Prüfungsprotokoll

Prüfung:	Vordiplom Informatik Teilprüfung Technische & Theoretische Informatik
Prüfer:	Prof. Seeger
Termin:	16. Oktober 2001, 11:15 Uhr
Note:	1,0

Die Prüfung:

Da Sie sich heute so schlecht fühlen (mir ging's miserabel, ich fühlte mich hundeelend) überlasse ich Ihnen die Wahl, womit Sie anfangen möchten.
Nun, da ich mich vorhin mit dem Beisitzer über das Halteproblem auf dem Parkplatz unterhalten habe, denke ich, könnten wir gleich damit weitermachen. [Lacht] (Ich erklärte das allgemeine Halteproblem und das es unlösbar ist. Dann bewies ich die Aussage indem ich zuerst die Unlösbarkeit des speziellen Halteproblems durch einen Widerspruchsbeweis zeigte und dann die Kontraposition anwendete.) Ich habe den Begriff Gödelnummer eben einfach so verwendet, soll ich ihn erklären?
Nö nö, welche Sprachen sind denn entscheidbar?
(Ich brachte die Chomsky-Hierarchy samt zugehörigen Chomsky-Grammatiken und zugehörigen Automaten. Ich erklärte Entscheidbarkeit und Semi-Entscheidbarkeit, das Wortproblem, expandierende Grammatik, etc. Es stellte sich heraus, das Prof. Gumm, bei dem ich die Theoretische Informatik gehört hatte, einige Begriffe anders eingeführt hat. Prof. Seeger war bei einigen meiner Begriffs-Definitionen zuerst irritiert. Als ich die Begriffe aber anders erklärte bzw. äquivalente Formulierungen benutzte, wurde alles klar.)
Welche Grammatik liegt normalerweise Compilern zugrunde?
Eigentlich beruhen alle Parser, die ich kenne, auf kontextfreien Sprachen. Das reicht, um die syntaktische Korrektheit eines Programms zu verifizieren. Wenn man aber noch so Dinge feststellen möchte, ob z.B. die Variablen vor dem ersten Benutzen schon initialisiert wurden, dann braucht man eine kontextsensitive Grammatik. Ein Parser zerlegt ein Wort in seine grammatikalischen Bestandteile und versucht, einen Herleitungsbaum zu erzeugen. yacc z.B. erzeugt aus einer kontextfreien Grammatik einen Parser, genauer: einen $LALR$-Parser. Es gibt verschiedene Arten von Parsern: top-down und bottom-up Parser. Die top-down Parser sind LL-Parser, d.h. sie lesen den Input von links nach rechts und gehen die grammatikalischen Regeln von links nach rechts durch. Sie betrachten also zuerst die linke Seite, dann die rechte Seite einer Regel. Wenn man sich den Herleitungsbaum anschaut [skizziere einen Baum], erkennt man, daß dieses Vorgehen den Herleitungsbaum von oben nach unten aufbaut. Bottom-up Parser lesen dagegen solange Zeichen aus dem Input - d.h. sie "shiften", bis sie die rechte Seite einer Regel erkennen, die sie dann durch die linke Seite ersetzen - also "reducen". Diese LR-Parser nennt man auch Shift-Reduce-Parser. Es kann bei ihnen z.B. zu Shift-Reduce- oder Reduce-Reduce-Konflikten kommen, und zwar immer dann, wenn der Parser nicht weiß,

was er in einer Situation tun soll. Dies kann man aber mittels lookahead und anderen Tricks verhindern...
Wissen Sie, wieviel lookahead bei einer Sprache wie Pascal nötig ist?
Ich würde sagen, eins müßte reichen.
[Überlegt] Hmm, ja, bei den älteren Pascal-Versionen trifft dies sicher zu... Ok, was könnten wir denn noch fragen... Berechenbarkeit!
Man kann jede noch so komplizierte Computeranwendung als eine Funktion $\varphi\colon \mathbb{N}^k \to \mathbb{N}$ auffassen. Dies geht aufgrund der Church'schen These (man kann danach jeden Algorithmus textuell formulieren und daraus die Gödelnummer des Algorithmus gewinnen). Diese Funktion ist genau dann berechenbar, wenn z.B. eine Turing-Maschine bei Ausführung der Funktion hält und auf dem Band dann nur noch $\varphi(x_1, \ldots, x_k) + 1$ viele Einsen nebeneinander stehen.
Wenn Sie an die Praktische Informatik denken, wie kann man den Begriff Berechenbarkeit noch formulieren?
Sie meinen die WHILE-Sprachen. Nun, die Church'sche These besagt doch, daß der Berechenbarkeitsbegriff unabhängig vom zugrunde liegenden Algorithmenkonzept ist. Es ist egal, ob man einen Algorithmus als Flußdiagramm, C-Programm, in der WHILE-Sprache oder auf einer Turing-Maschine betrachtet. Ist eine Funktion in einem Algorithmenkonzept berechenbar, dann auch in allen anderen. Soll ich Ihnen die WHILE-Sprachen näher erläutern?
Nein, ... Sie haben eben "in allen anderen" gesagt. Was ist mit den LOOP-Sprachen?
Die hatten wir beim Herrn Gumm nicht. Was ist mit denen?
[Erklärt mir, daß die nicht zu den äquivalenten Algorithmenkonzepten gehören] Aber wenn Sie das nicht hatten... (Wir redeten dann noch über Aufzählbarkeit und über die Überabzählbarkeit der nicht berechenbaren Funktionen. Irgendwann meinte Prof. Seeger:) Das ist ja das Schlimme! Von den nicht berechenbaren Funktionen gibt es überabzählbar viele. Die Menge der berechenbaren Funktionen ist dagegen verschwindend gering!
Naja, aber von denen gibt's trotzdem noch unendlich viele... [Beisitzer grinst]
Wissen Sie etwas zum Thema Komplexität?
Sicher! Man muß unterscheiden zwischen der Algorithmen-Komplexität und der Problem-Komplexität. Die Algorithmen-Komplexität sagt nichts darüber aus, ob es nicht vielleicht einen effizienteren Algorithmus für die Lösung eines Problems gibt. Wenn ein Problem aber in einer bestimmten Problem-Klasse liegt, dann kann kein Algorithmus zur Lösung des Problems effizienter sein als die Komplexität der Problem-Klasse. Wir hatten mehrere Komplexitätsklassen: P, NP und NPV, wobei NPV eigentlich eine besondere Untermenge von NP ist. In P liegen alle Probleme, die sich mit einem deterministischen Automaten oder Algorithmus in polynomialer Zeit lösen lassen. In NP liegen die Probleme, die sich mit einem nichtdeterministischen Algorithmus in polynomialer Zeit lösen lassen. Dabei heißt hier "nichtdeterministisch", daß der Algorithmus eine Art Delphi-Fähigkeit besitzt. Betrachten wir z.B. das Traveling Salesman Problem: Wenn ein Algorithmus in jedem Schritt und bei jeder Verzweigung "wüßte", welchen Weg er einschlagen müßte, um auf dem kürzesten Weg zur Lösung zu gelangen, dann würde der Algorithmus in polynomialer Zeit das Problem lösen können. Es gilt trivialerweise $P \subset$ NP. Um die Frage NP $\subset P$ zu beantworten, führt man eine weitere Klasse ein, die der NP-vollständigen

Probleme. Diese Probleme sind sozusagen die schwierigsten Probleme aus *NP*. Für sie gilt: Wenn es eine Lösung für ein *NPV*-Problem in polynomialer Zeit gäbe, dann würde dies auch für alle Probleme aus *NP* gelten. Dann wäre tatsächlich $P = \mathrm{NP}$.
Wie kann man zeigen, daß ein Problem NP*-vollständig ist?*
Man zeigt mit Hilfe eines polynomiellen Verifizierers, daß das Problem in *NP* liegt. Dann nimmt man ein bereits bekanntes Problem aus *NPV* her und zeigt, daß man das betrachtete Problem auf das Problem aus *NPV* in polynomialer Zeit reduzieren kann. Hierbei bedeutet "reduzieren", daß man eine Lösung des *NPV*-Problems in polynomialer Zeit in eine Lösung für das Problem aus *NP* transformieren kann.
Gut, kommen wir nun zur Technischen Informatik. Da gibt es gewisse "Grundelemente", mit denen man Schaltungen erstellen kann. Was braucht man dazu?
Mit den grundlegenden Schaltgliedern *AND*, *OR* und *NOT* kann man jede boolsche Schaltung realisieren.
Geht das nicht auch anders?
Klar! Es reichen auch schon alleine *NAND*- oder *NOR*-Glieder aus.
Wieso reichen die aus? Wie kann man z.B. ein NOT *mit ihnen realisieren?*
Ganz einfach, soll ich *NAND* oder *NOR* benutzen? [Zeichne die Operationstafel für *AND* auf und negiere die Einträge] So, daß ist jetzt die Tafel für das *NAND*. Wie man sieht, ergibt eine der Zeilen genau die Operationstafel für das *NOT*. Man muß also lediglich die eine Leitung auf 1 setzen und die andere wird dann negiert.
Sie haben das jetzt so gezeigt. Geht das auch anders?
[Oh, oh, ich weiß, daß man das auch direkt herleiten kann. Ich bin aber viel zu müde für solche geistigen Kapriolen, daher rechtfertige ich meine jetzige Erklärung:] Es gilt ja, daß zwei Schaltterme genau dann äquivalent sind, wenn ihre Schaltfunktionen gleich sind. Somit kann man die Gleichheit zweier Terme über eine Wahrheitstabelle beweisen. Also ist das, was ich hingeschrieben habe, ein gültiger Beweis.
[Läßt nicht locker] Überlegen Sie mal, was ist denn das Besondere an NAND *und* NOR-*Gliedern?*
[Ich hole aus:] In einem Distributiven Verband mit 0 und 1 gibt es einen Haufen Regeln: Idempotenz, Absorbtion, Assoziativität, Kommutativität, Distributivität und diese 0/1-Regeln. Es fehlt da aber noch die Negation. Wenn man den Distributiven Verband um die deMorganschen Regeln und einige Komplementärregeln erweitert, erhält man die Boolsche Algebra.
Ahh, deMorgan - darauf wollte ich hinaus.
[Ich schreibe den Term für ein *NAND*-Glied auf und wende deMorgan an:] $(x * y)' = x' * y'$
Gut. In der Technischen Informatik gibt es ja ein Problem, welches NP*-vollständig ist.*
Ja, das Problem, einen Schaltterm in eine äquivalente aber irredundante Summe von Primimplikanten zu bringen ist *NP*-vollständig. (Wir redeten noch etwas darüber.)
Nun was ganz anderes. Es gibt auch eine andere Art von Schaltungen, die bei Speichern von Bedeutung ist.
Flip-Flops, das sind rückgekoppelte Schaltungen.
Ja, und wie funktionieren die genau?
Das Besondere an ihnen ist, daß in ihre Eingabe ihr eigener letzter Ausgabewert miteinfließt.

Wie kann man aus Flip-Flops Speicherzellen machen?
Man braucht dazu außer den Flip-Flops noch etwas Extra-Hardware. Z.B. ist zur Adressierung ein Decoder notwendig, den man vor die Flip-Flops schaltet.
Wie sieht so ein Decoder aus?
(Hier war ich etwas verwirrt. Ich erzählte was von PLA's und deren zweistufigen Aufbau, Input und Output... dann Themawechsel:)
Wir hatten in der Vorlesung ein Modell von einer CPU behandelt, den Mikrocode-Simulator.
[Fange an zu zeichnen: zuerst ALU, dann X-,Y-,Z-Register und -Busse, dann weiter drumherum die Universalregister, RAM, etc.] Die ALU ist ein besonderer Baustein. Je nachdem, was für Werte an den Steuerleitungen anliegen, kann die ALU verschiedene Funktionen erfüllen. Wie der Name "Arithmetisch-Logische-Einheit" schon sagt, kann die ALU ihre Eingabewerte - hier das X- und das Y-Register - z.B. addieren oder mit *AND* verknüpfen. Das Ergebnis liegt dann im Z-Register. [Ich baue schrittweise die CPU auf und erläutere jeweils die Bedeutung - bis Prof. Seeger anfängt zu stöhnen...] Habe ich etwas falsch gemacht?
[Grinst] Nein, ganz und gar nicht. Mich freut nur, daß das alles so von alleine geht.
(Ich erläuterte weiterhin den Aufbau eines Mikrobefehls und die Bedeutung seiner Bits während der drei verschiedenen Prozessor-Phasen. Prof. Seeger wollte dann noch RAM und ROM sehen. Ich habe die Zeichnung dahingehend vervollständigt.)
Was ist der Grund dafür, daß diese Architektur niemals wirklich umgesetzt wurde? Intel hat mal zwei solcher Prozessoren tatsächlich gebaut, die waren aber nicht erfolgreich. Warum?
(Muß überlegen...) Naja, es fehlt z.B. eine Pipeline.
Richtig, und was noch?
(Keine Idee mehr. Prof. Seeger wollte auf den LIE-Zyklus hinaus, der die Ausführung ausbremst. Ich erklärte den LIE-Zyklus und wir sprachen dann noch über RAM und ROM bei PC's und über deren Zugriffsgeschwindigkeiten. Irgendwann kamen wir auf die Vorzüge der RISC-Architektur zu sprechen.)
Was für Vorteile hat die RISC-Architektur?
Das Pipelining wird durch die RISC-Eigenschaften begünstigt: So erlaubt das Load/Store-Prinzip die Bearbeitung von Daten ausschließlich in den Registern. Und die RISC-Befehle haben eine einheitliche Länge. Bei Intel-Prozessoren beispielsweise sind die Befehle nicht von gleicher Länge und es ist z.B. beim Additionsbefehl eine Speicheradresse als Operand erlaubt. Das bedeutet, daß bei der Abarbeitung eines Befehls manchmal ein Operand nachgeladen werden muß. Dies bremst die Abarbeitung in einer Pipeline aus.
Gut, dann machen wir jetzt Schluß.

Zur Prüfungsatmosphäre:

Wie oben schon angemerkt, ging es mir während der Prüfung wirklich miserabel. Es fiel mir schwer, mich zu konzentrieren. Ich mußte mich regelrecht zwingen, Prof. Seeger bei seinen Fragen zuzuhören. Meine Antworten redete ich teilweise wie im Halbschlaf vor mich hin, während ich mit den Händen meinen Kopf auf dem Tisch abstützte.

Der Prüfungsstress legte sich noch vor der ersten Frage, da Prof. Seeger etwas rumscherzte. Zwar wollte ich zwischenzeitlich aufgeben, aber dies lag keinesfalls am Prüfer oder an den Fragen. Prof. Seeger war sehr freundlich und sehr fair. Nach außen hin müßte die Prüfung wie ein entspanntes Gespräch gewirkt haben. Wenn man mal von meinem gesundheitlichen Zustand absieht, war es eine wirklich schöne Prüfung. Prof. Seeger ist ein äußerst angenehmer Prüfer.