GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Vorsemesterkurs Informatik

Sommersemester 2011

Programmieren und Programmiersprachen

SoSe 2011

Stand der Folien: 30. März 2011

Übersicht

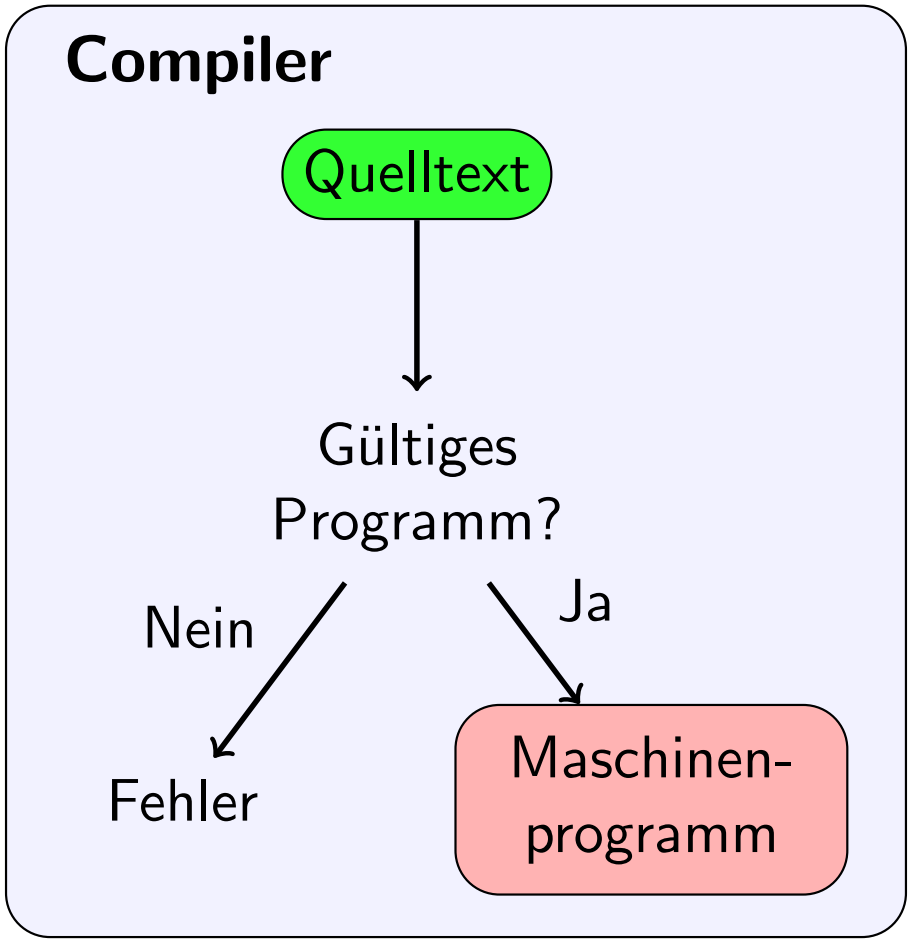
Höhere Programmiersprachen

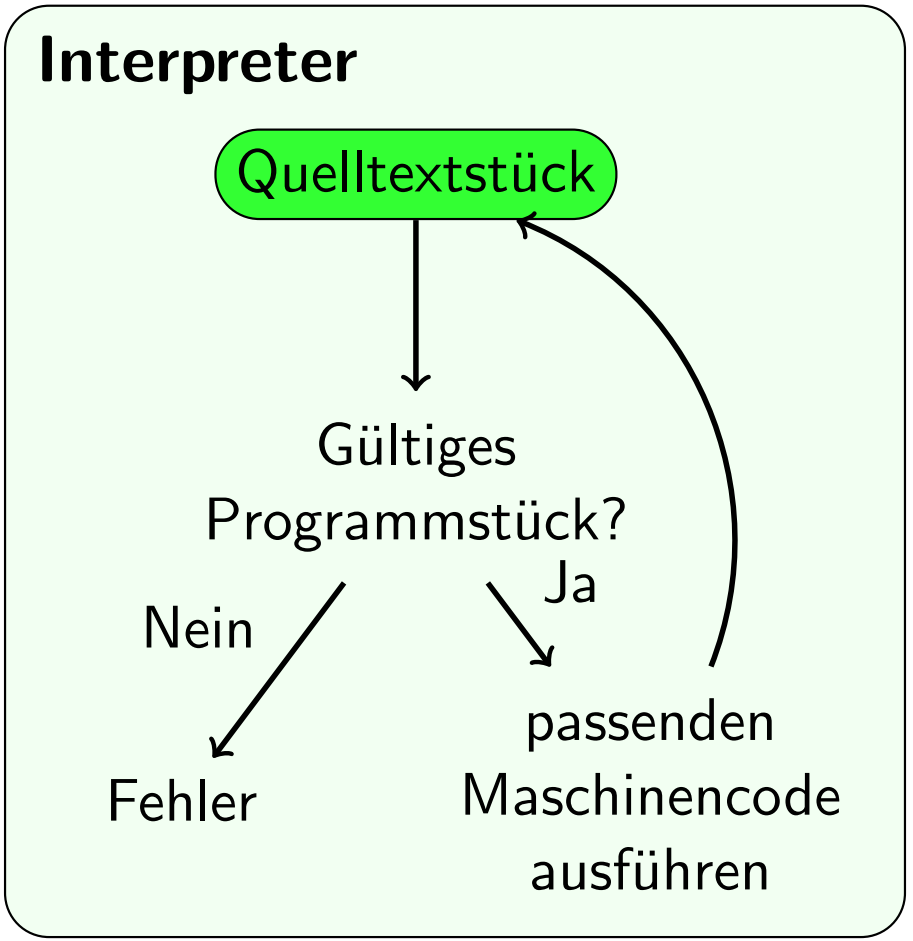
- Maschinenprogramme bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu unverständlich
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

Ausweg: Höhere Programmiersprachen

- Für den Mensch (meist) verständliche Sprache
- Abstraktere Konzepte, nicht genau am Rechner orientiert
- Der Rechner versteht diese Sprachen nicht
- Quelltext = Programm in höherer Programmiersprache

Compiler / Interpreter





- Übersetzung auf einmal
- Compilieren dauert
- Ausführung schnell

- Schnelle Übersetzung e. kleinen Stücks
- Gut zum schnellen Ausprobieren
- Ausführung eher langsam

Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!

Wir betrachen funktionale Programmiersprachen

Einordnung:

- Imperative Programmiersprachen
- Deklarative Programmiersprachen
  - Funktionale Programmiersprachen

Funktionale Programmiersprachen

- Programm = Menge von Funktionsdefinitionen
- Ausführung = Auswerten eines Ausdrucks
- Resultat = eine einziger Wert
- keine Seiteneffekte (sichtbare Speicheränderungen)!
  (insbes.: keine Programmvariablen, Zuweisung, . . .)
- Es gilt das Prinzip der *referentiellen Transparenz*: Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat.

Haskell

Haskell

- die pure funktionale Programmiersprache
- relativ neu: erster Standard 1990
- Benannt nach dem amerik. Mathematiker *Haskell B. Curry* (1900 - 1982)
- Haskell 98 veröffentlicht 1999, Revision 2003
- Haskell 2010, veröffentlicht Juli 2010

Die Informationsquelle: http://haskell.org

GHC und GHCi

- Wir verwenden den Glasgow Haskell Compiler, bzw. den Interpreter dazu: GHCi
- Homepage: `http://www.haskell.org/ghc`.
- Auf den RBI-Rechnern: Kommando `ghci` bzw. `/opt/rbi/bin/ghci`
- Selbst-Installieren: Am besten die Haskell Platform: `http://hackage.haskell.org/platform/`

Bedienung des GHCi

```
/opt/rbi/bin/ghci ⏎
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude>  1+1 ⏎
2
Prelude> 3*4 ⏎
12
Prelude> 15-6*3 ⏎
-3
Prelude> -3*4 ⏎
-12 Prelude> 1+2+3+4+ ⏎
<interactive>:1:8: parse error (possibly incorrect indentation)
```

Einige Interpreterkommandos

`:quit`	Verlassen des Interpreters
`:help`	Der Interpreter zeigt einen Hilfetext an, Übersicht über die verfügbaren Kommandos
`:load` *Dateiname*	Lädt Haskell-Quellcode der entsprechenden Datei, die Dateiendung von *Dateiname* muss `.hs` lauten.
`:reload`	Lädt die aktuelle geladene Datei erneut (hilfreich, wenn man die aktuell geladene Datei im Editor geändert hat).

Haskell-Quellcode

- im Editor erstellen
- Endung: `.hs`

Beispiel: Datei `hallowelt.hs`

```
wert = "Hallo Welt!"
```

```
> /opt/rbi/bin/ghci ⏎
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load hallowelt.hs ⏎
[1 of 1] Compiling Main  ( hallowelt.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

funktioniert nur, wenn `hallowelt.hs` im aktuellen Verzeichnis ist

Haskell-Quellcode

Beispiel: Datei `hallowelt.hs` liegt in `vorkurs/`

```
wert = "Hallo Welt!"
```

```
> /opt/rbi/bin/ghci ⏎
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load hallowelt.hs ⏎

<no location info>: can't find file: hallowelt.hs
Failed, modules loaded: none.
Prelude> :load programme/hallowelt.hs ⏎
[1 of 1] Compiling Main ( programme/hallowelt.hs, interpreted )
Ok, modules loaded: Main.
*Main> wert ⏎
"Hallo Welt!"
```

Nächstes Beispiel

```
zwei_mal_Zwei = 2 * 2

oft_fuenf_addieren = 5 + 5 + 5 + 5 +5 + 5 + 5 + 5 + 5 + 5 + 5

beides_zusammenzaehlen = zwei_mal_Zwei + oft_fuenf_addieren
```

```
Prelude> :load einfacheAusdruecke.hs ⏎
[1 of 1] Compiling Main  ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> zwei_mal_Zwei ⏎
4
*Main> oft_fuenf_addieren ⏎
55
*Main> beides_zusammenzaehlen ⏎
59
*Main> 3*beides_zusammenzaehlen ⏎
177
```

Funktionsnamen

müssen mit einem Kleinbuchstaben oder dem Unterstrich _ beginnen, sonst

```
Zwei_mal_Zwei = 2 * 2

...
```

```
Prelude> :load grossKleinschreibungFalsch.hs
[1 of 1] Compiling Main ( grossKleinschreibungFalsch.hs )

grossKleinschreibungFalsch.hs:3:0:
    Not in scope: data constructor ‘Zwei_mal_Zwei’
Failed, modules loaded: none.
```

Kommentare

Eine Quelltext-Datei enthält neben dem Programm:

- Erklärungen und Erläuterungen
- Was macht jede der definierten Funktionen?
- Wie funktioniert die Implementierung?
- Was ist die Idee dahinter? Man sagt auch:

$\rightarrow$ Der Quelltext soll **dokumentiert** sein!

Wie kennzeichnet man etwas als Kommentar in Haskell?

- Zeilenkommentare: `--` Kommentar...
- Kommentarblöcke: Durch `{-` Kommentar `-}`

Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

```
{- Hier steht noch gar keine Funktion,
   da auch die naechste Zeile noch im
   Kommentar ist

wert = "Hallo Welt"

   gleich endet der Kommentar -}

wert2 = "Hallo Welt"
```

Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

```
{- Hier steht noch gar keine Funktion,
   da auch die naechste Zeile noch im
   Kommentar ist

wert = "Hallo Welt"

   gleich endet der Kommentar -}

wert2 = "Hallo Welt"
```