Lösung zur Prüfung Informatik D-MATH/D-PHYS 4. 3. 2005

Dr. Bernd Gärtner

Aufgabe 1.

(a)

$$
\begin{aligned}
& \texttt{3.0 / 2 + 3 / 2 * 4.0} \\
\rightarrow\ & \texttt{3.0 / 2.0 + 3 / 2 * 4.0} \text{ (Konversion int nach double)} \\
\rightarrow\ & \texttt{1.5 + 3 / 2 * 4.0} \text{ (Exakte double-Division)} \\
\rightarrow\ & \texttt{1.5 + 1 * 4.0} \text{ (int-Division)} \\
\rightarrow\ & \texttt{1.5 + 1.0 * 4.0} \text{ (Konversion int nach double)} \\
\rightarrow\ & \texttt{1.5 + 4.0} \text{ (Exakte double-Multiplikation)} \\
\rightarrow\ & \texttt{5.5} \text{ (Exakte double-Addition)}
\end{aligned}
$$

(b)

$$
\begin{aligned}
& \texttt{3 / 4 > 0 \&\& 3.0 / 4.0 > 0 || 5.0 == -1.0f + 6.0f} \\
\rightarrow\ & \texttt{0 > 0 \&\& 3.0 / 4.0 > 0 || 5.0 == -1.0f + 6.0f} \text{ (int-Division)} \\
\rightarrow\ & \texttt{false \&\& 3.0 / 4.0 > 0 || 5.0 == -1.0f + 6.0f} \text{ (bool-Evaluation)} \\
\rightarrow\ & \texttt{5.0 == -1.0f + 6.0f} \text{ (Short-circuit-Evaluation)} \\
\rightarrow\ & \texttt{5.0 == 5.0f} \text{ (Exakte float-Addition)} \\
\rightarrow\ & \texttt{5.0 == 5.0} \text{ (Promotion nach double)} \\
\rightarrow\ & \texttt{true} \text{ (bool-Evaluation)}
\end{aligned}
$$

Aufgabe 2

```
double pow (double x, int i) {
  if (i==0)
    return 1.0;
  if (i<0)
    return pow(1.0/x, -i);
  return x * pow (x, i-1);
}
```

Die Precondition ist `i>=0 || x != 0.0`.

Aufgabe 3. Der Funktionsaufruf `g(n,f_)` berechnet zwei Werte: den eigentlichen Rückgabewert g_n sowie den Wert f_n des Referenzparameters `f_`. Für $n \geq 2$ gilt nach Konstruktion der Funktion

$$\begin{aligned} g_n &= g_{n-1} + f_{n-1}, \\ f_n &= g_{n-1}. \end{aligned}$$

Letztere Gleichung gilt sogar für $n = 1$, so dass wir für $n \geq 2$ die Gleichung $f_{n-1} = g_{n-2}$ in die erste Gleichung einsetzen können. Wir erhalten dann

$$g_n = g_{n-1} + g_{n-2}, \quad n \geq 2.$$

Wegen $g_0 = 0, g_1 = 1$ ist g_n also die n-te Fibonacci-Zahl. Hier ist die Postcondition:

```
// POST: gibt die n-te Fibonacci-Zahl F_n zurueck; fuer n>0 wird
//       F_{n-1} in f_ gespeichert
```

Aufgabe 4.

(a) Dieses Wort ist ableitbar: `p` → `p*p`→ `p*p*p` → `p*(p)*p` → `p*(p*p)*p`. Schliesslich werden noch alle `p` durch `1` ersetzt.

(b) Diese Wort ist nicht ableitbar. Dazu überlegen wir uns, dass für jedes ableitbare Wort die folgende Invariante gilt: Die Anzahl der `p`-Symbole plus die Anzahl der `1`-Symbole ist gleich der Anzahl der `*`-Symbole plus 1. Diese Invariante gilt offenbar für das Startwort und wird bei jeder Ableitung erhalten.

Da die Invariante aber für das gegebene Wort verletzt ist, kann dieses Wort nicht ableitbar sein.

(c) Dieses Wort ist ableitbar: `p` → `p*p`→ `(p)*p` → `(p*p)*p` → `((p)*p)*p` → `((p*p)*p)*p`. Schliesslich werden noch alle `p` durch `1` ersetzt.

Aufgabe 5.

(a) `int f(T x);` Beim Aufruf der Funktion wird der aktuelle Parameter kopiert; innerhalb der Funktion bezieht sich `x` auf diese Kopie; der aktuelle Parameter bleibt in jedem Fall unverändert. Diese Variante wird zum Beispiel verwendet, wenn man eine mathematische Funktion mit ganzzahligen Argumenten (Fakultät,...) realisieren will.

(b) `int f(T& x);` Beim Aufruf der Funktion wird eine Referenz auf den aktuellen Parameter kopiert; innerhalb der Funktion bezieht sich `x` auf den

aktuellen Parameter; der aktuelle Parameter kann also durch die Funktion verändert werden. Diese Variante wird zum Beispiel verwendet, wenn man einen String manipulieren will, also etwa eine Funktion schreiben will, die in einem gegebenen String alle Grossbuchstaben durch Kleinbuchstaben ersetzt.

(c) `int f(const T& x);` Beim Aufruf der Funktion wird eine Referenz auf den aktuellen Parameter kopiert; innerhalb der Funktion bezieht sich `x` auf den aktuellen Parameter; es ist allerdings nicht erlaubt, *schreibend* auf `x` zuzugreifen. Diese Variante wird zum Beispiel als Ersatz für Variante (a) verwendet, wenn der aktuelle Parameter sehr gross ist (String, Vektor). Es wird damit unnötiges Kopieren vermieden.

Aufgabe 6. Wenn wir *false* mit 0, *unknown* mit 1 und *true* mit 2 identifizieren, erhalten wir folgende Wertetabellen.

$\wedge$	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

$\vee$	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

Wir sehen nun, dass die Tabelle für $\wedge$ der Funktion min (Minimum zweier Zahlen) entspricht, während $\vee$ durch max (Maximum zweier Zahlen) ausgedrückt wird. Diese Beobachtung liegt der folgenden Implementierung zugrunde.

```
private:
  // Repraesentation eines Tribools
  int rep; // 0 = false, 1 = unknown, 2 = true
};

  Tribool::Tribool()
    : rep(1) // unknown
  {}

  Tribool::Tribool (bool b)
    : rep(2*b) // false -> 0, true -> 2
  {}

  Tribool operator&& (Tribool x, Tribool y) {
    Tribool result;
    // result is minimum of the two arguments
    if (x.rep < y.rep)
       result.rep = x.rep;
    else
```

```
      result.rep = y.rep;
    return result;
  }

  Tribool operator|| (Tribool x, Tribool y) {
    Tribool result;
    // result is maximum of the two arguments
    if (x.rep > y.rep)
       result.rep = x.rep;
    else
       result.rep = y.rep;
    return result;
  }
```

Es gibt natürlich andere mögliche Repräsentationen, die dann andere Implementierungen zur Folge haben.