PGdP Zusammenfassung

Ludwig Stecher

Blatt 1

- Aufruf des Compilers in der **Kommandozeile** und Ausführung des Programms:

```
javac Program.java
java Program
```

- Der erste Schritt (das Kompilieren) wandelt den Java-Code in **Java-Bytecode**, der in der Datei **Program.class** gespeichert wird.
- **Sonderzeichen**: Intern verwendet Java den Unicode-Zeichensatz **UTF-16**. Für Java-Code sollte jedoch **UTF-8** verwendet werden! Alle Zeichen (sogar Pinguine und andere Emojis) können dadurch einfach in den Code eingefügt werden. Auf manchen Betriebssystemen (*hust* Windows) führt die Ausgabe mancher Zeichen jedoch zu Problemen.

Blatt 2

- **MiniJava-Integer**: Die Zahlen `100_000` und `100000` sind äquivalent. Unterstriche dürfen jedoch nicht am Anfang oder Ende der Zahl stehen. Führende Nullen sind nicht erlaubt.
- **Grammatik**:

```
Number     ::= -? PDigit (Underscore Digit)* | 0
Underscore ::= _*
PDigit     ::= 1 | ... | 9
Digit      ::= 0 | Pdigit
```

- **Syntax von regulären Ausdrücken:**

Syntax	Erklärung	Beispiel
`A B`	A gefolgt von B	`Digit Digit`
`A*`	A kommt beliebig oft vor (mindestens 0 mal)	`Digit*`
`A+`	A kommt beliebig oft vor, aber mindestens 1 mal Entspricht `A A*`	`Underscore+`
`A?`	A kommt 0 mal oder 1 mal vor	`-?`
`(A)`	Gruppierung	`(Digit Underscore)*`
`A | B`	Es kommt entweder A oder B vor	`(PDigit | 0)*`

- Eine Grammatik besteht aus mehreren regulären Ausdrücken.

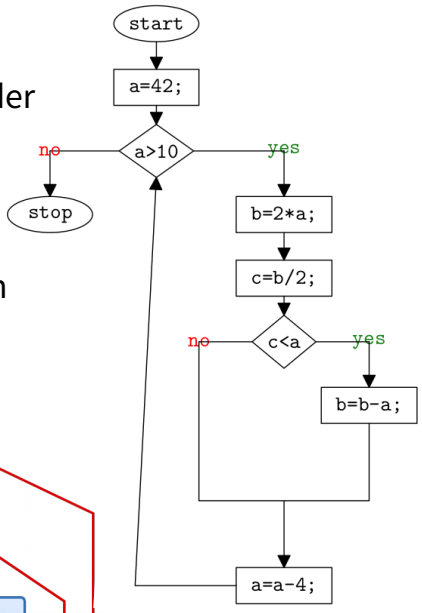
- **Beispiele** für reguläre Ausdrücke:
  - Alle Texte, die kein b enthalten: `a | c | ... | z`
  - Alle Texte, die mit a oder b beginnen und auf c enden: `(a | b) letter* c`
  - Alle Texte, die mit a beginnen und mit b enden oder mit b beginnen und mit a enden: `(a letter* b) | (b letter* a)`
- **Rekursive Grammatiken**: Diese bieten mehr Möglichkeiten, z.B.:

```
palindrom ::= a | b | (a palindrom? a) | (b palindrom? b)
```

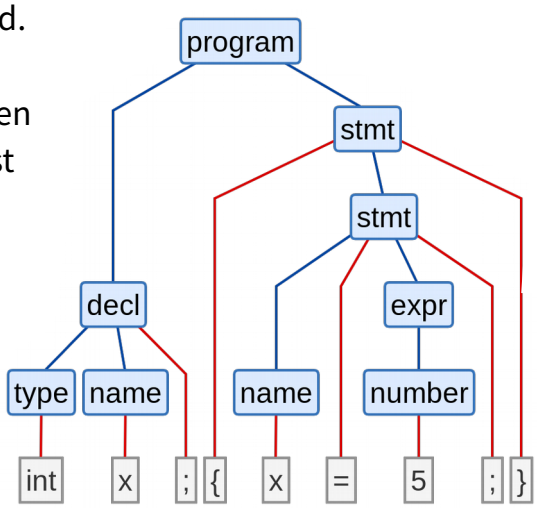
- **MiniJava**: MiniJava ist eine Teilmenge von Java. Das heißt, es gibt weniger Syntax, weniger Datentypen, dafür hat MiniJava eine sehr einfache Grammatik.

Blatt 3

- **Kontrollflussdiagramm**: Dies beschreibt die Ausführung der Anweisungen eines Programms. Start- und Stop-Knoten sind **rund**. Anweisungen sind **rechteckig**. Anweisungen mit Ein-/Ausgabe (d.h. read/write) sind **Parallelogramme**. Bedingungen sind Rauten, diese haben einen yes- und einen no-Pfad.



- **Syntaxbaum**: Ein Text, auf den eine Grammatik zutrifft, lässt sich in einen Syntaxbaum umwandeln. Rechts ist ein möglicher Syntaxbaum der MiniJava-Grammatik.



Blatt 4

- Strings sind unveränderliche (immutable) Zeichenketten.
- Ein Zeichen ist ein `char`, also ein 16-Bit langes Unicode-Symbol. Es kann also Werte zwischen 0 und 65535 annehmen und lässt sich in andere Zahldatentypen umwandeln.
- Zeichen schreibt man am besten als **Literale**, z. B. `'a'`, `'Z'`, `'@'`, `'\n'`
- **Wichtige** String-Methoden:
  - `charAt(int index)` gibt das Zeichen an der Position index zurück
  - `length()` gibt die Anzahl der Zeichen im String zurück

- Weitere, möglicherweise **nützliche** String-Methoden:
  - `toUpperCase()` gibt einen String zurück, in dem alle Kleinbuchstaben durch Großbuchstaben ersetzt wurden
  - `toLowerCase()` – Gegenteil von `toUpperCase()`
  - `equals(String s)` gibt zurück, ob die Zeichen in den Strings übereinstimmen
  - `equalsIgnoreCase(String s)` ist wie equals, ignoriert aber Groß-/Kleinschreibung
  - `toCharArray()` gibt die Zeichen als `char[]` zurück
  - `split(String s)` zerteilt den String in mehrere Stücke. Beispiel: `"AB CDE F".split(" ")` ergibt das Array `{"AB", "CDE", "F"}`
- **Zahlenbasen**:

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512

  - Zahlen in Java sind **binär**. Um eine Dezimalzahl ins Binärsystem zu übersetzen, schreibt man sich am besten die 2er-Potenzen auf. Dann kann man die Zahl in 2er-Potenzen zerlegen und die binäre Darstellung zusammensetzen: $1000 = 512 + 256 + 128 + 64 + 32 + 8 = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 = 1111101000_{(2)}$
  - Binär lässt sich leicht in **hexadezimal** umrechnen und umgekehrt: In hexadezimal entspricht 1 Zeichen 4 binären Zeichen:

    $156D_{(16)} = 0001\ 0101\ 0110\ 1101_{(2)}$ ($5 = 0101_{(2)}$, $6 = 0110_{(2)}$, etc.)
  - **Allgemein**: Um eine Zahl von Basis A in Basis B umzurechnen, teile die Zahl mit Rest durch B. Der Rest ist die 1. Ziffer von rechts. Wiederhole dies so oft mit dem Ergebnis der Division, bis die Division 0 ergibt:

    $78_{(16)}$ zu Basis 8:

    - $78_{(16)} / 8 = F_{(16)}$ Rest 0 → Die 1. Ziffer von rechts ist 0.
    - $F_{(16)} / 8 = 1_{(16)}$ Rest 7 → Die 2. Ziffer von rechts ist 7.
    - $1_{(16)} / 8 = 0_{(16)}$ Rest 1 → Die 3. Ziffer von rechts ist 1.
    - Ergebnis: $\mathbf{170_{(8)}}$
  - **Addition** und **Subtraktion**: Wie in der Schule gelernt.
    Beispiele zu Basis 8:

	5	2	3	6	1
+	1	3	6	2	4
Übertrag		1	1		
	6	6	2	0	5

	1	2	5	3	6	4
–		3	7	4	1	1
Übertrag	1	1	1			
		6	5	7	5	3

- **Multiplikation** und **Division** (ebenfalls zur Basis 8):

5	2	3	6	1	·	6	1
			5	2	3	6	1
	3	$_{3}7$	$_{1}6$	$_{2}6$	$_{4}4$	6	
Übertr.	1	1	1	1	1		
	4	0	4	1	0	4	1

5	2	3	6	/	5	=	1	0	3	7
5										
0	2	3								
	1	7								
		4	6							
		4	3							
			3							

Rest: 3

Blatt 5

- **Quiz**
  - `x/x != 1`, da bei Division durch 0 eine NullpointerException geworfen wird.
  - `0.3 != 0.1 + 0.1 + 0.1` wegen Rundungsfehlern bei Fließkommazahlen
    - Es können nur Summen aus 2er-Potenzen exakt dargestellt werden
  - 32-bit `float`-Werte können größer als 64-bit `long`-Werte werden. Dafür kann nicht jeder `int` bzw. `long` als `float` dargestellt werden, da bei großen `float`-Zahlen die Abstände zwischen Werten immer größer werden
  - Beachte, dass `float`- und `double`-Zahlen auch die Werte **NaN** (Not a Number) und ±**infinity** annehmen können!
  - Beachte, dass ganze Zahlen (`short`, `int`, `long`) durch Overflows das Vorzeichen wechseln können!
  - Jede Menge von Wörtern, die sich mit einer kontextfreien Grammatik beschreiben lässt, die nicht rekursiv ist, lässt sich auch mit einem regulären Ausdruck beschreiben und umgekehrt.
  - Eine Grammatik ist rekursiv, wenn ein Nichtterminal sich selbst enthalten kann, z. B.: `S ::= aAb`, `A ::= ab | S`
  - Ausdrücke werden von links nach rechts evaluiert. Wenn ein `String` mit einer Zahl konkateniert wird, wird die Zahl in einen `String` umgewandelt.
  - Die MiniJava-Grammatik, die in der Vorlesung vorgestellt wurde, ignoriert Punkt vor Strich. Im Gegensatz zu Java ist in MiniJava der Syntaxbaum und Auswertungsbaum von Programmen daher nicht eindeutig.

- **Auswertungssträucher**

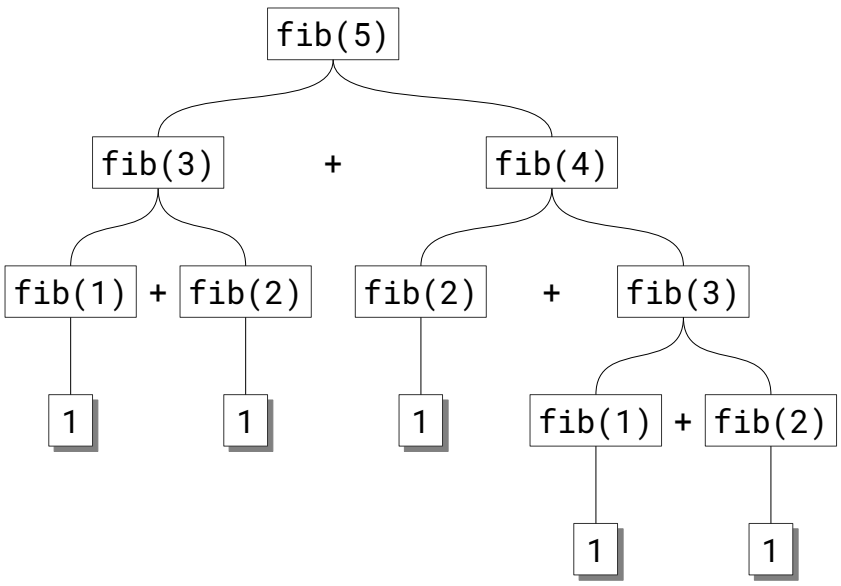
  Beachte: Beim Rechnen gilt Klammer vor Punkt vor Strich. Die Auswertung von Ausdrücken geht von links nach rechts. Eine Operation von zwei `int`s ergibt einen `int`, ggf. wird abgerundet. Ist mind. eine Fließkommazahl beteiligt, ist auch das Ergebnis eine Fließkommazahl.

Blatt 6

- **Rekursion**: eine Funktion ruft sich selbst auf
  - Die Funktion braucht ein Rekursionsende, so wie eine Schleife eine Abbruchbedingung braucht
  - Schleifen können immer in Rekursion umgewandelt werden
  - Nur endrekursive Funktionen können immer in Schleifen umgewandelt werden. Ein Funktion ist endrekursiv, wenn alle rekursiven Aufrufe die letzte Operation in der Funktion sind. Beispiel:

```java
private int rekFunktion(int a, int b) {
    if (a < 5) return 5;
    else if (b > 5) return rekFunktion(a / 2, b);
    else return rekFunktion(a, b * 2) + 5; // nicht endrekursiv!
}
```

  - Rekursion lässt sich oft als Baum veranschaulichen. Bei der Tiefensuche sollte das offensichtlich sein. Hier ein möglicher Baum der **Fibonacchizahlen**:

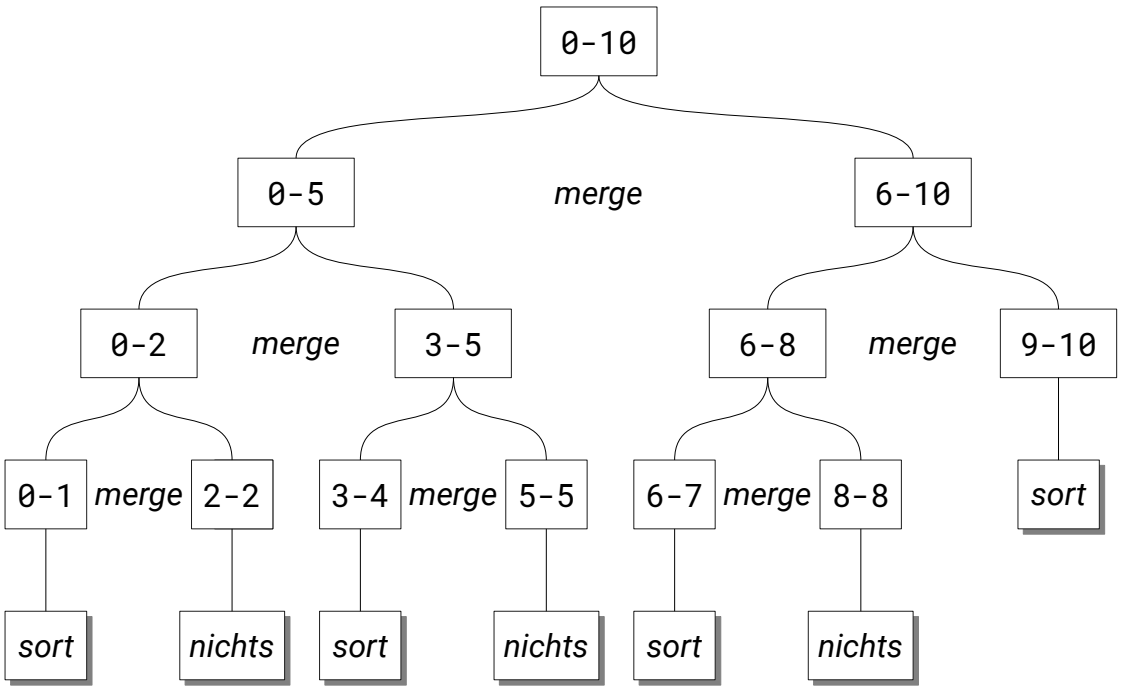


  - Baumstrukturen wie diese sind sehr einfach rekursiv zu implementieren. Mit jedem rekursiven Aufruf geht man 1 Ebene nach unten. Dadurch wird durch den Baum gegangen wie bei der Tiefensuche.

- Bei den Fibonacchizahlen muss man nur 2 Fälle unterscheiden:
  - `1`, falls `n <= 2`
  - `fib(n - 2) + fib(n - 1)`, sonst
- Java-Code:

```
public int fib(int n) {
    if (n <= 2) return 1;
    else return fib(n - 2) + fib(n - 1);
}
```

- So kann der Baum bei **Mergesort** aussehen (`0-10` bedeutet, das Array soll im Bereich von 0 bis 10 sortiert werden, also `a = 0, b = 10`):



- Dabei werden 3 Fälle unterschieden:
  - Mach nichts, falls a = b
  - Bring die Elemente in die richtige Reihenfolge, falls b – a = 1
  - Halbiere das Array, sortiere die Hälften rekursiv und merge sie, sonst
- Java-Code:

```
public void mergesort(int[] array, int a, int b) {
    if (b - a == 1) {
        int first = array[a], second = array[b];
        array[a] = Math.min(first, second);     // bring die Elemente in
        array[b] = Math.max(first, second);     // richtige Reihenfolge
    } else if (b - a > 1) {
        int half = (a + b) / 2;
        mergesort(array, a, half);      // sortiere 1. und 2. Hälfte
        mergesort(array, half + 1, b);  // rekursiv
        merge(array, a, half, b);       // merge die beiden Hälften
    }
}
```