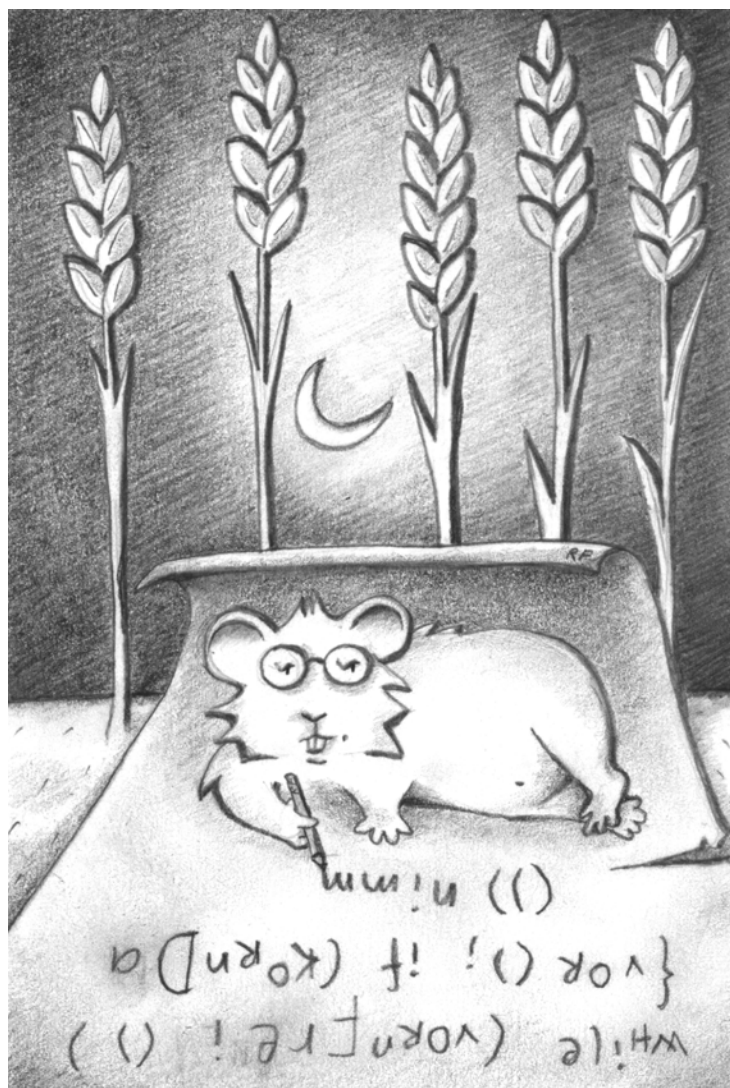
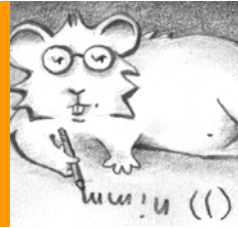




Kapitel 2

Programmiersprachen



Eine *Programmiersprache* ist eine zum Formulieren von Programmen geschaffene künstliche Sprache. In diesem Kapitel werden zunächst verschiedene Klassifikationen vorgestellt. Anschließend wird kurz erläutert, welche Aspekte bei der Definition einer Programmiersprache von Bedeutung sind. Detaillierter wird auf die Syntax von Programmiersprachen eingegangen.

2.1 Klassifikation von Programmiersprachen

Vielleicht fragen Sie sich jetzt: Wieso gibt es eigentlich nicht nur eine einzige Programmiersprache, mit der alle Programmierer arbeiten? Da Programmiersprachen anders als natürliche Sprachen, die sich über Jahrhunderte hinweg entwickelt haben, ja künstlich definiert werden müssen, hätte man sich doch von Anfang an auf eine einheitliche Programmiersprache festlegen können.

Zunächst kann man feststellen, dass es bestimmte Programmiersprachen gibt, die primär für Programmieranfänger definiert worden sind. Sie sind meistens sehr einfach gehalten, d.h. der Sprachumfang ist relativ gering. Sie sind leicht zu erlernen, eignen sich aber nicht besonders zum Lösen sehr komplexer Probleme. Hierfür werden sehr viel mächtigere Programmiersprachen benötigt.

Eine alternative Klassifizierung unterscheidet sogenannte niedere *Maschinensprachen* (*maschinen-nahe Programmiersprachen*) und höhere *problemorientierte Programmiersprachen*. Maschinensprachen ermöglichen die Erstellung sehr effizienter Programme. Sie sind jedoch abhängig vom speziellen Computertyp. Dahingegen orientieren sich die höheren Programmiersprachen nicht so sehr an den von den Computern direkt ausführbaren Befehlen, sondern eher an den zu lösenden Problemen. Sie sind für Menschen verständlicher und einfacher zu handhaben.

Ein weiterer Grund für die Existenz der vielen verschiedenen Programmiersprachen liegt in der Tatsache, dass die zu lösenden Probleme nicht alle gleichartig sind. So werden häufig neue Programmiersprachen definiert, die speziell für bestimmte Klassen von Problemen geeignet sind.

Den höheren Programmiersprachen liegen bestimmte Konzepte zugrunde, mit denen die Lösung von Problemen formuliert wird. Im Wesentlichen lassen sich hier fünf Kategorien – auch *Programmierparadigmen* genannt – unterscheiden:

- Imperative Programmiersprachen: Programme bestehen aus Folgen von Befehlen (PASCAL, MODULA-2).
- Funktionale Programmiersprachen: Programme werden als mathematische Funktionen betrachtet (LISP, MIRANDA).
- Prädikative Programmiersprachen: Programme bestehen aus Fakten (gültige Tatsachen) und Regeln, die beschreiben, wie aus gegebenen Fakten neue Fakten hergeleitet werden können (PROLOG).

- Regelbasierte Programmiersprachen: Programme bestehen aus „wenn-dann-Regeln“; **wenn** eine angegebene Bedingung gültig ist, **dann** wird eine angegebene Aktion ausgeführt (OPS5).
- Objektorientierte Programmiersprachen: Programme bestehen aus Objekten, die bestimmte (Teil-)Probleme lösen und zum Lösen eines Gesamtproblems mit anderen Objekten über Nachrichten kommunizieren können (SMALLTALK).

Nicht alle Programmiersprachen können eindeutig einer dieser Klassen zugeordnet werden. So ist bspw. LOGO eine funktionale Programmiersprache, die aber auch imperative Sprachkonzepte besitzt. Java und C++ können als imperative objektorientierte Programmiersprachen klassifiziert werden, denn Java- und C++-Programme bestehen aus kommunizierenden Objekten, die intern mittels imperativer Sprachkonzepte realisiert werden.

Programmierersprachen einer Kategorie unterscheiden sich häufig nur in syntaktischen Feinheiten. Die grundlegenden Konzepte sind ähnlich. Von daher ist es im Allgemeinen nicht besonders schwierig, eine weitere Programmiersprache zu erlernen, wenn man bereits eine Programmiersprache derselben Kategorie beherrscht. Anders verhält es sich jedoch beim Erlernen von Programmiersprachen anderer Kategorien, weil hier die zugrunde liegenden Konzepte stark voneinander abweichen.

2.2 Definition von Programmiersprachen

Programmierersprachen sind sehr exakte, künstliche Sprachen zur Formulierung von Programmen. Sie dürfen keine Mehrdeutigkeiten bei der Programmerstellung zulassen, damit der Computer das Programm auch korrekt ausführen kann. Bei der Definition einer Programmiersprache muss ihre *Lexikalik*, *Syntax*, *Semantik* und *Pragmatik* definiert werden:

- Lexikalik: Die Lexikalik einer Programmiersprache definiert die gültigen Zeichen bzw. Wörter, aus denen Programme der Programmiersprache zusammengesetzt sein dürfen.
- Syntax: Die Syntax einer Programmiersprache definiert den korrekten Aufbau der Sätze aus gültigen Zeichen bzw. Wörtern, d.h. sie legt fest, in welcher Reihenfolge lexikalisch korrekte Zeichen bzw. Wörter im Programm auftreten dürfen.
- Semantik: Die Semantik einer Programmiersprache definiert die Bedeutung syntaktisch korrekter Sätze, d.h. sie beschreibt, was passiert, wenn bspw. bestimmte Anweisungen ausgeführt werden.
- Pragmatik: Die Pragmatik einer Programmiersprache definiert ihren Einsatzbereich, d.h. sie gibt an, für welche Arten von Problemen die Programmiersprache besonders gut geeignet ist.

Die Lexikalik wird häufig in die Syntax mit einbezogen. Wie die Syntax einer Programmiersprache definiert werden kann, wird im nächsten Abschnitt detailliert erläutert. Die Semantik einer Programmiersprache wird in der Regel nur umgangssprachlich beschrieben, es existieren jedoch auch Möglichkeiten für eine formal saubere (mathematische) Definition. Für die Definition der Pragmatik einer Programmiersprache existiert kein bestimmter Formalismus. Sie wird deshalb umgangssprachlich angegeben.

2.3 Syntaxdarstellungen

Die Syntax einer Programmiersprache legt fest, welche Zeichenreihen bzw. Folgen von Wörtern korrekt formulierte („syntaktisch korrekte“) Programme der Sprache darstellen und welche nicht. Zur Überprüfung der syntaktischen Korrektheit eines Programms muss deshalb zuvor die Syntax der Programmiersprache formal beschrieben werden. Hierzu existieren verschiedene Möglichkeiten. In den folgenden zwei Unterabschnitten werden die zwei gängigsten Notationen vorgestellt, nämlich die *Syntaxdiagramme* und die *Backus-Naur-Form*.

Sowohl Syntaxdiagramme als auch die Backus-Naur-Form sind Techniken zur Darstellung sogenannter *kontextfreier Programmiersprachen*. Die meisten Programmiersprachen sind jedoch *kontextsensitiv*, d.h. es lassen sich nicht alle Regeln zur Beschreibung der Syntax mit den beiden Techniken beschreiben. Nicht formulierbare Eigenschaften der Syntax werden daher umgangssprachlich ergänzt.

2.3.1 Syntaxdiagramme

Bei den Syntaxdiagrammen handelt es sich um eine graphische und daher sehr übersichtliche Notation zur Definition der Syntax einer Programmiersprache. Syntaxdiagramme sind folgendermaßen definiert:

- Zur Beschreibung der Syntax einer Sprache existiert in der Regel eine Menge von Syntaxdiagrammen, die zusammen die Syntax definieren. In der Menge existiert genau ein *übergeordnetes Syntaxdiagramm*, bei dem die Definition beginnt.
- Jedes Syntaxdiagramm besitzt einen Namen (Bezeichnung).
- Jedes Syntaxdiagramm besteht aus runden und eckigen Kästchen sowie aus Pfeilen.
- In jedem rechteckigen Kästchen steht die Bezeichnung eines (anderen) Syntaxdiagramms der Menge von Syntaxdiagrammen (ein sogenanntes *Nicht-Terminalsymbol*).
- In jedem runden Kästchen steht ein Wort (*Token*, *Terminalsymbol*) der Sprache.
- Aus jedem Kästchen führt genau ein Pfeil hinaus und genau einer hinein.
- Pfeile dürfen sich aufspalten und zusammengezogen werden.
- Jedes Syntaxdiagramm besitzt genau einen Pfeil, der von keinem Kästchen ausgeht (eintretender Pfeil) und genau einen Pfeil, der zu keinem Kästchen führt (austretender Pfeil).

Token einer Programmiersprache werden durch die Lexikalik der Sprache definiert. Sie stellen die kleinsten zusammenhängenden Grundsymbole einer Sprache, bspw.:

- einfache und zusammengesetzte Symbole (+, <=, (, ...),
- Schlüsselwörter (reservierte Wörter),
- Bezeichner,
- Konstanten.

Trennzeichen (Zeichen für die Trennung von Token), wie Leerzeichen, Tabulatoren oder Zeilenumbrüche sowie Kommentare, werden im Allgemeinen nicht in den Syntaxdiagrammen berücksichtigt.

Mit Hilfe von Syntaxdiagrammen kann festgestellt werden, ob eine bestimmte Zeichenfolge (ein Programm) syntaktisch korrekt ist. Dazu fängt man bei dem eintretenden Pfeil des übergeordneten Syntaxdiagramm an und verfolgt die Pfeile. Erreicht man ein rundes Kästchen, so muss das entsprechende Token als nächstes in der Zeichenfolge auftreten. Erreicht man ein eckiges Kästchen, springt man in das entsprechend bezeichnete Syntaxdiagramm. Existieren alternative Wege, so wählt man den entsprechenden aus. Gibt es keinen Weg durch die Syntaxdiagramme, so ist das Programm syntaktisch nicht korrekt. Nach der Abarbeitung der Zeichenfolge muss der austretende Pfeil des übergeordneten Syntaxdiagramms erreicht worden sein. Sonst ist das Programm ebenfalls nicht syntaktisch korrekt.

Das Prinzip, nach dem Syntaxdiagramme arbeiten, lässt sich durch eine Analogie veranschaulichen: In einem Zoo gibt es eine Menge von Gehegen mit verschiedenen Tieren. Die Gehege können durch Besucher auf Wegen erreicht werden. Wegen der großen Besucherzahlen dürfen dabei die Wege jeweils nur in einer Richtung begangen werden. Es existieren Kreuzungen, an denen mehrere Wege eingeschlagen werden können. In Abbildung 2.1 werden anhand eines Syntaxdiagramms (Wegeplan des Zoos) die möglichen Wege durch den Zoo veranschaulicht. Die Gehege stellen dabei die Token dar. Aus Gründen einer besseren Übersichtlichkeit ist der Plan in zwei Teilpläne (Zoo, Säugetiere) aufgeteilt.

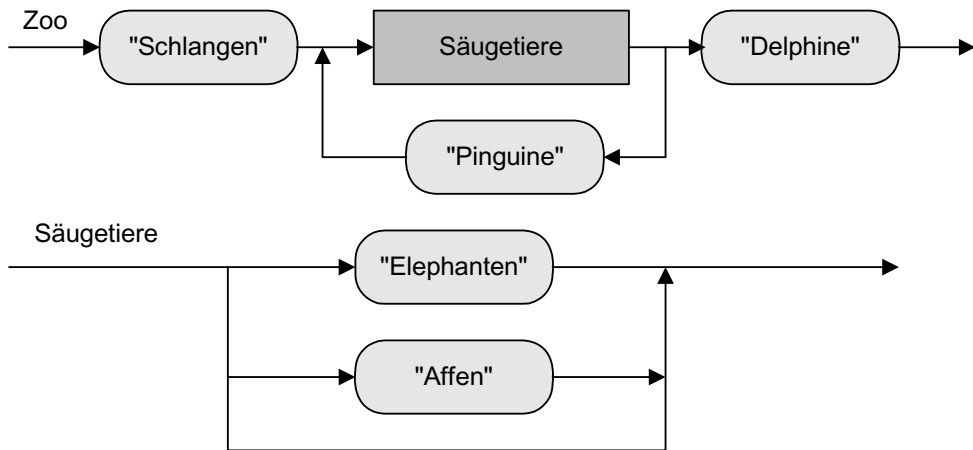


Abbildung 2.1: Wegeplan im Zoo als Syntaxdiagramm

Ein Photograph möchte nun im Zoo eine Fotoserie erstellen. Dabei muss (!) er jeweils ein Foto schießen, wenn er an einem Gehege vorbei kommt. Er orientiert sich an dem Wegeplan. Offenbar kann er die möglichen Bildsequenzen ermitteln, indem er die möglichen Wege durch den Zoo nachvollzieht. Erreicht er im Plan das eckige Kästchen *Säugetiere*, so zeigt der Teilplan *Säugetiere* den weiteren Weg, bis er diesen wieder verlässt und am Ausgang des eckigen Kästchens *Säugetiere* seinen Weg fortsetzt.

Mögliche Bildsequenzen sind zum Beispiel:

- Schlangen Delphine
- Schlangen Elephanten Pinguine Delphine

- $(\dots | \dots)$ bedeutet: Genau ein alternatives Symbol oder eine alternative Symbolfolge innerhalb der Klammern muss auftreten.

Das Zoo-Beispiel kann damit folgendermaßen formuliert werden:

		<Zoo>	:	:=	"Schlangen"		
			:	:=	<Säugetiere>		
			:	:=	{"Pinguine" <Säugetiere>}		
			:	:=	"Delphine"		
			:	:=			
		<Säugetiere>	:	:=	("Elefanten"		
			:	:=	"Affen"		
			:	:=	e		
			:	:=	)		
			:	:=			

Programmieren spielend gelernt mit dem
Java-Hamster-Modell

Boles, D.

2013, XIV, 380 S. 182 Abb., Softcover

ISBN: 978-3-8348-0640-6