RS1 HA zum 25.11.11

Claas Jaehrling, Sven-Hendrik Haase

November 25, 2011

Aufgabe 4.1

Das Problem hierbei ist, dass dieser Vergleich nur bei Zahlen, die Nahe bei einander liegen, funktioniert.
Es handelt sich um ein komplexes Problem, dass in großer Breite hier (`http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`) und hier (`http://stackoverflow.com/questions/3667932/why-use-float-floattointbi`) diskutiert wird.
Der Konsenz scheint zu sein, dass ein Epsilon eine unzuverlässige Wahl ist, während Epsilon in Kombination mit relativem Fehlerwert ein besserer Weg ist:

```
return Math.abs(a - b) < EPSILON * Math.max(Math.abs(a), Math.abs(b))
```

Aufgabe 4.2

(a)

ISO-8859-1 verwendet 8-bit Koderierung, also:

$$500000 * 8bit = 500000byte$$

Direkte Unicode-Kodierung basiert auf 16-bit, also:

$$500000 * 16bit = 1000000byte$$

Für UTF-8 benötigen wir die Anzahl der Sonderzeichen, da diese mit 16-bit Kodiert werden, die regulären Zeichen aber über 8-bit.
Anzahl der Sonderzeichen:

$$0.0056*500000+0.00287*500000+0.00616*500000+0.00308*500000 = 8855$$

Anzahl der regulären Zeichen:

$$500000 - 8855 = 491145$$

Also ergibt sich:

$$491145 * 8bit + 8855 * 16bit = 508855byte$$

(b)

Anzahl der Symbole:

$$0x4BDF - 0x3400 + 0x9FCF - 0x4E00 = 27054$$

(c)

Wir haben bei direkter Unicode-Kodierung wieder:

$$500000 * 16bit = 1000000byte$$

Für diesen Bereich von UTF-8 werden 24bit pro Zeichen benötigt:

$$500000 * 24bit = 1500000byte$$

Aufgabe 4.3

(a)

$$\begin{aligned} y &= 18 * x && (1)\\ &= (2^4) * x + (2^1) * x && (2)\\ &= (x << 4) + (x << 1) && (3) \end{aligned}$$

(b)

$$\begin{aligned} y &= 14 * x && (4)\\ &= (2^4) * x - (2^1) * x && (5)\\ &= (x << 4) - (x << 1) && (6) \end{aligned}$$

(c)

$$\begin{aligned} y &= -56 * x && (7)\\ &= (2^3) * x - (2^6) * x && (8)\\ &= (x << 3) - (x << 6) && (9) \end{aligned}$$

(d)

$$\begin{aligned} y &= 62 * (x + 2) && (10)\\ &= 62 * x + 124 && (11)\\ &= (2^6) * x - (2^1) * x + 124 && (12)\\ &= (x << 6) - (x << 1) + 124 && (13) \end{aligned}$$

Aufgabe 4.4

(a)

```
bitNand(x, y) {
    // Ansich: ~(a & b), aber:
    return ((~x) | (~y)); // De Morgan'sches Gesetze
}
```

(b)

```
bitXnor(x, y) {
    // Ansich: (a & b) | (~a & ~b), aber:
    return ~(~x | ~y) | ~(x | y); // De Morgan'sches Gesetze
}
```

(c)

```
getByte(x, n) {
    // Wir setzen eine 32-bit Architektur vorraus.
    // Ins kleinste Byte schieben und mit & herausholen.
    return (x >> (8*n)) & 0xff;
}
```

(d)

```
rotateLeft(x, n) {
    // Wir setzen eine 32-bit Architektur vorraus.
    // Geht nur mit unsigned ints.
    // Links rausgeschobene bits werden durch den Ausdruck auf der
    // rechten Seite des ORs wieder rechts reingeholt.
    return (x << n) | (x >> (32 - n));
}
```

(e)

```
abs(x) {
    return ;
}
```

Aufgabe 4.5

(a)

(14)