Visualisierung von ASN.1–Werten

Eine prototypische Implementierung

Jens Grabowski

Stefan Heymer

Beat Koch

13. November 1996
Bericht B-96-01

Zusammenfassung

Dieser technische Bericht beschreibt ein Programmpaket zur Visualisierung von Datenwerten, die in der Wertnotation der Datenspezifikationssprache ASN.1 beschrieben wurden. Das Paket besteht aus zwei Teilen, einer Übersetzungs- und einer Visualisierungskomponente. Die Übersetzungskomponente wurde in Prolog geschrieben. Sie erzeugt einen internen Zwischencode, der als Eingabe für die Visualisierungskomponente dient. Die Visualisierungskomponente wurde in Java programmiert.

Dieser Bericht ist ein Ergebnis einer bilateralen Kooperation zwischen der Siemens Schweiz AG und den Forschungsgruppen von Prof. Dieter Hogrefe an den Universitäten Lübeck und Bern. Die Autoren möchten allen Personen, die diese Arbeit möglich gemacht haben, ihren Dank ausdrücken.

Abstract

This technical report describes a program package for visualisation of data values defined in the value notation of the data specification language ASN.1. The package consists of two parts, a translation and a visualisation component. The translation component has been written in Prolog. It produces an internal intermediate code, which serves as input for the visualisation component. The visualisation component has been programmed in Java.

This report is an outcome of a bilateral cooperation between the Siemens Schweiz AG and the research groups of Prof. Dieter Hogrefe at the Universities of Lübeck and Berne. The authors express their thanks to all persons that made this work possible.

CR Categories and Subject Descriptors:
C.2.2 [Computer Communication Networks]: Network Protocols; D.3.0 [Programming Languages]: General

General Terms:
ASN.1, Java, Prolog, Visualisation

Inhaltsverzeichnis

Kapitel 1

Einführung

In diesem Bericht beschreiben wir ein Programmpaket zur Visualisierung von Datenwerten, die mit der standardisierten Spezifikationssprache ASN.1 definiert wurden. Dieses Programmpaket besteht aus zwei separaten Teilen:

- Einem Übersetzer, der die ASN.1–Spezifikationen einliest und in einen Zwischencode übersetzt, und
- einem Visualisierungsprogramm, das diesen Zwischencode einliest und in Baumform auf dem Bildschirm darstellt.

Ein Augenmerk lag bei der Entwicklung auf der Portabilität der gefundenen Lösungen. Aus diesem Grunde wurde die Visualisierungskomponente in der Programmiersprache Java erstellt. Die stellenweise uneindeutige Syntax der Spezifikationssprache ASN.1 führte dazu, daß der Übersetzer in Prolog entwickelt wurde, einer logischen Programmiersprache, die sich vor allem im Bereich der Computerlinguistik für die Implementierung von Grammatiken etabliert hat.

Die folgenden zwei Abschnitte dieses befassen sich getrennt mit den beiden Komponenten des Pakets. Interessenten an den beschriebenen Programmen mögen sich bitte an die Autoren unter den email–Adressen

`{grabowsk,heymer,bkoch}@itm.mu-luebeck.de`

wenden.

Kapitel 2

Ein Übersetzer für ASN.1

Dieser Abschnitt beschreibt einen in Prolog geschriebenen Übersetzer für die Spezifikationssprache ASN.1. Er ist Teil eines Programmpakets, das zur Visualisierung von ASN.1–Datenwerten dienen soll. Während der von Beat Koch in der Programmiersprache Java geschriebene Programmteil der Darstellung der Datenwerte einer Spezifikation aus einer Zwischendarstellung dient, soll der hier beschriebene Übersetzer eine ASN.1–Spezifikation in eben diese Zwischendarstellung übersetzen.

Dieser Abschnitt wurde von Stefan Heymer geschrieben.

2.1 Warum Prolog?

Für gewöhnlich werden Übersetzer mit den Werkzeugen `lex` und `yacc`, beziehungsweise deren GNU–Varianten `flex` und `bison` entwickelt. Ebenso gibt es eine Reihe weitere dedizierter Werkzeuge, die aber meist Verbesserungen der Standardwerkzeuge sind, oder andere Analysetechniken als die von `yacc` und `bison` durchgeführte LALR(0)–Analyse implementieren.

Warum wurde der hier beschriebene Übersetzer also in Prolog programmiert? Hierfür gibt es eine Reihe von Gründen:

- Die in [Steedman 1993] präsentierte Grammatik für ASN.1 ist an einigen Stellen nicht eindeutig, was eine *bottom–up*–Analyse, wie sie von Analyseprogrammen durchgeführt wird, die mit `yacc` oder verwandten Parsergeneratoren erzeugt werden, zum Teil beträchtlich erschwert.

- Die Fehlersuche in Prolog–Programmen wird durch die in den meisten Prolog–Interpretern eingebauten Debugger nach dem *box*–Modell sehr vereinfacht. Hiermit hat man die Möglichkeit, die Analyse eines Ausdruckes Schritt für Schritt zu verfolgen. Eine solche Möglichkeit bietet sich für Analysatoren, die mit den üblichen Parsergeneratoren erstellt werden, zumeist nicht.

- Die von den meisten Prolog–Interpretern zur Verfügung gestellten logischen Grammatiken (DCGs, *definite clause grammars*) stellen eine einfache Möglichkeit dar, Analyseprogramme auch für nicht–kontextfreie Sprachen zu programmieren.

- Da Prolog für gewöhnlich interpretiert wird, läßt es sich hervorragend für die schnelle Entwicklung von Prototypen einsetzen. Inzwischen gibt es auch Prolog–Übersetzer, die von Prolog in C–Code oder Maschinencode übersetzen, was die durch die Interpretation entstehenden Laufzeitnachteile kompensieren kann.

Alles in allem kann man sagen, daß die in diesem Dokument beschriebenen Arbeiten gezeigt haben, daß Prolog auch für die Erstellung von Übersetzern eine ernstzunehmende Alternative darstellt.

In den folgenden Unterabschnitten werden wir noch einmal gezielter auf zwei Teilbereiche eingehen, die für das Verständnis der Funktionsweise des erstellten Übersetzers wichtig sind: die Behandlung von Doppeldeutigkeiten in der behandelten Grammatik, und die Verwendung logischer Grammatiken für die Formulierung des Übersetzers.

2.1.1 Doppeldeutigkeiten und Nichtdeterminismen

Die in [Steedman 1993] angegebene Grammatik für ASN.1 ist an einigen Stellen nicht eindeutig. Das beste Beispiel hierfür bieten die beiden Produktionen für Mengen- und Sequenzwerte, nämlich

```
SetValue ::= "{" [NamedValue {"," NamedValue}] "}"
```

und

```
SequenceValue ::= "{" [NamedValue {"," NamedValue}] "}".
```

An dieser Stelle muß sich der Analysealgorithmus zwischen den beiden Alternativen entscheiden, „weiß“ aber noch nicht, welche Alternative die richtige sein wird. Da in ASN.1 auch die übliche Technik des *definition before use*, also der Definition eines Typen *vor* seiner Verwendung, nicht verpflichtend festgelegt ist, muß die endgültige Entscheidung für eine der beiden Alternativen bis zur Überprüfung der Typen verschoben werden — oder auf kompliziertere Weise gelöst werden. Hier muß für die „handelsüblichen“ Analysealgorithmen die kompliziertere Variante gewählt werden, denn ist ein (zum Beispiel) in C geschriebenes Programm erst bei der Typüberprüfung angekommen, gibt es eigentlich keinen Weg mehr *zurück*, um früher getroffene Entscheidungen zu korrigieren.

In Prolog ist hingegen bereits ein Verfahren zur Korrektur solcher Fehlentscheidungen Teil der Sprache selbst: das sogenannte *backtracking*. Kurz gesprochen beschreibt dies die Technik bei der Suche nach einer Lösung. Wird an einer Stelle der Suche festgestellt, daß eine Lösung nicht erreicht werden kann, so wird an den letzten Punkt zurückgegangen, an dem eine Entscheidung zwischen Alternativen stattgefunden hatte. An dieser Stelle wird dann die nächstmögliche Alternative ausgewählt.

Übertragen auf einen in Prolog geschriebenen Übersetzer für ASN.1 bedeutet dies, daß solche Doppeldeutigkeiten in der Grammatik während der Erstellung des Übersetzers nicht weiter betrachtet werden mußten — im Falle einer Doppeldeutigkeit muß das System einfach die weiteren möglichen Alternativen abarbeiten, oder alle möglichen Lösungen (das heißt Syntaxbäume und Ausgaben) für die eingegebene ASN.1–Spezifikation berechnen.

Diese Abarbeitungstechnik hat direkte Auswirkungen auf die Quellcodegröße für die Analysekomponente. Während die Parser–Komponente des in [Sample 1993] beschriebenen ASN.1–Übersetzers mit `yacc` programmiert wurde und kommentiert etwa 3000 Zeilen lang ist, benötigt der (unkommentierte) in Prolog geschriebene Analysator, wie er hier beschrieben wird, nur etwa 1000 Zeilen.

2.1.2 Logische Grammatiken

Neben der standardmäßig im Prolog–Sprachumfang enthaltenen Auswertung von Anfragen mit *backtracking* enthalten mehr und mehr Prolog–Implementierungen auch *logische Grammatiken*, also Grammatiken, die auf der in Prolog implementierten Logik basieren. Eine solche Art von Gram-

matik sind die verbreiteten *definite clause grammars (DCGs)*, neben denen es aber vor allen Dingen aus dem Bereich der Computerlinguistik weitere Grammatiktypen gibt. Für die in diesem Papier dargestellten Arbeiten beschränken wir uns jedoch auf DCGs.

Betrachten wir ein Beispiel. Eine DCG für einfache römische Zahlen kann wie folgt angegeben werden:

```
zahl(1) --> "I".
zahl(5) --> "V".
zahl(10) --> "X".
zahl(50) --> "L".
zahl(100) --> "C".
zahl(500) --> "D".
zahl(1000) --> "M".
```

Hier sieht man zum einen schon die Syntax–Schreibweise von Prolog, die *Produktionen*. Den Produktions–Operator `-->` kann man dabei als „entsteht aus“ lesen. Damit bedeutet zum Beispiel die erste Produktion, daß `zahl(1)` aus der Zeichenkette `"I"` entsteht.

Um die zusammengesetzten römischen Zahlzeichen verarbeiten zu können, benötigt man weitere Regeln:

```
roemisch(R)
  --> zahl(X), zahl(Y),
      { X < Y },
      !,
      roemisch(R1),
      { R is Y - X + R1 }.
roemisch(R)
  --> zahl(X), roemisch(R1),
      { R is X + R1 }.
roemisch(0)
  --> [].
```

Die zweite Produktion zeigt den normalen Verlauf des Aufbaus zusammengesetzter römischer Zahlen — die Zahlenwerte der einzelnen Zeichen werden zusammengezählt. Hier zeigen sich gleich zwei Eigenarten der DCGs: Nichtterminalsymbole wie `roemisch` und `zahl` können parametrisiert wer-

den, insbesondere mit Variablen, denen durch den Prolog eigenen Unifikationsmechanismus bei der Auswertung Werte zugeordnet werden. Außerdem sind auf der rechten Seite einer DCG–Produktion auch Prolog–Aufrufe wie `R is X+R1` möglich, sie sind dann in geschweiften Klammern anzugeben.

Die erste Produktion zeigt die Behandlung der Ausnahmen wie „IV“ oder „CM“. Hier wird nach zwei hintereinanderstehenden Zahlzeichen zuerst geprüft, ob das linke Zeichen kleiner ist als das rechte. Ist dies der Fall (wie zum Beispiel in „IV“), so sollen die beiden Zahlzeichen nicht zueinander additiert, sondern voneinander subtrahiert werden. Der Schnittoperator `!` unterbindet das *backtracking*, wenn die Auswertung bis zu diesem Punkt vorgedrungen ist — in diesem Falle werden die zweite und dritte Produktion nicht mehr als Alternativen angesehen. Die dritte Produktion behandelt letztlich nur die 0, für die es kein römisches Zahlzeichen gibt.

2.1.3 Vorteile logischer Grammatiken

Was zeigt uns nun dieser kurze Ausflug? Logische Grammatiken haben einige Vorteile:

- Die Verwendung von DCGs erlaubt es, sich auf die Grammatik zu konzentrieren, anstatt „Auswege“ aus mehrdeutigen Passagen der Grammatik suchen zu müssen.
- Die Verwendung von Prolog erlaubt durch das inherente *backtracking* eine knappe, übersichtliche und eingängige Darstellung von Grammatik und Auswertung.
- Die Parametrisierbarkeit der Nichtterminal in DCGs erlaubt es auch, in Prolog Sprachen zu erkennen und zu übersetzen, die durch die üblichen $LALR(0)$- und $LL(1)$–Übersetzer nicht behandelbar sind.
- Letztlich erlaubt es Prolog (bei schnittfreien Grammatiken) auch, mit ein- und derselben Grammatik sowohl Sätze einer Sprache zu erkennen, als auch sie zu erzeugen (von dieser Eigenschaft werden wir in diesem Papier allerdings keine Verwendung machen).

2.2 Der Übersetzer

Ein Übersetzer übersetzt eine Quelldatei in mehreren Phasen in einen sogenannten Zielcode. Dabei werden die folgenden Phasen durchlaufen:

- Der lexikalischen Analyse, die die Quelldatei in einen Strom sogenannter *lexikalischer Einheiten* oder *tokens* zerlegt,
- der syntaktischen Analyse, die den so gewonnenen Tokenstrom aauf Korrektheit bezüglich der übersetzten Sprache überprüft und einen Syntaxbaum erzeugt,
- der semantischen Analyse, die zum Beispiel die Korrektheit von Typen überprüft und einen Zwischencode erzeugt, und
- der Codeerzeugung, die aus dem Zwischencode und weiteren Informationen die Ausgabe des Übersetzers, den Code, erzeugt.

Für den hier betrachteten Übersetzer wurde aus Zeitgründen auf eine semantische Analyse verzichtet. Dafür kommt eine Brücke zwischen lexikalischer und syntaktischer Analyse hinzu. Wir beschreiben im folgenden den Programmaufbau für die einzelnen Phasen des Übersetzers. Dabei zeigen wir jeweils einige wichtige Klauseln.

2.2.1 Die lexikalische Analyse

Die lexikalische Analyse, also die Aufteilung des Eingabestroms in einen Strom lexikalischer Einheiten, der sogenannten Tokens, ist im beschriebenen Übersetzer ebenfalls als *definite clause grammar* ausgeführt. Startpunkt dieser Grammatik ist das Prädikat `lexemes`, das wie folgt definiert ist:

```
lexemes(L) --> spaces, lexemeList(L).
```

Dabei wird mit dem Prädikat `lexemeList` die eigentliche Liste von Tokens aufgebaut, die in der Eingabe enthalten ist. Das Prädikat `spaces` „überliest“ dabei sogenannten *white space*, also Leerräume oder Kommentare.

Die Liste der in der Eingabe enthaltenen Token wird durch eine rekursive Definition des Prädikats `lexemeList` erzeugt. Dieses ist definiert als:

```
lexemeList([L | Ls]) --> lexeme(L), !, spaces, lexemeList(Ls).
lexemeList([]) --> [].
```

Dabei dient die zweite Klausel als Abbruchkriterium, um die Rekursion am Ende des Eingabestroms zu beenden. In der ersten Klausel wird zuerst das am Anfang des Stromes stehende Token bestimmt, um danach zuerst den folgenden *white space* zu beseitigen, und dann die Liste der weiteren Lexeme zu bestimmen. Der in der ersten Klausel auftretende Schnitt `!` dient der Eliminierung von Auswahlpunkten: Es soll hier vermieden werden, daß ein bereits gefundenes Lexem durch *backtracking* auch in Teilabschnitten als Lösung gefunden wird.

Ein Lexem kann nun sein

- eine Zahl (`num`),
- eine Zeichenkette (`cstring`),
- eine binäre (`bstring`) oder hexadezimale (`hstring`) Konstante,
- ein Interpunktionszeichen, oder
- ein Bezeichner (`id`) oder reserviertes Wort.

Die ersten drei Fälle werden dabei unter dem Begriff „Konstante“ zusammengefaßt, während für den letzten Fall zwischen Bezeichnern und reservierten Worten unterschieden werden muß. Dies zeigt sich im Aufbau der entsprechenden Klauseln für das Prädikat `lexeme`:

```
lexeme(Token) --> word(W), { idToken(W, Token) }.
lexeme(Const) --> constant(Const).
lexeme('::=') --> "::=".
lexeme('{...,') --> "{...,", !.
lexeme('{') --> "{".
lexeme('}') --> "}".
lexeme('(') --> "(".
lexeme(')') --> ")".
lexeme('[') --> "[".
lexeme(']') --> "]".
lexeme('..') --> "..", !.
lexeme('|') --> "|".
```

```
lexeme(';') --> ";".
lexeme('.') --> ".".
lexeme(',') --> ",".
lexeme('<') --> "<".
lexeme('-') --> "-".
```

In der ersten Klausel wird zuerst das im Eingabestrom ganz vorne stehende Wort aus Buchstaben, Ziffern und Bindestrichen bestimmt, das nach den Regeln für ASN.1–Bezeichner aufgebaut ist, das dann mit dem Prädikat `idToken` auf die Einteilung zwischen Bezeichnern und reservierten Worten untersucht wird. Die zweite Klausel untersucht Konstanten, während die übrigen Klauseln die Interpunktionszeichen behandeln. Auch unter diesen Klauseln findet man Anwendungen des Schnittes, um eine Erkennung von Teilworten zu unterbinden.

Das Prädikat `idToken` ist keine DCG–Regel, sondern eine reine Prolog–Regel. Es ist wie folgt definiert:

```
idToken(String, Token) :-
  name(X, String),
  token(X, Token).
```

Um das zu einer Zeichenkette `String` gehörende Lexem `Lexem` zu bestimmen, wird zuerst die (als Liste von Zeichencodes gespeicherte) Zeichenkette mit dem Prädikat `name` in ein Prolog–Atom umgewandelt. Anschließend wird mit dem Prädikat `token` das zu diesem Atom passende Lexem bestimmt.

Das Prädikat `token` untersucht als erstes, ob es sich bei dem übergebenen Atom um ein reserviertes Wort handelt. Ist dies der Fall, wird das zu diesem reservierten Wort gehörende *token* geliefert. Hier muß wieder ein Schnitt verwendet werden, damit beim *backtracking* nicht reservierte Worte fälschlich als Bezeichner erkannt werden. Wenn die Einordnung des übergebenen Atoms fehlschlägt, kann dieses Atom nur ein Bezeichner sein. Damit ergibt sich die folgende Definition für das Prädikat `token`:

```
token(X, Token) :-
  reserved(X, Token),
  !.
token(X, id(X)).
```

Für die reservierten Worte wird ein einfaches „Wörterbuch“ mit dem Prädikat `reserved` aufgebaut. Dieses assoziiert hier nur die reservierten Worte

jeweils mit sich selbst. Dies ist ein relativ einfacher Ansatz der Ermittlung eines Tokens für ein reserviertes Wort. Anstelle dieses einfachen Ansatzes kann aber auch ein etwas ausgearbeiteterer gewählt werden, in dem die reservierten Worte mit Schlüßelnummern assoziiert werden. Für das „Wörterbuch“ ist im Prolog–Programm eine Reihe von Fakten abgelegt, von denen hier nur einige wenige angegeben werden:

```
reserved('ABSENT', 'ABSENT').
reserved('ANY', 'ANY').
reserved('APPLICATION', 'APPLICATION').
reserved('BEGIN', 'BEGIN').
...
reserved('T61String', 'T61String').
reserved('VideotexString', 'VideotexString').
reserved('VisibleString', 'VisibleString').
reserved('WITH', 'WITH').
```

Für die Ermittlung der Lexeme für Konstanten dient das Prädikat `constant`. Dieses reflektiert die schon oben angeführte Aufteilung der Konstanten in natürliche, binäre und hexadezimale Zahlen und Zeichenketten:

```
constant(C) --> num(C).
constant(C) --> bstring(C).
constant(C) --> hstring(C).
constant(C) --> cstring(C).
```

Wir zeigen hier als Beispiel nur das Prädikat `num`, mit dem natürliche Zahlen erkannt werden sollen, die Definitionen für die anderen Prädikate folgen analog:

```
num(num(N)) --> numstring(Number), { name(N, Number) }.
num(num('0')) --> "0".
```

Die erste Klausel dieser Definition bestimmt eine Zeichenkette, die eine voon Null verschiedene Zahl ohne führende Nullen darstellt, mit dem Prädikat `numstring`. Anschließend wird diese Zeichenkette in ein Prolog–Atom umgewandelt. Die entstehenden Atome sind allerdings nicht immer Zahlen, wie sie von Prolog verarbeitet werden können — zum Beispiel, wenn die Zeichenketten über den verarbeitbaren Rahmen hinausgehende Zahlen darstellen.

Eine Zahl wird mit dem Prädikat `numstring` erkannt. Dieses Prädikat ist wie folgt definiert:

```
numstring([D | Ds]) --> digit(D), { D \= '0' }, digits(Ds).
```

Hier wird gefordert, daß die erste Ziffer im Eingabestrom von Null verschieden sein muß. Dies schließt die Null als solche mit aus, was mit einer zweiten Klausel abgefangen werden könnte, dann aber einen weiteren Schnitt erfordern würde. Aus diesem Grund ist die Verarbeitung der Konstanten `0` in das Prädikat `num` hineingezogen worden.

Das Prädikat `digits` ist rekursiv definiert und liest eine Liste von Ziffern ein:

```
digits([D | Ds]) --> digit(D), digits(Ds).
digits([]) --> [].
```

Die Grammatikregeln für die einzelnen Ziffern erkennen jeweils eine einzelne Ziffer und geben den jeweiligen Zeichencode zurück:

```
digit('0') --> "0".
digit('1') --> "1".
digit('2') --> "2".
digit('3') --> "3".
digit('4') --> "4".
digit('5') --> "5".
digit('6') --> "6".
digit('7') --> "7".
digit('8') --> "8".
digit('9') --> "9".
```

Die Produktionen für binäre und hexadezimale Konstanten sind analog aufgebaut, ebenso die Produktionen für Zeichenkettenkonstanten.

2.2.2 Eine Brücke zwischen der lexikalischen und syntaktischen Analyse

Das Ergebnis der lexikalischen Analyse ist eine Liste von Lexemen. Auf dieser Liste wird in der nächsten Phase die syntaktische Analyse durchgeführt. Dabei müßte allerdings die Produktion

```
AnyType ::= "ANY" "DEFINED" "BY" identifier
```

nach [Steedman 1993] in Prolog als DCG–Produktion codiert werden als:

```
anyType --> [ 'ANY' ], [ 'DEFINED' ], [ 'BY' ], type.
```

Dies ist jedoch für die Codierung von Grammatikregeln relativ umständlich und unnatürlich. Deshalb wurde hier ein anderer Weg gewählt. Für die einzelnen reservierten Worte und Interpunktionssymbole wurden mnemonische Namen gewählt (die, auch weil sie Prolog–Konstanten sein sollten, mit einem kleinen „t“ beginnen), für die eine weitere Reihe von Produktionen definiert wurde. Für die Interpunktionszeichen sind dies die folgenden Produktionen:

```
tLeftBrace --> [ '{' ].
tRightBrace --> [ '}' ].
tLeftBracket --> [ '[' ].
tRightBracket --> [ ']' ].
tLeftParent --> [ '(' ].
tRightParent --> [ ')' ].
tLessThan --> [ '<' ].
tLeftBraceDotDotDotComma --> [ '{...,' ].
tComma --> [ ',' ].
tDot --> [ '.' ].
tSemicolon --> [ ';' ].
tDotDot --> [ '..' ].
tBar --> [ '|' ].
tMinus --> [ '-' ].
tGets --> [ '::=' ].
```

Für die reservierten Worte erfolgt dies analog zum schon im Abschnitt zur lexikalischen Analyse angeführten „Wörterbuch“, auch hier geben wir wieder nur einige der Produktionen an:

```
tABSENT --> [ 'ABSENT' ].
tANY --> [ 'ANY' ].
tAPPLICATION --> [ 'APPLICATION' ].
tBEGIN --> [ 'BEGIN' ].
tBIT --> [ 'BIT' ].
...
```

```
tTRUE --> [ 'TRUE' ].
tT61String --> [ 'T61String' ].
tVideotexString --> [ 'VideotexString' ].
tVisibleString --> [ 'VisibleString' ].
tWITH --> [ 'WITH' ].
```

Eine wichtige Funktion der Brücke zwischen lexikalischer und syntaktischer Analyse ist auch die Identifikation der verschiedenen Bezeichnertypen von ASN.1, also der Typ-, Wert-, und Modulreferenzen sowie weiterer Bezeichner. Dies geschieht mit den folgenden Produktionen:

```
identifier(Id) --> [ id(Id) ], { startsWithLower(Id) }.
modulereference(Id) --> [ id(Id) ], { startsWithUpper(Id) }.
typereference(Id) --> [ id(Id) ], { startsWithUpper(Id) }.
valuereference(Id) --> [ id(Id) ], { startsWithLower(Id) }.
```

In diesen vier Produktionen wird jeweils zuerst ein Bezeichner aus dem Tokenstrom als Terminalsymbol erkannt, um danach anhand des ersten Zeichens die Eingruppierung zu entscheiden. Diese Eingruppierung ist nicht scharf, vielmehr ist sie abhängig von der Position des entdeckten Bezeichners. Ein groß geschriebener Bezeichner kann also an der einen Stelle eine Modulreferenz sein, während er an der anderen Stelle eine Typreferenz sein kann. Diese Einteilung muß in der semantischen Analyse geklärt werden.

Die Prädikate `startsWithLower` und `startsWithUpper` ermitteln mittels des Prädikats `name` zuerst die Zeichenkettendarstellung zu einem Atom `Id` und stellen dann fest, ob das erste Zeichen dieser Zeichenkette ein Klein- beziehungsweise Großbuchstabe ist. Ihre Definitionen lauten:

```
startsWithLower(Id) :-
  name(Id, [First | _]),
  First > 96, First < 123.

startsWithUpper(Id) :-
  name(Id, [First | _]),
  First > 64, First < 91.
```

Als letzte Komponenten der Brücke zwischen lexikalischer und syntaktischer Analyse verbleiben vier Produktionen, die konstante Werte in die einfachere Notation übersetzen. Sie sind wie folgt definiert:

```
bstring(B) --> [ bstring(B) ].
cstring(C) --> [ cstring(C) ].
hstring(H) --> [ hstring(H) ].
number(N) --> [ num(N) ].
```

2.2.3 Die syntaktische Analyse

Nach der lexikalischen Analyse folgt die syntaktische Analyse, die auf die eben beschriebenen Brückenprädikate zurückgreift und mit deren Hilfe einen Syntaxbaum für die Eingabedatei erzeugt. Da eine vollständige Besprechung der syntaktischen Analyse weit über den Rahmen dieses Berichts hinausgehen würde, beschränken wir uns hier darauf, die Übersetzung nur einiger Grammatikregeln aus [Steedman 1993] in die ihnen entsprechenden DCG–Regeln anzugeben. Dabei gehen wir auch auf einige Spezialfälle ein, die einer aufwendigeren Behandlung bedürfen.

Eine ASN.1–Spezifikation für ein Spezifikationsmodul hat in EBNF–Darstellung den folgenden syntaktischen Aufbau, wenn man Makros aus dem Sprachumfang wegläßt:

```
Module ::= ModuleIdentifier "DEFINITIONS" [TagStyleDefault]
           "::=" "BEGIN"
             [Exports]
             [Imports]
             { TypeAssignment
             | ValueAssignment
             }
           "END".
```

Um diese Syntaxregel in eine DCG–Regel umsetzen zu können, wie sie in Prolog verwendet werden kann, müssen zuerst optionale Terme der Form `[...]` und wiederholte Terme der Form `{...}` beseitigt werden, ebenso müssen Alternativen `...|...` behandelt werden.

Das Vorgehen bei Alternativen ist relativ einfach: Sie werden entweder so umgestellt, daß der Alternativoperator `|` als äußerster Operator auftaucht, oder in eine eigene Regel gezogen. Ebenso verfährt man mit optionalen Termen, für jeden optionalen Term in einer Syntaxregel erzeugt man zwei Regeln, von denen die eine den optionalen Term enthält, die andere nicht. An manchen Stellen ist es aber unter Umständen sinnvoller, ein optionales Auf-

treten eines Termes in die entsprechende Regel für den Term hineinzuziehen. Beispiele hierfür sind die Terme `Exports` und `Imports` in der Syntaxregel für Module.

Wiederholt auftretende Terme werden grundsätzlich in eigene Regeln gezogen. Geht man nach diesem Verfahren vor, erhält man die folgenden Syntaxregeln als Ersatz für die oben angeführte Regel für Module:

```
Module ::=   (ModuleIdentifier "DEFINITIONS" TagStyleDefault
              "::=" "BEGIN"
                Exports
                Imports
                Assignments
              "END")
           | (ModuleIdentifier "DEFINITIONS"
              "::=" "BEGIN"
                Exports
                Imports
                Assignments
              "END").

Assignments ::=   (TypeAssignment Assignments)
                | (ValueAssignment Assignments
                | .
```

Die sogewonnenen Regeln können nun direkt in DCG–Regeln umgewandelt werden. Dabei wird für jede Alternative einer Syntaxregel eine neue DCG–Regel erzeugt, Nichtterminalsymbole der Grammatik werden (Prolog–üblich in Kleinschrift) übernommen, und anstelle der Terminalsymbole werden die oben beschriebenen Brückenprädikate verwendet. Damit erhält man für die oben beschriebenen Regeln die folgenden DCG–Klauseln:

```
module
  --> moduleIdentifier, tDEFINITIONS, tagStyleDefault,
      tGets, tBEGIN,
        exports,
        imports,
        assignments,
      tEND.
module
```

```
  --> moduleIdentifier, tDEFINITIONS,
      tGets, tBEGIN,
        exports,
        imports,
        assignments,
      tEND.

assignments
  --> typeAssignment, assignments.
assignments
  --> valueAssignment, assignments.
assignments
  --> [].
```

Behandelt man die gesamte Grammatik für den betrachteten Sprachumfang von ASN.1 auf diese Art, so erhält man einen Parser für ASN.1, der Spezifikationen auf ihre syntaktische Korrektheit überprüft. Diese Überprüfung liefert aber nur ein binäres Ergebnis, also ob ein syntaktischer Fehler vorliegt oder nicht. Wie erhält man also einen Syntaxbaum für eine ASN.1–Spezifikation?

An dieser Stelle kommt die Parametrisierbarkeit von DCG–Regeln eine große Bedeutung zu: Jeder DCG–Regel wird ein Parameter zugeordnet, in dem der Syntaxbaum aufgebaut wird. Durch das in Prolog verwendete Unifikationsverfahren erfolgt der Aufbau des Syntaxbaums dann in der Regel *bottom up*, allerdings ist es ebenso möglich, den Weg der syntaktischen Analyse zu „lenken“, indem man bestimmte Teile des Syntaxbaumes als „Vorgabe“ in diesen Parameter schreibt. So wäre es zum Beispiel möglich, die Typen der Werte als Hinweis dafür zu nehmen, mit welchen Regeln die syntaktische Analyse weiterlaufen soll. In ASN.1 ist dieser Weg aber durch das nicht vorgegebene *definition before use* allerdings versperrt.

Für den von uns aufgebauten Parser geben wir den Syntaxbaum durch einen Prolog–Term an. Die Knoten werden dabei durch einen Prolog–Funktor beschrieben, der die den Unterbäumen des Knotens entsprechenden Prolog–Terme als Arguments erhält. Für das Beispiel oben wäre dies ein Term

```
module(Id, Exports, Imports, Assignments),
```

wobei die vier Argumente des Funktors für den Modulbezeichner, die Export- und Importliste und die Liste der Typ- und Wertvereinbarungen ste-

hen. Für das Beispiel sehen die notwendigen Änderungen an den DCG–Klauseln wie folgt aus:

```
module(module(Id, Exports, Imports, Assignments))
  --> moduleIdentifier(Id), tDEFINITIONS, tagStyleDefault,
      tGets, tBEGIN,
        exports(Exports),
        imports(Imports),
        assignments(Assignments),
      tEND.
module(module(Id, Exports, Imports, Assignments))
  --> moduleIdentifier(Id), tDEFINITIONS,
      tGets, tBEGIN,
        exports(Exports),
        imports(Imports),
        assignments(Assignments),
      tEND.

assignments([TypeAssignment | Assignments])
  --> typeAssignment(TypeAssignment),
      assignments(Assignments).
assignments([ValueAssignment | Assignments])
  --> valueAssignment(ValueAssignment),
      assignments(Assignments).
assignments([])
  --> [].
```

Betrachten wir nun, wie Typ- und Wertvereinbarungen verarbeitet werden. Hierzu verwenden wir als Beispiel die folgende ASN.1–Spezifikation:

```
Test DEFINITIONS ::= BEGIN
  a INTEGER ::= 5
END
```

Für dieses Beispiel wird zuerst die Regel für Module durchlaufen. Von dieser aus werden die Export- und Importlisten verarbeitet, diese sind beide nicht vorhanden. Da sie für das Beispiel und das weitere Verständnis keine größere Rolle spielen, geben wir die entsprechenden Regeln hier auch nicht an. Als nächstes wird die Liste der Zuweisungen verarbeitet, hier findet sich eine Wertzuweisung. Die Regel für Wertzuweisungen lautet:

```
valueAssignment(valueAssignment(Id, Type, Value))
  --> valuereference(Id), type(Type), tGets, value(Value).
```

Das Prädkat `valuereference` erkennt dabei einen entsprechend den Regeln für Wertbezeichner aufgebauten Bezeichner, und unifiziert diesen dem entsprechenden Argument. Für das oben stehende Beispiel ist dies der Prolog–Term `id(a)`.

Für die Erkennung von Typen wird zwischen „eingebauten“ Typen wie `BOOLEAN` oder `INTEGER` und vom Benutzer definierten Typen unterschieden. Dies geschieht mit den folgenden Regeln:

```
type(builtin(Type))
  --> tags, builtinType(Type), subtypeSpecs.
type(defined(Id))
  --> tags, definedType(Id), subtypeSpecs.
```

Für unser Beispiel sind die Prädikate `tags` und `subtypeSpecs` nicht relevant, wir betrachten sie deshalb hier nicht. Für die Unterscheidung zwischen „eingebauten“ und vom Benutzer definierten Typen werden die entsprechenden Prädikate versuchsweise ausgewertet, schlägt ein Versuch fehl, so wird per *backtracking* die nächste Alternative untersucht. Im Syntaxbaum werden „eingebaute“ und selbstdefinierte Typen durch die Funktoren `builtin` und `defined` gekennzeichnet, diese Kennzeichnung muß dann bei der Typüberprüfung beachtet werden.

In unserem Beispiel hat die versuchsweise Auswertung des Prädikats `builtinType` Erfolg. Dieses Prädikat ist durch DCG–Regeln wie folgt definiert, wir geben hier der Kürze halber nicht alle Klauseln an:

```
builtinType(Type)
  --> booleanType(Type).
builtinType(Type)
  --> enumeratedType(Type).
builtinType(Type)
  --> integerType(Type).
...
builtinType(Type)
  --> anyType(Type).
```

Das Prädikat `builtinType` gibt die Verarbeitung lediglich an die entsprechenden Prädikate weiter, die die Syntax der eingebauten Typen definieren,

und unifiziert sein Argument entsprechend. In unserem Beispiel wird die Bearbeitung mit dem Prädikat `integerType` fortgesetzt, das wie folgt definiert ist:

```
integerType(structuredType('INTEGER', NamedNumbers))
  --> tINTEGER,
      tLeftBrace,
      namedNumbers(NamedNumbers),
      tRightBrace.
integerType(simpleType('INTEGER'))
  --> tINTEGER.
```

Hier hat die Untersuchung der zweiten Klausel Erfolg, bei der ersten Klausel wird zwar das Terminalsymbol `INTEGER` erkannt, danach müßte aber eine geschweifte Klammer folgen, die nicht in der Eingabe steht. Hier greift das *backtracking*, und es wird die Auswertung der zweiten Klausel versucht, die nur das Terminalsymbol `INTEGER` erfordert.

Die eben beschriebene DCG–Regel zeigt auch die Unterscheidung der Typen in strukturierte Typen, die mit dem Funktor `structuredType` markiert werden, und einfachen Typen, die mit dem Funktor `simpleType` markiert werden. Diese Unterscheidung ist für die Überprufung der Typen wichtig, sie wird ebenfalls in der Ausgabe der Zwischendarstellung benötigt, die die Eingabe für den ASN.1–Viewer ist.

In unserem Falle wird damit die Variable `Type` im Aufruf `type(Type)` in der Regel für Wertzuweisungen mit dem Term `simpleType('INTEGER')` unifiziert. In der Eingabe sind dann noch die Symbole `::=`, `5` und `END` zu verarbeiten. Nach der Syntaxregel für Wertzuweisungen fährt die Bearbeitung nach der Erkennung des Symbols `::=` mittels des Brückenprädikats `tGets` mit dem Prädikat `value` fort, das wie folgt definiert ist:

```
value(Value)
  --> builtinValue(Value).
value(definedValue(Value))
  --> definedValue(Value).
```

Auch hier wird analog zu den Regeln für die Erkennung von Typen zwischen „eingebauten“ und vom Benutzer definierten Typen unterscheiden. Auch hier hat die Untersuchung des Prädikats für eingebaute Werte Erfolg, für das wir wieder nur einige Klauseln angeben:

```
builtinValue(Value)
  --> booleanValue(Value).
builtinValue(Value)
  --> enumeratedValue(Value).
builtinValue(Value)
  --> integerValue(Value).
...
builtinValue(Value)
  --> anyValue(Value).
```

Analog zur Erkennung der Typen gibt das Prädikat `builtinValue` die Bearbeitung an die Prädikate für die Syntaxregeln der einzelnen Wertsorten ab und unifiziert dabei die Variable `Value` mit dem entsprechenden Argument des aufgerufenen Prädikats. Hier hat der Aufruf des Prädikats `integerValue` Erfolg, das wie folgt definiert ist:

```
integerValue(simpleValue('INTEGER', Number))
  --> signedNumber(Number).
integerValue(simpleValue('INTEGER', Id))
  --> identifier(Id).
```

Auch für Werte wird zwischen einfachen Werten (wie hier dem ganzzahligen Wert) und strukturierten Werten (zum Beispiel Mengen oder Strukturen) unterschieden. Für das von uns betrachtete Beispiel führt die erste Klausel zum Erfolg, sie unifiziert ihren ersten Parameter und damit die Variable `Value` im Aufruf der Regel für Wertzuweisungen mit dem Term `simpleValue('INTEGER', 5)`.

Damit ist die Bearbeitung der Wertzuweisung abgeschlossen, für sie wurde der Teilbaum

```
valueAssignment(id(a),
                simpleType('INTEGER'),
                simpleValue('INTEGER', 5))
```

ermittelt. Mit der Erkennung des noch ausstehenden Symbols `END` in der Eingabe werden dann sowohl die Verarbeitung der Liste der Typ- und Wertzuweisungen, als auch die Verarbeitung des Spezifikationsmoduls beendet. Damit erhält man den folgenden Syntaxbaum:

```
module(id('Test'),
       [],
       [],
       [valueAssignment(id(a),
                        simpleType('INTEGER'),
                        simpleValue('INTEGER', 5))])
```

Abweichend von der in [Steedman 1993] angegebenen Syntax wurde hier noch eine weitere Syntaxregel hinzugefügt und implementiert:

```
Specification ::= { Module }.
```

Dies bedeutet, daß eine ASN.1–Spezifikation aus einer Folge von ASN.1–Modulen besteht, die in der Eingabe zu finden sind. Dies hat seinen Sinn darin, daß so keine internen Darstellungen für separat übersetzte Module gespeichert und verwaltet werde muß. Das bedeutet, daß eine ASN.1–Spezifikatio immer *alle* verwendeten Module enthalten muß.

2.2.4 Bindung und Überprüfung der Typen

Nach der syntaktischen Analyse müßte für ASN.1 die Bindung der vom Benutzer vereinbarten Typbezeichner an die durch sie bezeichneten Typen und die Überprüfung der Spezifikation auf korrekte Typisierung folgen. Aus Zeitgründen entfiel in der Entwicklung dieser Schritt.

2.2.5 Ausgabe der Zwischendarstellung

Nach der syntaktischen Analyse (und im Prinzip der Typprüfung) folgt die Ausgabe des im nächsten Abschnitt spezifizierten Zwischencodes. Für diesen Zwischencode wird der Syntaxbaum durchlaufen, und in einem *depth first*–Durchlauf als Folge von inneren Knoten und Blättern ausgegeben. Diese Ausgabe geschieht durch zwei Prädikate `node` und `leaf`, die als Argumente jeweils die Ebene der Einrückung des Knotens, den auszugebenden Bezeichner, sowie gegebenenfalls Typ und Wert für den Bezeichner übergeben bekommen.

Module werden dabei mit ihrem Bezeichner auf Einrücktiefe 0 ausgegeben, die Wertzuweisungen folgen dann auf Tiefe 1. Typzuweisungen werden nicht ausgegeben, da nur Datenwerte visualisiert werden sollten.

Für eine Wertzuweisung werden der Bezeichner, Typ und Wert auf Ebene 1 ausgegeben, wenn der Wert ein einfacher Wert ist, ansonsten werden entsprechend die Komponenten eines strukturierten Wertes eine Ebene tiefer komponentenweise ausgegeben.

2.3 Zusammenfassung

In diesem Abschnitt haben wir beschrieben, wie mit Hilfe der logischen Programmiersprache Prolog ein Übersetzer von der Spezifikationssprache ASN.1 in einen Zwischencode geschrieben werden kann. Dabei zeigte sich, daß insbesondere Prolog für die Erzeugung von Übersetzern, speziell auch für solche für ASN.1, gut geeignet ist, und der entstehende Code knapp und kompakt ist. Die spezielle Auswertungsstrategie von Prolog erlaubt eine Konzentration auf die Grammatik, auch mehrdeutige Grammatiken können ohne größere Änderungen in Prolog formuliert werden.

Kapitel 3

Ein Betrachter für ASN.1–Wertdefinitionen

"ASN.1 Viewer" ist ein einfaches Programm zur Visualisierung von ASN.1-Wertdefinitionen. Als Eingabe liest das Programm eine Sequenz der im Preorder-Verfahren durchlaufenen Knoten eines Baumes, welcher die Wertdefinitionen beschreibt. Die Ausgabe entspricht ungefähr dem von PC-Programmen her bekannten Verzeichnisbaum. Teilbäume können durch Anklicken ein- und ausgeklappt werden.

Dieser Abschnitt ist von Beat Koch geschrieben worden.

Nach kurzen Installations- und Bedienungshinweisen befasst sich die vorliegende Dokumentation vor allem mit der Implementation des ASN.1 Viewers.

3.1 Installation

Die Beschreibung in diesem Abschnitt bezieht sich auf die Installation des ASN.1 Viewers unter Solaris 2.5. Auf anderen Systemen dürfte die Installation jedoch ähnlich sein.

3.1.1 Voraussetzungen

ASN.1 Viewer wird mit Hilfe eines Java-Interpreters ausgeführt, der auf dem System installiert sein muss. Der Interpreter muss Zugriff auf die Klassen-

bibliotheken des Java API haben.

3.1.2 Installation des ASN.1 Viewers

1. Kopiere alle Klassendateien des ASN.1 Viewers in ein Verzeichnis, zum Beispiel /home/bkoch/asnviewer.

   Die Dateien heissen ASNApplet.class, ASNCanvas.class, ASNFrame.class, ASNViewer.class, AboutDialog.class, Field.class und Message.class.

2. Setze die Umgebungsvariable CLASSPATH so, dass sie auch auf das Installationsverzeichnis zeigt. *Achtung:* Beim Starten des Programms können Probleme auftreten, falls auf dem System verschiedene Java API Klassenbibliotheken installiert sind. Gegebenenfalls müssen die Pfade in CLASSPATH umgeordnet werden.

   Beispiel: `CLASSPATH=/opt/local/java/lib/classes.zip:/home/bkoch/asnviewer:.`

3.2 Bedienung

Der ASN.1 Viewer wird durch den Befehl `java ASNViewer` gestartet. Es erscheint ein leeres Fenster mit Menüzeile.

Durch Auswahl des Menüpunktes "File – Open..." wird ein Dateiauswahl-Dialog angezeigt. Eine Datei kann ausgewählt werden; deren Inhalt wird anschliessend im Fenster ausgegeben. Wichtig: Beim Einlesen der Datei wird keine Validierung vorgenommen. Der Benutzer ist dafür verantwortlich, dass die von ihm gewählte Datei Datensätze in korrektem Format (siehe Abschnitt 3.5) enthält.

Die Teilbäume unter fett angezeigten Einträgen können durch einfaches Anklicken ein- bzw. ausgeklappt werden. Normalerweise wird beim Ausklappen nur die nächste Hierarchiestufe angezeigt. Wird während des Klickens die Shift-Taste gedrückt, so wird der komplette Teilbaum ausgeklappt.

Mit dem Menüpunkt "File – Quit" wird das Programm beendet. Unter "Help – About" können Informationen zum Programm abgerufen werden.

3.3 Entwicklungsumgebung

Der ASN.1 Viewer ist in Java programmiert. Als Entwicklungsumgebung dient das Java Developers Kit, Version 1.0.2 von JavaSoft, welches auf einer SUN SPARCstation unter Solaris 2.5 läuft.

Der gleiche Source-Code ist ebenfalls auf einem PC mit 386er Prozessor unter OS/2 WARP kompiliert und ausgeführt worden.

3.4 Programmaufbau

ASN.1 Viewer ist ein Java-Applet, welches in einen Frame eingebunden ist, damit es als selbständige Applikation ausgeführt werden kann. Grundsätzlich sollte es möglich sein, das Applet allein in eine HTML-Seite einzubinden und in einem Webbrowser laufen zu lassen. Bis jetzt ist jedoch darauf verzichtet worden, denn die Webbrowser schränken aus Sicherheitsgründen den Zugriff auf das Dateisystem des Hostrechners ein.

In den folgenden Abschnitten werden die zur Applikation gehörenden Klassen einzeln beschrieben.

3.4.1 public class AboutDialog extends Dialog

Informationsdialog für den ASN.1 Viewer. Zeigt die Versionsnummer des Programms in einem eigenen, modalen Fenster.

Quelldatei: `AboutDialog.java`

Konstruktoren

- **AboutDialog**

  `public AboutDialog(Frame parent)`

  Erstellt den kompletten Dialog. Dieser kann anschliessend durch Aufruf von `show()` angezeigt werden. Damit der About-Dialog bezüglich des Hauptfensters modal erscheinen kann, muss als Parameter eine Referenz auf den Frame des Hauptfensters übergeben werden.

Methoden

- **action**

  `public boolean action(Event e, Object arg)`

  Behandelt das Klicken auf den OK-Knopf. Klickt der Benutzer auf OK, so wird der Dialog gelöscht und die Resourcen freigegeben.

- **gotFocus**

  `public boolean gotFocus(Event e, Object arg)`

  Wird aufgerufen, sobald der About-Dialog den Focus erhält. Leitet den Focus an den OK-Knopf weiter, damit dieser per Tastatur bedient werden kann.

3.4.2 public class ASNApplet extends Applet

ASN.1 Viewer Applet. Enthält die Ausgabefläche für die Daten (Canvas), die dazugehörenden Scrollbars sowie das Objekt zur internen Repräsentation der darzustellenden Datei (`msg`).

Behandelt die Scrollbar-Ereignisse sowie das Verändern der Fenstergrösse.

Quelldatei: `ASNApplet.java`

Konstanten

- **HBAR_LINE_INCREMENT**

  `private static final int HBAR_LINE_INCREMENT = 20`

  Anzahl Pixel, um welche horizontal gescrollt wird, wenn der Benutzer links oder rechts des Schiebers auf die Scrollbar klickt.

- **MAX_CANVASWIDTH**

  `private static final int MAX_CANVASWIDTH = 1000`

  Maximale scrollbare Breite des Canvas.

Konstruktoren

- **ASNApplet**

  ```
  public ASNApplet()
  ```

  Erzeugt Instanzen für die privaten Objekte. Plaziert die Scrollbars rechts und unterhalb des Canvas.

Methoden

- **handleEvent**

  ```
  public boolean handleEvent(Event e)
  ```

  Behandelt die Scrollbar-Meldungen, indem entsprechende Methoden des ASNCanvas Objekts aufgerufen werden.

- **openFile**

  ```
  public void openFile(String fileName)
  ```

  Lässt die Datei einlesen, erzwingt das Neuzeichnen des Canvas und setzt die vertikale Scrollbar neu.

- **reshape**

  ```
  public synchronized void reshape(int x, int y, int width,
  int height)
  ```

  Berechnet die Anzahl Linien, welche auf dem Canvas angezeigt werden können. Zeichnet den Canvas neu und passt schliesslich die Anzeigewerte der Scrollbars an.

- **setHBar**

  ```
  public void setHBar()
  ```

  Berechnet die Anzeigewerte für die horizontale Scrollbar. Diese hängen von der Breite des Canvas sowie der maximal scrollbaren Breite der Ausgabefläche ab.

- **setVBar**

  ```
  public void setVBar()
  ```

Berechnet die Anzeigewerte für die vertikale Scrollbar. Diese hängen von der Höhe des Canvas sowie von der Anzahl sichtbarer Datensätze ab. Eigentlich sollte die vertikale Scrollbar verschwinden, falls die Anzahl anzuzeigende Datensätze kleiner ist als die Anzahl Zeilen auf dem Canvas, und nur bei Bedarf wieder erscheinen. Dies hat jedoch nicht korrekt funktioniert, wobei das Problem tendenziell im Java API bzw. in der Unkenntnis des Programmierers über die Internas desselben liegt. Die vertikale Scrollbar bleibt nun immer sichtbar, löst aber keine Ereignisse aus, falls der Canvas gross genug ist, um alle Datensätze anzuzeigen.

3.4.3 public class ASNCanvas extends Canvas

Zeichenoberfläche für die Ausgabe der Meldungsdaten. Enthält eine eigene Datenstruktur (`fieldVector`), in der für jede Zeile eine Referenz auf den darin angezeigten Datensatz gespeichert wird. Die Zeilen sind nummeriert; die oberste Zeile hat die Nummer 0.

Quelldatei: `ASNCanvas.java`

Konstanten

- **BORDER**

  ```
  protected static final int BORDER = 5
  ```

  Breite des linken Randes in Pixel.

- **INDENT**

  ```
  protected static final int INDENT = 30
  ```

  Einrückungsdistanz pro Hierarchieebene in Pixel.

- **NAMEWIDTH**

  ```
  protected static final int NAMEWIDTH = 200
  ```

  Breite der Namensspalte in Pixel.

- **VALUEPOS**

  ```
  protected static final int VALUEPOS = 600
  ```

Position der Wertespalte in Pixel.

Konstruktoren

- **ASNCanvas**

  `public ASNCanvas(Message msg)`

  Erzeugt die für die Ausgabe der Daten benötigten Fonts. Die Masse des fetten Fonts werden als Referenzwerte für die Berechnung der Anzahl sichtbarer Zeilen verwendet.

Methoden

- **drawLine**

  `protected void drawLine(Graphics g, int line, boolean highlit)`

  Zeichnet eine Datenzeile. Zuerst wird der Text ausgegeben, entsprechend der Hierarchiestufe eingerückt. Knoten des Baumes werden fett gedruckt, temporäre Datensätze (siehe dazu Seiten 34 und 40) werden kursiv angezeigt. Blätter des Baumes werden ohne spezielle Textattribute ausgegeben.

  Schliesslich werden die Verbindungslinien für übergeordnete Hierarchiestufen gemalt. Die entsprechende Information ist beim Einlesen der Meldung in den einzelnen Datenzeilen gespeichert worden.

  Der Parameter `highlit` wird im Moment nicht verwendet.

- **getFirstVisibleIndex**

  `public int getFirstVisibleIndex()`

  Zählt, wieviele sichtbare Datensätze zwischen der Wurzel des Baumes und dem auf der obersten Zeile ausgegebenen Datensatz liegen.

- **getLastVisibleLine**

  `public int getLastVisibleLine()`

  Gibt die Nummer derjenigen Zeile zurück, auf der der letzte Datensatz steht.

- **getLineHeight**

  `public int getLineHeight()`

  Als Zeilenhöhe wird die Gesamthöhe des aktuellen Referenzfonts (fetter Font) zurückgegeben.

- **init**

  `public void init(int lines)`

  Initialisiert die Variable `fieldVector`. Die Liste `msg`, die den Meldungsbaum repräsentiert, wird von der Wurzel an nach sichtbaren Datensätzen abgesucht. Die Referenzen der sichtbaren Datensätze werden in `fieldVector` gespeichert. Sind weniger Datensätze als Zeilen sichtbar, so wird den überflüssigen Zeilen das konstante leere Feld Message.EMPTYFIELD zugewiesen. Sobald `fieldVector` bereit ist, wird die Ausgabemethode `paint()` aufgerufen.

  `init()` wird aufgerufen, nachdem eine neue Meldung eingelesen oder nachdem die Grösse des Ausgabefensters geändert worden ist.

- **mouseDown**

  `public boolean mouseDown(Event e, int x, int y)`

  Ereignisbehandlungsmethode für das Drücken der Maustaste, wenn der Mauszeiger über dem Canvas steht. Einfache Klicks auf eine nichtleere Zeile werden durch Aufruf der Methode `singleClick()` beantwortet.

- **paint**

  `public void paint(Graphics g)`

  Ruft für jede angezeigte Zeile `drawLine()` auf. Löscht schliesslich den Bereich vom unteren Rand der untersten Zeile bis zum Fensterrand.

- **scrollAbsolute**

  `public void scrollAbsolute(int pos)`

  Wird aufgerufen, sobald der Benutzer den Schieber der vertikalen Scrollbar auf eine neue Position setzt. Ausgehend vom ersten Datensatz werden die ersten (`pos` - 1) sichtbaren Datensätze übersprungen. Danach wird `fieldVector` aufgefüllt und der Canvas neu gezeichnet.

- **scrollDown**

  ```
  public void scrollDown()
  ```

  Wird aufgerufen, sobald der Benutzer die Anzeige um eine Zeile nach unten scrollen will. Sucht — von der bisher letzten angezeigten Zeile ausgehend — nach der nächsten sichtbaren Zeile und hängt diese ans Ende von `fieldVector`. Anschliessend wird der erste Eintrag in `fieldVector` gelöscht und der Canvas neu gezeichnet.

- **scrollPageDown**

  ```
  public void scrollPageDown()
  ```

  Wird aufgerufen, sobald der Benutzer die Anzeige um eine Seite nach unten scrollen will. Füllt `fieldVector` — von der bisher letzten angezeigten Zeile ausgehend — vollständig neu mit den nächsten sichtbaren Zeilen auf. Gegebenenfalls werden Datensätze am Anfang von `fieldVector` eingefügt, um alle Zeilen zu füllen. Anschliessend wird der Canvas neu gezeichnet.

- **scrollPageUp**

  ```
  public void scrollPageUp()
  ```

  Wird aufgerufen, sobald der Benutzer die Anzeige um eine Seite nach oben scrollen will. Füllt `fieldVector` — von der obersten bisher angezeigten Zeile ausgehend — vollständig neu mit den vorhergehenden sichtbaren Zeilen auf. Gegebenenfalls werden Datensätze ans Ende von `fieldVector` angehängt, um alle Zeilen zu füllen. Anschliessend wird der Canvas neu gezeichnet.

- **scrollUp**

  ```
  public void scrollUp()
  ```

  Wird aufgerufen, sobald der Benutzer die Anzeige um eine Zeile nach oben scrollen will. Sucht — von der bisher ersten angezeigten Zeile ausgehend — nach der ersten vorhergehenden sichtbaren Zeile und fügt diese am Anfang von `fieldVector` ein. Anschliessend wird der letzte Eintrag in `fieldVector` gelöscht und der Canvas neu gezeichnet.

- **setXOffset**

  ```
  public void setXOffset(int xOffset)
  ```

Wird aufgerufen, sobald der Benutzer die horizontale Scrollbar betätigt. Die Variable `xOffset` bestimmt, welcher horizontale Abschnitt im Canvas gezeichnet werden soll. Nach dem Setzen der Variablen wird der Canvas neu gezeichnet.

- **singleClick**

  `private void singleClick(int line, boolean shift)`

  Behandelt einfaches Klicken auf eine Datenzeile. Falls der auf der Zeile `line` stehende Datensatz ein Knoten des Meldungsbaumes ist, so wird durch Aufruf von Message.toggleExpansion() die Sichtbarkeit des darunterliegenden Teilbaumes geändert. `shift` gibt dabei an, ob der Benutzer während dem Klicken die Shift-Taste gedrückt hat und somit den gesamten Unterbaum ausgeklappt haben will.

  Nachdem der Unterbaum ein- bzw. ausgeklappt worden ist, werden die sichtbaren Zeilen unterhalb der angeklickten Zeile neu bestimmt und ausgegeben. paint() wird nicht aufgerufen, um unnötiges Flackern zu vermeiden.

3.4.4 class ASNFrame extends Frame

Fensterrahmen für das ASN.1 Viewer Applet. Der Rahmen wird gebraucht, damit das Applet als selbständige Applikation ausgeführt werden kann.

Quelldatei: `ASNViewer.java`

Konstruktoren

- **ASNFrame**

  `public ASNFrame(String title, int width, int height)`

  Erzeugt eine Instanz der Klasse ASNApplet. Erzeugt die Menüleiste durch Aufruf von `setupMenuBar()`. Initialisiert und startet das Applet, sobald das Fenster in der durch `width` und `height` vorgegebenen Grösse angezeigt wird.

Methoden

- **action**

  `public boolean action(Event e, Object arg)`

  Behandelt Klick-Ereignisse auf Menüpunkte. Klickt der Benutzer auf "File – Quit", so wird das Programm ohne Rückfrage beendet. Bei "File – Open..." wird die Methode `openDialog()` aufgerufen. Klicken auf "Help – About..." schliesslich bewirkt die Erzeugung und Anzeige eines AboutDialog Objekts.

- **openDialog**

  `private void openDialog()`

  Zeigt einen System-Filedialog an und übergibt – sofern der Benutzer eine Datei ausgewählt hat — den kompletten Dateipfad an die Methode `ASNApplet.openFile()`.

- **setupMenuBar**

  `private void setupMenuBar()`

  Baut die Menüleiste zusammen und hängt sie an den Frame.

3.4.5 public class ASNViewer

Definiert die Methode main(); diejenige Funktion, welche durch den Java-Interpreter beim Starten der Applikation aufgerufen wird.

Quelldatei: `ASNViewer.java`

Methoden

- **main**

  `public static void main(String argv[])`

  Erzeugt eine Instanz der Klasse ASNFrame, wodurch ein Fenster angezeigt und das ASN.1 Viewer Applet gestartet wird.

3.4.6 class Field

Grundlegende Datenstruktur zur Beschreibung eines Datensatzes. Objekte der Klasse Field werden in einem Objekt der Klasse Message zu einer doppelt verketteten Liste zusammengefasst.

Quelldatei: `Message.java`

Konstanten

- **CORNER**

  ```
  public static final int CORNER = 3
  ```

  Linientyp: Senkrechte Linie von oben bis in die Mitte der Zeile, dann Linie gegen rechts.

- **CROSS**

  ```
  public static final int CROSS = 2
  ```

  Linientyp: Senkrechte Linie mit einer in der Mitte der Zeile gegen rechts abzweigende Linie.

- **LINE**

  ```
  public static final int LINE = 1
  ```

  Linientyp: Senkrechte Linie durch die ganze Zeile.

- **NOLINE**

  ```
  public static final int NOLINE = 0
  ```

  Linientyp: Keine Linie

Konstruktoren

- **Field**

  ```
  public Field(int level, boolean node, String name, String
  type, String value, boolean visible, boolean expanded,
  boolean temp)
  ```

Erstellt und initialisiert ein neues Field-Objekt. `level` gibt die Hierarchiestufe an, auf der der Datensatz steht. Der Wurzeldatensatz hat Level 0, tiefere Ebenen haben grössere Level. `node` bestimmt, ob der Datensatz ein Knoten des Baumes (`node` = true) oder ein Blatt ist. `name`, `type` und `value` enthalten die Strings für die drei Spalten, welche schliesslich angezeigt werden. `visible` bestimmt, ob der Datensatz sichtbar ist; `expanded` gibt an, ob ein Knoten ein- oder ausgeklappt ist. `temp` wird auf true gesetzt für Datensätze, welche intern durch das Programm erzeugt werden. Für eingelesene Datensätze ist `temp` immer false.

Methoden

- **attachArray**

  ```
  public void attachArray(int maxLevel)
  ```

  Weist dem Feld ein Array mit `maxLevel` Elementen zu, in welchem Informationen über die Hierarchielinien abgelegt werden, welche vor den eigentlichen Texteinträgen ausgegeben werden.

- **getLevel**

  ```
  public int getLevel()
  ```

  Gibt die Hierarchiestufe des Feldes zurück.

- **getLineStyle**

  ```
  public int getLineStyle(int level)
  ```

  Gibt den Linientyp (siehe oben) für die Hierarchieebene `level` zurück.

- **getName**

  ```
  public String getName()
  ```

  Gibt den String zurück, welcher in der Namensspalte angezeigt werden soll.

- **getNext**

  ```
  public Field getNext()
  ```

  Gibt eine Referenz auf das nächste Field-Objekt in der Liste zurück oder `null`, falls kein weiteres Field-Objekt existiert.

- **getPrev**

  `public Field getPrev()`

  Gibt eine Referenz auf das vorhergehende Field-Objekt in der Liste zurück oder `null`, falls kein vorhergehendes Field-Objekt existiert.

- **getType**

  `public String getType()`

  Gibt den String zurück, welcher in der Typenspalte angezeigt werden soll.

- **getValue**

  `public String getValue()`

  Gibt den String zurück, welcher in der Wertespalte angezeigt werden soll.

- **isExpanded**

  `public boolean isExpanded()`

  Gibt true zurück, falls der Teilbaum unter dem aktuellen Feld ausgeklappt ist, andernfalls wird false zurückgegeben.

- **isNode**

  `public boolean isNode()`

  Gibt true zurück, falls das Feld ein Knoten ist. False wird zurückgegeben, falls das Feld ein Blatt ist.

- **isTemp**

  `public boolean isTemp()`

  Gibt true zurück, falls das Feld intern durch das Programm erzeugt worden ist. Gibt false zurück, falls das Feld aus einer Datei eingelesen worden ist.

- **isVisible**

  `public boolean isVisible()`

  Gibt true zurück, falls das Feld als sichtbar markiert worden ist, andernfalls false.

- **setExpanded**

  `public void setExpanded(boolean e)`

  Setzt die Variable, welche bestimmt, ob der Teilbaum unter dem aktuellen Feld ausgeklappt (`e` = true) oder eingeklappt ist.

- **setLineStyle**

  `public void setLineStyle(int level, int style)`

  Setzt den Linientyp (siehe Konstanten oben) für die Hierarchieebene `level`.

- **setNext**

  `public void setNext(Field next)`

  Weist dem Feld eine Referenz auf das nächste Field-Objekt zu.

- **setPrev**

  `public void setPrev(Field prev)`

  Weist dem Feld eine Referenz auf das vorhergehende Field-Objekt zu.

- **setValue**

  `public void setValue(String value)`

  Weist dem Feld den String zu, welcher in der Wertespalte angezeigt werden soll.

- **setVisible**

  `public void setVisible(boolean v)`

  Markiert das Feld als sichtbar (`v` = true) oder unsichtbar.

3.4.7 public class Message

Datenstruktur zur Darstellung des Baumes der ASN.1-Wertedefinition. Implementiert eine doppelt verkettete Liste von Objekten der Klasse Field. Ein Element dieser Liste ist sichtbar, falls die Methode `Field.isVisible()` true zurückgibt.

Quelldatei: `Message.java`

Konstanten

- **EMPTYFIELD**

  ```
  public static final Field EMPTYFIELD = new Field(-1,
  false, , , , false, false, true)
  ```

  Leeres Feld; Dummy-Datensatz.

Konstruktoren

- **Message**

  ```
  public Message()
  ```

  Erzeugt ein neues, leeres Message-Objekt.

Methoden

- **add**

  ```
  public void add(Field newField)
  ```

  Hängt das Objekt `newField` an das Ende der Liste.

- **countVisible**

  ```
  public int countVisible()
  ```

  Gibt die Anzahl sichtbarer Elemente in der Liste zurück.

- **getFirst**

  ```
  Field getFirst()
  ```

  Gibt das erste Element der Liste zurück. Das erste Element ist immer die Wurzel des Baumes. Falls die Liste leer ist, wird `null` zurückgegeben.

- **getFirstOnLevel**

  ```
  private Field getFirstOnLevel(int level)
  ```

Gibt das erste Element der Liste mit Ebene `level` zurück. Existiert kein solches Element, so wird null zurückgegeben.

- **getNextOnLevel**

  ```
  private Field getNextOnLevel(Field field, int level,
  boolean checkHierarchy)
  ```

  Ausgehend vom Element `field`, suche das nächste Element mit Ebene `level`. Existiert kein solches Element, so wird `null` zurückgegeben.

  Falls `checkHierarchy` true ist, wird die Suche abgebrochen, sobald ein Element auf höherer Ebene gefunden wird. In diesem Fall wird `Message.EMPTYFIELD` zurückgegeben.

- **getNextVisible**

  ```
  public Field getNextVisible(Field field)
  ```

  Ausgehend vom Element `field` wird das nächste sichtbare Element gesucht. Existiert kein solches Element, so wird `null` zurückgegeben.

- **getPrevVisible**

  ```
  public Field getPrevVisible(Field field)
  ```

  Ausgehend vom Element `field` wird das erste vorhergehende sichtbare Element gesucht. Existiert kein solches Element, so wird `null` zurückgegeben.

- **toggleExpansion**

  ```
  public void toggleExpansion(Field field, boolean
  expandAll)
  ```

  Abhängig vom Zustand von `field` (ein- oder ausgeklappt) und dem Wert von `expandAll` wird das Sichtbarkeits-Flag der auf `field` folgenden Felder mit höherem Level geändert.

  Bemerkung: Der häufig auf einen Knoten folgende temporäre Datensatz (siehe Methode `readFromFile()`) hat den gleichen Level wie `field`. Damit die Schleife nicht schon beim ersten auf `field` folgenden Listen-Element abbricht, wird mit der Methode `Field.isTemp()` das Temporär-Flag überprüft.

- **readFromFile**

`public void readFromFile(String fileName)`

Liest die durch `fileName` identifizierte Datei ein. Die Datei muss Datensätze im in Abschnitt 3.5 beschriebenen Format enthalten. Beim Einlesen wird die Gültigkeit der Datensätze nicht überprüft! Lediglich die NumberFormatException wird beim Berechnen des Levels abgefangen, da die zur Konvertierung eines Strings in einen Integer verwendete Methode `Integer.parseInt()` recht heikel ist.

Alle Datensätze, bei welchen keine NumberFormatException aufgetreten ist, werden als neues Objekt der Klasse Field an die Liste angehängt. Falls das Wertefeld des neuen Objekts einen der drei Strings "CHOICE", "SET" oder "SEQUENCE" enthält, so wird das Wertefeld gelöscht und dafür ein zusätzliches Objekt an die Liste gehängt. Dieses "temporäre" Objekt enthält als Variablennamen einen der drei oben genannten Strings; es unterscheidet sich von "normalen" Objekten dadurch, dass ein Aufruf `Field.isTemp()` true zurückgibt. Diese Unterscheidung wird in der Methode `Message.toggleExpansion()` gebraucht.

Während dem Einlesen wird der grösste vorkommende Wert von Level in `maxLevel` gesichert. Sobald alle Datensätze eingelesen sind, wird jedem Element in der Liste ein Array der Grösse `maxLevel` zugewiesen. Für jede unterhalb der Wurzel liegende Hierarchiestufe werden schliesslich durch Aufruf von `setLineStyles()` die Hierarchielinien bestimmt.

- **removeAll**

  `public void removeAll()`

  Entfernt alle Elemente aus der Liste.

- **setLineStyles**

  `private void setLineStyles(int level)`

  Bestimmt die Linienattribute für alle Listenelemente auf der durch `level` vorgegebenen Hierarchieebene. Eine Beschreibung der vier möglichen Linientypen befindet sich unter "Konstanten" in Abschnitt 3.4.6.

  Algorithmus:

  1. prev = Erstes Element in der Liste mit Ebene `level`

2. next = Nächstes Element mit Ebene `level`
3. Setze präventiv den Linientyp für das erste Element auf CORNER.
4. Solange ein nächstes Element existiert (next ist nicht `null`):
5. Falls next EMPTYFIELD referenziert, ist die Suche nach dem nächsten Element durch ein Feld x mit höherer Hierarchie unterbrochen worden. Suche in diesem Fall das erste und zweite Element auf Ebene `level` hinter x. Setze den Linientyp für das erste Element auf CORNER. Gehe zurück zu Schritt 4.
6. next referenziert ein Feld aus der Liste: Setze den Linientyp von prev auf CROSS, denjenigen von next auf CORNER und den Linientyp aller zwischen prev und next liegenden Felder auf LINE. Setze danach prev = next und suche wieder das nächste Element mit Ebene `level`.
7. Gehe zurück zu Schritt 4.

- **visibleIndexOf**

  `public int visibleIndexOf(Field field)`

  Zählt die Anzahl sichtbarer Felder x zwischen dem ersten Feld der Liste und `field`.

3.5 Datensatzformat

Ein Datensatz besteht aus den folgenden fünf Feldern:

Bezeichnung	Typ	Bemerkung
Ebene	Integer	0: Wurzel des Baumes >0: Untergeordnete Ebene
Knoten	Boolean	true: Datensatz ist Knoten false: Datensatz ist Blatt
Variablenname	String	
Datentyp	String	
Wert	String	

Die Anordnung der Datensätze muss einem Preorder-Durchlauf des Baumes entsprechen.

Werden die Datensätze aus einer Datei eingelesen, so muss in dieser Datei jedes Feld auf einer eigenen Zeile stehen. Felder ohne Daten werden durch eine leere Zeile dargestellt.

3.6 Bekannte Probleme

Die folgenden Probleme sind beim Ausführen des Programms unter Solaris 2.5 aufgetreten.

- Erhält der "About"-Dialog nach seinem Aufruf zum ersten Mal den Fokus, wird der OK-Knopf nicht fokussiert. Wechselt der Fokus auf ein anderes Fenster und danach zurück zum "About"-Dialog, wird der OK-Knopf doch noch fokussiert.
- Klickt man auf den OK-Knopf im "About"-Dialog, so wird das gesamte Programm manchmal mit einer "segmentation violation" abgebrochen.

Literaturverzeichnis

[Sample 1993]
Michael Sample
Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler
Department of Computer Science, University of British Columbia, July 1993.

[Steedman 1993]
Douglas Steedman
Abstract Syntax Notation One (ASN.1): The Tutorial and Reference
Technology Appraisals Ltd., 1993.