2 Grundlagen

In diesem Kapitel behandeln wir einige Grundlagen, die für alle höheren Programmiersprachen relevant sind.

2.1 Beschreibung der Syntax

Zur Beschreibung einer Programmiersprache gehören zumindest die Syntax und die Semantik, denn ohne diese Beschreibungen kann man sie nicht zuverlässig verwenden (oder muss eine bestimmte Sprachimplementierung ausprobieren, was keine zuverlässige Methode ist). Zur exakten Beschreibung der Syntax gibt es etablierte Methoden, wie wir nachfolgend kurz darstellen.

Die üblichen Formalismen zur Beschreibung der Syntax von Programmiersprachen sind *formale Grammatiken*:

- Reguläre Grammatiken oder reguläre Ausdrücke werden für die Grundsymbole wie z.B. Zahlen, Bezeichner, Schlüsselwörter, u.ä. verwendet.
- Kontextfreie Grammatiken werden zur Beschreibung komplexer syntaktischen Strukturen (Klammerstrukturen, Blöcke) vewendet.
- Kontextabhängigkeiten werden häufig in der statischen Semantik festgelegt (informell oder mittels Inferenzsystemen).

Zur textuellen Notation von Grammatiken wird überwiegend die erweiterte Backus-Naur-Form (EBNF) benutzt. Wir definieren zunächst die Backus-Naur-Form (BNF), die erstmals für die Beschreibung von Algol-60 verwendet wurde.

Definition 2.1 (BNF (Backus-Naur-Form))**.** *Die* Backus-Naur-Form (BNF) *ist eine textuelle Notation für kontextfreie Grammatiken, die folgende Metasymbole verwendet:*

- *„::=“ definiert Regeln für Nichtterminalsymbole*
- *„|“ trennt Regelalternativen, d.h. verschiedene rechte Seiten für ein Nichtterminalsymbol*
- *„<...>“ beschreibt Nichtterminalsymbole*

Die durch umschließende Spitzklammern gekennzeichneten Nichtterminalsymbole sind beliebige alphanumerische Bezeichner. Alles andere sind Terminalsymbole (die manchmal auch durch Fettdruck oder Unterstreichen gekennzeichnet werden).

Beispiel 2.1 (Grammatik für Ausdrücke)**.**

```
<Exp>    ::= <Number> | <Id> | <Exp> <Op> <Exp> | ( <Exp> )
<Op>     ::= + | - | * | /
<Id>     ::= <Letter> | <Id> <Letter> | <Id> <Digit>
<Number> ::= <Digit> | <Number> <Digit>
<Letter> ::= a | b | c | ... | z
<Digit>  ::= 0 | 1 | 2 | ... | 9
```

Da für eine Sprachbeschreibung mit der BNF manchmal viele Nichtterminalsymbole notwendig sind, wurde mit der EBNF weitere Konstrukte eingeführt, die sich aber durch Verwendung neuer Nichtterminalsymbole in reine BNF übersetzen lassen.

Definition 2.2 (EBNF (Erweiterte Backus-Naur-Form))**.** *Die* erweiterte Backus-Naur-Form (EBNF) *unterscheidet sich von der BNF wie folgt:*

- *Nichtterminalzeichen beginnen mit Großbuchstaben*
- *Terminalzeichen werden mit einfachen Anführungszeichen quotiert (z. B.* `'+'`*) oder ohne weitere Angaben direkt notiert, wenn eine Verwechslung mit anderen Symbolen ausgeschlossen ist.*
- *Als Metasymbole werden benutzt:*
  - *„*`::=`*“: wie in BNF*
  - *„*`|`*“: wie in BNF*
  - *„*`(...)`*“: Gruppierung von Grammatikelementen*
  - *„*`{...}`*“: beliebig viele Vorkommen, auch 0 Mal*
  - *„*`[...]`*“: optionales Vorkommen, d.h. 0 oder 1 Mal*

Mittels EBNF können wir die Syntax für Ausdrücke kompakter definieren:

Beispiel 2.2 (EBNF-Grammatik für Ausdrücke)**.**

```
Exp    ::= Number | Id | Exp Op Exp | '(' Exp ')'
Op     ::= + | - | * | /
Id     ::= Letter { Letter | Digit }
Number ::= Digit { Digit }
```

Ein Problem von Grammatiken ist jedoch die mögliche Mehrdeutigkeit der Strukturbäume bzw. Ableitungen. Wir betrachten hierzu zwei mögliche (vereinfachte) Ableitungsbäume für den Ausdruck „`2 * 3 + v`“:

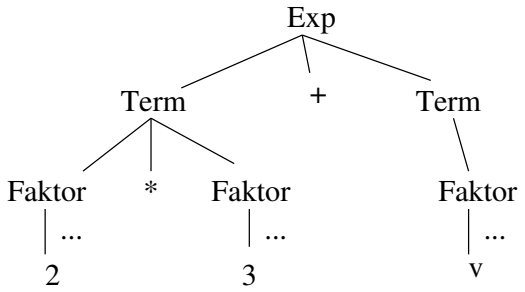
Semantisch ist der Unterschied zwischen beiden Bäumen relevant, denn diese würden bei der Auswertung zu unterschiedlichen Ergebnissen führen. Daher ist eine Auflösung der Mehrdeutigkeiten notwendig. Dieses kann durch verschiedene Methoden erreicht werden:

- Eindeutige Grammatiken ($LL(k)$-, $LR(k)$-Grammatiken, vgl. Compilerbau): Verändere die Grammatiken so, dass Ableitungsbäume immer eindeutig sind.

  Beispiel 2.3 (Eindeutige Grammatik)**.** Unsere Grammatik für Ausdrücke kann wie folgt in eine eindeutige Grammatik umgewandelt werden:

  ```
  Exp    ::= Term   { ( + | - ) Term }
  Term   ::= Faktor { ( * | / ) Faktor }
  Faktor ::= '(' Exp ')' | Number | Id
  ```

  Nun ergibt sich der folgende (vereinfachte) Ableitungsbaum für „`2 * 3 + v`“:



- Einführung von Prioritätsregeln („Punktrechnung vor Strichrechnung“) durch folgende Methoden:
  - spezielle „Operatorpräzedenzgrammatiken“
  - Transformation von Grammatiken, um Prioritäten zu codieren (s. o.)
  - in statischer Semantik auflösen

Eine weitere Darstellung für Grammatiken sind Syntaxdiagramme, mit denen Regeln für Nichtterminale graphisch dargestellt werden.

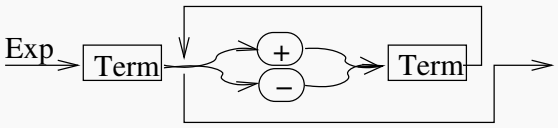
Definition 2.3 (Syntaxdiagramm)**.** *Ein* ***Syntaxdiagramm*** *ist eine graphische Darstellung kontextfreier Grammatiken. Es besteht aus folgenden Elementen:*

- *ein gerichteter Graph für jedes Nichtterminal*
- *eckige Box: Nichtterminalsymbol*
- *runde Box: Terminalsymbol*

Eine Ableitung für ein Nichtterminalsymbol ergibt sich durch einen Graphdurchlauf von links nach rechts.

Syntaxdiagramme sind für Anwender mit weniger formaler Ausbildung oft leichter zu verstehen.

Beispiel 2.4 (Syntaxdiagramm für das Nichtterminal `Exp`)**.**



2.2 Methoden zur Semantikbeschreibung

Bei der Semantikbeschreibung geht es darum, die Bedeutung der syntaktischen Konstrukte zu beschreiben. Bei informellen oder umgangssprachlichen Beschreibungen besteht oftmals das Problem der Eindeutigkeit, d.h. es ist eventuell nicht klar, welches Verhalten durch ein Sprachkonstrukt ausgelöst werden soll. Dies kann dazu führen, dass unterschiedliche Sprachimplementierungen für ein Programm unterschiedliche Ergebnisse liefern.
Um dieses Problem zu vermeiden, ist eine *formale Beschreibung* der Semantik wünschenswert. Hierzu gibt es z.B. folgende Methoden:

Axiomatische Semantik: Hier werden für jedes Sprachkonstrukt Vor-/Nachbedingungen angegeben, die immer gelten sollen. Dies ist auch unter dem Begriff *Hoare-Logik* bekannt (vgl. dazu die Vorlesung „Programmierung“).

Denotationelle Semantik: Hierbei wird jedem syntaktischen Konstrukt der Programmiersprache eine Bedeutung zugeordnet (“to denote”), d.h. eine denotationelle Semantik definiert eine Abbildung von synaktischen Konstrukten in passende semantische Objekte.

Operationale Semantik: Hierbei wird zunächst ein präzises Modell für Berechnungen und Berechnungsschritte definiert und darauf basierend wird eine formale Beschreibung von Berechnungsabläufen festgelegt.

Diese und auch andere Methoden werden zur Beschreibung von Programmiersprachen verwendet. Die axiomatische Semantik ist die abstrakteste, da hier nicht gesamte Berechnungen beschrieben werden, sondern nur die lokalen Effekte von Sprachkonstrukten definiert werden. Auf der anderen Seite findet sich die Idee der axiomatischen Semantik in der Praxis wieder, wenn z.B. Programmiersprachen die Formulierung und das Testen von Invarianten oder Vor- und Nachbedingungen unterstützen.
Die denotationelle Semantik baut auf komplexeren mathematischen Strukturen auf (z.B. Verbände, Fixpunkte), kann aber zur Implementierung einer Sprache verwendet werden, insbesondere wenn man funktionale Programmiersprachen verwendet. Dies wird später gezeigt.
Die operationale Semantik ist die konkreteste Beschreibung, weil hier direkt einzelne Berechnungsschritte beschrieben werden. Insbesondere kann man damit z.B. auch beschreiben, dass man eine Endlosschleife hat, wogegen bei den vorigen Arten der Semantikbeschreibung eine Endlosschleife etwas Undefiniertes ist. Die operationale Semantik

kann auch eine Grundlage zur Implementierung einer Sprache bilden, z.B. indem man einen Interpreter auf der Basis der operationale Semantik realisiert.
Im folgenden werden wir unterschiedliche Arten der Semantikbeschreibung verwenden, um Konstrukte aus existierenden Programmiersprache möglichst präzise zu beschreiben.

2.2.1 Strukturierte Operationale Semantik (SOS, (Plotkin, 1981))

Wir betrachten als erstes Beispiel die Beschreibung der Semantik einer einfachen imperativen Sprache, deren Syntax wie folgt definiert ist (das Nichtterminalsymbol `Stm` steht für „Statement“):

```
Stm ::= skip
      | Stm ; Stm
      | Var := Exp
      | if Exp then Stm else Stm
      | while Exp do Stm
```

Somit ist ein Programm dieser einfachen Sprache eine Folge von Anweisungen, wobei eine Anweisung entweder nichts tut („`skip`“) oder eine Zuweisung, eine Fallunterscheidung oder eine `while`-Schleife ist.

Imperative Programme manipulieren Werte von Variablen. Um dies zu beschreiben, benötigen wir einen **Zustand**, der beschreibt, welche Variablen welche Werte haben. Einen solchen Zustand können wir einfach als Abbildung

$$\sigma : Var \to Int \cup Bool \cup \ldots$$

modellieren, welche jeder Variablen einen Wert zuweist.
Ein **Berechnungsschritt** arbeitet die Anweisungen schrittweise ab und manipuliert dabei den Zustand. Daher können wir einen Berechnungsschritt formal als **Zustandsübergang**

$$\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$$

beschreiben (wobei S und S' aus `Stm` ableitbar sind).
Eine **Berechnung** beginnt in einem **Startzustand** (S ist das abzuarbeitende Programm und σ_{inital} ist ein Initialzustand, der z.B. für jede Variable einen Initialwert zuweist)

$$\langle S, \sigma_{initial} \rangle$$

und endet in einem **Endzustand**

$$\langle \texttt{skip}, \sigma \rangle$$

Wir beschreiben die Zustandsübergangsrelation „$\to$“ über den Aufbau von S (aus diesem Grund nennt man diese Form auch *strukturierte* operationale Semantik). Hierbei nehmen wir an, dass wir eine **Interpretationsfunktion** I zum Ausrechnen von Ausdrücken in einem Zustand haben, d.h. der *Wert eines Ausdrucks e in einem gegebenen Zustand* σ wird mit

$$I[\![e]\!]\,\sigma$$

bezeichnet. Die genaue Definition von I erfolgt später.

Transitionsaxiome und -regeln zur Interpretation von Anweisungen

- Axiom zur Abarbeitung einer *Zuweisung*:

$$\langle v := e, \sigma \rangle \to \langle \texttt{skip}, \sigma[v/I[\![e]\!]\,\sigma] \rangle$$

  Hierbei wird durch die Notation $\sigma[v/w]$ die *Abänderung* einer Funktion σ an der Stelle v bezeichnet, d.h. dies ist eine neue Funktion, die wie folgt definiert ist:

$$\sigma[v/w](x) = \begin{cases} w & \text{falls } x = v \\ \sigma(x) & \text{sonst} \end{cases}$$

  Dieses Axiom beschreibt also, dass die rechte Seite e ausgerechnet wird und der sich ergebende Wert $(I[\![e]\!]\,\sigma)$ als neuer Wert für v in der Umgebung gespeichert wird.

- Axiome zur Abarbeitung einer *bedingten Anweisung*:

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \to \langle S_1, \sigma \rangle \qquad \text{falls } I[\![b]\!]\,\sigma = true$$
$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \to \langle S_2, \sigma \rangle \qquad \text{falls } I[\![b]\!]\,\sigma = false$$

- Axiome zur Abarbeitung einer *Schleife*:

$$\langle \text{while } b \text{ do } S, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle \qquad \text{falls } I[\![b]\!]\,\sigma = false$$
$$\langle \text{while } b \text{ do } S, \sigma \rangle \to \langle S; \text{while } b \text{ do } S, \sigma \rangle \qquad \text{falls } I[\![b]\!]\,\sigma = true$$

- Axiom und Regel zur Abarbeitung einer *Sequenz von Anweisungen*:

$$\langle \texttt{skip}; S, \sigma \rangle \to \langle S, \sigma \rangle$$

$$\frac{\langle S_1, \sigma \rangle \to \langle S_2, \sigma' \rangle}{\langle S_1; S, \sigma \rangle \to \langle S_2; S, \sigma' \rangle}$$

  Folglich hat die Anweisung `skip` keine Wirkung und bei einer Sequenz muss (als Voraussetzung, s.u.) erst die linke Anweisung abgearbeitet werden.

Damit können wir eine **Berechnung** bezüglich eines Programms S_0 und einem Anfangszustand σ_0 als Folge

$$\langle S_0, \sigma_0 \rangle \to \langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle \to \dots$$

definieren, wobei jeder einzelne Schritt $\langle S_i, \sigma_i \rangle \to \langle S_{i+1}, \sigma_{i+1} \rangle$ aus den obigen Axiomen und Regeln folgt. Wir sprechen von einer **terminierenden Berechnung**, falls ein i existiert mit $S_i = \texttt{skip}$, ansonsten ist die Berechnung **nichtterminierend** (umgangssprachlich auch als „Endlosschleife“ bezeichnet).

2.2.2 Exkurs: Inferenzsysteme

Die obige operationale Semantik wurde durch Axiome und Regeln, d.h. ein *Inferenzsystem* definiert. Da Inferenzsysteme ein zentrales Mittel zur Beschreibung der statischen und dynamischen Semantik von Programmiersprachen sind, ist es wichtig, deren Bedeutung genau zu verstehen. Aus diesem Grund wiederholen wir an dieser Stelle aus der Logik die Begriffe der Inferenzsysteme und Ableitbarkeit.

Definition 2.4 (Inferenzsystem)**.** *Ein **Inferenzsystem** (**Deduktionssystem**, **Kalkül**) ist ein Paar $IR = (BL, R)$ mit*

- *BL (base language): Sprache, d.h. Menge von zulässigen Sätzen über einem bestimmten Alphabet*
- *$R = \{R_i\}_{i \in I}$ (I Indexmenge) ist eine Familie von Mengen $R_i \subseteq BL^{n_i}$ mit $n_i > 0$ (d.h. n_i-fache kartesische Produkte), wobei wir die beiden folgenden grundlegenden Klassen unterscheiden:*
  - *$n_i = 1$: Dann heißt R_i **Axiomenschema***
  - *$n_i > 1$: Dann heißt R_i **Regel- oder Inferenzschema**. Statt $(S_1, \ldots, S_{n_i}) \in R_i$ schreiben wir auch:*

    $$\frac{S_1 \ldots S_{n_i - 1}}{S_{n_i}}$$

    Dies wird auch gelesen als „aus $S_1 \ldots S_{n_i - 1}$ folgt S_{n_i}“.

Definition 2.5 (Ableitbare Sätze)**.** *Die Menge aller* Folgerungen *oder* beweisbaren/ ableitbaren Sätze *bzgl. eines Inferenzsystems IR ist die kleinste Menge F mit:*

- *Ist R_i ein Axiomenschema und $S \in R_i$, dann ist $S \in F$.*
- *Ist R_i Inferenzschema und $(S_1, \ldots, S_{n_i}) \in R_i$ und $S_1, \ldots, S_{n_i - 1} \in F$, dann ist auch $S_{n_i} \in F$ („falls die Voraussetzungen beweisbar sind, dann ist auch die Folgerung beweisbar“).*

Übliche Einschränkungen für Inferenzsysteme sind:

1. Die Indexmenge I ist endlich (d.h. es gibt nur endliche viele Schemata).
2. Jedes R_i ist entscheidbar.

Unter diesen Voraussetzungen gilt:

- Die Beweise sind effektiv überprüfbar (d.h. es ist entscheidbar, ob ein Beweis für eine Folgerung richtig ist).
- Die Menge aller Folgerungen ist aufzählbar.

Es ist allerdings in der Regel nicht entscheidbar, ob ein Satz aus dem Inferenzsystem ableitbar ist.

Beispiel 2.5 (SOS als Inferenzsystem)**.** Wir schauen uns noch einmal die oben angegebene Definition für die operationale Semantik imperativer Programme an. Diese Definition wird als Inferenzsystem wie folgt interpretiert:

- BL enthält Sätze der Form

  $$\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle$$

  wobei S_1, S_2 aus dem Nichtterminal `Stm` ableitbar ist und σ_1, σ_2 textuelle Darstellungen von Zuständen sind.

- Axiomenschema: Betrachte z. B. das Schema

  $$\langle \texttt{skip}; S, \sigma \rangle \to \langle S, \sigma \rangle$$

  Hierbei sind S und σ *Schemavariablen*, d.h. sie stehen für die Menge der Anweisungen und Zustände, wobei identische Variablen für identische Anweisungen bzw. Zustände stehen.

- Ein Regelschema ist z. B. bei der Sequenz zu finden:

  $$\frac{\langle S_1, \sigma \rangle \to \langle S_2, \sigma' \rangle}{\langle S_1; S, \sigma \rangle \to \langle S_2; S, \sigma' \rangle}$$

  Hier sind $S_1, S_2, S, \sigma, \sigma'$ Schemavariablen.

Somit ist unsere Schreibweise von Axiomen und Regeln nur eine endliche, kompakte Darstellung für eine unendliche Menge von Axiomen und Regeln. Daher werden diese auch *Schemata* genannt.

Behauptung: Wir wollen zeigen, dass

$$\begin{aligned}\langle \texttt{x:=1; y:=2}, \{\} \rangle &\to \langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle \\ &\to \langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\} \rangle\end{aligned}$$

eine korrekte Berechnung ist.

Beweis: Es gilt: $\langle \texttt{x:=1}, \{\} \rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1\} \rangle$ ist ein Axiom, also ableitbar. Die Anwendung der Regel mit diesem ableitbaren Satz als Voraussetzung ergibt:

$$\langle \texttt{x:=1; y:=2}, \{\} \rangle \to \langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle$$

ist ableitbar. Damit ist der erste Schritt korrekt. Der zweite Schritt

$$\langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\} \rangle$$

ist ein Axiom und damit auch korrekt. Der dritte Schritt

$$\langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\}\rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\}\rangle$$

ist ebenfalls ein Axiom und damit auch korrekt.