Praktische Informatik 3 WS 04/05

Bonusübungsblatt

Ausgabe: 26.01.05

Abgabe: 15/16.02.05

Christoph Lüth <cxl>
Shi Hui <shi>
Klaus Lüttich <luettich>
Wolfgang Machert <wmachert>
Christian Maeder <maeder>
Hennes Märtins <maertins>
Kai-Florian Richter <richter>
Dennis Walter <dw>
Diedrich Wolter <dwolter>

Diese Aufgabenblatt enthält insgesamt 10 Bonuspunkte, d.h. es gilt wie alle anderen Aufgabenblätter mit 8 Punkten als bestanden; 20 Punkte entsprechen der Bestnote.

14 *Haskell is Life!* *15 Punkte*

Zu dem Rennspiel des letzten Aufgabenblattes gehört natürlich noch ein kleines Demospiel, welches auf dem CeBIT-Messestand verteilt werden kann, um die vorbeiziehenden Devotionaliensammler zufriedenzustellen. Dabei fällt die Wahl auf einen Klassiker:

In dieser Aufgabe soll eine einfache Haskell-Version von Conway's *Game Of Life* implementiert werden. Das Spiel funktioniert wie folgt:

1. Als Spielfeld wird das Fenster in ein Gitter von mindestens 20 Zellen Breite und Höhe unterteilt. Jedes der Elemente dieses Gitters (sog. *Zellen*) kann gesetzt oder gelöscht (*lebendig* und *tot*) sein. Der Spieler kann durch Klicken in dieses Gitter einzelne Punkte setzen, oder gesetzte Punkte wieder löschen. Durch Betätigung der Leertaste wird das eigentliche Spiel gestartet: (*5 Punkte*)

2. Für alle Zellen des Gitters wird gleichzeitig nach folgenden Regeln ein neuer Wert (eine neue *Generation*) berechnet:

   - Eine Zelle mit genau drei lebendigen Nachbarn wird lebendig.
   - Eine Zelle mit mehr als drei oder weniger als zwei lebendigen Nachbarn stirbt.

   Hierbei sind die Nachbarn die Zellen direkt oder diagonal daneben (aber nicht die Zelle selbst). Nach der Berechnung der neuen Werte werden diese angezeigt, ggf. gewartet, und die nächste Iteration gestartet.

   Das Spiel läuft jetzt solange vor sich hin, bis keine Veränderung mehr eintritt. Die Anzahl der Generationen soll gezählt und auf dem Bildschirm ausgegeben werden. Die Geschwindigkeit, mit der neue Generationen errechnet werden, kann mit den Pfeiltasten herauf- oder herabgesetzt werden; ein Anfangswert von einer halben Sekunde erscheint sinnvoll. Mit der Leertaste kann das Programm bis zur Betätigung einer weiteren Taste angehalten, mit der Taste 'q' das momentane Spiel abgebrochen werden. (*10 Punkte*)

15 *Der Kleine Mathematiker* *15 Punkte*

Wie wir seit dem PISA-Test wissen, ist Deutschland mathematisches Notstandsgebiet, und weil Mathematik die Grundlage der modernen Wissenschaft ist, und weil, wie uns immer wieder vollmundig verkündet wird, wir in einer Wissensgesellschaft leben, wollen wir nicht unserer sozialen Pflichten entziehen und in dieser Aufgabe Haskell einsetzen, um die Mathematik einfacher zu machen.

Etwas konkreter formuliert wollen wir in Haskell ein Programm entwickeln, welches Funktionsdefinitionen einlesen, auswerten, und symbolisch differenzieren und integrieren kann.

Unser Programm soll nach dem Start folgende Kommandos von der Standardeingabe lesen und bearbeiten:

- Mit dem Kommando `def` wird eine Funktion definiert. `def` erhält als Argument einen Funktionsnamen, dann ein Gleichheitszeichen, und auf der rechten Seite einen Funktionsausdruck.

  Ein Funktionsausdruck kann aus folgenden Operatoren aufgebaut werden:

  - Addition, Subtraktion, Multiplikation, Division und Exponenation;
  - numerische Konstanten (Vorkommastellen, ggf. Dezimalpunkt und Nachkommastellen; kein Vorzeichen, keine Exponentialnotation), sowie Konstanten `pi` und `e`;
  - die trigonometrischen Funktionen `sin`, `cos` und `tan`;
  - der Parameter der Funktion, der immer `x` heißt;
  - eine vorher definierte Funktion, angewandt auf ein Argument;
  - sowie Klammern zur expliziten Vorfahrtsregelung.

  Folgendes sind beispielsweise legale Funktionsdefinitionen:

  ```
  def f = 3*x^2+ 5*x+ sin(x)
  def g = 45*(1/sin(x))
  ```

  (*5 Punkte*)

- Mit dem Kommando `derive` wird eine Funktion symbolisch nach `x` abgeleitet. Beispiel:

  ```
  derive f(x)
  derive sin(3*x)
  derive x^2+ 3*x+ 7
  ```

  (*3 Punkte*)

- Mit dem Kommando `integrate` wird eine Funktion symbolisch integriert (das ist nicht für alle Funktionen möglich). Beispiel:

  ```
  integrate x*5
  ```

  (*3 Punkte*)

- Mit `plot` wird eine Funktion gezeichnet (mittels HGL). `plot` erhält als Argument die Funktion, die gezeichnet werden soll, sowie den Anfangs- und Endwert für das Funktionsargument x. Zum Beispiel wird mit

  ```
  plot f from 0 to 13.0
  ```

  der Funktionsverlauf für `f` für $x = 0, \ldots, 13.0$ dargestellt. Das Fenster hat eine feste Größe. Die Skalierung in x-Richtung ergibt sich aus der Fenstergröße, die Skalierung in y-Richtung wird so berechnet, dass die jeweils größten und kleinsten Werte den oberen und unteren Rand bilden. (*4 Punkte*)

Hinweis: Definieren Sie zuerst eine Lexikalik und eine Grammatik für die Eingabesprache, das erleichtert die Implementierung eines Parsers ganz ungemein.

Lexikalik heißt, dass Sie einen Typen `Token` definieren, der die Schlüsselbegriffe der Eingabesprache modelliert, also hier zum Beispiel die arithmetischen Operatoren, Variablenname, Klammern, Schlüsselworte (`def`, `derive`, `integrate` und `plot`), Ziffernfolgen und Buchstabenfolgen (d.h. eine ununterbrochene Folge von Ziffern und Buchstaben). Definieren Sie sich eine Funktion

```
lex :: String-> [Token]
```

die eine Zeichenkette in eine Sequenz von Token übersetzt, und dabei Leerzeichen überliest. (Für den Lexer sind die Funktionen aus dem Standardmodul `Char` hilfreich.)

Definieren Sie dann eine abstrake Syntax. Diese besteht aus einem Datentypen `Cmd`, der die Kommandos repräsentiert, und einem Datentypen `Expr`, welche die zu modellierenden Operatoren enthält (siehe die Vorlesung vom 13.12.04). Der Parser besteht dann aus einer Funktion, welche Kommandos parsiert (`Maybe` hier nur, um den Fehlerfall zu modellieren):

```
parse :: [Token]-> Maybe Cmd
```

Zur Implementierung des Parsers für die Kommandosprache können Sie auf die Parserkombinatoren in der Vorlesung vom 13.12.04 zurückgreifen.

Dies ist Version 1.0 vom 2005/01/27 10:20:00.