Compiler
— Blatt 1 —

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik

Hochschule München

Sommersemester 2014

27.03.14 14:14

> Dieses Blatt ist ein Tutorial an dem Sie sich selbst entlang hangeln sollen. Wenden Sie sich bei Problemen/Fragen bitte an mich.

Wir programmieren in Haskell und nutzen das FP Haskell Center. Sie können alternativ dazu die Haskell Platform auf Ihrem eigenen Rechner installieren. Damit haben Sie den Glasgow Haskell Compiler und den dazugehörigen Interpreter und können mit einem beliebigen Editor Haskell-Dateien bearbeiten. Es gibt noch einige mehr oder weniger ausgereifte IDEs. Die meisten Haskeller nutzen Texteditoren, wie vim oder Emacs mit Haskell-Plugins.

Erste Schritte im FP Haskell Center

Account erzeugen

Gehen Sie auf `https://www.fpcomplete.com/ide` und erstellen Sie sich einen Account.

Dazu tragen Sie Ihre E-Mail-Adresse bei `Register` ein. Nutzen Sie am sinnvollsten Ihre `@hm.edu`-Adresse. Damit können Sie, wenn Sie wollen, auf die Personal-Lizenz als Free Academic Account upgraden. Damit können Sie Ihre Projekte in ein Git-Repository pushen.

Sie erhalten dann eine E-Mail und klicken einfach auf den Link. Sie landen damit direkt in der IDE die komplett im Browser läuft.

Klicken Sie dann erstmal oben auf Ihren generierten Username `fpuser4711...` und geben sich einen vernünftigen Username und ein Passwort, damit Sie sich beim nächsten

Mal vernünftig einloggen können. In Ihrem Profil können Sie außerdem noch die Editor Keymappings auf Vim oder Emacs einstellen (oder es einfach lassen wie es ist).

Anmerkung: Im FP Haskell Center müssen Sie Ihren Code immer compilieren lassen und ausführen. Das Compilieren passiert, wie in einer anderen IDE auch, ständig im Hintergrund, d.h. der Zeitaufwand ist nicht all zu groß. **Aber:** Sie müssen immer einen `main`-Funktion haben und können nur diese ausführen. Wenn Sie die Haskell Platform auf ihrem Rechner installiert haben, können Sie den Interpreter `ghci` nutzen um beliebige Funktionen direkt auszuführen.

Ein erstes Projekt erstellen

- Klicken Sie oben rechts auf Ihren Benutzernamen
- Klicken Sie dann auf "New Project"
- Erzeugen Sie dann ein leeres Projekt mit dem Titel "first steps"

Sie landen dann wieder in der IDE und sehen einen Welcome-Text.

Ein erstes Modul mit dem Namen `Main` ist bereits erzeugt. Klicken Sie darauf um es im Editor zu öffen.

Sie sehen dann folgenden Code:

```
1 -- | Main entry point to the application.
2 module Main where
3
4 -- | The main entry point.
5 main :: IO ()
6 main = do
7     putStrLn "Welcome to FP Haskell Center!"
8     putStrLn "Have a good day!"
```

Ein Haskell-Modul (= Haskell Source File, z.B. `Main.hs`) beginnt (in diesem Fall Zeile 2) immer mit `module ... where`, wobei für die drei Punkte der Modulname zu setzen ist.

Die Zeilen 1 und 4 sind Kommentare. In Haskell wird `--` statt `//` sowie `{-` und `-}` statt `/*` und `*/` genutzt. Das Pipe-Symbol ist ein Steuerzeichen für haddock.

In den Zeilen 5-8 wird die `main`-Funktion implementiert. Zeile 5 ist die sog. Typsignatur. Sie gibt an, dass die `main`-Funktion den Typ `IO ()`, gesprochen *IO Unit* hat. Die zwei Doppelpunkte bedeuten *hat den Typ*. Also "main hat den Typ IO Unit". `IO ()` wiederum bedeutet, die Funktion kann I/O machen und gibt nichts zurück (Unit heisst in anderen Sprachen übrigens `void`).

In Zeile 6 beginnt dann die Definition von `main`. Sie können sich für das `= do` eine öffende, geschweifte Klammer denken. Alles was dann weiter eingerückt ist als das `m` von `main`

gehört zu dieser Funktion. Die Funktion endet sobald etwas in der Spalte unter dem `m` steht oder, wie in diesem Fall, wenn die Datei zuende ist.

In den Zeilen 7 und 8 wird mit Hilfe der Funktion `putStrLn` ein `String` gefolgt von einem Zeilenumbruch ausgegeben.

Führen Sie die Anwendung aus, so sehen Sie in der Console die beiden Zeilen:

```
Welcome to FP Haskell Center!
Have a good day!
```

Erste Änderungen am Code

Führen Sie folgende Änderungen am Code durch und lassen Sie die Anwendung laufen. Überlegen Sie sich vorher was passieren könnte und überprüfen Sie Ihre Idee.

a. Kommentieren Sie die Typsignatur von `main` aus.
b. Ersetzen Sie die Funktionsaufrufe `putStrLn` durch `putStr`.
c. Ersetzen Sie die Funktionsaufrufe `putStr` durch `print`.
d. Ersetzen Sie den String `Have a good day` durch die Zahl 123.
e. Ersetzen Sie die Funktionsaufrufe `print` wieder durch `putStrLn`.

Im letzten Fall zeigt der Daumen nach unten und Sie sehen eine Fehlermessage, die in etwa so aussieht:

```
Main.hs:8:14:
    No instance for (Num String) arising from the literal `123'
    Possible fix: add an instance declaration for (Num String)
    In the first argument of `putStrLn', namely `123'
    In a stmt of a 'do' block: putStrLn 123
    In the expression:
      do { print "Welcome to FP Haskell Center!";
           putStrLn 123 }
```

Diese Fehlermeldung ist zunächst etwas schwer zu verstehen, denn Sie sagt uns nicht einfach: “Achtung: `putStrLn` kann nur auf Werte vom Typ `String` angewendet werden.”. Aber im Prinzip heisst es das nur etwas umständlicher. (Mehr Info bei Bedarf später.)

Haben Sie Probleme mit solchen Fehlermeldungen schauen Sie sich einfach mal die Typen an. Das geht im FP Haskell Center mit `Ctrl-i`.

Gehen Sie mit dem Cursor auf `putStrLn` und drücken Sie `Ctrl-i`. Dann wird Ihnen die folgende Message angezeigt:

```
putStrLn :: String -> IO ()
```


Dies wird gelesen als "putStrLn hat den Typ String nach IO Unit" und heißt, sie nimmt einen String als Parameter, macht I/O (in diesem Fall nur O) und gibt nichts zurück.

Sobald wir keine Fehler mehr im Programm haben, reicht übrigens für die Typinformation den Cursor an die entsprechende Stelle zu setzen. Machen Sie aus dem `putStrLn 123` wieder ein `print 123` und gehen Sie mit dem Cursor einfach nur irgendwo auf die 123. Dann sehen Sie unten:

```
123 :: Integer
```

Und ein `Integer` ist nunmal kein `String`!

Sie müssen sich in Haskell übrigens nicht merken was wie gecastet und wie automatisch aufgerufen wird. In Haskell wird **nichts** gecastet und automatisch aufgerufen. Und ja, das ist ein Feature und kein Bug!

Ein kleines bisschen Zahlenspielereien

Wir wollen nun erste eigene Funktionen schreiben.

Fügen Sie an Ihr `Main`-Modul die folgenden Zeilen an:

```
square :: Integer -> Integer
square n = n * n
```

Um die Funktion auch zu nutzen, ergänzen Sie die `main`-Funktion um die Zeile

```
    print $ square 123
```

Probieren Sie aus was passiert, wenn Sie das `$`-Zeichen weglassen.

Die Fehlermeldung ist diesesmal etwas besser verständlich und behauptet Sie wenden `print` auf zwei Parameter an. Der Compiler meint also der Ausdruck `print square 123` bedeutet, wende print auf die beiden Parameter `square` und `123` an.

In Haskell werden Parameter nämlich nicht in einer Parameterliste übergeben, sondern durch Whitespace getrennt hintereinander geschrieben.

Um es also richtig zu stellen, müssen Sie den Teilausdruck `square 123` klammern, also `print (square 123)`. Das geht auch. Da Haskeller aber noch fauler als andere Programmierer sind, schreiben sie ungern Klammern und haben einen Operator `$` der die öffnende Klammer ersetzt und die dazugehörige schließende entspricht einfach dem Ende des Ausdrucks (in diesem Fall dem Zeilenende).

Als nächsten Schritt wollen wir die Ausgabe etwas schöner machen und statt nur dem Ergebnis ausgeben:

```
square(123) = 15129
```


wir brauchen also einen `String square(123) =` und müssen das Ergebnis `15129 :: Integer` auch in einen String umwandeln und das ganze konkatenieren. Das ganze funktioniert dann mit

```
putStrLn ("square(123) = " ++ (show (square 123)))
```

oder wenn wir wenigstens ein paar Klammern vermeiden wollen mit

```
putStrLn $ "square(123) = " ++ (show $ square 123)
```

Die Funktion `show` entspricht in etwa der Methode `toString()` in Java. Sie wird aber nie implizit aufgerufen und ist auch nicht für alle Datentypen verfügbar. (Anmerkung: Das kann auch gar nicht sein, den in Haskell ist jede Funktion selbst auch ein Wert, der behandelt werden kann wie alle anderen Werte, z.B. als Ergebnis zurück gegeben, in eine Liste gespeichert, …. Und was soll den Bitte `show square` ausgeben? Die Parabel?)

Mit dem Operator `++` werden `String`s im Besonderen, aber auch beliebige Listen, konkateniert. Ein String in Haskell ist eine Liste von `Char`, geschrieben `[Char]`.

Schreiben Sie als nächstes eine Funktion `double` die einen Wert verdoppelt. Die Typsignatur ist übrigens nicht notwendig. Der Haskell-Compiler kann die Typen mit einem sog. Typ-Inferenz-Mechanismus selbst berechnen. Aber es gehört zum guten Stil (Dokumentation!) die Typsignaturen trotzdem anzugeben.

Fügen Sie zur `main`-Funktion einen Aufruf von `double` hinzu.

Fügen Sie noch die folgenden zwei Funktionen inkl. Implementierung und Aufruf in der `main` hinzu:

```
doubleThenSquare :: Integer -> Integer
squareThenDouble :: Integer -> Integer
```

Beide Funktionen sollen `double` und `square` nacheinander anwenden nur in verschiedener Reihenfolge.

Und jetzt das Ganze testen

Es lässt sich mathematisch beweisen, dass für ein beliebiges `n :: Integer` gilt

```
squareThenDouble n <= doubleThenSquare n
```

Wir wollen das mit dem QuickCheck-Framework automatisch für Zufallswerte testen.

Erzeugen Sie dazu ein zweites Modul mit dem Namen `Test`. Unter der Zeile `module ...` importieren Sie mit der folgenden Anweisung das QuickCheck-Modul.

```
import Test.QuickCheck
```

Implementieren Sie anschließend die obige Behauptung als sog. *Property*:

```
prop_squareDouble :: Integer -> Bool
prop_squareDouble n = squareThenDouble n <= doubleThenSquare n
```

Die Funktion nimmt einen `Integer` und berechnet den Wahrheitswert. Das gibt Ihnen gleich mal eine Fehlermeldung, denn die Funktionen `squareThenDouble` und `doubleThenSquare` gibt es im Modul `Test` gar nicht. Wir müssen einfach unter oder über dem QuickCheck-Import noch ergänzen:

```
import Main
```

oder besser:

```
import Main  (squareThenDouble, doubleThenSquare)
```

dann importieren wir nämlich nicht alles sondern nur die zwei Funktionen die wir benötigen.

Jetzt müssen wir nur noch irgendwie dem quickCheck-Framework sagen, dass er bitte unser Property testen soll. Dazu schreiben wir uns folgende `main`-Funktion:

```
main :: IO ()
main = quickCheck prop_squareDouble
```

Die Funktion `quickCheck` nimmt eine Funktion, in dem Fall `prop_squareDouble`, und füttert diese standardmäßig einigen Zufallswerten. Was für Werte benötigt werden, bekommt `quickCheck` durch den Typ der Funktion heraus. Wieviele Zufallswerte genutzt werden, kann als Parameter übergeben werden.

Jetzt müssen wir nur noch dem FP Haskell Center sagen, dass wir diesesmal nicht das `Main`-Modul sondern das `Test`-Modul ausführen wollen, wenn wir auf den “Play”-Button klicken. Dazu klicken wir einfach neben “Main” auf die Pfeilspitze nach unten und wählen dann das Modul `Test` aus.

Wenn Sie Ihren Test nun ausführen, sehen Sie eine Ausgabe, die Ihnen einfach zeigt wieviele Tests erfolgreich gelaufen sind.

Ändern Sie Ihr Property so, dass statt `<=` auf `<` getestet wird und testen Sie es.