Theoretische Informatik

Eine unverbindliche Übung

1 Einleitung

Als angehende Informatiker entwickeln Sie meist Programme. Dadurch haben Sie sich bereits fundierte Kenntnisse in der angewandten Informatik erworben. Für die Programmentwicklung benötigen Sie Programmiersprachen, Compiler, Betriebssysteme und eventuell auch Datenbanken. Anschließend lassen Sie das Programm auf einem Mikroprozessorsystem laufen. So kommen Sie auch mit der praktischen und der technischen Informatik in Kontakt. Es gibt aber ein weiteres Teilgebiet der Informatik, das im Lehrplan nicht erwähnt wird und daher in Ihrer Ausbildung weitgehend außen vor bleibt. Es ist die theoretische Informatik. Im Rahmen dieser unverbindlichen Übung soll Ihnen die Möglichkeit gegeben werden, sich auch mit diesem faszinierenden Teilgebiet der Informatik etwas zu beschäftigen.

2 Organisation

Der Kurs soll in Form einer einstündigen unverbindlichen Übung angeboten werden. Das bedeutet, dass für uns in etwa 36 Stunden zur Verfügung stehen und keine Noten vergeben werden. Die Teilnahme an der Übung wird aber natürlich im Zeugnis vermerkt, so dass Sie Ihr Engagement in diesem Gebiet nachweisen können.

Der Inhalt dieses Kurses wird hauptsächlich informell vermittelt. Die Grundgedanken der theoretischen Informatik werden anschaulich vorgestellt und anschließend anhand von Beispielen diskutiert. Da die Zusammenhänge meist komplex sind, werden einzelne Stunden meist nicht ausreichen, diese darzulegen. Es ist daher geplant, die Stunden zu blocken. Hierbei können wir mit maximaler Flexibilität vorgehen.

Die theoretische Informatik ist ein äußerst umfangreiche. Wir werden auf die folgenden acht Themengebiete einen ersten Blick werfen. In der folgenden Beschreibung werden diese anhand typischer Fragestellungen vorgestellt. So soll es Ihnen ermöglicht werden abzuschätzen, ob dieser Kurs für Sie von Interesse ist. Die Antworten zu den Fragen finden Sie im letzten Kapitel.

3 Inhalt

3.1 Unendlichkeit, Abzählbarkeit und Überabzählbarkeit

Immer diese vielen Variablen in den Programmen. Das ist ganz schön nervig. Geht das nicht auch anders? Und dann ist noch das Ding mit der Unendlichkeit. Ist eine halbe Ewigkeit wirklich kürzer als die ganze? Und gibt es vielleicht auch mehr als unendlich?

3.2 Funktionen, Potenzmengen und Entscheidbarkeit

Es gibt ja bekanntlich keine dummen Fragen, sondern nur dumme Antworten. Aber muss es auf jede Frage eine Antwort geben? Vielleicht gibt es Dinge, die prinzipiell im Dunklen bleiben müssen.

3.3 Nummerierung, Wortfunktionen und Berechenbarkeit

Immer diese vielen Datentypen in den Programmen: Ganze Zahlen, Fließkommazahlen, Zeichen, Zeichenketten, Felder. Aber braucht man die wirklich? Vielleicht könnte man sich ja auf einen Datentyp beschränken.

3.4 Prädikatenlogik und logisches Programmieren

Es nervt schon gewaltig, dass man immer eine Lösung für ein Problem finden muss, bevor man mit dem Programmieren beginnen kann. Geht das nicht anders? Könnte man dem Programm nicht sagen, was man will und das Programm sucht sich die Lösung selbst?

3.5 Formale Sprachen und endliche Automaten

Apples Siri funktioniert inzwischen schon recht gut, aber programmieren könnte man so nicht! Da bräuchte man schon eine verbale Programmiersprache. Aber warum eigentlich? Was ist der Unterschied zwischen einer natürlichen und eine formalen Sprache und was hat das eigentlich mit Automaten zu tun?

3.6 Lambda-Kalkül und funktionales Programmieren

Könnte man mathematische Aufgaben nicht komplett anders lösen? Vielleicht durch Textersatz? Überall wo *1+1* steht schreibt man eine *2* etc. Aber wie weit kommt man so? Kann man auf diese Weise nur einfache Aufgaben lösen oder eventuell sogar alle? Und gibt es Programmiersprachen, die so funktionieren?

3.7 Berechnungsmodelle, Churchsche These, unentscheidbare Probleme

Computer, Automaten, Touring-Maschinen und dann auch noch das Lambda-Kalkül. Alle sollen was mit Berechenbarkeit zu tun haben. Aber was soll das heißen und wie hängt das alles zusammen? Wenn schon nicht alles berechnet werden kann, dann wären ein paar Beispiele nicht berechenbarer Probleme schön, oder ist alles so kompliziert, dass man sich eh nichts darunter vorstellen kann?

3.8 Komplexität, O-Notation, NP-Vollständigkeit, P-NP-Problem

Jetzt haben sie noch nicht einmal das P-NP-Problem gelöst, obwohl es dafür eine Million Dollar Preisgeld gibt. Doch worum geht es denn da eigentlich? Soll was mit Komplexität und NP-Vollständigkeit zu tun haben, aber wer versteht das schon.

4 Weiteres Vorgehen

Anfang Oktober werden wir uns zu einer kurzen Besprechung treffen, bei der auf Fragen zum Kurs noch einmal kurz eingegangen und das weitere Vorgehen fixiert werden soll. Im Anschluss haben Sie die Möglichkeit, sich für die Übung verbindlich anzumelden. Diese kann nur dann stattfinden, wenn es mindestens acht verbindliche Anmeldungen gibt.

5 Die Antworten zu Kapitel 3

5.1 Unendlichkeit, Abzählbarkeit und Überabzählbarkeit

Mit Hilfe der Cantorschen Paarungsfunktion werden wir zeigen, dass jedes Programm nur eine einzige Variable benötigt. Mit einer Bijektion weisen wir nach, dass eine halbe Ewigkeit genau so lange dauert wie die ganze. So gelingt uns auch der Nachweis, dass es unterschiedlich große Unendlichkeit geben muss.

5.2 Funktionen, Potenzmengen und Entscheidbarkeit

Wenn es auf eine Frage eine Antwort geben soll, dann sollte man diese auch berechnen können. Mit Hilfe mengenbasierter Funktionen kann man leicht zeigen, dass nicht alles berechnet werden kann. Manches ist daher prinzipiell unbeantwortbar oder unentscheidbar, wie der theoretische Informatiker sagen würde.

5.3 Nummerierung, Wortfunktionen und Berechenbarkeit

Mit Nummerierungen wird es uns möglich sein, beliebige abzählbare Mengen auf die natürlichen Zahlen abzubilden. Verschiedene Datentypen braucht man daher eigentlich nicht und Computer rechnen ja sowieso nur mit *0* und *1*. Problematisch wird es erst dann, wenn man mit überabzählbaren Mengen zu tun hat. $\mathbb{R}$ ist ein derartiger Vertreter. Angewandte Informatiker wenden hierauf meist die Vogel-Strauß-Politik an.

5.4 Prädikatenlogik und logisches Programmieren

Genau hierfür gibt es das logische Programmieren. Man definiert zunächst das Problem und stellt dann Fragen an das System. Dieses beantwortet diese, ohne sagen müssen, wie die Antwort gefunden werden kann. Natürlich unterscheiden sich logische Programmiersprachen wie Prolog stark von den üblichen imperativen Programmiersprachen wie C. Der Grundbaustein derartiger Programmiersprachen sind Prädikate, mit denen wir uns hier beschäftigen wollen.

5.5 Formale Sprachen und endliche Automaten

Programmiersprachen müssen exakt und eindeutig beschreiben, wie ein Computer einen Algorithmus auszuführen hat. Natürliche Sprachen sind zwar universell, bieten aber auch Raum für Interpretation. Daher sind natürliche Sprachen für das Programmieren ungeeignet. In dieser Einheit werden lernen, wie formale Sprachen wie Programmiersprachen aufgebaut sind und dass zwischen formalen Sprachen und endlichen Automaten ein enger Zusammenhang besteht.

5.6 Lambda-Kalkül und funktionales Programmieren

Genau diese Idee setzt das Lambda-Kalkül um. Obwohl es sich lediglich um Textersatz handelt, ist das Lambda-Kalkül touringmächtig und kann somit alles berechnen, was berechenbar ist. Eine praktische Umsetzung des Lambda-Kalküls findet sich in den funktionalen Programmiersprachen. Falls es die Zeit zulässt, werden wir uns kurz mit Haskell beschäftigen.

5.7 Berechnungsmodelle, Churchsche These, unentscheidbare Probleme

Die Churchsche These besagt grob, dass man mit Touring-Maschinen das Maximum der Berechenbarkeit erreicht hat. Es gibt nichts, was eine Touring-Maschine nicht berechnen könnte, außer es ist prinzipiell unberechenbar. Untermauert wird diese These dadurch, dass es viele verschiedene Berechnungsmodelle wie Register-Maschinen, das Lambda-Kalkül oder die partiell-rekursiven Funktionen gibt, die alle Touring-mächtig sind. Unentscheidbare Probleme sind in der Tat immer recht abstrakt, aber mit Hilfe des Satzes von Rice zeigen wir beispielsweise, dass es kein allgemeingültiges Verfahren geben kann festzustellen, ob ein Programm eine Endlosschleife enthält. Dies ist auch der Grund, warum bei syntaktischen Fehlern immer eine Fehlermeldung ausgegeben wird, bei semantischen Fehlern, wenn überhaupt, meist nur Warnungen.

5.8 Komplexität, O-Notation, NP-Vollständigkeit, P-NP-Problem

Komplexität beschäftigt sich mit der Laufzeit von Programmen. Diese wird in der O-Notation angegeben, auf die wir intensiv eingehen werden. Als gerade noch praktisch lösbar, werden Probleme mit polynomialer Laufzeit, also P-vollständige angesehen. NP-vollständige Probleme benötigen exponentielle Laufzeit und gelten als praktisch unlösbar. Das nutzt man zum Beispiel bei Verschlüsselungsverfahren aus. Allerdings konnte noch keiner zeigen, dass P und NP wirklich verschieden sind. Vielleicht hatte ja nur noch keiner eine gute Idee NP-vollständige Probleme mit polynomialen Aufwand zu lösen. Dann wären alle NP-vollständigen Probleme auf einmal lösbar und die meisten Verschlüsselungsverfahren hätten ein echtes Problem.