Kodes festgestellt werden (*vergleichen Sie auch die Ausführungen zum Typ* CHOICE).

Der 32. Typ (*Kodierung* 1 1 1 1 1) erlaubt die Kodierung weiterer Datentypkennzeichen mit Zahlen $\geq$31 durch Anfügen weiterer Typbytes:



Die Datentypkennungen sind folgendermaßen zu einer Zahl zu kombinieren:



Jeweils sieben Bit stehen somit für die Kodierung zur Verfügung, während das achte Bit als Fortsetzungsbit zu lesen ist. Zu Lesen ist in den Richtungen HÖCHSTES

BYTE/BIT ---> NIEDRIGSTES BYTE/BIT, was für INTEL–gewohnte Leser etwas gewöhnungsbedürftig ist.

Das Datenfeld P/Z unterschiedet zwischen „primitiven“ Datentypen wie `INTEGER` usw. und „zusammengesetzten“ Typen wie `SEQUENCE` oder `SET`. Das Datenfeld solcher Variabler enthält dann wiederum ASN.1–Binärdatensätze, die rekursiv auf die gleiche Weise wie der Hauptdatenstrom ausgewertet werden können.

> **Aufgabe.** Rekursive Stromkodierung muss anders ausgewertet werden als rekursive Blockkodierung. Warum? Entwerfen Sie eine Strategie zur Auswertung von Binärdatensätzen.

Anmerkung. Gemäß Syntaxspezifikationen ist es den ASN.1–Kodeauswertern freigestellt, den `CHOICE`–Attributen Typkennziffern automatisch in eindeutiger Weise innerhalb eines bestimmten Kontextes zuzuordnen, wenn vom Anwender nichts anderes angegeben ist. Das vereinfacht zwar die Kennziffernvergabe für den Anwender, aber wehe, er nutzt dann die Möglichkeit, die Reihenfolge von Zeilen im Code zu ändern! `CHOICE`–Typkennzeichen sollten daher immer manuell vergeben werden.

Für die Längenkodierung stehen drei Möglichkeiten zur Verfügung. Für kurze Datenblöcke bis zu 127 Byte genügt ein Längenbyte zur Kodierung

Für darüber hinaus gehende Datenmengen wird im ersten Byte das höchste Bit=1 gesetzt und in den restlichen Bits die Anzahl der Längenbytes kodiert, die dann wie oben, aber in voller 8–Bit–Breite, zusammengesetzt werden. Die Kodierung für 201 Datenbytes besteht beispielsweise aus zwei Bytes mit dem Inhalt

Für Daten mit zunächst unbestimmter Länge (*Stromchiffrierung*) ist eine EOD–Kodierung aus zwei aufeinander folgenden Nullbytes vorgesehen:

Eine solche Kombination kann in primitiven Datentypen natürlich ebenfalls auftreten, zum Beispiel einem Bildinhalt als `OCTET STRING`. Die unbestimmte Längenkodierung ist daher auf Typen oder Variablen zu beschränken, bei denen keine Probleme auftreten können.[10]

> **Aufgabe.** Geben Sie die Kodierung eines `OCTET STRING` (*entspricht einem Binärpuffer*) der Länge 3.127 Byte an.

[10]Der „Nulltyp“ besitzt sogar eine besondere Kodierung (*siehe Tabelle*). Besondere Auswertungsprobleme sind aber nicht zu erwarten, vergleiche Fehler: Referenz nicht gefunden.

Bei der Datenkodierung ist die Einhaltung bestimmter Konventionen insbesondere bei Zahlentypen notwendig. Wir können hier nicht auf alle Datentypen Details eingehen, deshalb nur einige Beispiele (*eine erschöpfende Darstellung finden Sie in der X.690*):

- `BOOLEAN`: Der Wert Null ist als `false` zu betrachten, jeder andere Wert als `true`.
- `INTEGER`: Die Länge von ganzen Zahlen ist nicht begrenzt, das heißt auch lange Zahlen, wie sie für Verschlüsselungszwecke notwendig sind, müssen kodiert werden. Die Übertragung ist mit der geringstmöglichen Anzahl an Bytes durchzuführen, das heißt führende Nullbytes bei Typen mit auf den Maschinen fest eingestellten Wortlängen werden bei der Übertragung abgeschnitten. Außerdem muss die Kodierung das Vorzeichen enthalten, was wie bei dem C–Typ int durch das höchste Bit signalisiert wird. Die ASN.1–Kodierungsregel legt fest, dass die Bits des ersten (*höchsten*) Byte und das höchste Bit des zweiten

♦ nicht alle Null oder
♦ nicht alle Eins

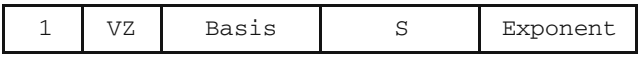
sein dürfen. Dazu wird das Zweierkomplement der Zahlen gebildet, womit auch das Vorzeichen korrekt und unabhängig von der jeweils verwendeten Zahlendarstellung auf den Rechnern rekonstruierbar ist:

```
2-Byte-Integerkodierung, negative Zahl:
0xFF80  ->  0x007F  ->  0x0080

4-Byte-Integerkodierung, positive Zahl
0x0000FF80  -> 0x00FF80  -> 0xFF007F  ->  0xFF0080
```

> **Aufgabe.** Implementieren Sie die Integerkodierung für zwei Arten interner Darstellung: Bei maschinengestützten Integertypen ist die Datenwortbreite fest vorgegeben (*meist vier oder acht Byte*) und das Vorzeichen ist im höchsten Bit gespeichert, so dass die Kodierung anhand des oben vorgestellten Beispiels durchgeführt werden kann. Bei langen Zahlen für Verschlüsselungszwecke ist die Größe nicht festgelegt und das Vorzeichen wird in einem eigenen Datenfeld gespeichert (*siehe Kapitel 12*), das heißt die Kodierung 0xFF00 kann auch eine positive Zahl kodieren.

- `REAL`: für die Kodierung von Fließkommazahlen existieren eine Reihe unterschiedlicher Möglichkeiten, von denen hier nur eine genannt sei. Das erste Byte gibt die Art der Kodierung an



  Mit Basis = 00 wird eine Kodierung im Binärformat ausgewählt, es sind aber auch andere Kodierungsarten, zum Beispiel als Hexadezimalstring, möglich. Die

Mantisse ist mit 2^s zu multiplizieren (*die Bitverschiebung ist unter bestimmten Umständen aufgrund der Exponentenkodierung notwendig*). Exponent gib die Anzahl der Exponentenbytes an (00 = ein Byte, 01 = zwei Byte, 10 = drei Byte, 11 = variable Anzahl), der Exponent wird als Zweierkomplement kodiert, sofern es sich um eine Binärkodierung handelt. Die restlichen Byte des Datensatzes enthalten die Mantisse als positive ganze Zahl.

- `BITSTRING` : Der Typ Bitstring überträgt eine beliebige Anzahl von Bits. Das erste Byte des Datensatzes gibt die Anzahl der unbenutzten Bits im letzten Datensatz an, wobei wieder vom höchsten zum niedrigsten Bit gezählt wird.

Die Kodierung für den Typ `OBJECT IDENTIFIER` nehmen wir uns getrennt vor. Wie im letzten Kapitel dargelegt, enthält er eine eindeutige Kennziffer zur Identifizierung einer Datenstruktur. Es handelt sich jedoch nicht einfach um eine große Zahl oder einen Binärstring, sondern der Typ besitzt eine komplexere Struktur, die es erlaubt, eine gewisse Ordnung bei der Vergabe universeller Kennzeichnungen herzustellen. Eine Objektkennung besteht aus einer Folge von hierarchischen Kennzeichnern, wobei jeder Kennzeichner ein oder mehrere Bytes belegen kann, was wieder durch das höchste Bit ausgedrückt wird (*Bit 8 = 1: Kennzeichnung wird fortgesetzt; Bit 8=0: letztes Byte der Kennzeichnung*). Vereinbarungsgemäß besteht der erste Eintrag aus zwei Kennzeichnern, die in der Form

$$(K_1 * 40) + K_2$$

zusammengesetzt sind und in der Hauptsache die Organisation und darin die Abteilung bezeichnen sollen, die die Objektkennung verwaltet hat. Die weiteren Kennzeichnungen bezeichnen den Eigentümer der Objektkennung, Algorithmen, Einsatzbereiche usw. Beispiel

```
OBJECT IDENTIFIER {joint-iso-itu-t(2) 100 3}
wird kodiert als  Kenn1:(40*2)+100 = 180, Kenn2=3

Binärkodierung:
--------------------------------
OBJECT IDENTIFIER     0x06
Länge                 0x03
Kennzeichner          0x813403
```

Würde beispielsweise diese Kennung einen verschlüsselten Datenaustausch nach einem bestimmten Modell bezeichnen, für das mehrere Verschlüsselungsmethoden möglich sind (*zum Beispiel eine Telnet–ähnliche Sitzung mit DES– oder AES–Verschlüsselung*), so ist diese als Folgekennung spezifizierbar

```
OBJECT IDENTIFIER {joint-iso-itu-t(2) 100 3 14}
```

Das führt zwar möglicherweise dazu, dass mit

```
OBJECT IDENTIFIER {joint-iso-itu-t(2) 100 3 14}
OBJECT IDENTIFIER {joint-iso-itu-t(2) 117 23 31}
```

der gleiche Algorithmus gemeint ist, aber die Berücksichtigung dieser „Feinheiten“ bei der Implementierung ist Angelegenheit des Programmierers und nicht von ASN.1.

Aufgabe. Geben Sie die Kodierung für die Objektbezeichner an.
Aufgabe. Gegeben sei die ASN.1–Spezifikation

```
Zahlung ::= SEQUENCE {
   oid     OBJECT IDENTIFIER,
   name    Name,
   bank    Bank,
   betrag  INTEGER
}  -- End
Name ::= [APPLICATION 5] VisibleString
Bank ::= [APPLICATION 6] VisibleString
```

Sowie der Datensatz

```
zahlung { oid { 1 10 137 4 }, name { "Müller" },
          bank { "Hausbank" }, betrag { 1703 } }
```

Stellen Sie die Binärkodierung zusammen.

Zusammengesetzte Typen wie `SET` oder `SEQUENCE` werden auf die gleiche Weise wie primitive Typen kodiert:

```
a SEQUENCE { b INTEGER ::= 15 }

-- wird kodiert durch

0x06  0x03    0x02 0x01 0x0F
```

Die drei „Datenbytes“ der `SEQUENCE` können vom Datenstrom entfernt und separat/rekursiv ausgewertet werden. Sofern `SEQUENCE OF`– oder `SEQUENCE (SIZE(..)`–Definitionen vorliegen, können die Attributlisten mehrfach abzuarbeiten sein. Die Interpretation der ASN.1–Binärdaten von

```
a INTEGER   b INTEGER
c SEQUENCE (SIZE(a,b)) OF INTEGER
```

erfordert dann zwingend die Kenntnis des ASN.1–Kodes und einigen internen Aufwand während der Datenauswertung, um die nacheinander eingehenden Teildaten

```
a{1} b{5} c{1,2,3,4,5}
a{6} b{9} c{6,7,8,9}
```

korrekt in die richten Speicherpositionen eines Datenfeldes zu übertragen.

14.3 Übersetzen von Quellkode: Interpreter–Modus

14.3.1 Parsen der Kodebestandteile

Um eine Anwendung ASN.1–kodierte Daten erzeugen oder lesen zu lassen, können wie Eingangs erwähnt zwei unterschiedliche Wege zur Auswertung von ASN.1–Code beschritten werden, die man etwas salopp als Interpretermodus und Compilermodus bezeichnen kann. Damit sind zwei grundsätzlich verschiedene Programmiermodelle verbunden.

Im Interpretermodus wird der ASN.1–Code mit einem Klassenauswerter bearbeitet, der eine Variablenliste erstellt, die ohne weitere Modifikation in der Lage ist, binärkodierte Daten in eine interne Struktur zu Lesen oder daraus zu Schreiben. Auf diese interne Struktur können Algorithmen zugreifen und die erforderlichen Datenmanipulationen vornehmen. Das Anwendungsprogramm für ein bestimmtes Protokoll wird somit parametriert, das heißt für die Ausführung einer neuen Aufgaben sind (*im Idealfall*) keine neuen Programme notwendig. Wir haben also eine dynamische Laufzeitauswertung vor uns.

Im Compilermodus wird der ASN.1–Code zunächst in Arbeitsmodule einer Programmiersprache übersetzt (*Compilezeitauswertung*). Mit Hilfe dieser Module kann anschließend ein bestimmtes Protokoll programmiert werden. Jedes implementierte Protokoll ist bei dieser Arbeitsweise somit ein individuelles Programm, und neue Protokolle können nur durch erneute Programmierung bereit gestellt werden. Zur Nutzung muss das Programm auf einer Maschine installiert werden.

Je nach Komplexität der Anwendung und Qualität der verwendeten Bibliotheken muss sich sowohl der intellektuelle als auch der arbeitstechnische Aufwand bei beiden Vorgehensweisen nicht unbedingt wesentlich unterscheiden. Wir werden uns zunächst mit dem Interpretermodus auseinander setzen. Da komplette lauffähige Anwendungen entstehen sollen, in denen die Daten nach bestimmten Algorithmen verarbeitet werden, sind natürlich im Interpretermodus auch einige Anpassungen auf der Algorithmenseite notwendig, und da sich der ASN.1–Code auf die Binärdatenkodierung beschränkt und keine Hinweise zu den Algorithmen beinhaltet, müssen wir zusätzlich einen Interpretermodus für die Algorithmenseite schaffen, das heißt zum ASN.1–Code kommt noch ein ALG–Code oder wie immer man das nennen möchte hinzu.

Für den Interpretermodus sind gewisse Anforderungen an den zu interpretierenden ASN.1–Code notwendig:

- Da wir Daten verarbeiten wollen, die zur Laufzeit einer Anwendung mit der Umgebung ausgetauscht werden, müssen die ASN.1–Kodes die zum Datenstrom gehörenden Variablen enthalten. Wir bezeichnen solche Kodes im weiteren als ASN.1–Programm.
- Der Datenaustausch in einem Protokoll vollzieht sich meist in mehreren Stufen, in denen ganz bestimmte Daten ausgetauscht werden. Für jede Stufe muss deshalb ein separates ASN.1–Programm vorliegen, das genau die benötigten Variablen enthält.

- Im Datenstrom stehen die Daten (*in den meisten Fällen*) in einer festgelegten Reihenfolge. Wir vereinbaren deshalb, dass die Variablen im Programm in genau der Reihenfolge auftreten müssen, in der sie auch im Datenstrom auftreten.

Kurzfassung. Ein ASN.1–Programm muss den Datenstrom eineindeutig beschreiben. Welche Typdefinitionen außer den für die Deklaration der Variablen benötigten das Programm enthält, ist unerheblich; im Rahmen der Interpretation müssen zwar alle Typen zunächst betrachtet werden, überschüssige Typauswertungen können aber anschließend gelöscht werden.

Wenn man sich nun an den Entwurf eines ASN.1–Interpreters begibt, ist die Festlegung einiger Rahmenbedingungen und allgemeiner Vorgehensweisen sinnvoll. Wie wir oben gesehen haben, ist ASN.1 eine recht komplexe Angelegenheit. Im ersten Ansturm nimmt man daher tunlichst Abstand von der Absicht, einen Programmmonolithen zu entwerfen, der mit allem klarkommt. Wir beschränken uns also bezüglich der Sprachelemente, die wir „verstehen“ wollen. Weiter ist es sicher sinnvoll, bei der Kontextauswertung zunächst mit kurzen Satzteilen auszukommen, idealerweise nur aus dem davor liegenden Wort zu ermitteln, ob das nächste gemäß Syntax zulässig ist. Das bedeutet aber auch, dass wir gewisse Kontexte (*vorläufig*) nicht zulassen. Was genau zulässig sein soll, ist im weiteren noch festzulegen. Konzeptionell ist es wünschenswert, einen erweiterungsfähigen Entwurf zu machen, der möglichst viele der für die Kontexauswertung notwendigen Informationen an zentralen Stellen in Form von Konstantenlisten sammelt, die leicht gepflegt werden können (*und nicht mit verwobenen individuellen* `if` *–Konstruktionen arbeitet, durch die nach einiger Zeit niemand mehr einen Durchblick gewinnen kann*). Ein praktisches Beispiel, das man Schritt für Schritt umsetzen kann, ist unter diesen Rahmenbedingungen recht sinnvoll, und unser Interpreter soll mindestens das folgende ASN.1–Programm bewältigen können (*wenn Sie genau hinschauen, werden Sie feststellen, dass das Programm schon einige Ferkeleien enthält*):

```
test [APPLICATION 2] TType

TType ::= SEQUENCE {
    satz INTEGER { 0..10 },
    p1   Teil1,
    p2   Teil2
    } -- end sequence

Teil1 ::= SET {
    s1  VisibleString,
    s2  GeneralString { "Ende Set"}
    } -- end set

Teil2 ::= CHOICE {
    i1 [0] INTEGER { 2 },
    i2 [1] INTEGER { 3 }
    } -- end choice
```

```
a INTEGER

test2 SEQUENCE (SIZE(1,a)) OF Teil2
```

Nicht berücksichtigen werden wir (*weitere Einschränkungen folgen gegebenenfalls bei Bedarf*):

- Produktionen, da vermutlich auch Ihre Studien ergeben haben, dass dies in der Praxis kaum auftritt.
- Benannte Konstante aus. Diese treten zwar im Zusammenhang mit `OBJECT IDENTIFIER` relativ häufig auf, lassen sich dort jedoch recht gut durch einen Texteditor berücksichtigen. Referenzen auf benannte Konstante sind aber so nicht eindeutig auflösbar.
- Zuweisungen von Werten zu Variablen. Das Arbeiten mit Werten soll nur im Rahmen von ASN.1–Textdaten oder ANS.1–Binärdaten erfolgen (*was bedeutet, dass konstante Vorbelegungen durch spezielle Daten realisiert werden müssen*).

  - Referenzen auf Variable oder Attribute von Variablen in Bereichsangaben mit der Ausnahme von **SIZE(..)** bei Feldgrößenangaben, hier aber auch eingeschränkt auf einfache Attribute, das heißt **a.b** greift auf daserste Attribut im Feld

    a SEQUENCE (SIZE(0,10)) OF {b INTEGER}
    zu und ist zulässig, ein indizierter Zugriff wie a[5].b ist nicht zulässig.

- Iterierte Typdefinitionen (*zum Beispiel* `Typ1::=Typ2 Typ2::= INTEGER`)
- Den Interpreter werden wir so gestalten, dass er sich bei Syntaxproblemen unter Angabe des Stelle im ASN.1–Programm, an der er sich gerade befindet, meldet. Stellt sich dann bei der Kontrolle heraus, dass es sich um nicht berücksichtigte Syntax handelt, kann nachgerüstet werden. Zugegebenermaßen besteht das Risiko bei dieser Vorgehensweise darin, irgendwann auf ein Programm zu stoßen, dessen Umsetzung über das Interpreterkonzept hinausgeht und der größere Redesignmaßnahmen notwendig macht (*siehe Strategieüberlegungen im einführenden Kapitel*).

Die Problemmeldestrategie müssen Sie bei der Entwicklung aktiv nutzen: Der Interpreter soll natürlich zunächst eine Positivauswertung durchführen, das heißt das korrekte Programm auch fehlerfrei auswerten. Funktioniert das bis zu einer bestimmten Zeile, so ist diese durch eine unzulässige Variante zu ersetzen. Der Interpreter muss dies mit einer Meldung quittieren und nicht etwa eine sinnlose Auswertung abliefern. Diesen zweiten Teil der Prüfungen führen Sie bitte in eigener Regie durch, ohne dass ich dies in den Aufgaben formuliere.

Als ersten Schritt können wir die benötigten Programmteile festlegen. Wie Sie am Beispielprogramm nachvollziehen können, sind folgende Schritte notwendig:

(a) Zur Vorbereitung der Auswertung wird das komplette ASN.1–Programm auf einen Datenpuffer eingelesen (*siehe Kapitel 9.6*), wobei „störende" Zeichenketten wie Kommentare, Zeilenvorschübe und gegebenenfalls auch mehrfache Leerzeichen hintereinander usw. entfernt werden. Wie Sie dies machen, überlasse ich Ihnen, da der Umfang der Aufgabe auch davon abhängt, ob Sie über das Beispielprogramm hinaus auch ASN.1–Module, das heißt Dateien mit Typdefinitionen nach ASN.1–Spezifikation, oder im gleichen Sinn, aber einfacher in der Ausführung, das Einbinden von Typdateien durch einen C–ähnlichen `#include`–Befehle zulassen. Der Datenpuffer enthält sämtliche Strukturdefinitionen (*auch nicht benötigte*) in beliebiger Reihenfolge sowie alle benötigten Variablen in der korrekten Reihenfolge.

(b) Vor die kontextbezogene Auswertung schalten wir eine „Textbearbeitung", die vorgegebene Zeichenketten im Datenpuffer durch andere ersetzt. Wir beseitigen damit zumindest das Definitionsproblem benannter Konstanten sowie alternative Schreibweisen einiger Schlüsselbegriffe in den auszuwertenden Kodes, die unser Auswertungssystem nicht erkennt.

(c) Im nächsten Schritt erfolgt die kontextbezogene Auswertung. Da die Reihenfolge der Sätze auf dem Puffer außer der absoluten Reihenfolge der Variablen nicht festliegt, werden die Variablendeklarationen in einer Variablenliste gesammelt, die Typdeklarationen in einer Typenliste. Die Variablenliste enthält danach möglicherweise noch Variablen oder Attribute, denen noch keine der Standardbinärkodierungen zugewiesen ist oder die möglicherweise selbst weitere noch nicht erkannte Attribute besitzen.

(d) In der Typauflösung wird in der Variablenliste rekursiv allen Variablen/Attributen mit nicht festgelegter Standardbinärkodierung aus der Typenliste der dort aufgeschlüsselte Typ zugeordnet. Die Variablenliste kann nun zumindest Binär- und Texkodierungen vollständig verstehen.

(e) Im Beispielprogramm ist die `SIZE`–Konfiguration der letzten Variablen von einer anderen Variablen abhängig, auf deren Wert zugegriffen werden muss. Diesen Variablenbezug gilt es nun aufzulösen. Das erfolgt in drei Schritten, die wir hier stichwortartig zusammenfassen:

   (e.1) Erstellen einer Liste der Variablen, auf deren Werte zugegriffen werden muss.

   (e.2) Notieren der Zugriffsanforderung in den in der Liste angegebenen Variablen und Rückmeldung.

   (e.3) Kontrolle in den zugreifenden Variablen auf den korrekten Datentyp und die Position der Fremdwerte vor den eigenen Werten im Datenstrom (*die Größenangabe muss zur Verfügung stehen, bevor die Auswertung des Feldes erfolgt. Man kann dies als weitere, nicht unbedingt von der Syntax vorgeschrieben Rahmenbedingung ansehen*).

Zusammengefasst erhalten wir damit einschließlich der Meldung von Problemen mit Hilfe eines einfachen Ausnahmemanagements folgenden Programmrahmen:

```
bool ASN_1::Compile_ASN1(string fname){
    DBuffer asn;
    bool result=true;
    ...
    fehler="";
    asn=read_file(fname);
    ASN_ExchangeFromList(asn);
    try{
        TextInterpreter(...,asn,vars,types);
        TypAufloesung(vars,types);
        GetReferenceList(vars,refs);
        NotifyReference(vars,refs,...);
        CheckReference(vars,...);
    } catch(string s) {
        fehler=fehler+s;
        vars->clear();
        result=false;
    }//endtry
    return result;
}//end function
```

Aufgabe. Die zu suchenden und die Ersatztexte können in zwei korrespondierenden Listen gesammelt werden. Mit Hilfe von STL–Algorithmen wird der Inhalt des Puffers auf das Auftreten von Strings aus der ersten Liste untersucht und Übereinstimmungen ausgetauscht. Implementieren Sie eine solche Methode.

Wir können damit gleich zu Punkt (c) unserer Arbeitsliste kommen. Die kontextbezogene Auswertung erfolgt Wort für Wort (*wobei wir noch genauer definieren müssen, was ein Wort ist und wie es erkannt wird*). Mehrere Worte fügen sich zu einem Satz zusammen, und ob ein Wort an einen vorhandenen Satz angehängt werden darf, wird möglichst durch letzte Wort, vielleicht auch durch das vorletzte oder das erste Wort festgelegt. Das erste Wort zu Beginn der Auswertung muss eine Variablenbezeichnung oder eine eigene Typbezeichnung sein:

```
meineErsteVariable ...
MeinErsterTyp ...
```

Alles andere ist unzulässig. Zu den unzulässigen Begriffen gehören auch die Standardtypen wie `INTEGER` usw. Liegt nun beispielsweise eine Variable vor wie in unserem ASN.1–Programm, so darf der Satz folgendermaßen fortgesetzt werden:

```
meineErsteVariable  MeinErsterTyp    -- eigene Typdefinition
meineErsteVariable  StandardTYP      -- Standardtypen
```

```
meineErsteVariable  [...]            -- Tagzuweisung
...
```

Führen wir die Überlegungen weiter, was danach wiederum zulässig ist, so sind für die erste Möglichkeit beispielsweise gültige Sätze:

```
meineErsteVariable  MeinErsterTyp  meineZweiteVariable
                -- 1
meineErsteVariable  MeinErsterTyp  MeinZweiterTyp
                -- 2
meineErsteVariable  MeinErsterTyp  ::=
                -- 3
meineErsteVariable  MeinErsterTyp  { .. }
                -- 4
...
```

In den Fällen `-- 1` und `-- 2` sind wir wieder am Startpunkt angelangt, die Fälle `-- 3` und `-- 4` müssten fortgesetzt werden, wenn uns nicht ein kurzer Blick auf unsere Einschränkungen davon überzeugen würde, dass wir diese Fälle nicht betrachten wollen.

> **Aufgabe.** Für die Praxis benötigen wir zunächst eine Methode, die ein Wort ausliest und zur weiteren Analyse bereitstellt. Implementieren Sie eine solche Methode. Das Wort wird in einem neuen Datenpuffer abgespeichert und gleichzeitig vom Kodepuffer entfernt. Im Laufe der Interpretation schwindet so der Inhalt des Kodepuffers, und die Interpretation ist beendet, wenn der Kodepuffer geleert ist (*falls nicht ein Fehler einen vorzeitigen Abbruch auslöst*).

Klassifizieren Sie dazu zunächst die verschiedenen Worttypen. Als Sondervereinbarung legen wir fest: Auch Klammerbereiche wie Inhaltslisten `{..}` betrachten wir als ein Wort. Der Inhalt besteht zwar seinerseits aus Worten, aber zu deren Interpretation kommen wir später.

Die so erhaltenen Worte teilen wir in Wortklassen ein. Die Klasse jedes Wortes muss in einer vorgegebenen Menge vorhanden sein, damit der Satz seine Gültigkeit behält. In C/C++ läuft das auf eine Bitmusterprüfung hinaus. Für unser ASN.1–Programm benötigen wir aufgrund unserer getroffenen Definition von einem Wort folgende Klassen[11]

```
enum asn_wordClass
    { asn_var              =   0x00001, // xA..z-_
      asn_referencedType   =   0x00002, // XA..z-_
```

[11] Spätestens jetzt sollten Sie die Aufgabe bearbeiten und in den Hinweise nachsehen, welche Überlegungen dahinter stehen. Trotzdem wird Ihre Liste der Worttypen vielleicht etwas anders aussehen als meine. Gemeinerweise habe ich die Gesamtaufgabe ja schon bearbeitet, um dieses Buch mit sinnvollem Inhalt zu füllen, und eine Reihe der dabei gemachten Erfahrungen wird sich immer wieder in den Beispielkodes vorab wiederfinden; die Unterschiede werden aber später aus dem Gesamtablauf klar.

```
    asn_primitiveType    =   0x00004, // BOOLEAN
    asn_constructedType  =   0x00008, // SEQUENCE
    asn_choiceType       =   0x00010, // CHOICE
    asn_zuweisung        =   0x00020, // ::=
    asn_tag              =   0x00040, // [ .. ]
    asn_content          =   0x00080, // { .. }
    asn_varcontent       =   0x40000, // { .. }
    asn_constraint       =   0x00400, // ( .. )
    asn_of               =   0x00800, // OF
    asn_empty            =   0x80000  //
  };//end enum
```

Um zu entscheiden, welcher Klasse ein gelesenes Wort angehört, wird ein zunächst Vergleich mit einer Liste der in ASN.1 fest definierten Worte durchgeführt und gegebenenfalls weiter die Art des ersten Zeichens geprüft. Jedem Wort können wir als Eigenschaften zuordnen, zu welcher Wortklasse es gehört, welche Wortklassen im Anschuss (*voraussichtlich*) zulässig sind und im Fall von Standarddatentypen, wie sie kodiert werden. Wir fassen die Eigenschaften in einer Struktur zusammen:

```
struct WordClassAttributes {
    char            tag;
    int             allow_next;
    asn_wordClass   wordClass;

    WordClassAttributes(): tag(0), allow_next(0),
                           wordClass(asn_empty){};
    WordClassAttributes(char tag, int next,
                asn_wordClass et):
                tag(tag), allow_next(next),
                wordClass(et){};
};//end struct
```

Die in ASN.1 reservierten Worte (*das sind nicht nur die Standardtypen, sondern auch Worte wie* OF *usw*.) und die zugehörenden Atribute, speichern wir in einer `map<..>` mit dem reservierten Wort als Schlüsselbegriff.

```
static map<string,WordClassAttributes> ptypes;
```

Die Suche nach reservierten Worten und deren Eigenschaften ist nun durch einen einfachen `find(..)`–Zugriff auf den Container möglich. Für die Wortklassen, die so nicht erfasst werden, legen wir entsprechende Konstante von `WordClassAttrbutes` an, um bei der Auswertung auf ein einheitlichen Datenmodell zugreifen zu können. Die Initialisierung der Liste erfolgt durch ein statisches Objekt einer Initialisierungsklasse, das im Konstruktor die Füllanweisungen für die Liste enthält. Zusätzlich kann das Einlesen der Ersatztexte (*Fehler: Referenz nicht gefunden*) und im Destruktor das Schreiben der Ersatzliste erfolgen. Einen Großteil der Parameter, die wir für die Auswertung benötigen, haben wir damit bereits an einer zentralen Stelle untergebracht.

```
Static class InitAll { ... } initAll;

InitAll::InitAll(){
    int construct_next= asn_content |
                        asn_varcontent |
                          asn_constraint | asn_of ;
    ...
    word_classes.insert(
     pair<string,WordClassAttributes>
      ("CHOICE",
         WordClassAttributes(0xff,
               asn_content|asn_varcontent,
                           asn_choiceType)));
    word_classes.insert(
     pair<string,WordClassAttributes>
      ("BOOLEAN",
        WordClassAttributes(0x01,primitiv_next,
                        asn_primitiveType)));
    ...       }//end constructor
```

Aufgabe. Implementieren Sie die Initialisierungsklasse vollständig. Vermutlich werden Sie den eine oder anderen Parametersatz noch nicht korrekt eingeben. Die notwendigen Korrekturen nehmen Sie im Verlauf der fortschreitenden Übersetzung der ASN.1–Vorlage vor.

Die Identifizierung der Wortklasse läuft in folgenden Stufen ab:

(a) Suche des kompletten gelesenen Wortes in der Wortliste – falls erfolglos:
(b) Suche des aus dem ersten Buchstaben bestehenden Unterstrings in der Wortliste (*Identifikation der Klammern und Texte*) – falls erfolglos:
(c) Prüfung auf reine Ziffernfolgen (*Identifikation von Werten*) – falls erfolglos:
(d) Unterscheidung zwischen Variablen und eigenen Typen anhand der Groß/Kleinschreibung des ersten Buchstabens.

Aufgabe. Implementieren Sie eine Methode zur Identifizierung der Wortklasse. Als Rückgabewert bietet sich der gefundene Wert von `WordClassAttributes` an. Für Variable, eigene Typen und Konstante definieren Sie Konstante von `WordClassAttributes` zur Rückgabe.

14.3.2 Konstruktion der Felddatentypen

Wir kommen damit zu einem etwas heiklen Thema, wie sich bei näherer Betrachtung zeigt: Mehrere Worte formen einen Satz, der einen eigenen Typ oder eine Variable beschreibt. Wir können einem Satz somit ein Objekt zuordnen, wobei wir

Typobjekte oder Variablenobjekte erhalten, die getrennt in seriellen Containern gesammelt werden. Im Falle einer SEQUENCE enthält ein Wort des Satzes rekursiv eingeschlossen weitere Sätze, die die inneren Struktur der SEQUENCE beschreiben. Die Objekte besitzen somit ebenfalls Container mit weiteren Objekten für die innere Struktur. Neben der Verwaltung einer Reihe allgemeiner Parameter müssen die Objekte später auch die Daten aufbereiten, was für jeden Datentyp individuell erledigt werden muss. In diesem Kapitel beschränkt sich die Individualität zunächst auf die Interpretation von Bereichsangaben.

Eine genauere Betrachtung zeigt schnell, dass die Individualität der ASN.1–Standardtypen doch so ausgeprägt ist, dass deren Berücksichtigung mit ein paar `if`–Abfragen im Code nicht zu erledigen ist, sondern besser für jeden Typ eine spezielle Klasse eingerichtet werden sollte. Für die Auswertungsumleitung auf einen speziellen Typ benötigen wir virtuelle Methoden, die wiederum Zeigervariable in unseren Containern verlangen, und außerdem steht der Typ erst relativ spät fest (*teilweise erst nach der Typauflösung im zweiten Durchgang*), so dass wir die Einträge des Containers umtypisieren müssen. Das hört sich ziemlich komplex an, aber für alles haben wir in den vergangenen Kapiteln bereits die notwendigen Werkzeuge entwickelt.

Beginnen wir mit einer Basisklasse für ASN.1–Objekte. Diese muss Attribute für die Sicherung von Variablen mit unbestimmtem (*das heißt selbst definierten*) Typ aufweisen. Eine Analyse unseres ASN.1–Vorlageprogramms führt zu der Definition

```
// Allgemeine ASN.1 - Objekte
struct ASN1: public ObjectReferenceCounter {
    ASN1();
    ~ASN1();
    virtual void init();
    virtual void set_att(const ASN1& asn);
    virtual bool codable(){return false;};
    virtual int  code_type(){return -1;};
    virtual void set_size(int s){size=s;};
             void set_tag(DBuffer& w);
    virtual bool load_content(DBuffer&w)
        {return false;};
    virtual string print_asn1(int stufe);
    string var_name,
           typ_name;
    asn1_classes klasse;
    int tag_type;
    vector<...> attributes;
    DBuffer data;
    int size;
    int referenced;
};//end class
```

Hinter diesen Definition stecken folgende Überlegungen (*Attribute und Methoden jeweils von oben nach unten*):

- Eine gültige Variable liegt vor, wenn Variablenname und Typname definiert sind. Der Variablenname wird in ASN.1–Textdaten benötigt, der Typname für die Typauflösung. Folgerichtig benötigen wir zwei String–Attribute für die Bezeichnungen.
- Klasse und Typnummer können individuell bereits bei eigenen Typen angegeben werden. Ein Kodierungstyp ist nicht notwendig, da wir ja später für jeden Typ spezielle Klassen ableiten.
- Unterlisten werden zwar nach unseren bisherigen Überlegungen nur für `SET`, `SEQUENCE` oder `CHOICE` benötigt, grundsätzliche Überlegungen zu einer späteren Erweiterung des rekursiven Auswertungsschemas lassen es jedoch geboten erscheinen, bereits hier einen Container anzulegen. Da wir anschließend noch überlegen müssen, wie der Container genau zu realisieren ist, ist hier zunächst ein Torso angegeben.
- Die interne Datensicherung erfolgt auf einem Feld des Typs `DBuffer`. Wir können beliebige Daten darauf unterbringen und durch eine Variable dieses Typs auch sehr leicht an andere Programmpositionen transferieren.
- Bei dem Feldtyp `SEQUENCE (SIZE(..)) OF` sind gegebenenfalls mehrere Werte eines Typs zu speichern. Durch Vergrößerung des Pufferattributs ist das kein Problem; die Größe des Datenfeldes wird in einem Attribut hinterlegt.
- Werte können in anderen Variablen benötigt werden. Durch das Attribut `referenced` geben wir an, welches Datenfeld an anderer Stelle benötigt wird. Das Attribut ist ein Mittelweg zwischen Zulässigem und Machbarem nach Erweiterung: Zulässig ist nach unseren Rahmenbedingungen nur ein Zugriff auf den ersten Wert, das Feld eröffnet prinzipiell den Zugriff auf jeden Wert, aber pro Variable nur auf einen.
- Im Laufe der Arbeit sind Kopier- und Rücksetzoperationen schon absehbar. Wir berücksichtigen dies durch eine virtuell Initialisierungsmethode `init()`.
- Beim Austausch von Objekten, den wir noch genauer untersuchen müssen, sind die Attribute zu kopieren. Vorgesehen wird die Methode `set_att(..)` für die Attribute der Basisklasse
- Für die Typauflösung und die Wertreferenzen werden zwei einfache Methoden deklariert, die angeben, ob es sich bereits um einen Standardtyp handelt (*Daten sind kodierbar*) und welcher dies ist.
- Bei Vergrößern oder Verkleinern des Datenbereiches sind typgebundene Initialisierungen durchzuführen. Die Grundmethode dazu legen wir hier an.
- Klasse und Typkennzeichnung werden bereits in der Basisklasse vollständig abgewickelt.
- Die Bereichsangaben bei Zahlen oder Strings sowie die Größenangaben von Feldern bieten sich zwar grundsätzlich für eine iterative oder rekursive Auswertung an; wir wollen solches aber für spätere Erweiterungen aufsparen und einstweilen die Interpretation vollständig den speziellen Typklassen überlassen (*betrachten*

Sie dies als weitere Rahmenbedingung). Die Methode `set_content(..)` kümmert sich um Bereiche, `SIZE`–Angaben usw.
- Zur Kontrolle unserer Bemühungen sehen wir eine `print(..)`–Methode vor, die uns eine Ansicht der gespeicherten Objekte einer Liste liefert.

Ein offener Punkt ist die Struktur des Containers, den wir hier als `vector<..>` deklariert haben. Um die Nutzung virtueller Methoden zu ermöglichen, muss er Zeiger enthalten. In Kapitel 7 finden wir folgende Lösung:

```
typedef     APtr<ASN1>          P_ASN1;
typedef     vector<P_ASN1>      ASN1_VarList;
```

Jetzt ist auch klar, warum ASN1 von `ObjectReferenceCounter` erbt. Es sind bei näherem Hinsehen aber noch Ergänzungen notwendig: Die Zeigerverwaltung arbeitet ausschließlich mit Referenzen, was aber auch bedeutet, dass im Programm

```
var_1  Spezialtyp
var_2  SEQUENCE {
   att_1  Spezialtyp, ...
```

`var_1` und `var_2.att_1` nach der Typauflösung auf die gleichen Objekte verweisen und bei der Verarbeitung von Daten die in `var_1` eingelesenen Daten bei der Auswertung von`var_2.att_1` überschrieben werden. Wir müssen deshalb Methoden vorsehen, die anstelle einer Referenz der nächsten `APtr<..>`–Variablen eine Kopie der Quellvariablen übergeben und später im Programmkode darauf achten, welche der beiden Methoden (*Kopie oder Referenz*) zu verwenden ist. Für die Erstellung einer Kopie implementieren wir zwei Methoden:

```
virtual APtr<ASN1> copy();
virtual void datacopy(ASN1& a);
```

Die Methode `copy()` erzeugt eine `APtr`–Variable, die eine Kopie der Quellvariablen enthält, die Methode `datacopy()` kopiert den Inhalt einer Variablen auf das Argument.

> **Aufgabe.** Auch wenn wir mit den Interpreter noch nicht loslegen, können Sie schon die ersten Methoden der Klasse ASN.1 implementieren, insbesondere die Methode zum Lesen der Klasse und der Typkennziffer. Begründen Sie auch, weshalb für die Erstellung einer Kopie einer Variablen zwei Methoden deklariert werden und geben Sie den Aufruftorso für eine erbende Klasse `ASN1_Integer::ASN1` an.

Die Auswertung der ASN.1–Vorlage kann nun durch eine rekursive Methode erfolgen. Für die Aufnahme der Ergebnisse der Auswertung (*Variable oder Typen*) stellen wir zwei Container bereit. Die Interpretermethode benötigt als Parameter die nächsten zulässigen Wortklassen, den Puffer mit dem ASN.1–Code sowie die Container, die im rekursiven Aufruf durch das Attribut `attributes` des ASN1–Objektes ersetzt wird. Zusätzlich sind aber weitere Steuerungselemente für die Rekursion notwendig:

(a) In der Hauptauswertung sind Variablen und Typdefinitionen zulässig.
(b) In der rekursiven, derzeit ausschließlich auf SEQUENCE–Typen beschränkten Auswertung sind ausschließlich Variable zulässig.
(c) Bei der rekursiven Auswertung des Typs SEQUENCE .. OF ist nur eine einzelne Typangabe zulässig, das heißt weder eine Variable noch eine weitere rekursive Verzweigung.

Zusammengefasst erhalten wir die Schnittstelle:

```
enum interpreterMode {
   imode_vars_types, imode_var_only,
   imode_one_type };

void TextInterpreter(interpreterMode ip, int allow,
                DBuffer& asn,
                ASN1_VarList& vars,
                ASN1_VarList& types){

// rekursiver Aufruf mit ASN1-Objekt  ao[12]
TextInterpreter(..,..,..,ao.content,ao.content);
```

Intern ist in einer Programmschleife das nächste Wort vom Puffer zu extrahieren, die Wortklasse des gelesenen Wortes zu ermitteln und durch UND–Verknüpfung mit dem Parameter allow die Zulässigkeit festzustellen und anhand der Wortklasse die Auswertung vorzunehmen, bei deren Abschluss die gültigen Wortklassen für den nächsten Durchlauf festgelegt werden.

> **Aufgabe.** Implementieren Sie das Gerüst für die Methode TextInterpreter(..). Werten Sie eine auftretende Variable aus (*weitere Auswertungen folgen später*). Was sollte im Fall von Fehlern passieren?

Für die weitere praktische Umsetzung des Interpreters sollen Sie etwas mehr auf sich gestellt sein. Sie können das rein theoretisch angehen oder sich mehr oder weniger experimentell durch das ASN.1–Programm hangeln und jeweils nach Fehlermeldung oder Ausdruck des Inhalts der Container Korrekturen vornehmen. Für den Ausdruck des Inhalts haben wir bereits oben in der Klasse ASN1 eine Methode vorgesehen, die durch eine allgemeine, einen Container bearbeitende Methode ergänzt werden kann (*ein wechselseitiger Aufruf der Klassenmethode durch die Containermethode und der Containermethode in der Klasse zur Ausgabe des Inhalts des internen Containers ist möglich*). Auch dazu benötigen Sie keine Hilfe, da der Ausdruck dem Vorlageprogramm entsprechen sollte.

---

[12] Beachten Sie hierzu die Fälle (b) und (d). Bei der Rekursion treten bei SEQUENCE Variable oder Typen auf. IN beiden Fällen ist die Auswertung in der Inhaltsliste der ASN1–Variablen abzulegen. Wegen der Unterscheidung des Interpreters zwischen zwei Listen ist das Listenattribut doppelt als Parameter anzugeben.

Wir kümmern uns hier lediglich noch um die ASN.1–Standardtypen. Die Anzahl der Klassen kann zunächst auf die im ASN.1–Programm verwendeten Typen beschränkt und später nach Bedarf ausgebaut werden. Wenn Sie berücksichtigen, dass `SEQUENCE` und `SET` viele Gemeinsamkeiten aufweisen und `CHOICE` ziemlich aus dem Rahmen fällt, ist folgende Klassenstruktur sinnvoll:

```
ASN1                                // Basisklasse
    ASN1_Integer
    ASN1_String
    ASN1_CHOICE
    ASN1_Field
        ASN1_Sequence
        ASN1_Set
```

14.3.3 Bereichsdefinitionen

ASN1_Integer und ASN1_String müssen Bereichsdefinitionen aufnehmen. Wir deklarieren dazu folgende Attribute:

```
ASN1_Integer:
    vector<pair<int,int> > intervals;

ASN1_String:
    vector<DBuffer> strs;
```

Zulässige Werte für Zahlen werden in Form von Intervallen gespeichert; bei Stringvariablen werden zulässige Strings gespeichert.

```
ivar INTEGER { 13, 17..21 }
    // Intervalle:  pair(13,13),  pair(17,21)
```

Sind die Attribute leer, so sind alle Werte zulässig. Nicht berücksichtigt (*und damit Raum für Ihre eigenen Erweiterungen*) ist der Ausschluss von Zahlen oder Strings oder die Einschränkung von Schreibweisen wie die Verwendung von Großbuchstaben in Strings ohne Einschränkung des Inhalts. Bei diesbezüglichen Erweiterungen sollte jedoch immer zunächst die X.680 konsultiert werden, um im Rahmen der Norm zu bleiben.

Die Speicherung eines Feldes von Zahlenwerten in einer `ASN1_Integer`–Variablen durch Bereitstellung eines hinreichend großen Platzes auf dem Datenpuffer haben wir bereits angerissen. Bei Stringvariablen kann eine entsprechende Anzahl von Textkonstanten (*durch Hochkommata begrenzte Texte*) hintereinander auf dem Puffer abgelegt werden. Zum Lesen wird die Wortlesemethode verwendet; auch beim Ändern eines Eintrags ist sequentiell vorzugehen.

Aufgabe. Entwickeln Sie ein Methode zum Lesen beziehungsweise Ersetzen bestimmter Strings in einem Stringfeld. Die Methode wird zwar erst später benötigt, stellt hier aber schon eine ganz gute Übung dar. Die Speichermethode mutet zwar im Vergleich mit dem Container für die zulässigen String primitiv an, scheint mir aber für die Praxis hinreichend zu sein. Alternativ können Sie natürlich auch an Lösungen mit Parameterstrings (*da bestehen allerdings in der bisherigen Version der Parameterstrings gewisse Einschränkungen bei der Darstellung von Texten, die auch zu überwinden/umgehen wären*) oder mit Vektorcontainers auf dem Datenpuffer arbeiten.

Die Datentypen `SEQUENCE` und `SET` unterscheiden sich nur bei der Auswertung von Daten, so dass wir hier eine gemeinsame Zwischenklasse definieren können, die Attribute für die Feldgröße enthält:

```
string sfrom,sto;
int ifrom,ito;
```

Wir sehen Stringattribute für die Aufnahme der Parameter in der `(SIZE(..,..))`–Deklaration vor, bei denen es sich voraussetzungsgemäß sowohl um Zahlen als auch um Variablennamen handeln kann. Im Rahmen des später zu betrachtenden Datenaustausches werden diese Werte in ganzzahlige Werte übersetzt. Wir sehen hier folgende Logik vor:

(a) Bei `SET`– oder `SEQUENCE`–Vereinbarungen ohne Felderweiterung gilt `(from=0,ito=1)`.
(b) Bei `SET`– oder `SEQUENCE`–Vereinbarungen mit definiter Größenangabe `(SIZE)` werden die Attribute `ifrom` und `ito` jeweils nach Anforderung des Datensatzes belegt. Da nicht auszuschließen ist, dass Felder in mehreren Teilen übertragen werden, zum Beispiel

```
a INTEGER b INTEGER c SEQUENCE (SIZE(a,b))OF...

1. Satz:  a=1, b=10 ;  2. Satz: a=11, b=20
```

wird im Attribut `size` der höchste Indexwert gespeichert und die Attribute mit diesem Wert dimensioniert.[13]
(c) Bei `SET OF` oder `SEQUENCE OF` werden beide Grenzen auf (–1) gesetzt. Die Dimensionierung der Attribute muss daher während des Datenaustausches nach Bedarf durchgeführt werden.

Nach unseren Rahmenbedingungen ist es nicht vorgesehen, Felder selbst zum Gegenstand anderer Felder zu machen. Optional können Sie auch ein Indexmodell entwickeln, dass ASN.1– Programme der Art

[13] In ASN.1 bedeutet SIZE(a,b) die Zählung for(i=a;i<=b;++). Die Attribute ifrom und ito sollten jedoch genäß C–Konvention verwendet werden.

```
a SEQUENCE (SIZE(1,10)) OF Type1
Type1 ::= SEQUENCE (SIZE(1,5)) of INTEGER
```

zulässt.
Beim Datentyp `CHOICE` müssen wir feststellen können, welcher der Auswahltypen gelesen wurde oder zu schreiben ist. Da der Datenpuffer bei diesen Datentyp ansonsten nicht benötigt wird, legen wir auf ihm ein Feld von ganzen Zahlen an, die jeweils das aktive Attribut angeben (*nicht aktiv: Inhalt* (-1)). Damit ist auch für den Datentyp `CHOICE` eine Verwendung in Feldern gewährleistet, zudem können wir unser Schnittstellenmodell mit Algorithmen nahtlos beibehalten: Der Datenpuffer kann auch von Anwendungen dimensioniert werden, was die ASN1–Objekte in der `set_size(..)`–Methode berücksichtigen können, und auch die Spezifizierung eines aktiven `CHOICE`–Typs ist durch einen Algorithmus möglich, ohne dass eine zusätzliche Schnittstelle benötigt wird.

14.3.4 Elimination selbstdefinierter Typen

Als letzter Posten der Arbeitsliste der Interpretermethode ist nun noch der Ersatz des zunächst verwendeten ASN1–Objektes durch eines der spezialisierten zu realisieren. In der Klasse `WordClassAttributes` haben wir durch das Attribut `tag` bereits spezifiziert, um welches Spezialobjekt es sich handelt; die Kopiermethoden für den bereits aufgearbeiteten Inhalt stehen ebenfalls bereit. Um weiterhin die speziellen Daten an einer zentralen Stelle sammeln zu können, führen wir für jede ASN1–Klasse eine statische Erzeugungsmethode ein:[14]

```
struct ASN1_Integer: public ASN1 {
    ...
    static P_ASN1 set_codable_type(P_ASN1& pa);
```

Die Methoden sammeln wir in einer Liste, die im Initialisierungsobjekt zusammen mit den Wortklasseneinträgen initialisiert werden kann:

```
static const int anz_asn1_makefunctions=32;
static struct ASN1_MakeFunction {
    ASN1_MakeFunction(){fnc=0;};
    P_ASN1 (*fnc)(P_ASN1&);
} make_function_list[anz_asn1_makefunctions] ;
// Für den Typ CHOICE kann ein beliebiger
// Kodierungstyp außerhalb der anderen
// Standardtypen vergeben werden.
        make_function_list[0x00].fnc=
          &ASN1_Choice::set_codable_type;
```

[14] Für die Zwischenklasse ASN1_Field benötigen wir keine Erzeugungsmethode. Warum?

Durch einen Indexzugriff in der Interpretermethode wird nun das ursprüngliche Basisobjekt passend überschrieben

```
ao=make_function_list[w_att.tag].fnc(ao);
```

> **Aufgabe.** Implementieren Sie die statischen Methoden. In den Kopiermethoden für die Attribute der Objekte sind nun zwei Fälle zu berücksichtigen: Bei der Kopie eines Objektes wird ein Objekt des gleichen Typs erzeugt, so dass alle Felder kopiert werden können. Beim Überschreiben eines Basisobjektes dürfen aber nur die Felder des Basisobjektes kopiert werden; ein Zugriff auf spezielle Attribute führt zu Laufzeitfehlern. Sie können diese Fälle durch bereits vorhandene Eigenschaften der Klassen unterscheiden. Außerdem ist in der Interpretermethode nach dem Abspeichern eines ASN1–Objektes das Arbeitsobjekt wieder als Basisobjekt zu initialisieren.

Damit sollte der Interpreter den ASN.1–Code vollständig abarbeiten können und der Ausdruck der Variablenliste folgendes Aussehen besitzen:

```
test [APPLICATION 2] TType {}
a [UNIVERSAL 2] INTEGER {}
test2 [UNIVERSAL 16] SEQUENCE (SIZE(1,a)) OF {
     [undefined -1] Teil2 {}
}
```

Falls Ihre Liste noch Ungereimtheiten aufweist, nehmen Sie vor der weiteren Arbeit die erforderliche Korrekturen in Ihren Quellen vor.

In dieser Liste ist allerdings nur die Variable `a` vollständig deklariert. Bei `test` ist unklar, was sich hinter `TType` verbirgt, bei `test2` ist `Teil2` nicht geklärt. Außerdem sollte überprüft werden, ob die Variable `a` in `(SIZE(1,a))` auch existiert und vor der Variablen `test2` deklariert ist, da andernfalls mit nicht unerheblichen Problemen bei der Datenbearbeitung zu rechnen ist.

Lösen wir gemäß unserer Arbeitsliste zunächst die Typen auf: Die Typbezeichnungen von Variablen oder Attributen von Variablen mit `codable()==false` werden im Typcontainer gesucht und die Objektvariable durch eine Kopie des gefundenen Objekts überschrieben überschrieben. Der Attributinhalt muss dabei selektiv kopiert werden:

- Der Variablenname des alten Objektes bleib erhalten, ebenso die Typbezeichnung.
- Sofern Klasse und Typkennziffer spezifiziert sind, werden diese ebenfalls beibehalten. Sind diese Attribute nicht gesetzt worden, werden Klasse und Typkennzeichen des gefundenen Objekts verwendet.

> **Aufgabe.** Implementieren Sie eine Funktion für die Typauflösung. Die in jedem Objekt enthaltene Unterliste kann rekursiv mit dieser Methode bearbeitet werden.

14.3.5 Auflösung der gegenseitigen Abhängigkeiten

Bevor wir uns das Ergebnis anschauen, erstellen wir noch die Liste der Variablen, die von anderen Variablen aufgerufen werden (*im derzeitigen Ausbauzustand trifft dies nur auf Felder im Rahmen der* `(SIZE(..))`*–Vereinbarung zu*). Die Variablennamen werden in einem Vektorcontainer gesammelt. Dazu wird eine Methode implementiert, die einen Container des Typs `ASN1_VarList` bearbeitet und die virtuelle Methode

```
virtual void ASN1::AskReference(vector<string>& vs)
```

aufruft, die Einträge in den Container vornimmt und den objekteigenen Container in einer Kreuzrekursion wieder von der Hauptmethode bearbeiten lässt. Den eigentliche Grund für das Vorziehen der Listenerstellung werden Sie bemerken, wenn Sie das Ergebnis der Typenauflösung ausdrucken und mit der Endversion vergleichen:

```
test [APPLICATION 2] TType {           -- SEQUENCE
    satz [UNIVERSAL 2] INTEGER {0..10} -- INTEGER
    p1 [UNIVERSAL 17] Teil1 {          -- SET
        s1 [UNIVERSAL 26] VisibleString }
                          -- VisibleString
        s2 [UNIVERSAL 27] GeneralString
             {"Ende Set"}   -- GeneralString
    }
    p2 Teil2 {}{                       -- CHOICE
        i1 [CONTEXT 0] INTEGER {2}     -- INTEGER
        i2 [CONTEXT 1] INTEGER {3}     -- INTEGER
    }
}
a [UNIVERSAL 2] INTEGER {}              -- INTEGER

test2 [UNIVERSAL 16] SEQUENCE (SIZE(1,a)) OF {
    Teil2 {}{                           -- CHOICE
        i1 [CONTEXT 0] INTEGER {2}      -- INTEGER
        i2 [CONTEXT 1] INTEGER {3}      -- INTEGER
    }
}
```

Die `CHOICE`–Attribute weisen nämlich noch nicht die Klasse `CONTEXT` auf. Die Listenerzeugung gibt uns die Möglichkeit, dies zu korrigieren und zugleich zu prüfen, ob alle Attribute unterschiedliche Typkennziffern aufweisen. Da sich an der Variablenliste nichts mehr ändert, ist nun der richtige Zeitpunkt für diese Korrektur.

> **Aufgabe.** Implementieren Sie die Methoden zur Listenerstellung unter Berücksichtigung der Sonderfunktion für den Typ `CHOICE`.

Die Auswertung des ASN.1–Programms wird abgeschlossen durch die folgenden beiden Methoden:

(a) Die rekursive Methode

```
void NotifyReference(vector<ASN1>& var,
                     vector<string>& vs, string s)
```

setzt in diejenigen Variablen, die in der soeben erzeugten Liste vs der benötigten Variablenreferenzen auftreten, das Attribut referenced auf den Wert 0.[15] Die Methode verzweigt rekursiv auf die Attribute der Variablen, wobei das Stringattribut s die Variablennamen rekursiv ergänzt. Da nur Strings überprüft werden, sind spezielle Methoden in den Klassen ASN1 oder ASN1_Content nicht notwendig.

Aufgabe. Implementieren Sie die Methode.

(b) Die Methode `void CheckReference(..)` überprüft nun abschließend, ob alle Beziehungen aufgelöst werden können. Dazu erhält Sie eine „Ereignisliste“, die den vollständigen Namen der Variablen (*auch hier wieder Rekusion*) und den Datentyp enthält:

```
vector<pair<string,int»
```

Weist das Attribut `referenced` einen Wert ≠(-1) auf, so wird ein Eintrag in dieser Liste erzeugt.[16]

Nach Eintrag der Beziehung prüft jede Variable mittels der virtuellen Klassenmethode `CheckReference(..)`, ob sie eine externe Variablenreferenz benötigt und sich diese in der Liste mit dem passenden Typ befindet. Sofern die Beziehung nicht aufgelöst werden kann, ist entweder die Daten liefernde Variable nicht oder hinter der nutzenden deklariert, und es wird eine Ausnahme geworfen. Das gleiche gilt, wenn der Typ der Variablen nicht den Anforderungen entspricht.

Aufgabe. Implementieren Sie die Methoden. Fassen Sie gleichzeitig alle Aktionen in einer Auswertemethode zusammen.

Die Überprüfung, ob die Referenzen korrekt hergestellt werden, führen wir erst im nächsten Schritt, der Bearbeitung von ASN.1–Daten, durch.

Abschließend fassen wir alles in einer Klassen zusammen, die eine Variablenliste verwaltet und sämtliche hier diskutierten Details nach außen kapselt:

```
class ASN_1 {
public:
    ASN_1();
    ~ASN_1();
    /*! Übersetzen einer ASN.1-Kodedatei in eine
```

[15] Eigentlich von (–1) auf den Feldindex des Wertes, der später an eine andere Stelle übertragen werden soll. Vergleiche dazu auch die Anmerkungen bei der Erstellung der Arbeitsliste.

[16] Hierzu kann eine virtuelle Methode code_type() implementiert werden.

```
        Variablenliste.  */
    bool   Compile_ASN1(string fname);
    /*! Ausgabe einer Variablenliste als
        ASN.1-Code */
    string VarList();
    /*! Ausgage des Fehlerstrings.  */
    string ErrorList();
protected:
    void * varlist;
    string fehler;
};//end class
```

Die Klasse können wir nun nach Bedarf durch neue Methoden ergänzen. Die Variablenliste ist hier als `void *`–Zeiger deklariert, damit auch die mit der Klasse `ASN1` gekoppelten Typen nicht veröffentlicht werden müssen.

Aufgabe. Implementieren Sie die Rahmenklasse.

14.4 Prüfung von Datensätzen

Eine erste Einsatzmöglichkeit von ASN.1–Objektbäumen, die ohne eine Bindung an Datenobjekte einer Anwendung auskommt, ist die Überprüfung, Konvertierung und Speicherung von Datensätzen. Da Binärdaten- und Textdatenkodierung fast den gleichen Arbeitsablauf erfordern, beschränke ich die Diskussion hier der Lesbarkeit halber auf Textdaten. Einzulesen und anschließend wieder Auszugeben ist der Datensatz

```
test { satz { 5 },
       p1 { s1 { "leer" },
            s2 { "Ende Set" } },
       p2 { i2 { 3 } } }
a { 2 }
test2{ i1 { 2 },
       i2 { 3 } }
```

Für die Schreib- und Leseroperationen deklarieren wir in der zuletzt implementierten „Generalklasse“ vier Methoden

```
// Textein- und Ausgabe
bool TextRead(DBuffer s);
bool TextWrite(DBuffer& s);

// Binärdatenein- und Ausgabe
bool BinaryRead(DBuffer db);
bool BinaryWrite(DBuffer& db);
```

Das Lesen eines Datensatzes besteht für jede Variable in der Liste (*und rekursiv für jedes Attribut einer Variablen*) aus zwei Operationen:

(a) Das erste gelesene Wort muss mit dem Namen der Variablen übereinstimmen und das zweite Wort muss vom Typ `{...}` sein. Diese Prüfung ist für alle Typen außer `CHOICE` gleich und wird durch eine in der Klasse ASN1 deklarierte Methode erledigt.
(b) Die Auswertung des zweiten Wortes ist typabhängig und wird durch eine virtuelle Methode abgewickelt, in der außer für den Typ `CHOICE` die Methode (1) aufgerufen wird.

Die Methoden benötigen als Parameter den Puffer mit dem Datensatz, den Index für die Abspeicherung des Wertes im internen Pufferfeld, einen Container zum Eintrag der Daten, die in anderen Variablen benötigt werden sowie einen String mit dem im Rahmen einer Rekursion aufgebauten Variablennamen:

```
virtual void TRead(DBuffer& w, int index,
              vector<pair<string,DBuffer> >& vref,
              string vnam){};
void CTRead(DBuffer&v, DBuffer&w, int index);
```

Die Weitergabe von Daten an nutzende Variable ist in dieser Schnittstelle bereits erklärt: Wie bei der Prüfung, ob die Beziehung hergestellt werden kann, wird der Variablenname sowie der Datenpuffer in einen Container gestellt (*dass der Puffer Daten des korrekten Typs enthält, ist ja bereits geprüft*).[17]

In den Methoden muss natürlich überprüft werden, ob die Indexangaben korrekt sind und die vorgegebenen Wertebereiche von den einzulesenden Daten eingehalten werden. Das sind mehr oder weniger Fleißaufgaben, die ich ohne großen Kommentar Ihnen überlassen kann. Auch das Auslesen von Werten aus dem Referenzpuffer ist an sich nichts Neues, da im Grunde nur die Typprüfung zum Abschluss der ASN.1–Programmauswertung durch das Überspielen des Wertes ersetzt werden muss. Ich beschränke mich daher auf die zwei etwas anspruchsvolleren Fälle `SET` und `CHOICE`.

Bei `SET`–Typen ist die Reihenfolge der Attribute nicht festgelegt, aber es müssen alle Attribute auftreten. Mit Hilfe eines `set<..>`–Containers ist deshalb Buch zu führen, welche Attribute bereits gelesen wurden. Die Überprüfung, welches Attribut als nächstes im Datensatz vorkommt, erfolgt durch Auslesen der Variablenbezeichnung und Suchen des zuständigen Attributes in der Liste. Für `(ito-ifrom)>1` muss dieser Vorgang mehrfach wiederholt werden.

Aufgabe. Implementieren Sie die Methoden für die Standardtypen sowie den `SET`–Typ. Führen Sie parallel die Überlegungen auch für binäre Daten durch.

[17] Das Gesamtverfahren weist eine deutliche Verwandschaft zu der Synchronisation von Objekten in Dialogmenues durch Ereignismeldungen auf. Einmal entwickelte Techniken können an vielen Stellen gewinnbringend wiederverwertet werden.

CHOICE–Daten unterscheiden sich im CHOICE–Objekt von den anderen Verarbeitungsschritten dadurch, dass keine Worte gelesen werden (*siehe oben*). Die Feststellung, welches Attribut im Datensatz kodiert ist, kann ähnlich wie bei SET durch Prüfen der einzelnen Attribute oder durch Fangen von Ausnahmen erfolgen:

```
void ASN1_Choice::TRead(DBuffer& w, int index,
vector<pair<string,DBuffer> >& vref, string vnam){
    ASN1_VarList::iterator it;
    int * coded;
    DBuffer v;
    if(index>=size)
        set_size(index+1);
    DecodeReference(vref);
    v=w;
    coded=reinterpret_cast<int*>(data.begin());
    for(it=attributes.begin();
                it!=attributes.end();++it){
        try{
            (*it)->TRead(w,index,vref,vnam);
            coded[index]=
                distance(attributes.begin(),it);
            return;
        }catch(string s){
            w=v;
        }//endcatch
    }//endfor
}//end function
```

Hierbei wird unterstellt, dass die Variablen bei einem Fehler beim Leseversuch eine Ausnahme werfen.

Die Implementation der Schreibmethoden ist im Grunde noch einfacher und umfasst folgende Teilschritte:

(a) Prüfung auf externe Variablenreferenzen und Lesen der Fremdwerte vom Container.
(b) Prüfen der vorhandenen Werte auf Einhaltung der vorgegebenen Grenzwerte.
(c) Schreiben der Daten auf den Datenstrom.
(d) Eintrag der aktuellen eigenen Werte in den Referenzcontainer, so weit erforderlich.

```
void ASN1_Integer::TWrite(DBuffer& w, int index,
vector<pair<string,DBuffer> >& vref, string vnam){
    string s;
    int * li;
    vector<pair <int,int> >::iterator it;
    if(index>=size)
        ErrorFunction(24,var_name);
```

```
    li=reinterpret_cast<int*>(
            (data.begin()+(index*sizeof(int))));
    if(intervals.size()>0){
        for(it=intervals.begin();
                    it!=intervals.end();++it)
            if(it->first<=*li && *li<=it->second)
                goto OK;
        ErrorFunction(22,var_name);
    }//endif
OK: w.add_back(var_name.c_str(),var_name.length());
    w.add_back('{');
    s=value2string(*li);
    w.add_back(s.c_str(),s.length());
    w.add_back('}');
    if(referenced>=0){
        vref.push_back(pair<string,DBuffer>
                        (vnam+var_name,data));
    }//endif
}//endif
```

Die Methoden sind korrekt implemetiert, wenn die Eingabedaten nach Lesen und Schreiben wieder erscheinen. Die Daten befinden sich zwar jetzt noch in den Datenpuffern der ASN1–Objekte und stehen anderen Anwendungen noch nicht zur Verfügung, trotzdem ist das Erreichte bereits als Anwendung nutzbar, in dem Daten auf die Einhaltung bestimmter einfacher Rahmenbedingungen überprüft werden können, vorausgesetzt, Sie haben Ihre Tests unter den oben beschriebenen Bedingungen durchgeführt und sich vergewissert, dass falsche Daten auch zu einer Meldung durch die Anwendung führt.

14.5 Datenbank und Anwendungsverknüpfung

14.5.1 Ein einfaches Datenbankmodell

Wie schon eingangs bemerkt, werden in vielen Anwendungen nacheinander unterschiedliche Sätze von ASN.1–Variablen benötigt. Beispielsweise wird in Sicherheitsprotokollen die Identität des Kommunikationspartners abgefragt, die Angaben im Bedarfsfall bei einem Dritten überprüft, die Sitzungsparameter ausgehandelt und anschließend eine verschlüsselte transparente Verbindung aufgebaut, die bei längerer Dauer zwischenzeitlich auch das Aushandeln neuer Sitzungsparameter erfordern kann. Das kann beispielsweise folgendermaßen aussehen:

```
HashAlgorithList ::= SEQUENCE {
    algListID  OBJECT IDENTIFIER { 2 3 1771 1 2 },
    knownAlgs  SEQUENCE OF OBJECT IDENTIFIER
} -- Ende der Liste
```

Wie oben schon vermerkt wurde, dient der Datentyp `OBJECT IDENTIFIER` zur eindeutigen Identifizierung eines ASN.1–Programms und einer ASN.1–Datensatzes. Der Wert des Attributes `algListID` legt hier beispielsweise fest, dass es sich um eine Liste unterstützter Hash–Algorithmen handelt. Wenn es also gilt, verschiedene ASN.1–Programme zu verwalten, sind die `OBJECT IDENTIFIER` offensichtlich ein geeignetes Verwaltungskriterium.

> **Aufgabe.** Unsere bisherige Auswertung von ASN.1–Daten beruht auf der Annahme, dass das zugehörende ASN.1–Programm bekannt und geladen ist, das heißt wir können statische Dialoge, in denen die Reihenfolge der unterschiedlichen Datensätze (*auch ohne* `OBJECT IDENTIFIER`) genau festliegt, oder dynamische Dialoge, die einen bekannten Start besitzen und am Ende jedes Satzes verkünden, welchen Typ der nächste Satz aufweist, bearbeiten.

In der Realität muss das jedoch nicht stimmen. Wenn Sie an das Ereignismodell denken, ist es nicht unproblematisch, jeweils eine Abstimmung aller über den nächsten Schritt herbeizuführen. Es muss also auch mit Datensätzen gerechnet werden, die nicht angekündigt sind, sondern sich selbst identifizieren. Entwickeln Sie ein Modell, das Datensätze identifizieren und nach Laden des zugehörenden ASN.1–Programms interpretieren kann.

Unter solchen Bedingungen ist es natürlich nicht angebracht, bei jeder Sitzung aufs neue die ASN.1–Quellen zu übersetzen, sondern ein gespeichertes Bild des jeweiligen Variablencontainers hochzuladen. Zu diesem Zweck konstruieren wir eine einfache Datenbank mit indexgesteuertem Zugriff.[18] Der Index (*Schlüssel*) ist die Identifikationsnummer eines `OBJECT IDENTIFIER`.
Der Schnittstelle der Datenbank geben wir folgendes Aussehen:

```
class SimpleDBase {
public:
    SimpleDBase();
    ~SimpleDBase();

    bool create(string name);
    bool connect(string name);
    bool disconnect();

    bool get(const DBuffer& key, DBuffer& dat);
    bool put(const DBuffer& key, DBuffer& dat);
    bool remove(const DBuffer& key);

    void reconstruct();
    ...
```

[18] Sie können natürlich auch versuchen, eine „echte“ Datenbank zu verwenden (*MySQL oder PostGRE sind beispielsweise kostenlos verfügbar*). Die wollen aber auch erst mal bedient und überredet werden, unsere Daten zu schlucken. Wie hier gezeigt wird, ist es aber auch recht einfach, eine brauchbare Lösung aus vorhandenen Mitteln zu konstruieren.

Daten und Schlüssel übergeben wir als Datenpuffer, um beliebige Daten abspeichern zu können. Die Bank enthält nur eine Tabelle, auf die mit einem Schlüssel zugegriffen wird, und besteht aus zwei Dateien.[19] Die Indexdatei „`name.idx`“ enthält das Indexsystem und wird vollständig geladen, die Datei „`name.dat`“ enthält die Daten, die jeweils aus der Datei gelesen oder in die Datei geschrieben werden. Die Indexdatei wird erst bei Arbeitsende wieder gesichert.[20] `create(..)` erzeugt eine neue Datenbank (*es darf keine mit dem angegebenen Namen existieren*), `connect(..)` öffnet eine existierende und `disconnect()` schließt die Datendatei und sichert die Indexdaten. Für den Transport der Daten und der Schlüssel verwendet wir Datenpuffer. Die Daten werden jeweils an das Ende der Datei geschrieben, im Index wird die Schreibposition und die Satzlänge gespeichert. Zur Implementation des Index eignet sich eine STL–`map`, und wir erhalten folgende Attributliste der Datenbank:

```
typedef pair<int,int> ipair;
typedef map<DBuffer,ipair>  KeySet;
typedef pair<DBuffer,ipair> opair;

FILE * dfile;
string dname;
KeySet keyset;
```

In Datenbanken mit häufigem Austausch von Datensätzen entstehen bei dieser Vorgehensweise (*Schreiben des Datensatzes an das Ende der Datendatei und Überschreiben der Satzzeiger im Index*) im Laufe der Zeit längere ungenutzte Datenblöcke. Die Methode `reconstruct()` „komprimiert“ die Daten wieder, indem die Sätze nach vorne kopiert werden.[21]

Die Indexdatei enthält pro Indexsatz die Einträge:

```
-- Länge des Schlüssels
-- Schlüssel, Anzahl Bytes gemäß erstem Feld
-- Position des Datensatzes
-- Länge des Datensatzes
```

---

[19] Die „echten“ Datenbanken erlauben bekanntlich viele Tabellen, auf denen komplexere Ordnungshierarchien aufgebaut werden. Das benötigen wir hier jedoch gar nicht, also können wir auch auf den Luxus ganz verzichten.

[20] Selbst wenn wir große ASN.1–Programmlisten verwalten müssen, werden wohl kaum mehr als einige Hundert verschiedene Programme zusammenkommen, deren Schlüssel sich bequem im Hauptspeicher eines Rechners verwalten lassen. Den für die Verwaltung von Millionen von Datensätzen notwendigen Aufwand können wir daher auch einsparen.

[21] Eigentlich ist das nicht geplant: Wenn eine Variablenliste einmal deklariert ist, sollte keine Notwendigkeit bestehen, sie zu ändern (*ein neue Version ist ein weiterer kompletter neuer Eintrag und kein Ersatz*). In der Entwicklungsphase kann die Restrukturierungsmethode aber ganz hilfreich sein.

Sie wird in der Methode `disconnect()` geschrieben:

```
dfile=fopen((dname+".idx").c_str(),"wb");
for(it=keyset.begin(),et=keyset.end();it!=et;++it){
    li=it->first.size();
    fwrite(&li,1,sizeof(int),dfile);
    fwrite(it->first.begin(),1,li,dfile);
    fwrite(&it->second.first,1,sizeof(int),dfile);
    fwrite(&it->second.second,1,sizeof(int),dfile);
}//endfor
li=0;
fwrite(&li,1,sizeof(int),dfile);
fclose(dfile);
```

Aufgabe. Schreiben Sie dazu passende Methoden `create(..)` und `connect(..)`. Ergänzen Sie abschließend die restlichen Methoden zum Schreiben, Lesen und Löschen von Datensätzen sowie zur Reorganisation.

14.5.2 Anwendung auf die ASN.1–Objekte

Mit Hilfe dieser Datenbank können wir nun die ASN.1–Bäume speichern und gezielt wieder laden. Auch dieser Vorgang wird wieder rekursiv ausgeführt. Wir bringen zunächst einem ASN.1–Objekt bei, seinen Inhalt an einen Datenpuffer anzufügen beziehungsweise ihn vom Anfang eines Puffers zu lesen und ihn dabei gleichzeitig zu entfernen. Wir implementieren dazu die Methoden

```
class ASN_1 {
    bool Load(DBuffer db);
    DBuffer Store();
    ...
};//end class

class ASN1 {
    virtual bool LoadObject(DBuffer& dat);
    virtual DBuffer StoreObject();
    ...
};//end class

DBuffer StoreD(ASN1_VarList& va);
bool LoadD(ASN1_VarList&va, DBuffer& db);
```

Jedes Objekt schreibt zunächst seine Typkennung auf den Puffer, anschließend seine Attribute und rekursiv die Unterobjekte. Hier ist wieder eine Kreuzrekursion zwischen `ASN1::StoreObject()` und `StoreD(..)` vorgesehen. Beim Rücklesen stellt die Methode `LoadD(..)` zunächst fest, um welchen Objekttyp es sich

handelt, und erzeugt mit Hilfe der Make–Funktion, die wir schon im Interpreter verwendet haben, ein Objekt, das die restlichen Daten einliest und in der Variablenliste gespeichert wird.

Aufgabe. Das Sichern und Rücklesen von Objekten gehört ja nun schon zu den „Standardübungen“, so dass weitere Worte unnötig sind. Implementieren Sie also die genannten Methode und vergleichen Sie sie mit der Objektfabrik.

14.5.3 Verknüpfung mit anderen Datenobjekten

Zum Abschluss dieses Kapitels folgt noch eine Aktion, die nichts für schwache Nerven ist. Bislang stehen die Variablenlisten ja ziemlich einsam in der Gegend. Um echte Anwendungen konstruieren zu können, die mehr machen als einen ASN.1–Datensatz zu kontrollieren, müssen wir noch eine Zugriffsmöglichkeit auf die Daten schaffen. Hierzu besorgen wir uns einen Zeiger auf das Datenelement einer ASN.1–Variablen:[22]

```
DBuffer* DataPtr(ASN1_VarList& va, string varname){
    ASN1_VarList::iterator it;
    DBuffer * ptr;
    for(it=va.begin();it!=va.end();++it){
        if(varname.compare((*it)->var_name.c_str())==0)
            return &(*it)->data;
        ptr=DataPtr((*it)->attributes,
                varname+(*it)->var_name+".");
        if(ptr!=0)
            return ptr;
    }//endfor
    return 0;
}//end function
```

Hierbei gehen wir von folgenden Randbedingungen aus:

(a) Die ASN.1–Variablenliste wird erzeugt oder geladen, bevor ein Anwendungsteil einen Zugriff auf die Daten verlangt.
Die Anwendungsteile müssen also in einer bestimmten Reihenfolge ausgeführt werden. Verstöße sind insofern zunächst unkritisch, als sonst eben Nullzeiger von `DataPtr(..)` zurück gegeben werden.
(b) Die Anwendungsteile kennen die Namen der ASN.1–Variablen, auf die sie zugreifen möchten. Hiervon sollte man wohl in der Regel ohne schlechtes Gewissen ausgehen können.

[22] Den Einbau in bestehende Strukturen nehmen Sie bitte selbst vor.

(c) Beide (*Anwendung und ASN.1–Objekte*) haben die gleiche Vorstellung von den auf dem Puffer befindlichen Daten. Das kann natürlich schon etwas heikel werden, wenn eigene spezielle Datentypen, beispielsweise lange ganze Zahlen, verwendet werden. Durch die Verwendung von `DBuffer` als Vermittler ist aber bei ordnungsgemäßer Verwendung der Klasse zumindest sichergestellt, dass bei der Speichernutzung kein Unfug angestellt wird.

(d) Die ASN.1–Variablenliste wird während der Nutzungsdauer nicht verändert oder gelöscht. Zu solchem Tun besteht normalerweise auch kein Anlass, ein Abweichen hätte aber fatale Folgen, da die Zeiger auf die Puffer ihre Gültigkeit verlieren, ohne dass die Anwendung davon Kenntnis erlangt.[23]

Punkt (d) dieser Liste gibt Anlass, bei konkreten Anwendungen durch Trägerobjekte dafür zu sorgen, dass die ASN.1– und die nutzenden Datenobjekte hinsichtlich ihrer Lebenszeit synchronisiert werden. Wir werden dies im nächsten Abschnitt für Filterketten diskutieren. Bei Verzicht darauf begibt man sich in die dunklen Abgründe der C–Programmierung mit Zeigern, deren Gültigkeit nur noch der Sorgfalt des Programmierers untersteht. Als Mindestsicherung sollten Ausnahmen genutzt werden:

```
DBuffer* b= ... ;
try{
    *b=...;
    ...
}catch(...){
    cerr << "Synchronisationsverlust" << endl;
    exit();
}//endexception
```

Für den Programmablauf bringt das keinen Gewinn, denn im Gegensatz zu den Ausführung in Kapitel 7 handelt es sich hier tatsächlich um Fehler und nicht um Ausnahmen. Wenn die Laufzeitumgebung den Synchronisationsverlust bemerkt, ist mit ziemlicher Sicherheit schon einiges im Speicher kaputt gegangen, und das Programm sollte auf jeden Fall beendet werden. Zumindest weiß man aber, an welcher Stelle man nach den Fehlern zu suchen hat.

Ein echtes „Ausnahmemanagement“ bietet sich an, wenn `DataPtr(..)` einen Nullzeiger zurück liefert. Falls Punkt a) der obigen Liste korrekt bearbeitet wurde, bedeutet das nämlich, dass Daten- und ASN.1–Modell nicht zusammen passen. Da wir es mit einer Laufzeitkonstruktion zu tun haben, über deren Zustandekommen wir uns nicht den Kopf zu zerbrechen haben, sollten wir nur variable Lösungsstrategien anbieten – und dazu ist die Ausnahmeklasse ja konstruiert worden. Die spielt sich aber in den Datenobjekten ab, auf die wir hier nicht weiter eingehen, so dass der

[23] Natürlich kann man Gegenmaßnahmen ergreifen. Anstatt sich darauf zu stürzen ist aber sicher etwas Programmierdisziplin oder Weiterlesen angesagt.

Hinweis genügen muss. Für Kontrollen oder eine Konfliktbehebung bieten wir hier lediglich noch eine Methode an, die alle vorhandenen ASN.1– Datennamen liefert:

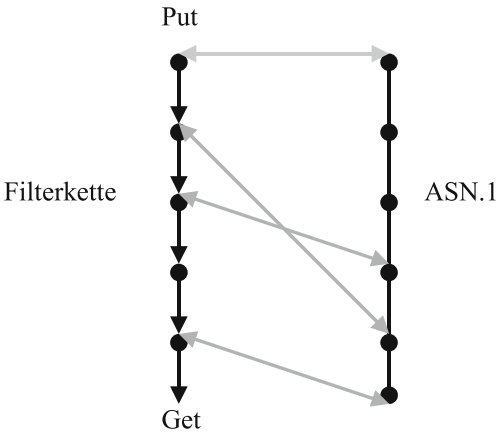
```
DBuffer ASN_1::AllDataPointers(){
    ASN1_VarList * va =
             reinterpret_cast<ASN1_VarList*>(varlist);
    DBuffer db;
    db.add_back("ASN1_Ref(",strlen("ASN1_Ref("));
    db.add_back(GetAllPointers(*va, ""));
    db.add_back("),",2);
    return db;
}//end function
```

14.6 Verknüpfung mit den Filterklassen

Unsere ASN.1–Variablenlisten können nun in beliebige Anwendungen integriert werden. Datenübertragungsprotokolle lassen sich aber auch häufig durch eine Kombination aus Filterketten und ASN.1–Programme beschreiben, so dass wir dies noch etwas systematisieren wollen. Nachdem die Daten von den Algorithmen einer Filterkette verarbeitet sind, sind sie in einer bestimmte Reihenfolge auf den ASN.1–Datenstrom zu bringen. Im Kapitel über Filterketten haben wir einige Werkzeuge bereit gestellt, um das Ergebnis komplexerer Bearbeitungen am Ende der Filterkette entnehmen zu können. Dabei können natürlich auch Kodierungsmaßnahmen nach ASN.1 durchgeführt werden, so dass der fertige ASN.1–Datenstrom entsteht. Diese vermischte Verarbeitung – Algorithmen und ASN.1–Kodierung – passt jedoch weniger zu den bisher entwickelten Werkzeugen, sondern lässt sich eher durch eine Übersetzung eines ASN.1–Kodes in ein C++–Klassenmodell und nachfolgende programmtechnische Einbeziehung der Algorithmen realisieren. Dieser Option werden wir uns im nächsten Teilkapitel zuwenden. Betrachten wir aber Filter und ASN.1–Kodierung als parallele Vorgänge, so können wir wieder mit unseren Werkzeugen arbeiten.

Die Bezeichnung „parallele Vorgänge“ ist so zu interpretieren, dass die Bearbeitung durch die Filterkette parallel zur Arbeit einer Variablenliste durchgeführt wird, jedoch nicht, dass die betreffenden Objekte auch im jeweiligen Container einen vergleichbaren Platz einnehmen. Weit vorne liegende Filterergebnisse können auf ASN.1–Variablen der hinteren oder mittleren Plätze zugreifen:

Hierdurch wird die Verarbeitung möglicherweise sogar einfacher, da in der Filterkette das Trennen und Wiederzusammenführen der Daten großenteils durch direkte Übertragung von Datenteilen auf die ASN.1–Variablen ersetzt werden kann. Für den Transfer der Daten definieren wir eine spezielle Filterklasse, die ähnlich an eine ASN.1–Variable gebunden wird, wie diese untereinander Kontakt herstellen. Sie besitzt als Attribute den Namen „ihrer“ ASN.1– Variablen, einen Zeiger auf deren Datenpuffer und Parameter, welche Daten übertragen werden sollen.



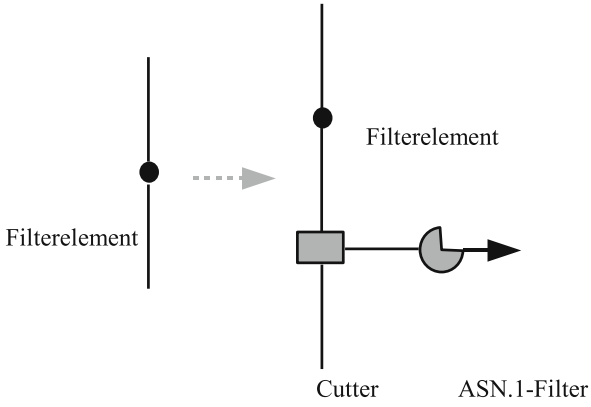

```
NewFactoryClass(ASNFilter,MFilter)

    int Add(MFilter* outQ);
    int Insert(MFilter* outQ, int n);
    bool GetStatus(int oid, DBuffer& db);
    bool SetParameter(int oid, DBuffer& db);
protected:
    string asn_varname;
    int ofs, len, asn_ofs;
    DBuffer * asn_db;

    ASNFilter& operator=(const MFilter&);
    void Prot_Parameter(DBuffer& db);
    void Prot_Status(DBuffer& db);
    void Prot_PutChain();
    void Prot_GetChain();
};//end class
```

Objekte der Klassen sind Datenquellen beziehungsweise Datensenken, das heißt es können keine weiteren Filterelemente an sie angehängt werden. Deshalb werden die Methoden `Add(..)` und `Insert(..)` überschrieben. Sofern sie nicht am Ende einer Kette stehen, sondern wie in unserem Bild Daten aus einer Kette abzweigen, sind sie somit folgendermaßen einzusetzen:

Das Cutterobjekt verdoppelt in seiner Standardkonfiguration den Datenstrom, kann aber auch für zusätzliche Filterfunktionen parametriert werden. Das ASN.1–Filterobjekt überträgt bei jedem Datenabschluss den Inhalt des Ausgabepuffers auf die ASN.1–Variable beziehungsweise liest vor jeder Leseoperation die Daten aus. Um Parametrierungen von Cutterobjekten so weit wie möglich einzuschränken, wird die Datenübertragung durch drei statische Parameter gesteuert:



- `ofs` gibt die Position auf dem Ausgabepuffer ab, ab der Daten kopiert werden sollen,
- `len` spezifiert, wie viele Bytes zu kopieren sind, und
- `asn_ofs` gibt die Zielposition auf dem Puffer der ASN.1–Variablen an.

Aufgabe. Implementieren Sie die Methoden zum Schreiben und Lesen von Daten. Im Normalfall werden jeweils sämtliche Daten kopiert, wobei gegebenenfalls auch die Größe des Puffers der ASN.1–Variablen angepasst wird.

Für die Bindung der Filterobjekte an die ASN.1–Variablenliste vereinbaren wir folgende Vorgehensweise:

(a) Die Filterobjekte werden werden auf die in Kapitel 10.2 eingeführt Weise mit dem Parameterstring

```
ASN1_Var(_variablenname_,_ofs_,_len_,_asn_ofs_)
```

parametriert. Der letzten Parameter können auch fehlen (*das heißt die Parameteranzahl kann 1, 2, 3 oder 4 betragen*). Bestehende Bindungen und Parameterierungen werden dabei aufgelöst.

(b) Mit der Methode `GetStatus(..)` kann der Name der ASN.1–Variablen abgerufen werden. Die Methode wird so modifiziert, dass bei `oid==-1` eine Liste sämtlicher in de Kette parametrierter Variablennamen abgerufen werden kann. Diese Methode dient allerdings nur zu Kontrollzwecken und wird für die Bindung nicht benötigt.

(c) In der Klasse `ASN_1` implementieren wir die Methode `AllData Pointers()`, die einen Datenpuffer mit folgendem Parameterstring erzeugt:

```
ASN1_Ref(_var_1_(_adr_1),_var_2(_adr_2),...)
```

Er enthält sämtliche Variablennamen der Liste zusammen mit den Adressen der Datenpuffer (*über ein geeignetes Format, zum Beispiel* `int`*, in einem String gewandelt*).

(d) Die Filtermethode `SetParameter(..)` wird so modifiziert, dass bei `oid==-1` der Parameterstring (c) ausgewertet werden kann.

Aufgabe. Implementieren Sie nun die Parametrierungsmethoden.

Wir verbinden nun Filterketten und ASN.1–Variablenlisten in einer Trägerklasse, die wir datenbankfähig machen. Die Trägerklasse soll folgende Eigenschaften besitzen:

(a) Filterketten und ASN.1–Variablenlisten werden im Dialog mit einer Anwendung parametriert und in der Datenbank gesichert.
(b) Filterketten und ASN.1–Variablenlisten werden per Befehl aus der Datenbank geladen.
(c) Filterketten und ASN.1–Variablenlisten werden aus dem ASN.1–Datensatz ermittelt und aus der Datenbank geladen.

Beginnen wir mit (a). Eine ASN.1–Variablenliste erhalten wir aus einem ASN.1–Programm. Eine lineare Filterkette aufzubauen ist auch kein Problem, wobei wir dem Trägerobjekt noch nicht einmal fertige Filterobjekte übergeben müssen, sondern nur den Klassennamen; den Rest erledigt die Objektfabrik. Die Parametrierung von Filterobjekten ist ebenfalls kein Problem. Der Aufbau verzweigter Ketten ist komplexer: „vormontierte“ Kettenteile werden mit Cuttern oder Mergern zu der Gesamtfilteranwendung verbunden. Wir schaffen und dazu Raum, indem wir einen Kettencontainer zur Verwaltung von Teilketten sowie die dazu notwendigen Bedienungsmethoden deklarieren. Die Trägerklasse bekommt damit das Aussehen:

```
class AF_Carrier {
public:
    AF_Carrier();
    ~AF_Carrier();

    void clear();
    void clear_queue(int qu);
    int add_front(int queue, MFilter * mf);
    int add_back(int queue, MFilter * mf);
    bool add_queue(int dqu, int squ);
    bool set_parameter(int qu, int n, string par);
    bool set_parameter(int oid, string par);
    bool set_put_chain(int qu);
    bool set_get_chain(int qu);
    bool load_asn(string fname);
    void bind();
```

```
protected:
    typedef MFilter * PFilter;
    typedef APtr<MFilter> AFilter;
    ASN_1 asn;
    vector<AFilter> kette;
    vector<PFilter> kende;
};//end class
```

Der Rückgabewert der Methode `add_front(..)` (*Eine vorhandene Kette wird an das neue Filterobjekt angefügt, das zum neuen Kettenanfang wird*) beziehungsweise `add_back()` (*Anfügen eines neuen Objektes an das Ende einer Kette*) ist die Objektnummer des neuen Objektes, mit der die Parametrierung vorgenommen werden kann. Die Methode `clear_queue(int)` löscht eine Kette aus dem Container, wobei die folgenden Ketten um eine Position aufrücken.

Aufgabe. Implementieren Sie die Methoden.

Die Filterketten werden entweder als Schreibketten oder als Leseketten parametriert. Datensenke für eine Schreibkette beziehungsweise Datenquelle für eine Lesekette ist die ASN.1–Variablenliste. Um korrekt zu funktionieren, müssen die Ketten entsprechend abgeschlossen sein. Kontrollen sehe ich hier nicht vor (*Sie können in Ihren Versionen natürlich etwas hierfür vorsehen*). Sind die Filterketten und die ASN.1–Variablenliste komplett eingerichtet, werden die Bindungen zwischen beiden Teilen durch die Methode `bind()` hergestellt. Dazu ist die Klasse `ASN_1` durch die Methode `DBuffer AllDataPointers()` zu ergänzen, die den oben beschriebenen Parameterstring mit den Adressen der Datenpuffer erzeugt. Damit ist die Anwendung arbeitsfähig.

Für die Verarbeitung von Daten benötigen wir den folgenden Methodensatz, der wohl nicht weiter kommentiert werden muss:

```
/*! Datenverarbeitung */
void FilterPut(int qu, DBuffer& d);
DBuffer FilterGet(int qu);
void MessageEnd(int qu);
int OutMessages(int qu);

void ASNPut(DBuffer& db);
DBuffer ASNGet();
```

Teil (a) unserer Anforderungen an das Trägerobjekt ist hiermit komplett erledigt. Wie die Daten in die Verarbeitung gelangen oder aus ihr wieder entnommen werden, wollen wir hier nicht weiter diskutieren; wir können aber auf diese Weise ein vollständiges Protokoll erzeugen, ohne eine Zeile Programmkode schreiben zu müssen. Wir können somit zu Teil (b) kommen und Ketten sowie ASN.1–Variablenliste in einer Datenbank ablegen. Dazu genügt der Methoden- und Attributsatz:

```
/*! Datenbankfunktionen */
bool set_database(string s);
```

```
bool load_db(DBuffer& key);
bool store_db(DBuffer& key);
```

Als Datenbank verwenden wir die im letzten Teilkapitel entwickelte einfache Datenbank. Der Zugriffsschlüssel ist ein `OBJECT IDENTIFIER`. Für die Speicherung der Filterketten stehen zwei unterschiedliche Möglichkeiten zur Verfügung:

- Die Kettenparametrierung kann protokolliert werden: Da die gesamte Parametrierung nur Textstrings benötigt, können alle Methodenaufrufe der Reihe nach in einem Parameterstring gespeichert werden, der in der Datenbank gesichert wird. Bei der Rekonstruktion aus der Datenbank werden alle Aufrufe in der gleichen Reihenfolge noch einmal wiederholt.

  Diese Methode ist sehr einfach umzusetzen, aber nicht sonderlich elegant, da auch alle Umwege wiederholt werden müssen (*die geringe Verarbeitungsgeschwindigkeit dürfte allerdings kaum eine Rolle spielen*). Eine Zusatzmaßnahme ist für die Parametrierung mit Hilfe der Objekt–ID zu treffen, da bei einem erneuten Aufbau natürlich andere Nummern vom System vergeben werden. Werden die Identifikationsnummern der erzeugten Filterobjekte in einem Vektorcontainer abgespeichert und im Protokoll die Indexnummer im Container anstelle der Objekt–ID gespeichert, so wird der Parametrierbefehl bei der Rekonstruktion mit der im Vektorcontainer gespeicherten ID (*der Container wird natürlich bei der Rekonstruktion zunächst gelöscht und dann wieder aufgebaut*) korrekt ausgeführt, da die Indexposition eines bestimmten Objektes konstant bleibt.
- Die Ketten können mit Hilfe Objektfabrik gesichert und gelesen werden. Die Vorgehensweise entspricht derjeniger der Dateisicherung von Fabrikobjekten; da wir den Datenpuffer aber erst nach der Objektfabrik entwickelt haben, herrscht hier noch Nachholbedarf.

  Diese Lösung verlangt zwar etwas mehr Aufwand als die erste, ist aber sicher eleganter, da die fertigen Ketten ohne jegliche Umwege gesichert werden.

Wir werden hier die zweite Möglichkeit diskutieren; sie können aber natürlich auch die erste umsetzen. Wir setzen hier bei der Filterbasisklasse an; alternativ kann natürlich auch die Objektfabrik erweitert werden. Zum Schreiben der Daten auf einen Puffer führen wir eine virtuelle Methode ein, die ebenfalls etwas weniger Aufwand als in der Objektfabrik beinhaltet.[24]

[24] Die Versionsprüfung fehlt hier beispielsweise, ebenso die Trennung in eine allgemeine Verwaltungsmethode und eine spezielle Datensicherungsmethode. Die Implementation der Methoden habe ich bei den Filtern vorgenommen, da hier ohnehin Maßnahmen zur Berücksichtigung der Verkettung notwendig sind. Sie können dies selbständig auf die Objektfabrik ausdehnen, da hier ohnehin eine Ergänzung notwendig ist, denn wir benötigen bei der hier gewählten Verarbeitungsart einen freien Zugriff auf den Standardkonstruktor der Filter, den die Objektfabrik eigentlich nicht zur Verfügung stellt. Sie können dies beispielsweise durch ein zweites Makro, das die Konstruktoren im public–Bereich führt, erreichen.

```
virtual DBuffer MFilter::ObjectToBuffer() const{
    DBuffer db;
    db.add_back(ClassName().c_str(),
                    ClassName().length()+1);
    db.add_back(ClassName().c_str(),
                    ClassName().length()+1);
    db.add_back(put_chain);
    db.add_back(get_chain);
    if(chain!=0){
        db.add_back(1);
        db.add_back(chain->ObjectToBuffer());
    }else{
        db.add_back(0);
    }//endif
    return db;
}//End function
```

Die Klassenbezeichnung wird wieder doppelt erfasst, um bei der Rekonstruktion die Klasse sicher erkennen und erzeugen zu können. Die angeketteten Objekte werden ebenfalls auf den Puffer geschrieben. Erbende Klassen rufen diese Methode auf und schreiben anschließend ihre Daten auf den Puffer.

> **Aufgabe.** Für die Rekonstruktion wird eine statische Methode zum Erzeugen eines Objektes und eine virtuelle Methode zum Lesen der Daten benötigt. Implementieren Sie die Methoden. Die gelesenen Daten werden vom Puffer entfernt, so dass die nächste Methode ihre Auswertung am Beginn der Puffers beginnt.

Damit lassen sich alle Daten eines Protokolls auf einem Puffer unterbringen und sichern:

```
Pufferstruktur:
===============
Anzahl der Filterketten (int)
... Kette 1 ...
... Kette 2 ...
...
... Kette n ...
ASN.1-Variablenliste
```

> **Aufgabe.** Komplettieren Sie nun die Datenbankmethoden. Die Datenbank kann bis zur Änderung des Namens geöffnet bleiben, so dass schnelle Zugriffe möglich sind. Vergessen Sie nicht, zum Abschluss die Bindung zwischen Ketten- und ASN.1–Objekten herzustellen.

Kümmern wir uns nun um Aufgabe (c) . In Fehler: Referenz nicht gefunden haben Sie ja bereits eine Methode implementiert, um nach bestimmten Regeln `OBJECT IDENTIFIER` in einem binären ASN.1–Datensatz zu identifizieren. Nennen wir sie

```
vector<DBuffer> Scan_ASN1_OID(DBuffer db);
```

und kontrollieren wir die Implementation: Es ist klar, dass `OBJECT IDENTIFIER (OID)` nur dann ohne weitere Informationen erkannt werden können, wenn sie die Normalkodierung

```
class= UNIVERSAL   Type=6
Bitkodierung:  00 0 00110
```

aufweisen. Ein OID tritt als

- Eintrag in einem Puffer auf (und ist dann für die Struktur des gesamten Puffers verantwortlich)– oder
- der Puffer enthält nur zusammengesetzte Datentypen mit einem OID als erstem Attribut.

Zusammengesetzte Typen lassen sich an der Bitkodierung `xx 1 xxxxx` erkennen. Die Methode liefert somit alle OIDs in der ASN.1–Binärkodierung. Damit nachfolgend die passenden Anwendungen aus der Datenbank geladen werden können, müssen die Schlüssel binärkodierte OIDs sein.

Bei der Überprüfung eines ASN.1–Binärdatensatzes muss davon ausgegangen werden, dass mehrere OIDs identifiziert werden und der Datensatz mehr oder weniger willkürlich aus diesen Objekten rekrutiert wurde, das heißt aus der Datenbank lassen sich zwar (*hoffentlich*) passende Protokolle zu allen gefundenen OIDs laden, es existiert jedoch kein Gesamtobjekt. Um den Datensatz komplett auswerten zu können, laden wir deshalb alle Objekte hintereinander in die Anwendung:

```
// Fügt weitere ASN.1 - Variablen an die
// vorhandene Liste an
bool ASN_1::AddLoad(DBuffer db);

// Fügt weitere Anwendungen an. Die neue
// Filterkette beginnt an der im Rückgabewert
// angegebenen Position
int AF_Carrier::add_load_db(DBuffer& key);
// Führt den Pufferscan durch und lädt die
// Anwendung
typedef pair<DBuffer,int> sc_chain;
typedef vector<sc_chain>  sc_container;
sc_container AF_Carrier::ScanAndLoad(DBuffer db);
```

Die Methode `ScanAndLoad(..)` führt den Ladevorgang vollständig durch und liefert als Rückgabewert alle notwendigen Informationen für die Anwendung: Im ersten Feld von `sc_chain` ist der OID eingetragen, im zweiten Feld der Index des Beginns der dazugehörenden Filterkette, so dass die Anwendung weiß, was zu tun ist.

Damit haben wir nun auch ein einfache Möglichkeit in der Hand, eine Anwendung zu erweitern: Treten in einem Protokoll unbekannnte OIDs auf, so genügt

es, die speziellen ASN.1–Programme nachzuschieben, um wieder alles auswerten zu können. Bereits implementierte Teile müssen dabei nicht angefasst werden. Auch gegenüber unterschiedlichen Reihenfolgen von Daten in einem Satz sind die Anwendungen nun unempfindlich.

Wir schließen die Untersuchungen durch eine letzte Erweiterung ab, in dem wir den ASN.1–Variablen zwei Attribute `rd_active` und `wr_active` spendieren, die es erlauben die Variable beim Lesen oder Schreiben zu überspringen (*Standard: die Variablen sind aktiv*). Mit

```
struct asn_active {
    string vname;
    bool rd,wr;
};//end struct

void ASN_1::SetActivity(const asn_active& act);
void ASN_1::SetActivity(vector<asn_active>& act);
```

sowie den entsprechenden Pendant in der Klasse `AF_Carrier` können einzelne oder ganze Gruppen von Variablen geschaltet werden, wobei klar ist, dass aus dem Abschalten einer zusammengesetzten Variable natürlich das Abschalten sämtlicher Attribute folgt, ohne dass dies angegeben werden muss.

Aufgabe. Implementieren Sie die Methoden zum Laden von Protokollen nach identifizierten OIDs und zum Schalten von ASN.1–Variablen.

Mit kompletten Anwendungen muss ich Sie nun allerdings alleine lassen, da hierzu natürlich jeweils ein kompletter Satz an Filtern (*mit Algorithmen*) und ASN.1–Datenbehandlungsmethoden notwendig ist. Bezüglich der Algorithmen habe ich schon auf Möglichkeiten verwiesen, sich bei fertigen Bibliotheken zu bedienen. Für Versuche sind aber auch Vergleichsdaten notwendig, denn es nützt nur begrenzt, wenn eine Anwendung Ihre Daten Lesen und Schreiben kann, aber andere nicht versteht. Suchen Sie daher nach Daten mit bekanntem Inhalt, beispielsweise ASN.1–kodierten Zertifikaten, und testen Sie damit Ihre Implementation.

14.7 Compilezeit – Implementation

Wir gehen jetzt auf die zweite Möglichkeit, ASN.1–Code in eine Anwendung umzusetzen, ein. Durch die Ähnlichkeit zu Programmiersprachen besteht die Möglichkeit, ASN.1–Typvereinbarungen in Klassendefinitionen zu übersetzen und diese Klassen zum Ausgangspunkt der Anwendungsprogrammierung zu machen. Die Klassen besitzen bereits alle Funktionalitäten, um ASN.1–Daten zu erzeugen oder einzulesen, gegebenenfalls bereits mit Ausnahmemanagement für die Behandlung von Fehlern beim Datenaustausch. Sie müssen „nur noch“ mit den einzelnen Algorithmen verbunden werden, wobei bei geschicktem Grundaufbau das „nur noch“ durchaus seine Berechtigung hat. Die charakteristischen Unterschiede zu unser ersten Vorgehensweise sind damit:

- Es werden ASN.1–Strukturdefinitionen (ASN.1–Module) übersetzt, während Variable nicht berücksichtigt werden (*diese Funktion übernimmt die anschließende Anwendungsprogrammierung*).
- Die Verbindung mit den Algorithmen erfolgt durch eine Programmierung. Jede spätere Änderung oder Erweiterung muss ebenfalls programmiert werden.

Wir sehen uns das an dem gegenüber unseren vorherigen Untersuchungen etwas einfacheren Beispiel

```
Teil1 ::= CHOICE {
    s1  BOOLEAN,
    s2  INTEGER
    } -- end set

Teil2 ::= CHOICE {
    i1 [0] IMPLICIT INTEGER,
    i2 [1] IMPLICIT INTEGER
    } -- end choice

TType ::= SEQUENCE {
    satz INTEGER,
    p1   Teil1,
    p2   Teil2
    } -- end sequence
```

etwas genauer an. Das ASN.1–Modul wird Typ für Typ in C++–Code umgesetzt, wobei ich hier gleich das Endergebnis präsentiere

```
class Teil1 : public AsnChoice{
public:
    ...
    enum {choiceS1,  choiceS2} m_nChoice;
    AsnBool m_s1;
    AsnInt m_s2;
protected:
    void DecodeChoice(DBuffer& bt);
    void EncodeChoice(DBuffer& bt) const;
};//end class

class Teil2 : public AsnChoice{
public:
     ...
    enum {choiceI1,  choiceI2} m_nChoice;
    AsnInt m_i1;
    AsnInt m_i2;

protected:
    void DecodeChoice(DBuffer& bt);
```

```
    void EncodeChoice(DBuffer& bt) const;
};//end class

class TType : public AsnSequence{
public:
    TType(int nTag=defaultTag,
           int nClass=defaultCls);
    AsnInt m_satz;
    Teil1 m_p1;
    Teil2 m_p2;
protected:
    void DecodeElements(DBuffer& bt);
    void EncodeElements(DBuffer& bt) const;
};//end class
```

Die Umsetzung eines ASN.1–Moduls in ein C++–Modul ist einfacher als die Übersetzung von ASN.1–Programmen. Da wir es mit einem Programmiersystem zu tun haben, können wir großzügiger mit den Auswertungsparametern umgehen. Zunächst ist es sinnvoll, in den C++–Dateien zu notieren, welche ASN.1–Module verwaltet werden:

```
// C++ - Header - File:
// ===================
// asn-managed-file:  _asn_file_1
// asn-managed-file:  _asn_file_2 ...
```

Diese Initialisierung der C++–Dateien kann von Hand vorgenommen werden.[25] Die Übersetzung von ASN.1–Typen in C++–Klassen nehmen wir selektiv vor: In einem ASN.1–Modul befinden sich unter Umständen viele nicht benötigte Typen, und die gewünschten Typen stehen in einer für die C–Programmierung nicht unbedingt geeigneten Reihenfolge. Wir steuern Typenumsetzung und Reihenfolge manuell, indem in einem String alle gewünschten Typen in der korrekten Reihenfolge vorgegeben werden:[26]

```
bool asn2C_addTypes(string c_file, string typenames)
```

Der Compiler öffnet die (*ohne Extension*) angegebene C++–Headerdatei und prüft zunächst, ob die angegebenen Typen nicht bereits definiert sind. Das kann durch Testen auf Vorliegen von `class _typename_` in der Datei erfolgen; für die weitere Arbeit ist es aber sinnvoll, die Klassen durch Kommentare in Sektionen zu trennen und alternativ diese zu verwenden:

---

[25] Mit ein bisschen Snobismus kann man natürlich auch eine Initialisierungsmethode dazu implementieren. Was nun günstiger ist – sich zu erinnern, wie die Methode heißt und zu bedienen ist oder schnell die Zeilen aus einer vorhandenen Datei zu kopieren – muss jeder selbst entscheiden.

[26] Hier den Endtyp vorzugeben und alles weitere dem Compiler zu überlassen, wäre allerdings ein wenig zu viel des Snobismus, wenn man sich hier nur ein einfaches, aber effektives Werkzeug konstruieren möchte.

```
// start-asn-section: TType
class TType : ...
...
// end-asn-section
```

Sind die Typen noch nicht erfasst, liest der Compiler die aufgelisteten ASN.1–Module ein (*hierzu können Sie die Methoden aus dem ersten Durchgang verwenden*) und überprüft, ob dort Definitionen der Form `_type_ ::=` vorliegen.

Die Auswertung kann nun in der gleichen Weise vorgenommen werden, wie dies im ersten Durchgang erfolgt ist, wobei wir einschränkend rekursive direkte Felddeklarationen ausschließen. Sehen wir uns an einem Beispiel an, warum:

```
Type_1 ::= SEQUENCE { feld ::= SEQUENCE { ... } ... }
```

Für das Attribut `feld` ist eine Strukturvereinbarung notwendig, die innerhalb der Klasse `Type_1` angelegt werden muss und für die sich der Compiler einen Namen überlegen darf. Die Deklaration ist daher aufzuspalten in [27]

```
Feld ::= SEQUENCE { ... }
Type1 ::= SEQUENCE { feld Feld  ... }
```

Aufgabe. Die Erzeugung der C++–Dateien sollte unter Verwendung der Erfahrungen aus dem ersten Modell problemlos durchführbar sein.

Bei der Bearbeitung der Ergebnisse, insbesondere nach Ergänzung weiterer Klassen, mit dem C++–Compiler kann das Problem auftreten, dass die Klassen noch nicht in der richtigen Reihenfolge stehen. Bei kleineren Aufgaben wird dieses Problem von Hand durch Verschieben der Sektionen geregelt, bei größeren Aufgaben kann aber auch eine Automatik implementiert werden:

Aufgabe. Die Sektionen können als Textblöcke eingelesen und sortiert werden. Die Sortierreihenfolge

```
section_1 < section_2
```

ist gleichbedeutend mit

```
section_2.textstring.find(section_1.name+" ")>0
```

Ordnen Sie die Sektionen nach dieser Methode in der Header–Datei neu an.

[27] Beides, sowohl die Definition von Strukturen in Klassen als auch die Vergabe von Namen für diese Strukturen, ist im Grunde kein allzu großes Problem, so dass Sie diese Beschränkung eher als Empfehlung betrachten können. Allerdings leidet die Übersichtlichkeit nicht unerheblich unter der Schachtelung, und im Gegensatz zu unserem ersten Modell müssen wir mit dem Ergebnis der ASN.1–Umsetzung die Anwendungsprogrammierung ja erst noch vornehmen. Überhaupt empfiehlt es sich, bei Typschachtelungen „den Ball flach zu halten“, um in der Programmierung nicht häufig auf Attributzugriffe der Art `a.b.c.d.e.f=15` angewiesen zu sein.

Auch bei der Änderung von Typen ist das Sektionskonzept recht hilfreich. Aus der vorhandenen Datei können die Typen ausgelesen und die ASN.1–Module neu übersetzt werden. Entstehen dabei andere Ergebnisse, werden die Sektionen in den Dateien ausgetauscht. Führen Sie dies als `Aufgabe` durch.

Außerhalb der Sektionen kann nun auch weiterer Code untergebracht werden, der bei den Ergänzungen oder Änderungen unverändert bleibt (*falls Sie daran nicht gedacht haben, müssen Sie Ihre Editorfunktionen noch einmal erweitern*).

Um den Entwicklungsrahmen komplett zu machen, müssen lediglich noch Implementationen für die Standardtypen `AsnChoice, AsnSequence, AsnInteger` usw. geschrieben werden. Auch hier können die Ergebnisse aus dem ersten Modell übernommen werden, wobei die Attribute in Abhängigkeit von den verwendeten Algorithmen auch anders (*direkter*) gestaltet werden können. Das weitere ist dann Anwendungsprogrammierung, wobei ich Sie aus den gleichen Gründen wie im ersten Modell sich selbst überlasse.