Vom Gatter zum Compiler: Im Unterricht durch sieben Abtraktionsebenen

Urs Lautebach[1]

Abstract: Vorgestellt wird ein Unterrichtsgang, der die Funktionsweise von Computern über viele Abstraktionsebenen hinweg erschließt und erlebbar macht. Die Schüler erstellen dabei auf jeder Ebene Soft- oder Hardware selber. Ausgehend von logischen Verknüpfungen bauen die Schüler (am Simulator oder mit Steckkästen) zunächst Addierwerke und andere ausgewählte Teile, soweit sie auch im Prozessorsimulator "Mikrosim" sichtbar sind. Dort werden dann Datenflusssteuerung, Neumann-Zyklus und Mikro- sowie Assemblerprogrammierung behandelt. Weil aus Schülersicht erst die Hochsprache wieder vertrautes Terrain darstellt, entsteht schließlich ein Compiler zumindest für die Sprache der arithmetischen Ausdrücke: Die Schüler simulieren dessen Syntaxanalyse zunächst "unplugged" mit einem Kartenspiel. Daran angelehnt programmieren sie den Compiler, der beliebige Ausdrücke in Assemblerprogramme übersetzt: Nach dem Assemblieren sind diese Programme im Simulator lauffähig. Der Prozessor berechnet dann vor den Augen der Schüler den Wert des Ausdrucks. Der Reiz des Unterrichtsganges ist der transparente und für die Schüler erlebbare Durchstich sozusagen vom Gatter zum Compiler.

Keywords: Logische Schaltungen, Rechenwerk, Mikroprozessor, Mikrosim, Mikrocode, Assembler, Compiler, Unterrichtseinheit, Abstraktionsebenen

1 Einleitung

Die Frage "Ja, aber wie FUNKTIONIERT denn nun ein Computer?" wird in der Schule meist nur punktuell behandelt: Der Unterricht schafft sozusagen einzelne "Inseln des Verstehens", etwa logische Schaltungen, Mikroprogrammierung oder Hochsprachen.

Der hier vorgestellte Weg hingegen baut auch zwischen diesen Inseln so viele Brücken wie möglich: Das Ziel ist ein Durchstich durch viele Abstraktionsebenen, eben "vom Gatter zum Compiler". Dieser Durchstich ist in der Breite immer unvollständig, in der Tiefe aber durchgängig, weil die Schüler der allmählich zunehmenden Abstraktion bewusst folgen können. Sie haben dabei auf jeder Ebene Gelegenheit, Komponenten selbst zu bauen und auf der nächsthöheren Ebene auch weiter zu verwenden.

Alle Elemente des Unterrichtsganges können natürlich auch einzeln unterrichtet werden. Besonders reizvoll wird der Unterrichtsgang im Zusammenhang, wenn es gelingt, einen Bogen über alle Abstraktionsebenen zu schlagen. Aber auch für sich allein sind die Elemente durchaus als allgemeinbildend anzusehen, weil alle dazu beitragen, die Arbeits-

[1] Faust-Gymnasium Staufen, Krichelnweg 1, 79219 Staufen, urs.lautebach@fgst.de
OpenPGP-Schlüsselfingerprint: 5F95 477B 22A8 2D9D 66DA 3D0A 6E23 BC3B 8CCE ACB1

weise des Computers zu entmystifizieren.

Die unterrichtliche Wiederverwendung wird auch dadurch erleichtert, dass die meisten Elemente nicht an bestimmte Werkzeuge gebunden sind. Lediglich das eigens erstellte Assemblerwerkzeug ist gezielt auf den Simulator MIKROSIM abgestimmt.

Darüber hinaus soll auch angeregt werden, die Lernenden im Informatikunterricht (mit dem genannten „Durchstich“ vor Augen) Zusammenhänge über viele Abstraktionsebenen hinweg herzustellen zu lassen. Denn ein Schlüsselmerkmal der Informatik ist ja, dass sie auf vielen solcher Ebenen „stattfindet“. Darum sollte die Fähigkeit, die jeweils passende Ebene gezielt auszuwählen, eines der Ziele jeder Informatikausbildung sein. Der geschilderte Unterrichtsgang lässt die Lernenden verschiedene Ebenen bewusst kennenlernen und gedanklich miteinander verbinden.

2 Unterrichtsgang

2.1 Logische Gatter

Anknüpfend an den aus einer Programmiersprache bekannten UND-Operator beginnt der Unterrichtsgang bei logischen Verknüpfungen, Gattern und den entsprechenden Wertetabellen. Dafür kommen der Simulator Logicsim von www.tetzl.de oder (falls verfügbar) Logikstecksysteme zum Einsatz.

Nach Einführung des Binärsystems bauen die Schüler mindestens Halb-, Volladdierer und Addierwerke; als Leitgedanke bietet sich „Rechnen mit Strom“ an (Gallenbacher, „Abenteuer Informatik“ [Ga12]). Je nach Zeitbudget entstehen auch noch andere Bauteile, die den Schülern in der nächsten Phase als fertige Bausteine innerhalb von Mikrosim wieder begegnen. Das können etwa Flipflops, Subtrahierer, Multiplexer oder umschaltbare Rechenwerke sein. Für die Arbeit mit Mikrosim muss das Hexadezimalsystem beherrscht werden; meistens ist auch eine Einführung des Zweierkomplements sinnvoll.

2.2 Prozessormodell

Diese nächste Phase stützt sich wesentlich auf „Mikrosim“, einen Simulator für einen einfachen von-Neumann-Prozessor mit 8-Bit Wort- und Adressbreite (Konrad Dammeier, Pflege mittlerweile Thomas Schaller, siehe [DS02]). Er besteht für die Schüler sichtbar aus Registern, Bussen, Steuer- und Rechenwerk, was eine nahtlose Anbindung an das zuvor Behandelte gewährleistet. Die Schüler lernen hier erst die Datenflusssteuerung innerhalb eines Prozessors kennen und automatisieren sie dann über den Mikroprogrammspeicher.

Für eine Auswahl des Maschinenbefehlssatzes bauen sie schließlich Mikroprogramme

für einfache (MOV AX, BX) und komplexere Assemblerbefehle (ADD [BX+nn]) bis hin zu bedingten Sprüngen (JPZ) und Stackbefehlen (z.B. PUSH AX, POP AX). Dieser Teil lehnt sich stark an entsprechende Materialien von M. Makowsky, K. Dammeier und anderen an.

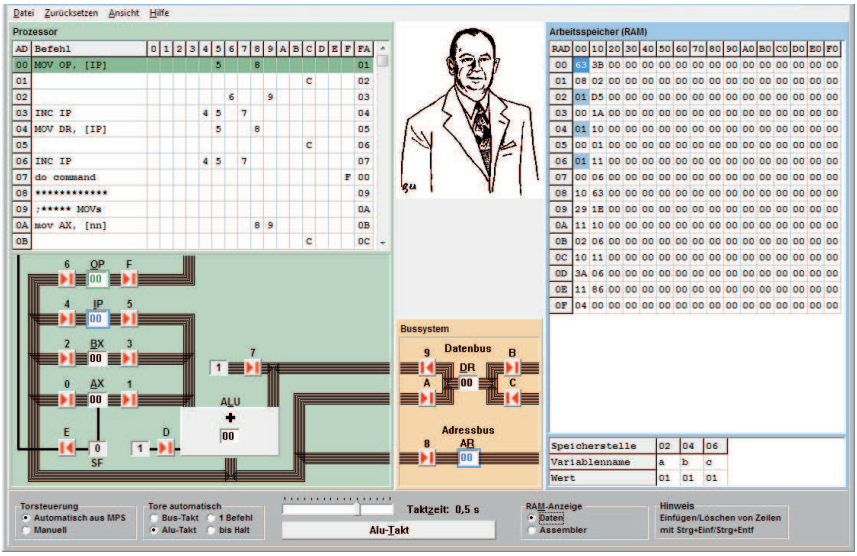

Abb. 1: Simulator „Mikrosim“. Links (grün) der eigentliche Prozessor mit Mikroprogrammspeicher und darunter Registern, Bussen, Toren und Rechenwerk (ALU). Rechts 256 Byte RAM. Der Simulator kann bis auf Ebene einzelner Tore von Hand gesteuert werden, aber auch Takte, Befehle oder ganze Programme in wählbarer Geschwindigkeit automatisch abarbeiten.

Darüber hinaus werden spezielle Stackarithmetikbefehle wie ADDSTACK („entferne die beiden obersten Stackeinträge und lege ihre Summe wieder auf den Stack“) eingeführt, um später Postfixausdrücke elegant auswerten zu können.

Auch in dieser Phase werden nur so viele Mikroprogramme erstellt, dass den Schülern klar wird, wie der Prozessor zu seinem Assemblerbefehlssatz kommt. Den vollständigen (und weitgehend fehlerfreien) Befehlssatz für die nächste Phase bekommen die Schüler gestellt. Daran lernen sie Grundzüge der Assemblerprogrammierung.

2.3 Assembler

Die Doppelverwendung des Wortes „Programm“ in Mikroprogramm(-speicher) und Assemblerprogramm ist für Schüler verwirrend, die Abgrenzung dementsprechend schwierig. Hier hilft die Klarstellung, dass der linke Teil des Prozessors „von Intel-Ingenieure entworfen“ und programmiert wird, die ihn dann fertig eingegossen und unveränderbar verkaufen; das RAM rechts hingegen ist auch für Endkunden zugänglich und veränder-, also programmierbar. An dieser Stelle wechselt der Schüler also aus der

Ingenieurs- in die Kundenrolle.

Mikrosim bietet für die Assemblerprogrammierung ein elegant integriertes Werkzeug, das RAM-Inhalte automatisch als Maschinencodes interpretiert und sofort die entsprechenden Mnemonics darstellt. Es ist unkompliziert, eignet sich aber nur für sehr kurze Programme bis etwa 5 Zeilen. Der Autor hat daher ein eigenes erstellt, das im Assemblerquelltext auch Sprunglabel, Konstanten und Kommentare verarbeiten kann; außerdem kommt es damit zurecht, dass Assemblerbefehle bei verschiedenen Schülern an unterschiedlichen Mikroadressen stehen und deswegen auch unterschiedliche Opcodes haben können.

Die Schüler sollen nach dieser Phase gerade so viel Assembler beherrschen, dass sie später Assemblerprogramme vom Compiler auch automatisch erzeugen lassen können. Dafür reicht es, sehr einfache Programme wie Zuweisungen, Entscheidungen und einfache Arithmetik von Hand assemblieren zu können:

```
if(a < 3)                          MOV AX, [var_a:]
{                                  SUB 3
    b = 5;                         JPS else:
}                                  MOV AX, 5
else                               MOV [var_b:], AX
{                                  JP ende:
    c = c + 3;                 else: MOV AX, 3
}                                  ADD [var_c:]
                                   MOV [var_c:], AX
                               ende: ...
```

Abb. 2: Einfaches Java-Programm, Übersetzung in Assembler

Stärkere Schüler bekommen etwa Schleifen als Zusatzaufgabe, z.B. für eine Multiplikationsroutine. Unabhängig davon und mit Blick auf den Leitgedanken „Wechsel der Abstraktionsebenen“ sollen die Schüler beim Programmieren in Assembler aber auch spüren, dass komfortable Hochsprachen durchaus keine Selbstverständlichkeit sind.

Bis hierher hat der Unterricht zwar vorgeführt, wie man aus Gattern einen in Assembler programmierbaren Rechner baut; aber erst der Brückenschlag zur Hochsprache führt die Schüler zurück auf vertrautes Terrain und vervollständigt damit den Durchstich.

2.4 Compiler

Aus didaktischer Sicht reicht für den Compiler die Sprache der arithmetischen Ausdrücke (also „7+2“ oder „5*(6-(8-5))“): Sie ist allen vertraut, die zugehörige Grammatik ist mit nur 11 Regeln noch übersichtlich. Andererseits ist diese Sprache aber wegen der

Rekursion immerhin kontextfrei. Das ist auch ein wichtiges Merkmal realer Programmiersprachen, das die Konstruktion von Compilern maßgeblich beeinflusst.

Die Schüler wandeln solche Ausdrücke zunächst von Hand in Postfix-Schreibweise um (für die obigen Beispiele also „7, 2, +“ bzw. „5, 6, 8, 5, -, -, *“) und lernen deren Vorteile kennen: Erstens ist sie auch ohne Klammern immer eindeutig, zweitens kann der Postfixausdruck direkt als Anweisungsfolge für eine Stackmaschine gelesen werden. Mit den genannten Assemblerbefehlen zur Stackarithmetik steht eine solche Stackmaschine den Schülern bereits zur Verfügung.

Der letzte Schritt ist dann natürlich die Automatisierung der Infix-Postfix-Übersetzung. Mittlerweile händigt der Autor den Schülern die dafür fertig vorbereitete Grammatik aus: Sie erlaubt eine Analyse durch rekursiven Abstieg mit 1-Token-Vorausschau („lookahead“).

```
(1) expr       → term restExpr EOF
(2) term       → faktor restTerm
(3) restExpr   → +  term restExpr
(4)            | -  termrestExpr
(5)            | empty
(6) restTerm   → *  faktor restTerm
(7)            | /  faktor restTerm
(8)            | empty
(9) faktor     → ZAHL
(10)           | VAR
(11)           | ( expr )
```

Abb. 3: Grammatik für arithmetische Ausdrücke (nach [Gu93])

Der resultierende Parser ist ein LL(1)- oder recursive-descent-Parser. Die leider überhaupt nicht intuitive Struktur der Grammatik gewährleistet, dass Operatoren mit ihren Präzedenzen und der gewünschten Linksassoziativität richtig interpretiert werden. Man kann das kurz kommentieren, die Konstruktion der Grammatik aber weglassen. Es hat sich nicht bewährt, Linksrekursion und Eindeutigkeit an dieser Stelle auch noch zu behandeln.

Den Ablauf der Analyse („parsing“) erleben die Schüler zunächst anhand eines Kartenspiels, in dem jede der 11 Produktionen in Form von Karten vorkommt. Mit den folgenden „Spielregeln“ ergibt sich die Funktionsweise eines recursive-descent-Parsers:

- Man schaut in der Eingabe nur ein Terminal voraus, der Rest bleibt abgedeckt;
- man legt Produktionen in Form von Papierkarten auf einen Stapel;
- man markiert auf der obersten Karte jeweils den Stand ihrer Abarbeitung;
- fertig abgearbeitete Karten wirft man weg und arbeitet auf der Karte weiter, die darunter sichtbar wird, und zwar an der vorher markierten Stelle;

- man streicht „verbrauchte“ Terminale aus der Eingabe heraus.

Auf diese Weise wird die Eingabe „unplugged“ auf Gültigkeit geprüft („geparst“). Hier hat es sich bewährt, auch ungültige Ausdrücke wie „4*)5-7“ parsen zu lassen, damit klar wird, dass und wann solche Fehler bei der Analyse auffallen.

Im nächsten Schritt wird die Grammatik so erweitert, dass sie in den richtigen Momenten auch Ausgaben tätigt. Damit können die Schüler dann schon auf Papier einen „Unplugged-Compiler“ durchspielen, der Infix- in Postfixausdrücke umwandelt. Schon nach wenigen Durchläufen wird klar, dass der Vorgang tatsächlich weder weit reichende Vorausschau, noch Kreativität oder Intuition verlangt – dass er also ohne weiteres automatisierbar ist.

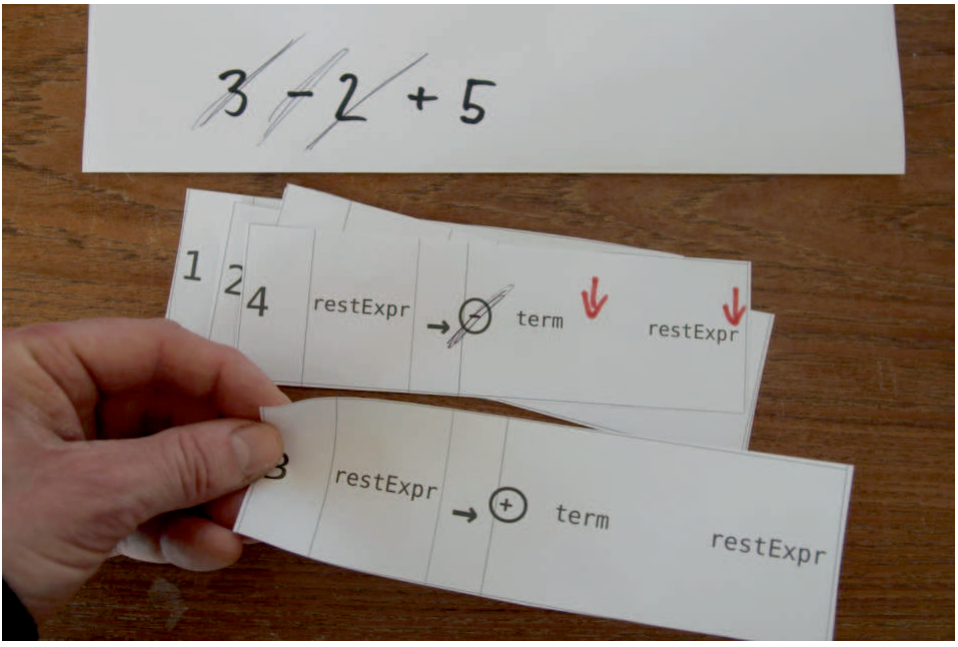

Abb. 4: Kartenspiel für den „Unplugged-Compiler“. Oben die Eingabe, aus der drei Zeichen bereits verbraucht sind. In der Mitte der Kartenstapel; die neue Karte wird gerade aufgelegt.

An diesem Punkt, bekommt die Lerngruppe den Compiler als vorbereitetes Java-Projekt. Darin sind unter anderem Datei-Ein-/Ausgabe, lexikalische Analyse und eine rudimentäre Fehlerbehandlung schon fertig programmiert; die Schüler ergänzen „nur noch“ den Kern des Compilers, also die eigentlichen Grammatikregeln mit Ausgaben (nach Assembler) exakt so, wie sie das zuvor am Kartenspiel geübt haben.

Der fertige Compiler erzeugt dann für jeden Ausdruck ein Programm in der Assemblersprache, die die Schüler zuvor am Simulator selbst gebaut haben. Nach dem Assemblieren ist es auf diesem Prozessor auch tatsächlich und beobachtbar lauffähig.

Die Einheit ist an dieser Stelle eigentlich zu Ende, das Ziel erreicht: Die Schüler haben vom AND-Gatter bis zum Compiler auf jeder Abstraktionsebene selbst Hand angelegt, nach und nach einen (im Simulator) funktionsfähigen Computer aufgebaut und sogar eine Komponente eines Betriebssystems selbst programmiert. Die Übersetzung funktioniert, man kann den übersetzten Programmen beim Ablaufen zuschauen und sie liefern das korrekte Ergebnis. In einer für die Schüler transparenten Art Weise baut all das letztlich auf die Gatter, mit denen die Einheit begonnen hat; auf dem Weg von dort bis zum Compiler erleben sie jede Menge raffinierte Informatik.

2.5 Zusammenfassung

Mit einem Umfang von gut 20 Doppelstunden ist die Einheit für Schüler aber doch schwer zu überblicken. Um die angestrebte Transparenz trotzdem zu gewährleisten, lässt der Autor sie an diesem Punkt eine Zusammenfassung erstellen, die möglichst viele Zusammenhänge innerhalb des ganzen Unterrichtsverlaufs wiedergeben soll. Einige Leitfragen helfen den Schülern bei der Strukturierung:

- Welche Bauteile setzen sich aus welchen anderen zusammen?
- Wo ist Mathematik drin, wo Informatik?
- Welche Softwarewerkzeuge sind beteiligt? Was verarbeiten sie, was liefern sie und an wen?
- Wo ist eigentlich die Grenze zwischen Soft- und Hardware?
- Wo ist menschliche Kreativität im Spiel, was kann automatisch ablaufen?

Die Schüler wählen die Form dieser Zusammenfassung selbst. Sie wurde schon als Mindmap, Flowchart, Prezi, Landkarte und mehrfach auch als handgezeichneter Comic gestaltet. Sichtung der Heftaufschriebe, Auswahl der Inhalte und Konzeption der Darstellung nehmen nochmals Unterrichtszeit in Anspruch, die hier aber gut investiert ist.

Zumindest im Nachhinein gelingt damit fast allen Schülern der gedankliche Brückenschlag über die behandelten Ebenen hinweg; er kommt sonst allenfalls im Informatikstudium zustande und dort auch nicht in dieser gebündelten Form.

2.6 Mögliche Erweiterungen

Der gesamte Unterrichtsgang wurde im vierstündigen Kurs der gymnasialen Oberstufe bisher fünf Mal gehalten und im Lauf der Zeit auf die Teile reduziert, die für den angestrebten Durchstich wesentlich sind. Natürlich gibt es viele denkbare und reizvolle Erweiterungen. So kann man die vom Compiler übersetzte Sprache auch noch schrittweise ausbauen: Statt eines Ausdrucks übersetzt er dann eine Zuweisung

```
variable = ausdruck;
```

und als nächstes eine Kette von Zuweisungen. Leistungsstarke Schüler dürfen dem Compiler eventuell noch boole'sche Ausdrücke wie

```
ausdruck < ausdruck
```

und dann Kontrollstrukturen wie

```
while(boolescherausdruck)
{
    zuweisungskette
}
```

beibringen. In dieser Bonusaufgabe spielen Theorie und Praxis auf lehrreiche Weise zusammen, weil man für jede Änderung der Grammatik die neuen first- und follow-Mengen der einzelnen Regeln ermitteln muss, um die Entscheidungen des Parsers sauber programmieren zu können (für arithmetische Ausdrücke bekamen die Schüler diese Mengen fertig vorgegeben).

Weil das mühsam ist und auch erkennbar unkreativ, also eine Tätigkeit, die man automatisieren sollte, hat der Autor in der Vergangenheit noch einen Parsergenerator vorgeführt, der genau das tut: Er transformiert eine maschinenlesbar notierte Grammatik in einen RD-Parser. Es ist gar nicht schwer, damit den Umfang der übersetzten Sprache nochmal deutlich zu erweitern, etwa um Variablendeklarationen, diverse Kontrollstrukturen und andere Sprachmittel. Das fasziniert leistungsstarke Schüler, ist aber für das übergeordnete Lernziel nicht nötig.

3 Materialien

Die verwendeten Softwarewerkzeuge, Übersichten, Beispiele, Arbeitsblätter und Musterlösungen stellt der Autor auf Anfrage per E-Mail zur Verfügung. Rückmeldungen, Vorschläge und Kritik sind willkommen.

Literaturverzeichnis

[DS02] Dammeier, K.; Schaller, T.: https://www.informatik.schule-bw.de/index.php?id=55&aid=55&mid=47.

[Ga12] Gallenbacher, J.: Abenteuer Informatik – IT zum Anfassen von Routenplaner bis Online-Banking, Spektrum, 2012.

[Gu93] Gumm, H.-P.: Vorlesungsskript Informatik IV, 1993.