Informatik I WS 02/03

Erste Übungsstunde am 29. Oktober 2002

Einführung

Über mich

Mein Name ist Matthias Sala und meine Email `salam@student.ethz.ch`. Ich wohne in Zürich und Basel und studiere im fünften Semester Informatik.

Über meine Übungsstunden

Die Teilname an meinen Übungsstunden ist freiwillig. Wenn Ihr einige der Übungen abgebt, dann erhaltet Ihr auch das Testat.

Informationen, Material und diese Übungsunterlage findet Ihr unter meiner Internetseite `www.n.ethz.ch/student/salam`.

Wenn Ihr Fragen habt, fragt einfach in der Übungstunde oder schreibt mir eine Email.

Enhanced Backus Naur Form – EBNF

Was ist EBNF?

In der Informatik benötigt man oft eine syntaktische Beschreibung, zum Beispiel für eine Programmiersprache wie Oberon. Die erweiterte Backus Naur Form ist eine syntaktische Beschreibungssprache, welche eine standardisierte Grammatik hat.

Wie sieht eine EBNF aus?

Syntaxregeln

Die EBNF setzt sich aus mindestens einer oder auch mehreren Syntaxregeln zusammen. Eine Syntaxregel besteht aus einem Namen (links), einem Gleichheitszeichen und einer Reihe von Symbolen. Eine Syntaxregel wird immer mit einem Punkt abgeschlossen:

```
Buch = Titelseite Inhaltsverzeichnis Text.
```

Oft spricht man auch lediglich von Regel statt Syntaxregel.

Symbole und Terminalsymbole

Symbole sind "Verweise" auf Syntaxregeln, deren Name gleich lautet wie das Symbol. Beispielsweise das Symbol `Titelseite` im vorherigen Beispiel verweist auf eine Regel mit Namen `Titelseite`.

Terminalsymbole sind abschliessende Symbole, die nicht mehr auf weitere Regeln verweisen. Dank ihnen kommt man mit der EBNF auch irgendwann mal zu einem Ende. Terminalsymbole werden immer in Anführungs- und Schlusszeichen geschrieben, zum Beispiel:

```
SatzA = "Irgendein Satz kann hier stehen".
```

Konkatenation

Symbole können einfach aneinandergereiht, indem sie mit einem Abstand getrennt werden.

```
A = B C.
```

aber nicht: `A = BC.`

Alternative

Wenn man mehrere Symbole zur Auswahl stellen möchte, dann kann man dies mit dem senkrechten Strich Zeichen tun. Bei einer solchen oder-Auswahl kann immer nur ein und genau ein Symbol ausgewählt werden. Zum Beispiel kann ein Buchstabe folgendermassen definiert werden:

```
Vokal = "A" | "E" | "I" | "O" | "U".
```

Option

Wenn ein Symbol nur fakultativ ist, dann schreibt man das Symbol in einer eckigen Klammer. Zum Beispiel:

```
Paar = "Hans liebt Susanne" [" sehr!"].
```

Repetition

Soll ein Symbol null-, ein-, oder unendlich viel mal vorkommen, dann schreibt man das Symbol in einer geschweiften Klammer. Zum Beispiel:

```
Namenliste = { Vorname ";" Nachname ZEILENWECHSEL }.
```

Gruppierung

Wenn Symbole gruppiert werden sollen, so dass klar wird, welcher Operator (Alternative, Option etc.) welche Symbole mit einbezieht, dann verwendet man die normalen runden Klammern:

```
A = B | C | D.
```

ist unterschiedlich zu

```
A = ( B | C ) | D.
```

Rekursion

Oft möchte man Dinge verschachteln. Zum Beispiel in der Mathematik können Additionen und Multiplikationen verschachtelt werden, wobei sie in ihrer Form immer noch gültig bleiben:

```
Rechnung = Term [PlusOperator Term].

Term = Faktor [MalOperator Faktor].

Faktor = (Zahl {Zahl}) | "(" Rechnung ")".

Zahl = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
"8" | "9".

PlusOperator = "+".

MalOperator = "*".
```

Das heisst also, dass beim fetten "Rechnung" erneut die obere Regel Rechnung angewendet wird, was zur Folge hat, dass eine verschachtelte Rechnung generiert werden kann, zum Beispiel:

```
1   +   5   *   (   4   +   3   )
```

wobei `4 + 3` erneut auf die Syntaxregel `Rechnung` verweist.

Ein Beispiel

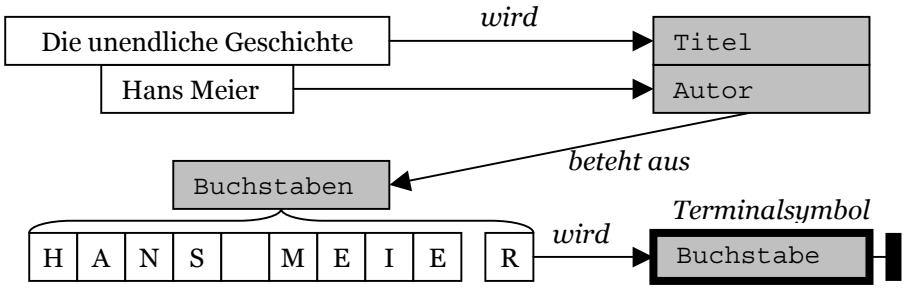
Nehmen wir das Beispiel vom Anfang; wir wollen eine neue Sprache spezifizieren, mit welcher wir Bücher verfassen können. Bücher bestehen aus Titelseiten, Impressumseite, Inhaltsverzeichnissen, Vorworten, Kapiteln, Stichwortverzeichnissen und Quellenangaben. Einige dieser Teile sind zwingend, zum Beispiel der Umschlag besteht immer aus Titelseite und Klappentext. Andere Teile sind optional, wie zum Beispiel das Stichwortverzeichnis, das oft fehlt. Wieder andere können in verschiedener Anzahl vorkommen, so gibt es keine Regel, wieviele Kapitel ein Buch haben sollte.

Die EBNF würde dann folgendermassen aussehen:

```
Buch = Titelseite Inneres Klappentext.

Titelseite = Ueberschrift Autor Bild.

Ueberschrift = Buchstaben.

Autor = Buchstaben.

Bild = Bildpunkte.

Inneres = [ Impressumseite ] Inhaltsverzeichnis
{ Vorwort } Text [ Stichwortverzeichnis | Index ]
[ Quellenverzeichnis ].

Klappentext = Buchstaben.

Impressumseite = Buchstaben.

Buchstaben = Buchstabe { Buchstabe }.

Buchstabe = "A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
"Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" |
"Z".
```

usw.

Wie man oben sieht, müssen alle Teile mit Regeln genau spezifiziert werden. Am Schluss müssen alle Symbole mit Terminalsymbolen beschrieben sein. Das heisst, wenn ich genug lange mein Buch in diese Teile zerlege, gelange ich irgendwann zu jedem einzelnen Buchstaben, den ich dann mit der untersten Regel mit Terminalsymbolen beschreiben kann, zum Beispiel:



Mehr Material

ISO Definition der EBNF (19 Seiten)

`http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`

Anweisungen in Oberon

Was ist Oberon?

Oberon ist eine prozedurale Programmiersprache, welche auf verschiedenen Plattformen verfügbar ist. In der Vorlesung werden wir Oberon.NET verwenden. Da sich Oberon.NET aber noch in der Entwicklung befindet, müsst Ihr nicht unbedingt dieses benutzen. Beim VIS könnt Ihr eine CD beziehen, welche noch das herkömmliche Native Oberon enthält.

Die Syntax der Anweisungen

If Anweisung

Die If Anweisung wird verwendet, um anhand einer Bedingung, welche erfüllt wird oder nicht, unterschiedlich zu reagieren:

```
IF hunger = "ja" THEN
   essen
ELSE
   fasten
END;
```

oder wenn man noch genauer unterscheiden möchte:

```
IF hunger = "ja" THEN
   essen
ELSIF hunger = "sehr"
   sofort_essen
ELSE
   fasten
END;
```

Case Anweisung

Die Case Anweisung ist vergleichbar mit dem If, aber beim Case können mehrere Varianten angegeben werden:

```
CASE hunger OF
     "sehr": sofort_essen
   | "normal": gelegentlich_essen
   ELSE fasten
END;
```

Der Vorteil von Case liegt in der schnelleren Ausführung als If mit einer Unzahl Elsif. Zudem wird der Quelltext übersichtlicher, wenn man Case benutzt. Als Faustregel benutzt man ein Case erst wenn mehr als fünf Unterscheidungen nötig sind. Vorher reicht eigentlich das If.

While Anweisung

Wenn man einen Teil des Quelltextes mehrmals ausführen möchte, dann verwendet man zum Beispiel eine While Anweisung. Bei einer While Anweisung wird solange der Code zwischen DO und END ausgeführt, bis die Bedingung nicht mehr erfüllt ist. Ein While kann auch nie ausgeführt werden, wenn die Bedingung schon das erste mal nicht zutrifft. Ein Beispiel:

```
WHILE hunger = "ja" DO
   ess_vorgang
END;
```

Repeat Anweisung

Ein Repeat wird einmal ausgeführt und dann solange wiederholt, bis die Bedingung erfüllt wird.

```
REPEAT
   fasten
UNTIL hunger = "ja";
```

Wichtig ist, dass das Repeat in jedem Fall mindestens einmal ausgeführt wird. Deshalb ist es beispielsweise nicht möglich statt `warte` den `ess_vorgang` in die Repeat Schlaufe zu setzen, weil es möglich wäre, dass man sich dann überisst, weil man gar keinen Hunger hat. Also fehleranfällig und deshalb gefährlich ist:

```
REPEAT                          (* FEHLERANFÄLLIG *)
   ess_vorgang
UNTIL hunger = "nein";
```

Loop Anweisung

Eine Loop Anweisung wird verwendet, wenn man eine ewige Schlaufe möchte, welche sich im wiederholten Code selbst mittels dem Kommando `EXIT` abbricht:

```
LOOP
   IF hunger = "ja" THEN essen END;
   IF hunger = "statt für immer" THEN EXIT END
END;
```

Dieser Loop wird so lange wiederholt, bis der Hunger für immer weg ist. Es ist also möglich, dass der Loop nie aufhört. Es ist zudem schwierig zum Beweisen, dass ein konkretes Stück Code mit einer Loop Anweisung darin auch wirklich immer beendet. Um Fehlerfreiheit und annähernde Haltegarantie zu gewährleisten, verwendet man die schwächsten und stärksten Vor- und Nachbedingungen.

Mehr Material

Offizielle Oberonseite

`http://www.oberon.ethz.ch`

Kurze Definition der Sprache Oberon

`http://www.oberon.ethz.ch/oreport.html`

Hoare Triple

weakest pre-/strongest postcondition, invariant

schwächste Vor-/stärkste Nachbedingung, Invariante

Was ist das?

Jede Codezeile, welche Variablen verwendet, ist anfällig auf Eingabefehler. Zum Beispiel ist es nicht erlaubt eine Division durch Null durchzuführen. Vorbedingung sind Bedingungen, die eingehalten werden müssen, damit der folgende Code fehlerfrei ausgeführt werden kann. Die Nachbedingung ist die Bedingung, welche nach dem Ausführen eines Codes erfüllt sein muss. Stimmt sie nicht mit der Wirklichkeit überein, so hat sich irgendein Fehler ereignet.

Das Hoare Triple sieht wie folgt aus:

$\{P\}\ S\ \{Q\}$

P ist die Vorbedingung, S der Algorithmusabschnitt und Q die Nachbedingung. Als Satz bedeutet es,

> *wenn P wahr ist bevor S ausgeführt wird, dann ist, wenn die Ausführung von S terminiert, Q danach wahr.*

Zum obigen Beispiel der Division:

Die schwächste Vorbedingung für die Division $A \div B$ ist immer $B \neq 0$, weil nie durch 0 dividiert werden darf.

Eine Analogie zum Hoare Triple ist eine Zugfahrt von Zürich nach Basel. Ich muss am Zürcher Hauptbahnhof mit einem gültigen Billett in den Zug einsteigen, um mit dem Zug von Zürich nach Basel fahren zu können. Das weiss ich im Voraus, und dies ist meine Vorbedingung. Wenn ich dann mit dem Zug in Basel ankomme, weiss ich, dass ich in Basel sein muss (das ist die Nachbedingung Q). Es geht gar nicht anders, es sei denn der Zug hätte einen Unfall gehabt, was bedeuten würde, dass der Vorgang S abgebrochen wurde.

Die Invariante

Eine Invariante ist das, was sich durch den Algorithmus nicht ändert bzw. immer gleich bleibt. Im folgenden Beispiel ist die Invariante (x = y), weil am Anfang und am Ende die Bedingung erfüllt ist:

(x = y)	{ x (x + 1 }	(x = y + 1)
(x = y + 1)	{ y (y + 1 }	(x = y)
(x = y)	{ x (x + 1;	
	y (y + 1;}	(x = y)

Die Invariante muss aber nicht zwischen den einzelnen Zeilen des Algorithmus zutreffen, sondern nur über den Ganzen gesehen.

Leider gibt es kein Verfahren mit dem man die Invariante immer finden kann. Meist ist es sogar extrem schwierig eine Invariante aus einem Algorithmus zu extrahieren, so dass es einem formellen Beweis gleichkommt.

Die Invariante in der Analogie mit der Zugfahrt von Zürich nach Basel wäre zum Beispiel, dass ich das Billett auf mir trage, weil ich mein Billet von der Abfahrt bis zur Ankunft auf mir tragen muss wegen dem Billettkontrolleur.

Wie lösen...

...wenn die postcondition gegeben ist?

Wir nehmen ein Beispiel:

{?} x := y + 5; y := y + 1 {x ≤ 7; y = Y}

Uns ist also folgender Algorithmus gegeben:

- Addiere 5 zu y und speichere in x
- Inkrementiere y um 1

Und unsere Endbedingung ist:

- x soll kleiner gleich 7 sein
- y soll gleich der Konstante Y sein

Wir gehen also folgendermassen vor:

- setze die Endbedingung von hinten, also rückwärts in den Algorithmus ein
- die Gleichung am Schluss ist dann die schwächste Vorbedingung☐ XE "schwächste Vorbedingung" ☐

Also konkret:

Y = y + 1 (y + 1 = Y

7 ≥ y + 5 (y ≥ 2

Jetzt haben wir zwei Gleichungen, welche als schwächste Vorbedingung gelten müssen:

{y + 1 = Y; y ≥ 2}

Also immer wenn y grösser gleich 2 ist und y um 1 kleiner als Y ist, der Algorithmus mit dieser Eingabe ausgeführt wird, wird die Endbedingung erfüllt.

...wenn die precondition gegeben ist?

Wir nehmen folgendes Beispiel:

{x ≥ 5} x := 2 * x {?}

Wir gehen folgendermassen vor:

- setze die Vorbedingung von vorne her, also vorwärts in den Algorithmus ein
- die Gleichung am Schluss ist dann die **stärkste** Nachbedingung

Also konkret:

x ≥ 2 * 5 → x ≥ 10

Demnach ist die stärkste Nachbedingung x ≥ 10. Immer wenn die Vorbedingung erfolgreich getestet und der Algorithmus durchgeführt wurde, muss x beim Austritt aus dem Algorithmus grösser gleich 10 sein.

Mehr Material

Kurze Erklärung zum Hoare Triple in englisch

http://c2.com/cgi/wiki?HoareTriple

Wirklich gute Ausführliche Erklärung des Themas

http://www.csi.uottawa.ca/ordal/papers/peter/node15.html

Theorem Prover für Hoare Triple in Java

http://www.site.uottawa.ca/~vsawma/vixtp/

Hoare Proof of an Array Reversing Program

http://www.csee.wvu.edu/~callahan/cs336/arrayreverse.html

Slide Show zum Thema

http://cs.gmu.edu/~bjamison/slides/06.Verification.pdf