

Kolloquium - Informatik

Jarek Mycan

15. Juni 2025

Frage 1: Welche Eigenschaften muss ein Algorithmus haben?

Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder zur Berechnung eines Wertes. Damit ein Verfahren als Algorithmus gilt, muss es folgende Eigenschaften erfüllen:

1. **Finitheit:**
Der Algorithmus besteht aus endlich vielen, wohldefinierten Anweisungen.
2. **Ausführbarkeit (Effektivität):**
Jeder einzelne Schritt muss in endlicher Zeit durchführbar sein – entweder von einem Menschen oder einer Maschine.
3. **Eindeutigkeit (Determinismus):**
Zu jedem Zeitpunkt darf es nur genau eine eindeutig definierte Folgeanweisung geben. Es darf keine Mehrdeutigkeit auftreten.
4. **Endlichkeit (Terminierung):**
Der Algorithmus muss nach endlich vielen Schritten zu einem Ergebnis kommen. Es darf kein unendlicher Ablauf entstehen.
5. **Allgemeinheit:**
Der Algorithmus soll nicht nur für eine konkrete Eingabe funktionieren, sondern für eine ganze Klasse von Problemen bzw. Eingabewerten.

Frage 2: Warum ist die iterative Lösung eines Algorithmus oft effizienter als eine rekursive Lösung?

In vielen Fällen ist eine **iterative Lösung effizienter** als eine rekursive, weil sie **weniger Speicher und Rechenzeit** benötigt. Der Hauptgrund liegt in der Art und Weise, wie rekursive Aufrufe vom System verarbeitet werden:

1. **Speicherverbrauch durch den Aufrufstack:**
Jeder rekursive Funktionsaufruf wird auf dem sogenannten Call-Stack gespeichert. Das führt bei vielen Rekursionsebenen zu hohem Speicherverbrauch und kann sogar zu einem *Stack Overflow* führen.
2. **Overhead durch Funktionsaufrufe:**
Bei jedem rekursiven Aufruf entstehen zusätzliche Operationen wie das Speichern von Rücksprungadressen und Parametern. Das verlangsamt die Ausführung im Vergleich zu einer einfachen Schleife.
3. **Tail-Call-Optimierung fehlt oft:**
In manchen Programmiersprachen (z. B. Java) wird keine sogenannte *Tail-Call-Optimierung* durchgeführt. Das bedeutet, auch bei scheinbar „einfachen“ Rekursionen wächst der Stack weiter an.
4. **Iterative Lösungen sind oft einfacher zu kontrollieren:**
Schleifen erlauben in vielen Fällen eine bessere Kontrolle über Abbruchbedingungen und Zustände – vor allem bei großen Eingabemengen.

Frage 3: In welchen Fällen könnte Rekursion einer Iteration vorzuziehen sein?

Rekursion ist in bestimmten Fällen nicht nur zulässig, sondern sogar vorteilhaft oder natürlicher als eine iterative Lösung. Das gilt insbesondere bei Problemen, die sich von Natur aus rekursiv strukturieren lassen. Typische Anwendungsfälle:

1. **Baum- und Graphstrukturen:**
Rekursion eignet sich sehr gut für das Durchlaufen von Bäumen (z. B. binäre Suchbäume) oder rekursiven Strukturen wie Verzeichnissen. Jede Rekursionsebene entspricht einem Knoten oder Zweig.
2. **Divide-and-Conquer-Strategien:**
Algorithmen wie **Quicksort**, **Mergesort** oder **Binary Search** nutzen das Prinzip der rekursiven Zerlegung in Teilprobleme, die unabhängig voneinander gelöst werden.
3. **Backtracking-Probleme:**
Probleme wie das **Rucksackproblem**, **Sudoku** oder das **8-Damen-Problem** profitieren von rekursiven Lösungen, da sie schrittweise Alternativen ausprobieren und zurückkehren können.
4. **Eleganz und Lesbarkeit:**
In vielen Fällen (z. B. Fibonacci-Zahlen, Fakultät) ist die rekursive Form kompakter und leichter zu verstehen – besonders zu Lern- und Demonstrationszwecken.
5. **Keine explizite Stack-Verwaltung:**
In rekursiven Lösungen übernimmt der System-Stack die Verwaltung der Zwischenergebnisse, was bei komplexen Pfadverfolgungen hilfreich ist.

Frage 4: Was versteht man unter einem effizienten Algorithmus? Welche Maßstäbe werden zur Effizienzbewertung verwendet?

Ein **effizienter Algorithmus** ist ein Verfahren, das ein Problem korrekt löst und dabei möglichst wenig Rechenzeit und Speicherplatz benötigt. Die Effizienz eines Algorithmus wird in der Informatik anhand folgender Maßstäbe bewertet:

1. **Zeitkomplexität (Laufzeitverhalten):**

Sie beschreibt, wie die Rechenzeit des Algorithmus in Abhängigkeit von der Eingabemenge wächst. Gängige Notationen sind z. B.:

- $\mathcal{O}(1)$: konstant
- $\mathcal{O}(n)$: linear
- $\mathcal{O}(n \log n)$: log-linear
- $\mathcal{O}(n^2)$: quadratisch
- $\mathcal{O}(2^n)$: exponentiell

2. **Platzkomplexität (Speicherbedarf):**

Zeigt, wie viel Speicherplatz der Algorithmus je nach Eingabegröße benötigt.

3. **Skalierbarkeit:**

Wie gut lässt sich der Algorithmus auf größere Datenmengen oder parallele Verarbeitung übertragen?

4. **Stabilität und Zuverlässigkeit:**

Insbesondere bei Sortier- oder numerischen Verfahren kann es wichtig sein, dass bestimmte Eigenschaften erhalten bleiben.

5. **Implementierungskomplexität:**

Effiziente Algorithmen sollten auch möglichst einfach, wartbar und nachvollziehbar implementierbar sein.

Frage 5: Wie analysiert man die Zeitkomplexität eines Algorithmus?

Die **Zeitkomplexität** eines Algorithmus beschreibt, wie stark seine Laufzeit in Abhängigkeit von der Eingabegröße n wächst. Um sie zu analysieren, geht man typischerweise wie folgt vor:

1. **Zähle elementare Operationen:**

Man zählt die Anzahl der Schritte (z. B. Vergleiche, Zuweisungen), die der Algorithmus in Abhängigkeit von der Eingabegröße n durchführt.

2. **Unterscheide verschiedene Fälle:**

- **Best Case:** günstigster Fall (z. B. gesuchte Zahl steht an erster Stelle)
- **Worst Case:** ungünstigster Fall (z. B. lineare Suche durchs gesamte Feld)
- **Average Case:** mittlerer Aufwand über viele typische Eingaben

3. **Verwende die Landau-Notation (\mathcal{O} -Notation):**

Sie abstrahiert von konstanten Faktoren und niedrigrangigen Termen und beschreibt das asymptotische Verhalten der Laufzeit bei großen Eingaben.

4. **Analysiere Schleifen und Rekursionen:**

- **Einzelne Schleifen:** Anzahl der Iterationen direkt ablesen (z. B. $\mathcal{O}(n)$)
- **Verschachtelte Schleifen:** multipliziere die Schleifendurchläufe (z. B. $\mathcal{O}(n^2)$)
- **Rekursive Aufrufe:** Rekursionsgleichung aufstellen und lösen (z. B. $T(n) = T(n-1) + c \Rightarrow \mathcal{O}(n)$)

5. **Ignoriere konstante Faktoren und niedrigere Terme:**

Beispiel: Bei $3n + 5$ zählt in der Komplexität nur $\mathcal{O}(n)$.

Frage 6: Was ist der Unterschied zwischen 2^n und n^k algorithmisch gesehen?

Der Unterschied zwischen 2^n (exponentielles Wachstum) und n^k (polynomielles Wachstum) ist entscheidend für die Effizienz und praktische Anwendbarkeit eines Algorithmus:

1. **Polynomieller Algorithmus n^k :**

- Hier steigt die Laufzeit mit der Eingabegröße n in polynomieller Weise, z. B. quadratisch (n^2), kubisch (n^3) oder allgemein n^k , wobei k eine feste Zahl ist.
- Solche Algorithmen gelten als *praktisch effizient lösbar* (auch „in Polynomialzeit“).

2. **Exponentialer Algorithmus 2^n :**

- Hier verdoppelt sich die Laufzeit bei jeder zusätzlichen Einheit von n , was zu einem extrem schnellen Wachstum führt.
- Solche Algorithmen sind in der Regel *nicht effizient* und gelten als unpraktisch für große Eingaben.

3. Mathematischer Vergleich:

$$\lim_{n \rightarrow \infty} \frac{n^k}{2^n} = 0 \quad \text{für jedes feste } k$$

Interpretation: Die exponentielle Funktion 2^n wächst wesentlich schneller als jede polynomielle Funktion n^k . Das bedeutet: Für große n ist ein Algorithmus mit $\mathcal{O}(2^n)$ sehr viel langsamer als einer mit $\mathcal{O}(n^k)$.

4. Beispiele:

- **Polynomiell:** Sortieralgorithmen wie **Mergesort** oder **Heapsort** ($\mathcal{O}(n \log n)$), **Dijkstra-Algorithmus**.
- **Exponentiell:** Brute-Force-Lösungen für das **Rucksackproblem**, **Hamiltonkreis-Problem**, **SAT-Problem**.

Fazit: Während polynomielle Algorithmen auch bei großen Eingaben effizient arbeiten können, sind exponentielle Algorithmen oft nur für kleine Eingaben praktikabel – oder müssen durch Heuristiken oder approximative Verfahren ersetzt werden.

Frage 8: Wie kann man den Speicherbedarf eines Algorithmus reduzieren?

Der Speicherbedarf eines Algorithmus kann durch verschiedene Maßnahmen reduziert werden. Ziel ist es, möglichst wenig Hauptspeicher (RAM) zu belegen und unnötige Datenhaltung zu vermeiden. Mögliche Strategien:

1. **In-place-Verarbeitung:** Daten werden direkt im vorhandenen Speicher bearbeitet, ohne zusätzliche Kopien anzulegen. *Beispiel:* Sortieralgorithmen wie **Quicksort** arbeiten in-place.
2. **Vermeidung unnötiger Datenstrukturen:** Keine überflüssigen Arrays, Listen oder Objekte anlegen – z. B. statt einer Kopie nur mit einem Index arbeiten.
3. **Verwendung von Iteration statt Rekursion:** Rekursive Algorithmen erzeugen für jeden Aufruf einen Stack-Frame \rightarrow hoher Speicherverbrauch. Eine iterative Variante spart Stack-Speicher.
4. **Speicherfreigabe (bei Sprachen mit manueller Speicherverwaltung):** Belegte Speicherbereiche nach Gebrauch sofort wieder freigeben (z. B. in C mit **free()**).
5. **Verwendung platzsparender Datentypen:** Nur so große Datentypen verwenden, wie nötig. *Beispiel:* **byte** statt **int**, falls Zahlenbereich ausreicht.

6. **Daten komprimieren:** Bei großen Datenmengen können Speicherformate oder Kompression eingesetzt werden.
7. **Memoisierung nur selektiv einsetzen:** Zwischenspeichern (Caching) spart Rechenzeit, kann aber viel Speicher verbrauchen. → Abwägen zwischen Zeit- und Platzkomplexität.

Frage 9: Algorithmus zum Tausch zweier Variablen ohne zusätzliche Variable

Ziel: Zwei Zahlenwerte sollen miteinander getauscht werden, ohne eine dritte Hilfsvariable zu verwenden.

Pseudo-Code (mittels Addition und Subtraktion):

Eingabe: Zwei Zahlen a und b

```
a ← a + b  
b ← a - b  
a ← a - b
```

Ausgabe: Vertauschte Werte von a und b

Beispiel:

- Anfangswerte: $a = 5, b = 9$
- Schritt 1: $a = 5 + 9 = 14$
- Schritt 2: $b = 14 - 9 = 5$
- Schritt 3: $a = 14 - 5 = 9$

Ergebnis: Jetzt gilt $a = 9, b = 5$

Alternative Methode: Tausch mit bitweiser XOR-Operation

Diese Methode verwendet die bitweise XOR-Verknüpfung, um zwei Variablen ohne zusätzliche Speicherzelle zu vertauschen.

Pseudo-Code (mittels XOR):

Eingabe: Zwei Zahlen a und b

```
a ← a XOR b  
b ← a XOR b  
a ← a XOR b
```

Ausgabe: Vertauschte Werte von a und b

Beispiel:

- Anfangswerte: $a = 5, b = 9$
- Binärdarstellung:

$$a = 0101$$

$$b = 1001$$

- Schritt 1: $a = a \oplus b = 0101 \oplus 1001 = 1100 \rightarrow a = 12$
- Schritt 2: $b = a \oplus b = 1100 \oplus 1001 = 0101 \rightarrow b = 5$
- Schritt 3: $a = a \oplus b = 1100 \oplus 0101 = 1001 \rightarrow a = 9$

Ergebnis: Jetzt gilt $a = 9, b = 5$

Frage 10: Was versteht man unter einem Attributwert, und wann ist er atomar?

Ein Attributwert ist der konkrete Wert, den ein Feld in einer Tabelle besitzt, z.B. Anna im Attribut Vorname. Atomar bedeutet, dass der Wert nicht weiter aufteilbar ist. Zum Beispiel ist die Adresse "Hauptstr. 12, Berlin, 10115" *nicht atomar*, weil sie aus mehreren logischen Komponenten besteht (Strasse, Stadt, PLZ). Nur wenn diese in separaten Feldern vorliegen, gelten sie als atomar.

Frage 11: Welche Probleme können in nicht normalisierten Datenbanken auftreten?

Es können sogenannte Anomalien auftreten:

- **Einfügeanomalie:** Es ist nicht möglich, einen neuen Elternteil zu speichern, ohne gleichzeitig einen Schüler einzutragen.
- **Löchanomalie:** Wird ein letzter Schüler eines Elternteils gelöscht, geht auch die Elterninformation verloren.
- **Änderungsanomalie:** Wird z.B. die Telefonnummer eines Elternteils geändert, muss dies in allen Zeilen erfolgen, in denen der Elternteil erscheint.

Frage 12: Was bedeutet eine funktionale Abhängigkeit in einer Tabelle?

Ein Attribut A ist funktional abhängig von Attribut B , wenn für jeden Wert von B genau ein Wert von A existiert. Beispiel: Die Telefonnummer ist funktional abhängig von der ElternID, denn jeder Elternteil hat genau eine Telefonnummer.

Frage 13: Welche Bedingungen müssen erfüllt sein, damit sich eine Relation in der 1., 2. bzw. 3. Normalform befindet?

- **1NF:** Alle Attributwerte sind atomar (nicht weiter aufteilbar).
- **2NF:** Die Relation ist in 1NF, und jedes Nichtschlüsselattribut ist voll funktional vom Primärschlüssel abhängig (keine partielle Abhängigkeit bei zusammengesetzten Schlüsseln).
- **3NF:** Die Relation ist in 2NF und alle Nichtschlüsselattribute sind direkt vom Primärschlüssel abhängig (keine transitive Abhängigkeit).

Frage 14: Warum ist die Trennung von Adresse in Straße, Stadt und PLZ wichtig für die Normalisierung?

Weil die Adresse sonst ein zusammengesetzter Attributwert ist, der nicht atomar ist. Für Abfragen, Sortierungen oder die Verarbeitung einzelner Teile (z.B. nur nach PLZ suchen) ist eine getrennte Speicherung notwendig.

Frage 15: Warum ist die Einführung einer Zuordnungstabelle (z.B. SchuelerEltern) notwendig?

Weil die Beziehung zwischen Schülern und Elternteilen **m n** ist: Ein Elternteil kann mehrere Kinder haben, und ein Kind kann mehrere Bezugspersonen (z.B. Mutter, Vater) haben. Dies lässt sich in relationalen Datenbanken nur mit einer Zwischentabelle korrekt abbilden.

Frage 16: Was versteht man unter einer transitiven Abhängigkeit, und wie wird sie beseitigt?

Wenn ein Nichtschlüsselattribut A von einem anderen Nichtschlüsselattribut B abhängt, und B vom Primärschlüssel abhängt, liegt eine transitive Abhängigkeit vor.

vor. Sie wird durch Auslagerung in eigene Tabellen beseitigt. Beispiel: Adresse ist transitiv von SchuelerID über den Namen des Schülers abhängig.

Frage 17: Was ist der Unterschied zwischen Primär- und Fremdschlüssel?

- **Primärschlüssel:** Eindeutiger Identifikator für jeden Datensatz (z.B. SchuelerID).
- **Fremdschlüssel:** Verweist auf den Primärschlüssel einer anderen Tabelle, um eine Beziehung herzustellen (z.B. AdresseID in der Tabelle **Schueler**).

Frage 18: Wann spricht man von einer 1:n- oder m:n-Beziehung zwischen Tabellen?

- **1:n:** Ein Elternteil kann viele Schüler haben (1 Elternteil zu n Schülern).
- **m:n:** Ein Schüler kann mehrere Elternteile haben und ein Elternteil mehrere Schüler. Diese Beziehung wird durch eine Zwischentabelle modelliert.

Frage 19: Wie würde man die Beziehung zwischen Schülern und Elternteilen modellieren, wenn ein Elternteil mehrere Kinder hat?

Mit einer Zwischentabelle (z.B. **SchuelerEltern**), die die IDs beider Parteien enthält. So kann man beliebig viele Verknüpfungen darstellen.

Frage 20: Wie würde eine SQL-Abfrage aussehen, um alle ...?

Frage 21: Wie kann man mit SQL mehrere Tabellen sinnvoll verknüpfen?

Durch Verwendung von JOIN-Befehlen (INNER JOIN, LEFT JOIN etc.), bei denen Tabellen über gemeinsame Schlüssel (Fremd-/Primär) verbunden werden.

Frage 22: Welche Rolle spielt die Normalisierung bei der Wartung und Erweiterung eines DB?

Normalisierung reduziert Redundanzen und Anomalien, was zu:

- geringeren Speicherbedarf,
- weniger Fehlern bei Datenpflege,
- vereinfachter Erweiterbarkeit führt.

Frage 23: Welche Erweiterungen der Datenbankstruktur wären denkbar, wenn z.B. auch Lehrkräfte und Unterrichtsdaten integriert werden sollen?

Einbindung neuer Tabellen:

- Lehrkraft(LehrerID, Name, Fach)
- Unterricht(Klasse, LehrerID, Fach, Raum)
- Stundenplan(Klasse, Tag, Uhrzeit, Fach, LehrerID)

Frage 24: Gibt es Szenarien, in denen man bewusst auf eine vollständige Normalisierung verzichtet?

Ja, z.B. bei Performance-Optimierung oder Analyse Zwecken. Eine zu stark normalisierte Struktur führt zu vielen JOINS, die bei großen Datenmengen Rechenzeit kosten. Daher kann man für bestimmte Reports bewusst "kontrollierte Redundanz" zulassen.

Frage 25: Was versteht man unter einer formalen Sprache?

Eine **formale Sprache** ist eine Menge von Wörtern über einem bestimmten Alphabet, die nach festen Regeln definiert sind.

Frage 26: Wie unterscheidet sich eine formale Sprache von einer natürlichen Sprache?

Unterschied zwischen einer formalen Sprache und einer natürlichen Sprache:

Eine *formale Sprache* ist eine Menge von Zeichenketten, die nach genau definierten Regeln gebildet werden. Sie wird häufig in der theoretischen Informatik und Mathematik verwendet. Beispiele sind Programmiersprachen oder reguläre Ausdrücke.

Eine *natürliche Sprache* hingegen ist eine von Menschen gesprochene Sprache, die historisch gewachsen ist und oft Mehrdeutigkeiten, Unregelmäßigkeiten

und Kontextabhängigkeiten aufweist. Beispiele sind Deutsch, Englisch oder Chinesisch.

Mathematisch betrachtet ist eine formale Sprache eine Teilmenge der Menge aller möglichen Zeichenketten über einem Alphabet:

$$L \subseteq \Sigma^*$$

wobei Σ ein Alphabet und Σ^* die Menge aller endlichen Zeichenketten über diesem Alphabet ist.

Formale vs. Nicht-Formale Sprachen

1. **Formale Sprachen** sind präzise definiert und basieren auf strikten Regeln, oft in der Mathematik, Logik und Informatik. Sie bestehen aus einem **endlichen Alphabet** und einer **exakten Grammatik** (z. B. reguläre Sprachen, kontextfreie Sprachen).

Beispiele:

- Programmiersprachen (Python, Java, C++)
- Mathematische Logik (Prädikatenlogik, Aussagenlogik)
- Reguläre Ausdrücke

2. **Nicht-Formale Sprachen** sind Sprachen, die nicht streng durch formale Regeln definiert sind. Sie enthalten oft Mehrdeutigkeiten, Kontextabhängigkeiten und historische Entwicklungen.

Beispiele:

- **Natürliche Sprachen** wie Deutsch, Englisch, Chinesisch
- **Gesprochene Dialekte** ohne einheitliche Grammatik
- **Alltagssprache** mit umgangssprachlichen oder metaphorischen Bedeutungen
- **Poesie und literarische Ausdrucksformen**, die oft absichtlich von grammatischen Normen abweichen

Frage 27: Was ist eine Grammatik, und aus welchen Komponenten besteht sie?

- einer Menge von **Nichtterminalen** (Variablen),
- einer Menge von **Terminalen** (Symbolen des Alphabets),
- einer Menge von **Produktionsregeln**, die beschreiben, wie Nichtterminale ersetzt werden können,
- einem **Startsymbol**, von dem die Ableitung beginnt.

Frage 28: Erklären Sie den Unterschied zwischen Terminal- und Nichtterminalsymbolen.

Terminale sind die unzerlegbare Symbole der Sprache, während **Nichtterminale** als Platzhalter für andere Zeichen oder Sequenzen dienen.

Frage 29: Was ist der Unterschied zwischen einer regulären und einer kontextfreien Grammatik?

Eine **reguläre Grammatik** definiert reguläre Sprachen und kann durch endliche Automaten erkannt werden. Eine **kontextfreie Grammatik** erlaubt rekursive Definitionen und wird durch Kellerautomaten verarbeitet.

Eine **kontextfreie Grammatik (CFG)** ist eine Grammatik, bei der alle Produktionsregeln die Form $A \rightarrow \alpha$ haben, wobei A ein Nichtterminal und α eine beliebige Zeichenfolge aus Terminalen und/oder Nichtterminalen ist.

Frage 30: Stellen Sie einen Vergleich zwischen Regulären und Kontextfreien Grammatik.

Reguläre Grammatik

Vorteile:

- **Einfach zu verstehen und zu verarbeiten** – Reguläre Grammatiken können mit **endlichen Automaten** erkannt werden.
- **Effiziente Verarbeitung** – Sie benötigen wenig Speicher, da kein Stack nötig ist.
- **Einfache Umsetzung** – Häufig genutzt in Suchalgorithmen und Compilern für Mustererkennung.

Nachteile:

- **Begrenzte Ausdruckskraft** – Keine verschachtelten Strukturen möglich, z. B. keine Klammerausdrücke $((()))$.
- **Nicht für alle Sprachen geeignet** – Beispielsweise kann die Sprache $\{a^n b^n \mid n \geq 1\}$ nicht mit einer regulären Grammatik beschrieben werden.

Kontextfreie Grammatik (CFG) **Vorteile:**

- **Mehr Ausdruckskraft** – Kann auch **verschachtelte Strukturen** darstellen, z. B. mathematische Ausdrücke $((a + b) * c)$.
- **Geeignet für Programmiersprachen** – Sprachen wie Python oder C werden mit CFGs definiert.

- **Erkennt rekursive Strukturen** – Sprachen wie $\{a^n b^n \mid n \geq 1\}$ lassen sich damit einfach beschreiben.

Nachteile:

- **Aufwendiger zu verarbeiten** – Erfordert **Kellerautomaten** mit Speicher, was aufwendiger ist als ein endlicher Automat.
- **Parsing ist schwieriger** – Das Erkennen von CFGs, z. B. in Compilern, ist komplizierter als bei regulären Grammatiken.
- **Nicht immer effizient** – Einige Parsing-Verfahren für CFGs haben eine hohe Rechenzeit.

Zusammenfassung

- **Reguläre Grammatiken** – Einfach, schnell, aber begrenzt.
- **Kontextfreie Grammatiken** – Mächtiger, aber komplexer zu verarbeiten.

Für einfache Muster wie Telefonnummern oder Suchmuster reichen **reguläre Grammatiken**. Für komplexere Strukturen wie Programmiersprachen oder mathematische Ausdrücke braucht man **kontextfreie Grammatiken**.

Frage 31: Sind Programmiersprachen wie beispielsweise Java oder C++ kontextfrei oder regulär.

Nein, **nicht alle Programmiersprachen sind kontextfrei**. Während viele Aspekte einer Programmiersprache mit einer **kontextfreien Grammatik (CFG)** beschrieben werden können, gibt es einige wichtige Eigenschaften, die **über kontextfreie Sprachen hinausgehen**.