Robert Nowak

Mathematica Softwaretool Entwicklung
für die Datenkonversion von GDSII Layoutdaten

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fakultät Informationstechnik & Elektrotechnik

Wien - Mittweida, 2009

Robert Nowak

Mathematica Softwaretool Entwicklung für die Datenkonversion von GDSII Layoutdaten

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fakultät Informationstechnik & Elektrotechnik

Wien - Mittweida, 2009

Erstprüfer: Prof. Dr.-Ing. Werner Günther

Zweitprüfer: Dipl.-Ing. Stefan Eder-Kapl

Vorgelegte Arbeit wurde verteidigt am:

Nowak, Robert:
Mathematica Softwaretool Entwicklung für die Datenkonversion von GDSII Layoutdaten,
Mittweida, Hochschule Mittweida, Fakultät Informationstechnik & Elektrotechnik,
Diplomarbeit, 2009

Ziel der Diplomarbeit ist es, eine Sammlung von Mathematica Softwaretools für die Datenkonversion von GDSII Layoutdaten zu implementieren.GDSII ist ein Standard Layout Format der Halbleiterindustrie. Mathematica ist ein hoch entwickeltes Softwaresystem welches die Entwicklung nahezu beliebiger Softwareanwendungen ermöglicht. Die Arbeit erläutert zunächst das GDSII-Datenformat. Danach folgt eine Einführung in das Mathematica Softwaresystem und in die Mathematica Programmierung. Weitere Kapitel beschäftigen sich mit der Implementierung der Tools. Schließlich folgt ein Kapitel mit einer Demonstration der entwikkelten Tools.

Vorwort

Die vorliegende Diplomarbeit wurde im Sommersemester 2009 als Abschlussarbeit meines Studiums, an der Hochschule Mittweida, Fakultät Informationstechnik & Elektrotechnik, in Wien angefertigt.

Ich möchte mich recht herzlich bei allen Personen bedanken, welche an der Entstehung der Arbeit mitgewirkt haben. Mein Dank gilt insbesondere Herren Prof. Dr.-Ing. Werner Günther, welcher das Manuskript mit großem Einsatz durchgesehen hat und mit seinen konstruktiven Anmerkungen die Qualität dieser Arbeit positiv beeinflusst hat. Besonders bedanken möchte ich mich auch bei meinem betrieblichen Betreuer Herrn Dipl.-Ing. Stefan Eder-Kapl, welcher mir mit fachlichem Rat zur Seite gestanden ist. Ganz besonderer Dank gilt auch meiner Frau Romana und meiner Tochter Natalie, welche mich jederzeit bei meinem Vorhaben unterstützt haben.

1 Einleitung

1.1 Motivation

Das GDSII-Streamformat ist eines der wichtigsten Datenformate bei der Herstellung Integrierter Schaltkreise. Das Format spezifiziert die Beschaffenheit der so genannten Layoutdaten. Diese Layoutdaten stellen eine Beschreibung der geometrischen Formen dar, welche die Funktionalität der Schaltungen auf physikalischer Ebene entscheidend mitbestimmen.

Will man direkte Einflussnahme auf die elektrischen/physikalischen Eigenschaften der herzustellenden Schaltungen nehmen, so ist das unter anderem über die gezielte geometrische Spezifikation bzw. Formgebung der Schaltungsteile möglich. Diese freie und individuelle Formgebung der Schaltungsteile ist insbesondere dann möglich, wenn man in der Lage ist GDSII-Daten zu erstellen welche die Geometrien der individuell gestalteten Schaltungsteile enthalten.

Ein weiteres Anwendungsgebiet ist jenes der Charakterisierung von Mikro-Lithographie-Apparaten. Diese Apparate führen den wichtigen Schritt der lithographischen Strukturübertragung für die Chipherstellung aus. Hierbei werden bei den gängigen Verfahren so genannte Fotomasken, das sind naiv betrachtet Vorlagen in der Art von Diafotos, auf mit Fotolack beschichteten Halbleitermaterialien, auf optischem Wege übertragen. Für die genaue Charakterisierung der Leistungsfähigkeit dieser Apparate werden Fotomasken mit unterschiedlichsten Testmustern benötigt. Diese Testmuster müssen, will man die Leistungsgrenzen der Lithographie-Apparate ausloten, ebenfalls frei spezifiziert werden können. Auch dafür ist es notwendig individuelle GDSII-Layouts zu erstellen.

Nun existieren für die direkte Layouterstellung bereits eine Reihe von Programmen, so genannte Layouteditoren. Diese Programme sind vom Prinzip her den allgemein bekannten (Vektor-)Grafik-Programmen sehr ähnlich. Sie ermöglichen es dem Benutzer mittels Mauseingabe nahezu beliebige geometrische Formen zu zeichnen. Nachteilig bei dieser Art der Geometrie Eingabe ist, dasd komplexere Formen mit der Maus oft nur sehr mühsam zu erstellen sind, auch die Möglichkeiten für eine präzise Koordinatenspezifikation sind selten befriedigend. Meist fehlt auch die Möglichkeit geometrische Formen parametrisiert zu erstellen, um so z.B. verschiedene Seitenlängen geometrischer Formen nicht fix vorgeben zu müssen und jederzeit bequem ändern bzw. festlegen zu können.

Um Layoutdaten in der geschilderten Qualität erstellen zu können helfen die Layout-Editoren nur bedingt weiter. Für diesen Zweck existieren jedoch bereits Tools in Form von Software-

bibliotheken für gängige Programmiersprachen wie z.B. Java oder C++. Diese Tools versetzen den Benutzer in die Lage die gewünschten Layouts, durch Erstellung von Programmen, in einer der erwähnten Programmiersprachen zu generieren.

Mit Mathematica existiert seit vielen Jahren ein extrem vielseitiges und mächtiges Software-System. Durch die Programmierbarkeit des Systems ist eine Vielzahl von Aufgabenstellungen damit lösbar.

Über die Existenz einer Softwarebibliothek für das Mathematica-System für die Generierung von Layoutdaten, insbesondere im GDSII-Format, ist dem Autor nichts bekannt. Dieser Umstand stellt die Motivation dar, im Rahmen dieser Arbeit, ebenso eine Softwarebibliothek für das Mathematica-System zu entwickeln.

Mit dem zu schaffenden Tool sollen die Daten-Export/Import Möglichkeiten des Mathematica Software Systems, um das in der Halbleiterindustrie wichtige GDSII-Datenformat erweitert werden. Damit erhält der Designer ein Tool in die Hand, welches die gesamte Mathematica Funktionalität für den Layoutprozess zur Verfügung stellt.

1.2 Aufgabenstellung

Der Autor dieser Diplomarbeit ist bei der Fa. IMS-Nanofabrication AG, Wien beschäftigt. Die Firma hat die Entwicklung innovativer Micro-(Nano)-Lithografieapparate für die Micro-(Nano)-Industrie bzw. für die Halbleiterindustrie zum Ziel.

In diesem Zusammenhang besteht immer wieder Bedarf an individuell spezifizierten Testmustern in Form von GDSII-Daten. Zum Einem werden die benötigten Muster nach Bedarf entworfen, zum Anderen werden Muster auch von Partnerfirmen bereitgestellt. In letzterem Fall ist es gewünscht, die zur Verfügung gestellten Muster zu inspizieren und wenn für bestimmte Zwecke notwendig, entsprechend zu modifizieren.

Der Autor hat die Aufgabe übernommen, diesen Bedarf, durch die Implementierung geeigneter Softwaretools, mit möglichst großer Flexibilität zu erfüllen. Im Betrieb ist eine Reihe von Mitarbeitern mit dem Mathematica-System vertraut, dadurch bietet sich der Einsatz von Mathematica für eine mögliche Lösung des Problems an.

Vom betrieblichen Betreuer der Diplomarbeit wurden die folgenden Vorgaben gestellt:

Es soll eine Mathematica-GDSII Softwarebibliothek erstellt werden, welche zusammen mit dem Mathematica-System folgende Funktionalitäten bereitstellt bzw. unterstützt:

- Entwicklung von Mathematica Funktionen für die Erstellung gültiger GDSII-Daten. Hierbei geht es nicht um die Konversion eines Datenformates nach GDSII, sondern vielmehr darum, den Nutzer in die Lage zu versetzen, GDSII-Daten in einer konstruktiven Art und Weise, durch erstellen von Mathematica-Programmcodes unter zu Hilfenahme der implementierten Funktionen, zu erzeugen.
- Darstellung von GDSII-Daten in einer textuellen, vom Benutzer lesbaren Form. GDSII-Daten sind spezifikationsgemäß immer binär, um dem Nutzer GDSII-Daten besser zu veranschaulichen, wird eine umkehrbar eindeutige Abbildung der GDSII-Binärdaten auf eine textuelle Darstellungsform implementiert.
- Speichern von generierten, binären GDSII-Layoutdaten als gültige GDSII-Dateien. Mit diesem Schritt stehen die Layoutdaten bereit, um diese mit Hilfe einer Lithografieapparatur, auf ein geeignetes Substrat physikalisch zu übertragen
- Einlesen von GDSII-Binärdateien in das Mathematica-System. Es handelt sich hierbei um die Aufgabe der Konversion von GDSII-Daten in geeignete Mathematica-Datenstrukturen. Dieser Schritt ist notwendig, um eine weitere Manipulation der Daten mit Hilfe des Mathematica-Systems zu ermöglichen.
- Analyse von gelesenen GDSII-Daten. GDSII-Daten sind im Allgemeinen aus hierarchischen Strukturen aufgebaut. Diese Hierarchien können mitunter komplexe Ausmaße annehmen. Aus diesem Grund ist es notwendig die auftretenden Strukturen und Hierarchien analysieren zu können.
- Filterung gelesener GDSII-Daten nach spezifizierbaren Kriterien. Das GDSII-Format stellt eine Reihe von Attributen zur Verfügung, welche den in den Daten enthaltenen Objekten zugewiesen werden können. Durch die Filterung der Daten nach diesen Attributen, steht dem Nutzer ein weiteres Hilfsmittel für die Analyse von GDSII-Daten zur Verfügung.
- Konvertierung gelesener und analysierter GDSII-Daten in Mathematica-Grafiken. Da GDSII insbesondere auch als Vektorgrafik-Format zu verstehen ist, wird ein Tool benötigt die Daten grafisch darzustellen.

Weiters soll diese Arbeit folgende Themen bereitstellen bzw. aufbereiten:

- Eine Einführung in das Mathematica-System
- Eine Einführung in die Mathematica Programmierung
- Eine Einführung in das GDSII-Format
- Eine Erläuterung der durchgeführten Implementierung und der dabei verwendeten Konzepte und Methoden
- Erläuterungen zu der Funktionsweise der implementierten Funktionen und funktionelle Tests der selbigen

2 Das GDSII-Stream-Format

2.1 Die Bedeutung des GDSII-Formates beim Schaltungsentwurf

Ein wichtiger Schritt bei der Entwicklung Integrierter Schaltungen, ist jener des Layout-Prozesses. Das Ergebnis dieses Prozesses ist der geometrische Entwurf elektronischer Schaltungen, also die geometrische Formgebung der Schaltungsteile wie Metallleitungen, Gates, Kontakte etc. Entsprechend der physikalischen Erfordernisse besteht ein komplettes Layout im Allgemeinen aus einer Vielzahl verschiedener Layer (Schichten). Die Rolle von GDSII besteht hierbei darin, eine Spezifikation für das Datenformat der benötigten geometrischen Objekte bereitzustellen. Bei den Objekten handelt es sich überwiegend um Polygone, welche in GDSII, in Form von Koordinaten-Listen spezifiziert werden. Somit ist das GDSII-Format den Vektorgrafik-Formaten zuzuordnen. [1], [2], [3], [4].

Einer der ersten Schritte beim Entwurf ist die Umsetzung einer Schaltungsidee in eine formelle Beschreibung der zu entwickelnden Schaltung. Dieser Schritt wird oft als Designentry bezeichnet. Im Falle analoger Schaltungen besteht dieser Schritt üblicherweise aus dem rechnergestützten Zeichnen eines Schaltplanes, Schematics genannt.

Einfache digitale Schaltungen können entweder wie zuvor als Schematic entworfen werden oder aber mit Hilfe einer HDL Sprache (Hardware Description Language z.B. VHDL, VERILOG, etc.) auf der Gate Level Abstraktionsebene beschrieben werden. Hierbei werden digitale Bauteile (Gatter, Gates) über HDL Netzlisten miteinander verbunden.

Komplexere Digitalschaltungen werden ebenfalls mittels einer HDL Sprache, allerdings auf RTL Ebene (Register-Transfer-Level) entworfen. Der RTL-Entwurf beschreibt die gewünschten Schaltungen auf einer höheren Abstraktionsebene als dies beim Gate Level Design der Fall ist. Hierbei wird mittels des HDL Design die zu realisierende Schaltung aus mehreren simultan arbeitenden Prozessen entworfen. Zum Einsatz kommen dabei zwei Typen von Prozessen. Zum Einen sind dies rein kombinatorische Prozesse welche, ähnlich wie bei der herkömmlichen Software Programmierung, Eingangsgrößen mittels arithmetischer und/oder logischer Operatoren zu Ausgangsgrößen verknüpfen. Zum Anderen handelt es sich um getaktete Prozesse welche in Registern ihre Zustände speichern und über eine kombinatorische Logik selbige Register ansteuern und ändern um den Zustand des Prozesses zu wechseln bzw. einen Zustand zu speichern.

Ein wie zuvor beschriebenes RTL Design wird in einem der nachfolgenden Prozessschritte von einem Synthesetool in ein Gate Level Design synthetisiert.

Als ein weiterer Prozessschritt folgt sowohl für den analogen als auch für den digitalen Fall die Generierung einer Netzliste. Eine Netzliste beschreibt einerseits die in einer Schaltung vorkommenden Bauteile mit deren Anschlüssen und andererseits die Netze (Leitungen) welche die Bauteile bzw. deren Anschlüsse miteinander verbinden, sowie allfällige Attribute der Bauteile und Netze.

Ein weiterer Schritt in der Entwurfskette besteht aus der Umsetzung der Netzliste in ein Layout. Dieser Entwicklungsschritt wird als Place & Route bezeichnet. Das Place & Route Softwaretool entnimmt dabei die benötigten Bauteile aus einer Bauteilbibliothek, in welcher die Teile als geometrische Objekte in den verschiedenen notwendigen Layern vorliegen und platziert (Place) diese an geeigneter Stelle im zu erstellenden Layout.

Die Bauteile müssen im zweiten Teil des Place & Route Prozessschrittes geeignet miteinander verbunden (Route) werden. Die Verbindungen bestehen ebenfalls aus geometrischen Objekten. Im Allgemeinen sind das Liniensegmente von geeigneter Linienbreite welche, in einem geeigneten Layer mittels passender Leiterbahnführung, die zu verdrahtenden Anschlüsse zweier oder mehrer Bauteile miteinander verbinden.

Zu den erwähnten Bauteilbibliotheken ist anzumerken, dass die darin enthaltenen komplexeren Bauteile in der Regel hierarchisch aus einfacheren Bauteilen mittels der zuvor geschilderten Methodik entworfen werden. Auf der untersten Hierarchieebene stehen dann, im digitalen Fall, einfache Gatter oder einzelne Schalttransistoren, diese einfachen Bauteile werden geometrisch anhand der technischen Erfordernisse bzw. physikalischen Notwendigkeiten entworfen.

2.2 Ein Layout Beispiel an Hand eines CMOS-NAND-Gatter

Als Beispiel für den Entwurfsprozess dient die einfache Digitalschaltung eines CMOS-NAND-Gatter. Die CMOS-Technologie ist heutzutage die meistgenutzte Technologie der Halbleiterindustrie. Wesentliches Merkmal der CMOS-Technologie ist deren Aufbau aus zueinander komplementär verschalteten NMOS- und PMOS-Transistoren mit dem daraus resultierenden Vorteil des zumindest im statischen Betrieb nahezu nicht vorhandenen Stromflusses.

So wie alle gängigen industriellen Halbleiter-Prozesse zur Herstellung Integrierter Schaltungen, werden auch CMOS-Schaltungen mit Hilfe eines Planar-Prozesses realisiert. Bei einem Planarprozess werden die einzelnen Schaltungselemente in Form von aufeinander gestapelten dünnen Schichten auf einem geeigneten Trägersubstrat aufgebracht bzw. strukturiert. In der

Regel handelt es sich bei dem Trägermaterial um p (oder auch n) dotiertes Silizium. Auf dem Silizium-Träger, welcher als Wafer bezeichnet wird, werden in einer Abfolge von Prozessschritten Materialstrukturierungen vorgenommen.

Die Strukturierung erfolgt hierbei in der Regel durch Aufbringen eines photoempfindlichen Lackes auf dem zu prozessierenden Wafer und anschließender Belichtung des selbigen mit einer, die Struktur definierenden, Photomaske. In weiteren Schritten werden die belichteten Teile der Lackschicht entwickelt und anschließend in einem Ätzprozess entfernt.

Nach dem Ätzprozess besteht auf dem Trägermaterial eine Lacktopologie welche als Maskierung für den unmittelbar folgenden Prozessschritt dient. In solch einem Folgeschritt können nun, die nicht mit Lack geschützten Oberflächenbereiche z.B. mit Hilfe so genannter Iionenimplanter dotiert werden. Die Lackmaskierung kann aber auch für andere Prozessschritte wie z.B. weitere Ätzprozesse oder z.B. das Aufdampfen von Metallschichten etc. verwendet werden.

Durch eine wiederholte Abfolge von Prozessschritten, analog zu den oben beschriebenen, entsteht der typische Schichtaufbau der Planartechnik. Der Lagenaufbau eines CMOS-Prozesses ist in Abbildung 3 „Ein CMOS-Layer-Stack“ zu sehen.

Anhand eines CMOS-NAND wird nun gezeigt, wie sich der Entwurf des Gatters für den Schaltungsdesigner darstellt. Zunächst wird das Schematics aus NMOS und PMOS Transistoren erstellt. Die Transistoren müssen dann in weiterer Folge als geometrisch-physikalische Strukturen realisiert werden. Im Beispiel wurden die Transistoren im Schematics bereits in Voraussicht der physikalischen Layout Erfordernisse entsprechend gezeichnet und platziert; diese Vorgangsweise ist in der Praxis natürlich nicht üblich und erfolgt hier nur aus Gründen der Anschaulichkeit. Es sei auch festgehalten, dass die entworfene Schaltung keinen Anspruch auf tatsächliche Funktionalität erhebt. Für eine real funktionierende Schaltung müssen eine Reihe von Designrules eingehalten werden um einer bestimmten Halbleitertechnik gerecht zu werden. Weiters sind je nach Technologie noch verschiedene Hilfsstrukturen wie z.B. Kontaktierungen des Substrates bzw. der Wannen mit VDD bzw. GND oder z.B. Antilatchup-Schaltungen einzufügen, um zu funktionsfähigen Schaltungen zu gelangen. Die elektrischen Eigenschaften einer solchen Schaltung sind letztlich von der verwendeten Technologie und dem geometrischen Layout bestimmt.

Ergebnis des Layout-Prozesses ist eine geometrische Beschreibung, in erster Linie in Form von Polygonstrukturen, der verschiedenen notwendigen Layer wie z.B. Metall, Polysilizium,

etc. Die Geometrie jedes einzelnen Layers wird üblicherweise in weiterer Folge auf Photomasken übertragen, welche als Vorlagen in photolithographischen Prozessen dazu dienen die entsprechenden Strukturen auf geeignetes Halbleitermaterial zu übertragen.

Die geschilderte Entwicklungskette vom Schematics zum Layout wird in Abbildung 1 „Ein CMOS Nand-Gatter“ in Form dreier aufeinander folgender Bilder, sowie der zugehörigen Netzliste in Tabelle 1 „Netztliste CMOS-NAND-Gatter“ veranschaulicht. Es ist an dieser Stelle erwähnenswert, dass aus dem Layout zumindest nicht auf triviale Weise, auf die ursprüngliche Netzliste zurück geschlossen werden kann. Ein Grund dafür ist unter anderem jener, dass verschiedene Schaltungsteile im Layout, wie in Abbildung 1 „Ein CMOS Nand-Gatter“ ersichtlich, zu gemeinsamen geometrischen Objekten verschmelzen können.

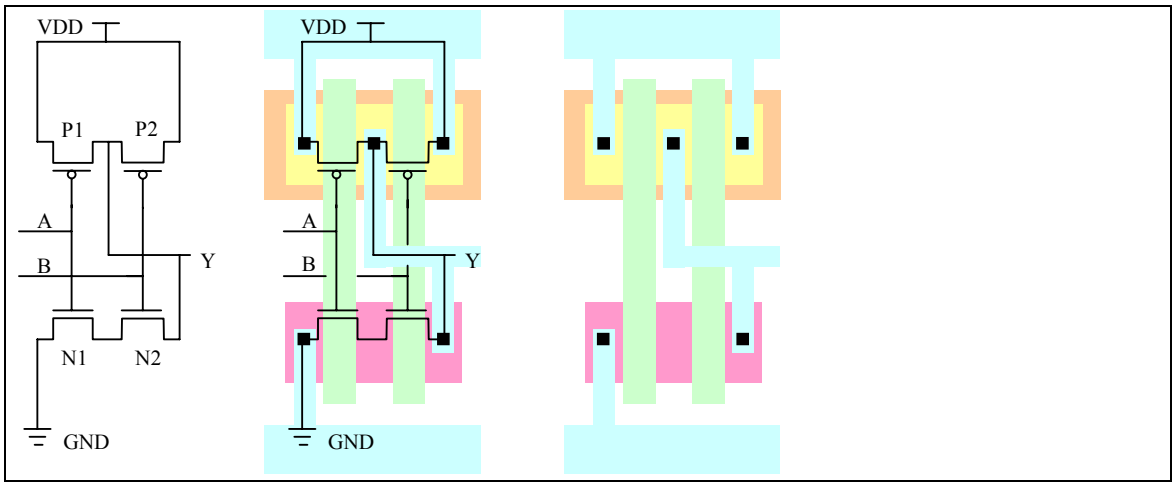

Abbildung 1 „Ein CMOS Nand-Gatter“

Bauteil	**Anschluss**	**Netz**
P1	1	VDD
	2	Y
	3	A
P2	1	Y
	2	VDD
	3	B
N1	1	GND
	2	X
	3	A
N2	1	X
	2	Y
	3	B

Tabelle 1 „Netztliste CMOS-NAND-Gatter“

Wie erläutert, besteht ein Layout aus mehreren Layern. Dieser Umstand ist in Abbildung 2 „Die individuellen Layer eines CMOS-Gatter“ veranschaulicht. Die Layer wiederum definie-

ren einen Stapel von Materialschichten wie in Abbildung 3 „Ein CMOS-Layer-Stack“ symbolisch dargestellt.

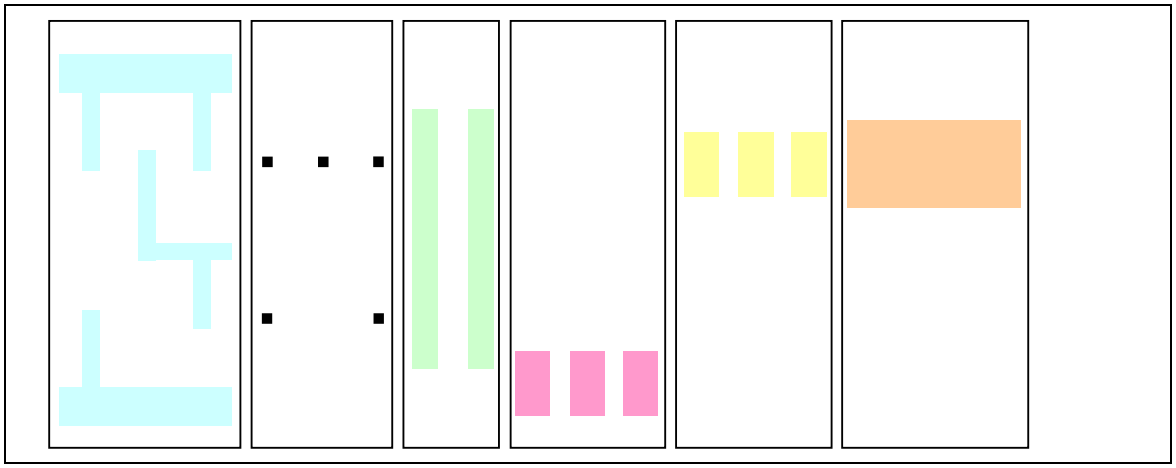

Abbildung 2 „Die individuellen Layer eines CMOS-Gatter“

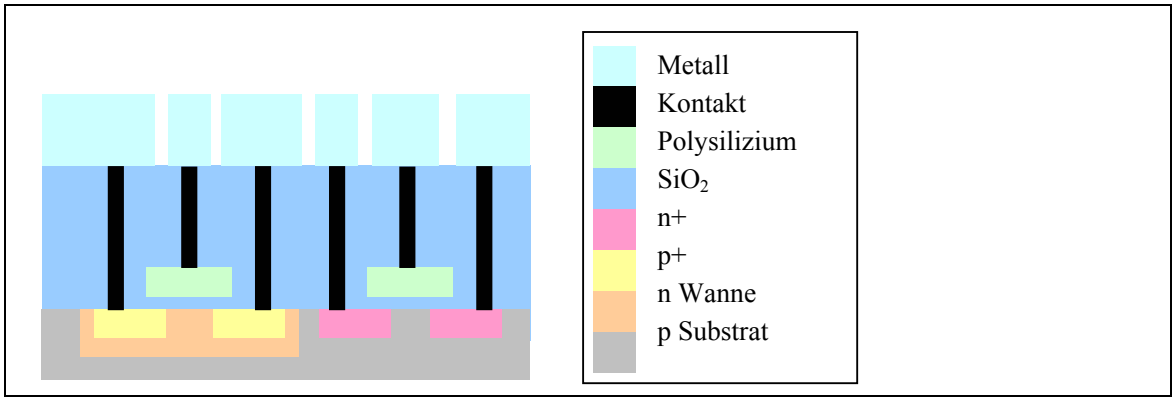

Abbildung 3 „Ein CMOS-Layer-Stack“

In Kapitel 4 „Praktische Anwendung und Demonstration der Tools“ wird das vorgestellte NAND-Gatter wiederholt bemüht werden, um dort die entwickelten Tools an Hand des Gatters vor zu führen.

2.3 Ursprung und Historie des GDSII-STREAM-FORMAT

In der Anfangszeit der Mikroelektronik wurde der Entwurf Integrierter Schaltungen noch per Hand mittels transparenten Folien und darauf appliziertem lichtundurchlässigem Klebeband hergestellt. 1971 wurde von Calma (Sunnyvale, California, USA) das Graphical Design System kurz GDS eingeführt. 1978 folgte GDSII. GDSII war ein CAD Softwaresystem zum rechnergestützten Entwurf Integrierter Schaltungen. Das System verwendete das GDSII Stream Format zur Speicherung der Layouts. GDSII als Softwaresystem ist schon lange aus der Halbleiterindustrie verschwunden, nach wie vor aktuell ist jedoch das GDSII Stream Format. Heutzutage wird der Begriff GDSII kurzerhand als Synonym für das GDSII Stream Format verwendet. Erstaunlicherweise hat es bis 2003 gebraucht, um den bislang ersten und einzigen ernst zu nehmenden Nachfolger für GDSII Stream auf den Plan zu rufen, das OASIS - Open Artwork System Interchange Standard Format. Trotz der Konkurrenz ist GDSII bis heute das wohl am weitesten verbreitete Layout Format der Halbeiterindustrie.

2.4 Aufbau des GDSII-STREAM-FORMAT

GDSII ist eines der Standard-Formate der Halbleiterindustrie für die Repräsentation grafischer/geometrischer Layout-Daten. Das Format besteht, gemäß logischer Sichtweise, aus einer baumartigen Hierarchie, benannter GDSII-STRUCTURE's, wie in Abbildung 4 „GDSII-Struktur Hierarchie“ symbolisch dargestellt. Jede der Strukturen besteht im Allgemeinen aus einer Anzahl von GDSII-ELEMENT's. Die Elemente fallen jeweils in eine von zwei Kategorien. Diese Kategorien sind: 1. geometrische Objekte wie z.B. Polygone, welche auf einem GDSII-LAYER platziert werden und 2. Referenzen. Die Referenzen verweisen hierbei, über die Spezifikation eines Referenz-Namens, auf eine mit eben diesem Namen benannte weitere Struktur.

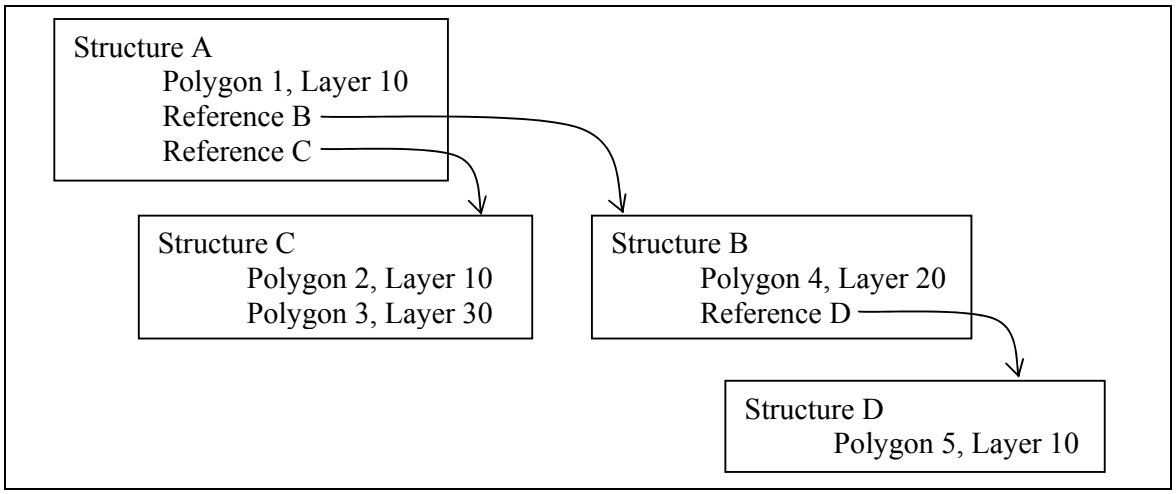

Abbildung 4 „GDSII-Struktur Hierarchie“

Die technische Realisation der gezeigten GDSII-STRUCTURE-Hierarchie basiert auf einer sequentiellen Abfolge von geeigneten, gemäß GDSII-Spezifikation, binären, GDSII-RECORD’s. Jeder GDSII-RECORD besteht aus einem binären Prolog gefolgt von den binären Nutzdaten, des RECORD’s. Im Prolog wird unter anderem auch die GDSII-RECORD-TYPE des RECORD’s festgelegt. Die gesamte Abfolge zusammengehöriger GDSII-RECORD’s wird auch als GDSII-Datensatz bezeichnet. Im Allgemeinen wird ein GDSII-Datensatz in einer GDSII-Datei gespeichert.

Binär-Prolog *1*	Binär-Nutzdaten *1*
Binär-Prolog *2*	Binär-Nutzdaten *2*
…	…
Binär-Prolog *n*	Binär-Nutzdaten *n*

Tabelle 2 „GDSII-Datensatz mit n GDSII-RECORD’s“

Da die binäre Form der GDSII-RECORD’s wenig anschaulich ist, wird in dieser Arbeit überwiegend eine textuelle Form für die Darstellung selbiger gewählt. Bei dieser Darstellung wird jeder einzelne GDSII-RECORD, auf einer eigenen Textzeile gezeigt. Bei der textuellen Form wird zunächst der GDSII-RECORD-TYPE als Klartext dargestellt, es folgt ein Leerzeichen und abschliessend folgen die Nutzdaten in Textform. Um die Anschaulichkeit der Darstellung weiter zu erhöhen, werden an geeigneten Stellen zusätzlich Einrückungen sowie Leerzeilen eingefügt.

Um zu einem raschen Verständnis des GDSII-Formates zu gelangen, folgt nun beispielhaft in einfacher aber vollständiger GDSII-Datensatz, welcher eine einzige Struktur in Form eines Quadrates wie in Abbildung 5 „Eine einfache GDSII-STRUCTURE“ ersichtlich repräsentiert.

```
HEADER 600
BGNLIB
LIBNAME
UNITS 1.e-6 1.

BGNSTR
STRNAME UnitSquare
BOUNDARY
  LAYER 0
  DATATYPE 0
  XY {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}
  ENDEL
ENDSTR
ENDLIB
```

Tabelle 1 „Ein vollständiger beispielhafter GDSII-Datensatz“

Der Datensatz besteht, wie in Tabelle 1 „Ein vollständiger beispielhafter GDSII-Datensatz“ ersichtlich ist, aus genau 13 GDSII-RECORD’s von unterschiedlichem GDSII-RECORD-TYPE. Diese RECORD’s werden nun detaillierter erläutert:

- Der Datensatz beginnt korrekterweise mit einem GDSII-HEADER-RECORD, welcher mit seinen Nutzdaten die GDSII-VERSION mit 6.00 festlegt.
- Es folgt der notwendige GDSII-BGNLIB-RECORD.
- Weiters folgt der GDSII-LIBNAME-RECORD.
- Anschließend folgt ein GDSII-UNITS-RECORD welcher die GDSII-USER-UNITS mit 10^-6 Meter (1 Mikrometer) und die GDSII-DATABASE-UNITS mit 1 festlegt.
- Jetzt wird mit dem GDSII-BGNSTR-RECORD der Beginn einer Struktur definiert
- Nun wird durch den GDSII-STRNAME-RECORD der definierten Struktur, der Name „UnitSquare“ zugewiesen.
- Die Struktur besteht aus genau einem GDSII-ELEMENT, diese wird mit einem GDSII-BOUNDARY-RECORD als ein geschlossenes Polygon festgelegt.
- In weiterer Folge wird dem GDSII-ELEMENT mit Hilfe eines GDSII-LAYER-RECORD die Layernummer 0 zugewiesen
- Ebenso wird dem GDSII-ELEMENT nun mit einem GDSII-DATATYPE-RECORD die Datatypenummer 0 zugewiesen.
- Der folgende GDSII-XY-RECORD und die darin als Nutzdaten enthaltenen Koordinaten beschreiben, wie leicht ersichtlich ist, ein im Koordinatenursprung platziertes

Quadrat mit einer Kantenlänge von 1. Genau genommen beträgt die Kantenlänge, auf Grund der spezifizierten GDSII-USER-UNITS, 1 Mikrometer.

- Schließlich wird mit dem folgenden GDSII-ENDEL-RECORD das GDSII-BOUNDARY-ELEMNT abgeschlossen.
- Weiters wird mit dem ENDSTR-RECORD die mit „UnitSquare“ benannte Struktur abgeschlossen.
- Letztendlich wird mit dem GDSII-ENDLIB-RECORD der GDSII-Datensatz beendet.

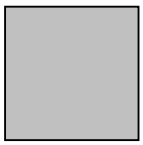

Abbildung 5 „Eine einfache GDSII-STRUCTURE“

2.5 Die EBNF des GDSII-STREAM-FORMAT

Wie Tabelle 1 „Ein vollständiger beispielhafter GDSII-Datensatz“ vermuten lässt, muss eine bestimmte Reihenfolge bei den eingesetzten GDSII-RECORD’s eingehalten werden um einen zumindest syntaktisch korrekten GDSII-DATENSATZ zu erhalten. Das exakte Regelwerk für die korrekte Abfolge von GDSII-RECORD’s wird mittels der Extended Bachus Nauer Form, kurz EBNF genannt, spezifiziert.

Die Syntaxen der gängigsten Datenformate, Programmiersprachen, Kommunikationsprotokolle, etc. können formal als (kontextfreie) Sprachen angesehen werden, welche mittels so genannter kontextfreier Grammatiken, erzeugt werden können. Die EBNF ist eine Metasprache zur Darstellung von kontextfreien Grammatiken und eignet sich somit bestens dazu, die GDSII-Syntax zu spezifizieren. Die EBNF für eine bestimmte Grammatik besteht aus einer endlichen Menge von Ableitungsregeln welche angeben wie Nonterminal-Symbole in endlich vielen Schritten abzuleiten sind, so dass letztlich eine Folge von Terminal-Symbolen entsteht welche einen syntaktisch gültigen Satz der zugrunde liegenden Sprache darstellen.

Bei der für das GDSII-STREAM-FORMAT verwendeten EBNF werden folgende Metasymbole verwendet:

Metasymbol	**Bedeutung**
=	Definition
.	Regelende
[ ]	Klammerung für Optionales Symbol
( )	Klammerung für Symbolgruppierung
{ }	Klammerung für null oder mehrmals vorkommendes Symbol
|	Alternative (Oder)

Tabelle 4 „Metasymbole der EBNF“

Alle Wörter in Großbuchstaben werden in der hier verwendeten EBNF als Terminal-Symbole aufgefasst und sind über ihre Bezeichnung genau einem GDSII-RECORD-TYPE zugeordnet. Alle Wörter in Kleinbuchstaben werden als Nonterminal-Symbole mittels der EBNF soweit abgeleitet, bis ausschließlich Terminal-Symbole, das sind in diesem Fall genau die GDSII-RECORD’s, vorliegen. Der Aufbau der GDSII-RECORD’s an sich, ist nicht Teil der EBNF, die Spezifikation ist Kapitel 2.6 zu ersehen.

EBNF des GDSII Stream Format	
stream_format =	HEADER BGNLIB LIBNAME [REFLIBS][FONTS][ATTRTABLE] [GENERATIONS][format_type] UNITS {structure} ENDLIB.
format_type =	FORMAT [MASK {MASK} ENDMASKS].
structure =	BGNSTR STRNAME [STRCLASS]{element} ENDSTR.
element =	(boundary | path | sref | aref | text | node | box) {boundary | path | sref | aref | text | node | box} {property} ENDEL.
boundary =	BOUNDARY [EFLAGS][PLEX] LAYER DATATYPE XY.
path =	PATH [EFLAGS][PLEX] LAYER DATATYPE [PATHTYPE][WIDTH] XY.
sref =	SREF [EFLAGS][PLEX] SNAME [strans] XY.
aref =	AREF [EFLAGS][PLEX] SNAME [strans] COLROW XY.
text =	TEXT [EFLAGS][PLEX] LAYER textbody.
node =	NODE [EFLAGS][PLEX] LAYER NODETYPE XY.
box =	BOX [EFLAGS][PLEX] LAYER BOXTYPE XY.
textbody =	TEXTYPE [REPRESENTATION][PATHTYPE][WIDTH][strans] XY STRING.
strans =	STRANS [MAG][ANGLE].
property =	PROPATTR PROPVALUE.

Tabelle 5 „EBNF des GDSII-STREAM-FORMAT“

In den folgenden Erläuterungen zur GDSII-EBNF werden die verwendeten GDSII-RECORD’s mittels „// Erläuterungstext“ kommentiert, dies ist weder ein Teil des GDSII-STREAM-FORMAT, in welchem keine Kommentare vorgesehen sind, noch ein Teil der EBNF.

Wie aus der EBNF ersichtlich ist, besteht eine gültige GDSII-STREAM-FORMAT-Datei zumindest aus folgenden, jeweils exakt einmal vorkommenden (Ausnahme: {*structure*}), sequentiell angeordneten Terminalen, in unserem Fall sind das immer GDSII-RECORD’s:

HEADER	// Header des Stream
BGNLIB	// Beginn des Stream
LIBNAME	// Name für den Stream (nicht notwendigerweise der Dateiname)
UNITS	// Einheiten für die Umrechnung von Koordinaten in Echtmaße
{*structure*}	// Beliebige Anzahl von Strukturen
ENDLIB	// Ende des Stream

Das in der obigen Abfolge von GDSII-RECORD's zwischen UNITS und ENDLIB gelegene {*structure*} spezifiziert gemäß EBNF eine beliebige Anzahl (auch null ist möglich) von GDSII-Strukturen, welche letztendlich ebenfalls aus GDSII-RECORD's aufgebaut sind.

structure wiederum besteht gemäß der BNF mindestens aus folgenden RECORD's:

```
BGNSTR          // Beginn der Struktur
STRNAME         // Name der Struktur (wichtig für eine eventuelle Referenzierung)
{element}       // Beliebige Anzahl von Elementen für die Struktur
ENDSTR          // Ende der Struktur
```

element muss, um beispielsweise ein (geschlossenes) Polygon zu spezifizieren, aus mindestens folgenden RECORD's aufgebaut sein:

```
BOUNDARY        // Ein Polygon
LAYER           // Der Layer für die Platzierung des Polygons (Element)
DATATYPE        // Der Datatype für das Polygon (Element)
XY              // Die Koordinaten der Polygonpunkte
ENDEL           // Ende des Polygons (Element)
```

Alle weiteren möglichen GDSII-Streams sind analog, wie oben kurz angerissen, aus der GDSII-EBNF und aus der im nächsten Kapitel anschließenden GDSII-RECORD-Spezifikation ableitbar.

2.6 Das Binärformat der GDSII-RECORD's

GDSII-RECORD's besitzen, gemäß GDSII-Spezifikation, ein Binär-Daten-Format. Bei diesem Format besteht ein RECORD aus einem Prolog und darauf folgenden Nutzdaten. Sowohl Prolog als auch Nutzdaten liegen in binärer Form vor. Die Binärdaten der GDSII-RECORD's sind überwiegend in Form von 16 BIT WORD's organisiert. Hierbei sind die WORD's in, der so genannten, „Bigendian" (HIGH-BYTE vor LOW-BYTE) Reihenfolge abgelegt. Bedingt durch die variable Länge der Nutzdaten ist die Länge eines GDSII-RECORD's ebenfalls variabel. Von logischer Sichtweise betrachtet besteht ein GDSII-RECORD aus den vier aufeinander folgenden Teilen: RECORDLENGTH, RECORDTYPE, DATATYPE, DATA.

WORD Nr..	High BYTE	Low BYTE
1 Prolog	RECORDLENGTH 2**n*	
2 Prolog	RECORDTYPE *r*	DATATYPE *d*
3 Data	DATAWORD 1	
.	.	
.	.	
n Data	DATAWORD *n* -2	

Tabelle 6 „GDSII-RECORD Binärformat"

Wie aus Tabelle 6 „GDSII-RECORD Binärformat“ ersichtlich ist, wird jeder GDSII-RECORD von einem Prolog eingeleitet. Der Prolog beginnt immer mit der RECORD-LENGTH, einer 16 BIT INTEGER Zahl, welche die Gesamtlänge des RECORD’s in 8 BIT BYTE angibt. Als Nächstes folgt Der RECORDTYPE, eine 8 BIT INTEGER Zahl, welche die RECORDTYPE-Nummer des betreffenden RECORD’s spezifiziert. Der letzte Teil des Prologs ist der DATATYPE, eine 8 BIT INTEGER Zahl, welche die DATATYPE-Nummer des RECORD’s festlegt.

Anschließend an den Prolog folgen die Nutzdaten. Das Format der Nutzdaten ist durch den DATATYPE des RECORD’s spezifiziert, die Länge der Nutzdaten ergibt sich aus der RECORDLENGTH abzüglich der 2 WORD’s des Prologs.

Wie aus Tabelle 7 „GDSII-DATATYPE’s“ ersichtlich ist, kennt GDSII, 7 verschiedene DATATYPE’s. Diese Typen umfassen verschiedene Ganzzahlen, Gleitpunktzahlen Bitarrays und Zeichenketten. Die Spezifikation der Typen erfolgt wie aus Tabelle 6 „GDSII-RECORD Binärformat“ ersichtlich ist, über das DATATYPE-Feld des RECORD’s.

DATATYPE	Value
NO_DATA	0x00
BIT_ARRAY	0x01
INTEGER_2	0x02
INTEGER_4	0x03
REAL_4	0x04
REAL_8	0x05
STRING	0x06

Tabelle 7 „GDSII-DATATYPE’s“

Das GDSII-Format kennt 60 verschiedene RECORD-Typen, welche jeweils durch das RECORDTYPE-Feld, wie in Tabelle 6 „GDSII-RECORD Binärformat“ ersichtlich ist, spezifiziert werden. Einige der möglichen RECORD-Typen, wie z.B.: HEADER, BOUNDARY, etc. wurden bereits vorgestellt. Der genaue Aufbau der wichtigsten RECORD-Typen ist im Anhang unter 8.1 „Die wichtigsten GDSII-RECORD’s“ zu finden. Eine vollständige Liste aller GDSII-RECORD-TYPE’s ist im Anhang unter 8.2 “Tabelle der GDSII-RECORD-Typen“ zu finden.

3 GDSII-Mathematica Tools und Implementierung

3.1 GDSII Erfordernisse – Mathematica Funktionalität

Um die Erfordernisse der Aufgabenstellung aus Kapitel 1.2 zu erfüllen, ist es notwendig einiges an Funktionalität durch die zu implementierenden Tools bereit zu stellen.

Für den Mathematica basierten GDSII-Export wird Folgendes an Funktionalität benötigt:

- Eine Anzahl von low-level Funktionen für die Generierung von GDSII-RECORD's.
- Eine Anzahl von high-level Funktion, welche jeweils sequentielle Abfolge zusammengehöriger GDSII-RECORD's erzeugt.
- Eine Funktion für die textuelle Darstellung von GDSII-RECORD's.
- Eine Funktion für die hexadezimale Darstellung von GDSII-RECORD's.
- Eine Funktion für die binäre, GDSII-Konforme Darstellung von GDSII-RECORD's.
- Eine Funktion für den Export von GDSII-RECORD's. als GDSII-Datensatz

Für den Mathematica basierten GDSII-Import wird weiters an Funktionalität benötigt:

- Eine Funktion zum Import binärer GDSII-RECORD's aus einem GDSII-Datensatz in geeignete Mathematica Strukturen.
- Eine Funktion zur Analyse importierter GDSII-RECORD's/GDSII-Datensatz.
- Eine Funktion zur Konvertierung eines analysierten GDSII-Datensatzes in eine Mathematica-Grafik.
- Eine Funktion zur Filterung von importierten GDSII-Daten aufgrund ausgewählter GDSII-Attribute

Die Funktionen für den Export und den Import sowie eine Reihe von weiteren benötigten Hilfsfunktionen werden alle basierend auf Mathematica Standardfunktionen entwickelt. Die folgenden Kapitel beschreiben die notwendigen Konzepte und Schritte für deren Implementierung.

3.2 Kurzübersicht der Mathematica-Programmierung

Um den weiteren Ausführungen dieser Arbeit folgen zu können, ist ein zumindest punktuelles Verständnis von Teilbereichen der Mathematica-Programmierung notwedig. Zu diesem Zweck werden nun auszugsweise Konzepte der Mathematica-Programmierung, insbesondere solche die in der Arbeit Verwendung finden kurz, vorgestellt. [5], [6]

Mathematica und insbesondere dessen Programmierung beruhen auf dem Konzept dass sämtliche Mathematica Objekte, seien es Daten, Grafiken, Funktionen, etc. aus Mathematica-Expressions's aufgebaut sind. Eine Mathematica-Expression besteht im Allgemeinen aus einem *Head* und einer *Sequence* von Elementen. Eine Expression wird in der Form: *A[B, C, D]* dargestellt. Hierbei bildet *A* den *Head* der Expression und *[B, C, D]* bilden die dazugehörige *Sequence* von Elementen. Die wichtigste Mathematica-Datenstruktur stellt *List* dar. *List* ist selbstverständlich eine Expression und dient zur Aufnahme von Listelementen. Diese entspricht grob den Arrays aus bekannteren Programmiersprachen. Die Form dafür ist wie folgt: *List[e1, e2, e3]*, wobei hierfür üblicherweise die vollständig gleichwertige Kurzform *{e1, e2, e3}* bevorzugt wird. Auch Funktionen bzw. Funktionsaufrufe werden als Expression's realisiert. Dies sei am Beispiel der trigonometrischen Funktion *Sin[ ]* gezeigt. Der Aufruf *Sin[x]* bewirkt wie zu erwarten die Berechnung des gewünschten Funktionswertes. Der Aufruf einer Funktion kann auch in der Postfix-Operator-Form erfolgen: *x//Sin*, dies ist vollkommen gleichwertige mit der zuvor gezeigten Form. Die Postfixform erhöht, insbesondere bei Funktionsaufrufketten, die Lesbarkeit von Mathematica Code, so kann beispielsweise die Form: *Sin[Sqrt[ArcTan[x]]]* gleichwertig auch als *x//ArcTan//Sqrt//Sin* dargestellt werden. Für Benutzer definierte Funktionen mit lokalen Variablen steht *Module* zur Verfügung. Die Verwendung ist wie folgt:

```
Norm[x_, y_] := Module[{h},
   h=x*x+y*y;
   Sqrt[h]
]
```

Tabelle 8 „Eine benutzerdefinierte Mathematica-Funktion"

Das Beispiel Tabelle 8 „Eine benutzerdefinierte Mathematica-Funktion" ist wie folgt zu verstehen: Definiert wird eine Funktion mit Namen *Norm*, diese akzeptiert 2 formale Paramter: *x* und *y*. Die Funktion bedient sich intern der lokalen Variable *h* und liefert als Funktionsergebnis die letzte enthaltene Anweisung, also *Sqrt[h]* zurück.

Diese gegebene Kurzübersicht sollte es ermöglichen, die weiteren Ausführung, dieser Arbeit nachvollziehen zu können. Eine umfassendere Einführung in das Mathematica-System ist in Kapitel 9, „Anhang: Das Mathematica Software System" zu finden.

3.3 Hilfsfunktionen und Hilfsstrukturen für die Mathematica GDSII Tools

In diesem Kapitel werden eine Reihe von Hilfsfunktionen und Hilfsstrukturen entworfen und vorgestellt, welche bei der Implementierung der Tools benötigt werden. Es werden hier nicht alle verwendeten und benötigten Hilfsfunktionen vorgestellt, sondern in erster Linie nur jene, welche für ein Verständnis der Tools notwendig sind.

- *GDSRECORDS*

Wie bereits mehrfach erläutert wurde, sind GDSII Dateien aus unterschiedlichen RECORD's aufgebaut. Aus diesem Grund wird zunächst eine Datenstruktur, welche die Grundeigenschaften aller möglichen GDSII-RECORD-Typen beschreibt, definiert. Die Anzahl der zu unterstützenden GDSII-RECORD's beträgt 60. Dementsprechend wird eine 60-elementige Mathematica-Liste erstellt, um die benötigten Informationen aufzunehmen. Die Liste wird mit dem Namen *GDSRECORDS* angelegt. *GDSRECORDS* besteht also aus 60 Elementen, jedes dieser Elemente ist seinerseits wiederum eine Liste mit jeweils 4 Elementen, die da wären:

1. Die Typnummer des GDSII-RECORD's als hexadezimaler String.
2. Der zur Typnummer zugehörige symbolische Typname als String.
3. Die Datentypnummer des GDSII-RECORD's als hexadezimaler String.
4. Der zur Datentypnummer zugehörige symbolische Datentypname als String.

Um den Sachverhalt besser zu veranschaulichen, wird an dieser Stelle beispielhaft der GDSII-RECORD-Typ mit der Nummer 0x10 (dezimal 16) vorgestellt. Es handelt sich hierbei um den GDSII-XY-RECORD, er wird in GDSII überwiegend verwendet um die XY Koordinatenpaare von Polygonen aufzunehmen. Der entsprechende Eintrag in der *GDSRECORDS* Liste besteht aus der 4-elementigen Liste mit den Werten: *{0x10, XY, 0x03, INTEGER_4}*. Die 4 Element haben die folgende Bedeutung: 0x10 und XY spezifizieren die Typnummer und den symbolischen TypeNamen 0x03 und INTEGER_4 spezifizieren den Datentyp des XY RECORD's. Zu beachten ist hierbei, dass in den GDSII-Dateien nur die Typnummern und Datentypnummer vorkommen. Die symbolischen Nummern kommen in den Dateien nicht vor und sind nur im Rahmen der hier entwickelten Tools von Bedeutung.

GDSRECORDS ist also eine Mathematica-Liste welche vom nachfolgenden, auszugsweise dargestellten Mathemtica-Code aufgebaut wird:

```
GDSRECORDS = {
        { 0x00, HEADER, 0x02, INTEGER_2 },
        .
        .
        { 0x10, XY, 0x03, INTEGER_4 },
        .
        .
        {0x3B, LIBSECUR, 0x02, INTEGER_2 }
}
```

Tabelle 9 „GDSRECORD's"

Dabei wird dem Mathematica-Symbol *GDSRECORDS* die entsprechende Mathematica-Liste zugewiesen. Die Liste besteht aus einer einleitenden geschwungenen Klammer, gefolgt von den Elementen der Liste welche durch Kommas getrennt sind und einer abschliessenden geschwungenen Klammer. Die einzelnen Elemente sind ihrerseits als Listen mit jeweils 4 Elementen erkennbar. Beispielhaft sind oben nur der allererste GDSII-RECORD „HEADER" mit der Nummer 0x00, der RECORD „XY mit der Nummer 0x10 (dezimal 16) sowie der allerletzte RECORD „LIBSECUR" mit der Nummer 0x3B (dezimal 59) vollständig dargestellt. Das ergibt insgesamt 60 RECORD-Typen.

- *GDSiiLookup[ ]*

Diese Hilfsfunktion erwartet als Parameter einen RECORD-TypeNamen. Dieser Parameter ist der symbolische Name eines GDSII-RECORD-Types. Aufgerufen liefert die Funktion, mit Hilfe der Liste *GDSRECORDS*, einen hexadezimalen String aus Typnummer und Datentypnummer des abgefragten Parameters. Beispielhaft schauen für den „HEADER" RECORD-Typ Aufruf und Rückgabeergebnis der Funktion wie folgt aus:

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiLookUp[„HEADER"]	*„0002"*

Das Ergebnis des Aufrufes ist der 4-stellige Hexadezimal-String : *„0002"*. Die ersten beiden Stellen *„00"* liefern hierbei die Typnummer, die letzten beiden Stellen *„02"* liefern die Datentypnummer.

- *GDSiiRECORD[ ]*

Die Hilfsfunktion generiert, wie auch schon ihr Name vermuten lässt, GDSII-RECORD's. Die Funktion erwartet zwei Parameter. Diese sind zu einem *RecordName*, mit der gleichen Bedeutung wie zuvor bei *GDSiiLookUp[ ]* und als weiteren Parameter *Content*, welcher die zugehörigen Nutzdaten eines GDSII-RECORD's spezifiziert. *Content* ist hierbei als Hexadezimal-String anzugeben. Die Funktion liefert ebenfalls einen Hexadezimal-String als Ergebnis. Dieser String besteht einerseits aus dem gleichen hexadezimalen Rückgabewert wie er auch von *GDSiiLookUp[ ]* geliefert wird, allerdings wird nun noch der Hexadezimal-String *Content* angehängt sowie die Gesamtlänge des GDSII-RECORD vorangestellt. Beispielhaft werden Aufruf und Rückgabeergebnis der Funktion für den *„HEADER"* RECORD-Typ betrachtet:

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiRECORD [„HEADER", GDSiiInteger2Hex[600]]	*„000600020258"*

Zunächst fällt auf, dass eine weitere Funktion bemüht wird: *GDSiiInteger2Hex[ ]*. Diese einfache Hilfsfunktion wandelt Integerzahlen in ihre 2 Byte Hexadezimalform um und wird an dieser Stelle nicht näher erläutert. Der Rückgabewert ist ein Ergebnis in der Form eines 12-stelligen Hexadezimal-String. Der besseren Verständlichkeit wegen erfolgt eine Gruppierung des Ergebnisses in seine GDSII Bestandteile: *„0006 00 02 0258"*.

Gemäß GDSII bilden die ersten 4 Stellen, (4 Tetraden, eine Tetrade entspricht 4 BIT entsprechen einer Hexadezimalstelle) eine Zahl vom GDSII-DATATYPE: INTEGER_2 welche die Gesamtlänge des RECORD's spezifiziert. In diesem Fall ist das „*0006"*, der betreffende RECORD hat also eine Länge von 6 BYTE oder auch 12 Tetraden. Das ist in Übereinstimmung mit dem Resultat als 12-stelligen hexadezimalen String. Gefolgt wird die Längenangabe von den 2 Stellen welche die GSII-Typnummer bilden, in diesem Fall mit *„00"* einen GDSII-HEADER-RECORD-Typ codierend. Als Nächstes folgen die 2 Stellen mit der GDSII Datentypnummer, an dieser Stelle *„02"* womit das Format der Daten mit INTEGER_2 festlegt wird. Abschliessend folgt, in diesem konkreten Fall eines HEADER-RECORD's, eine INTEGER_2 Zahl als Nutzdaten des RECORD's, welche die GDSII-Versionsnummer spezifiziert. Die Zahl hierbei ist *„0258"* das ist die hexadezimale Darstellung der Zahl 600, welche, neben bei bemerkt, die derzeitig höchste gültige GDSII-Versionsnummer darstellt.

Die *GDSiiRECORD[ ]* Funktionen erlauben es also, GDSII-RECORD's in Form von Hexadezimal-Strings zu erzeugen.

- *GDSiiTextual[ ]*

Die Funktion *GDSiiTextual[ ]* extrahiert die textuellen Formteile aus Listen von GDSII-RECORD's, wie sie von den noch vorzustellenden low-level GDSII-Funktionen erstellt werden.

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiTextual[RecordList]	*TextualRecordList*

- *GDSiiHexadezimal[ ]*

Die Funktion *GDSiiHexadezimal[ ]* extrahiert die hexadezimalen Formteile aus Listen von GDSII-RECORD's, wie sie von den noch vorzustellenden low-level GDSII-Funktionen erstellt werden.

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiHexadezimal [RecordList]	*HexadezimalRecordList*

- *GDSiiBinary[ ]*

Die Funktion *GDSiiBinary[ ]* leistet den Schritt der Umwandlung, einer ganzen Liste von hexadezimalen GDSII-RECORD's, in eine Liste von 1-BYTE Integerwerten. Die Funktion erhält als Parameter eine Liste von GDSII-RECORD's als Hexadezimal-Strings und liefert als Rückgabewert eine Liste von, den Hexadezimalwerten entsprechenden, 1-BYTE Integerwerten.

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiBinary[HexadezimalRecordList]	*BinaryIntegerList*

Solch eine Liste von Integerwerten kann, in einem weiteren Schritt mittels der Standard Mathematica Funktion *BinaryWrite[ ]*, in eine Datei geschrieben werden, so dass man damit in der Lage ist, gültige GDSII-Dateien zu generieren.

3.4 GDSII Exportfunktionen

Mit dieser Arbeit sollen sowohl Import-, als auch Exportfunktionen für das GDSII-Format erstellt werden. In einer ersten naiven Vorstellung scheinen diese beiden Gruppen von Funktionen von ähnlicher oder gleicher Komplexität bezüglich ihres Implementierungsaufwandes zu sein. Diese Vorstellung ist aber nicht richtig. Beim Implementieren der Exportfunktionen besteht im Prinzip jegliche Freiheit, ein Schema von Mathematica-Konstrukten zu wählen, welches in der Lage ist, konstruktiv die vom Benutzer gewünschte GDSII-Daten zu synthetisieren. Im umgekehrten Fall der Importfunktionen müssen Dateien mit de facto unbekannten GDSII-Inhalten analysiert werden. Dabei kann bestenfalls von der Annahme ausgegangen werden, dass es sich zumindest um korrekte GDSII-Daten handelt. Es muss aber jegliche nur denkbare gültige GDSII-Datei korrekt analysiert und importiert werden können. Doch auch im Fall von Dateien mit nicht korrektem GDSII-Inhalt werden zumindest aufschlussreiche Fehlermeldungen durch die Importfunktionen erwartet. Mit der Erkenntnis, dass die Exportfunktionen wohl leichter zu implementieren sind, wird zunächst mit der Betrachtung eben dieser Funktionen begonnen

Ziel der Exportfunktionen ist es, dem Nutzer ein einfach zu handhabendes Werkzeug zur Verfügung zu stellen, mit welchen er in die Lage versetzt wird, geometrische Gebilde nach seinen Vorstellungen in GDSII-Daten umzusetzen.

Aufbauend auf den zuvor eingeführten Hilfsfunktionen kann nun der notwendige Code für den Aufbau von gültigen GDSII-Daten, implementiert werden. Für die Erzeugung von GDSII-Daten mittels Mathematica wird ein einfaches Schema gewählt. Das Schema sieht im Wesentlichen zwei Kategorien von Funktionen vor. Die erste dieser beiden Kategorien, ist eine Reihe von zu implementierenden low-level Funktionen, welche es ermöglichen alle notwendigen Typen von GDSII-RECORD’s zu erstellen. Dies erlaubt eine einfache und übersichtliche 1:1 Entsprechung, zwischen jeweils einem im GDSII-Format vorgesehenen GDSII-RECORD und einer geeignet implementierten Mathematica Funktion, welche im Stande ist, den betreffenden RECORD zu generieren.

Die zweite Kategorie von Funktionen welche implementiert werden, erhalten treffender Weise die Bezeichnung higher-level Funktionen. Die higher-level Funktionen bauen auf den low-level Funktionen auf und stellen, wie später noch ersichtlich wird, eine höhere Funktionalität mit besserem Komfort (als die low-level Funktionen) zur Verfügung.

- *GDSii[ ]*

Die low-level Funktionen werden nach einem einfachen Muster entworfen: Diese Funktionen erhalten alle den gleichen Namen. Dieser Funktionsnamen lautet: *GDSii*. Das gewählte Muster macht vom Konzept des „Method Overloading“ Gebrauch, hierbei können mehrere Funktionen (Method’s) mit dem gleichen Funktionsnamen definiert werden, wobei sich die Funktionen allerdings durch ihre akzeptierten Parameter unterscheiden müssen (in der Softwareentwicklung spricht man in diesem Zusammenhang von der Signatur einer Funktion). Jede der implementierten *GDSii[ ]* Funktionen liefert als Funktionsrückgabewert eine Mathematica-Liste mit zwei Elementen. Beide Elemente stellen den gleichen GDSII-RECORD dar, allerdings in zwei verschiedenen Formen. Das erste Element ist eine textuelle, für den menschlichen Nutzer leicht lesbare Form des GDSII-RECORD, das zweite Element stellt denselben RECORD als hexadezimalen String dar. In diesem Dokument wird im Allgemeinen die textuelle Form, also das erste Element dargestellt. Um die Lesbarkeit der textuellen Form noch weiter zu erhöhen, werden ausgewählten generierten RECORD’s jeweils geeignete Leerzeilen oder Leerzeichen-Einrückungen vorangestellt.

Das hexadezimale Element hingegen dient als Vorstufe für die binäre Form wie sie für ein gültiges GDSII-Format notwendig ist.

Funktionsaufruf und Funktionsrückgabewert sind wie folgt:

Funktionsaufruf	**Funktionsrückgabewert**
GDSii[RecordTypeString, Data]	*{TextualGDSIIRECORD, HexGDSIIRECORD}*

Die gewählten Definitionen der low-level GDSii Funktionen haben im Allgemeinen immer folgende Form, wobei der Implementierungs-Code nicht explizit gezeigt wird und nur durch „Pünktchen“ angedeutet ist: *GDSii[RecordTypeString_, Data_] := …*

Im Speziellen schauen die Definitionen für die low-level Funktionen beispielhaft wie folgt aus: *GDSii[„HEADER“, Ver_] := …*

Diese low-level Funktion generiert, wenig überraschend, einen GDSII-RECORD vom Typ HEADER, welcher als Nutzdaten die Versionsnummer *Ver* enthält. Der Aufruf, bzw. die Nutzung der Funktion sieht beispielhaft wie folgt aus:

Funktionsaufruf	**Funktionsrückgabewert**
GDSii[„HEADER“, 600]	*{HEADER 600, 000600020258}*

Der Aufruf generiert einen GDSII-RECORD vom Typ HEADER mit der GDSII-Versionsnummer 600 als Nutzdaten. Man beachte das zweiteilige Ergebnis: zum Einem die textuelle, für den Benutzer lesbare Form, zum anderen die Form als hexadezimalen String. Für die weitere Nutzung des generierten GDSII-RECORD's wird die passende Form, textuell oder hexadezimal, ausgewählt.

Als weiteres Beispiel dient die Funktion zur Erstellung von XY GDSII-RECORD's:
GDSii[„XY“, KoordList_] := …

Diese Funktion generiert einen GDSII-RECORD vom Typ XY. Dieser RECORD-Type findet Verwendung, um beispielsweise Listen von Koordinatenpaaren zur Spezifikation von Polygoneckpunkten zu erzeugen. Folgend wird ein Beispiel für die Verwendung der Funktion angegeben, um die Eckpunkte eines Quadrates zu spezifizieren:

Funktionsaufruf	**Funktionsrückgabewert (textuelle Form)**
GDSii[„XY“, {0,0},{1,0},{1,1},{0,1},{0,0}}]	*XY {0,0},{1,0},{1,1},{0,1},{0,0}}*

Man beachte, dass wie oben bereits erwähnt und aus Gründen der Übersichtlichkeit, nur die textuelle Form des Funktionsrückgabewertes dargestellt ist. Generiert wird ein XY GDSII-RECORD, welcher ein im Koordinatenursprung 0/0 positioniertes Quadrat mit der Seitenlänge 1 spezifiziert. Angemerkt sei hier, dass Polygone in GDSII immer geschlossen spezifiziert werden müssen, aus diesem Grund sind der erste und der letzte Polygoneckpunkt immer identisch. Wie aus den beiden obigen Beispielen ersichtlich ist, unterscheiden sich die low-level *GDSii[ ]* Funktionen immer durch ihren ersten Parameter, welcher als String den symbolischen GDSII-RECORD-Namen des zu erstellenden RECORD's beinhaltet.

In Analogie zu den beiden vorgestellten Funktion, wird zu jedem der möglichen GDSII-RECORD's je eine *GDSii[ ]* Funktion definiert bzw. implementiert.

Um die Erstellung von GDSII-Daten zu erleichtern werden, wie bereits erwähnt, aufbauend auf den low-level Funktionen, eine Reihe von higher-level Funktionen realisiert. Wie schon aus den früher vorgestellten Syntaxdiagrammen bzw. der BNF Definition des GDSII-Formates ersichtlich ist, gibt es eine wiederkehrende Vorzugsreihenfolge in der Abfolge von

GDSII-RECORD's innerhalb des Formates. Genau dieser Umstand kann vorteilhaft für die Implementierung der higher-level Funktionen genutzt werden. Diese sind nämlich im Stande die erwähnten, oftmals gebrauchten Abfolgen von GDSII-RECORD's zu erzeugen.

Es werden insgesamt die folgenden higher-level Funktionen implementiert:

Funktion (higher-level)	**Erzeugte GDSII-RECORD's Abfolgen**
GDSiiBGNLIB[ ]	HEADER, BGNLIB, LIBNAME, UNTS
GDSiiBOUNDARY[ ]	BOUNDARY, LAYER, DATATYPE, XY, ENDEL
GDSiiBOX[ ]	BOX, LAYER, BOXTYPE, XY, ENDEL
GDSiiPATH[ ]	PATH, LAYER, DATATYPE , PATHTYPE, WIDTH, XY, ENDEL
GDSiiNODE[ ]	NODE, LAYER, NODETYPE, WIDTH, XY, ENDEL
GDSiiTEXT[ ]	TEXT, LAYER, TEXTTYPE, WIDTH, XY, STRING, XY
GDSiiSREF[ ]	SREF, SNAME, XY, ENDEL
GDSiiAREF[ ]	AREF, SNAME, COLROW, XY, ENDEL
GDSiiSTRANS[ ]	STRANS, MAG, ANGLE

Tabelle 10 „higher-level Funktionen"

Die aufgelisteten Funktionen werden jeweils mit verschiedenartigen Parametern aufgerufen. Als Ergebnis liefern sie jedoch allesamt jeweils eine Abfolge von mehreren GDSII-RECORD's in Form einer Mathematica *Sequence,* um so einen längeren Abschnitt einer GDSII-Datei zu erzeugen.

Beispielhaft wird die Funktion *GDSiiBOUNDARY[ ]* näher vorgestellt. Diese Funktion ergänzt die low-level Funktion *GDSii[„BOUNDARY"]* und ermöglicht die vollständige Spezifikation eines GDSII-STRUCTURE-ELEMENT- BOUNDARY, welches in der EBNF als „boundary" bezeichnet ist. An dieser Stelle ist es hilfreich, die EBNF der „boundary" zu wiederholen:

```
boundary =  BOUNDARY [EFLAGS][PLEX] LAYER DATATYPE XY.
```

Aus der EBNF ist ersichtlich, dass eine „boundary“ immer, zumindest die folgenden RE-CORD’s in genau der angegebenen Reihenfolge enthalten muss:

BOUNDARY, LAYER, DATATYPE, XY. Nachdem „boundary“ wie aus der vollständigen EBNF unter: Tabelle 5 „EBNF des GDSII-STREAM-FORMAT“ ersichtlich ist, immer Teil eines „element“ ist, wird als Abschluss zwingend auch noch ein ENDEL-RECORD benötigt. All dem wird die folgende Implementierung der Funktion *GDSiiBOUNDARY[ ]* gerecht:

```
GDSiiBOUNDARY[Poly_List,Layer_Integer:0,Datatype_Integer:0] :=
Sequence@@{GDSii["BOUNDARY"],
        GDSii["LAYER", Layer],
        GDSii["DATATYPE", Datatype],
        GDSii["XY", Poly],
        GDSii["ENDEL"]
}
```

Tabelle 11 „GDSiiBOUNDARY[ ]“

Wie an Hand der Definition ersichtlich ist, akzeptiert die Funktion bis zu drei Parameter, wovon der erste obligatorisch angegeben werden muss, die restlichen zwei sind optional. Bei den optionalen Parametern handelt es sich um die LAYER-Nummer und die DATATYPE-Nummer. Beide Nummern werden defaultmäßig mit 0 festgelegt sofern sie nicht explizit spezifiziert werden. Beispielhaft wird die Verwendung der Funktion für die Generierung eines einfachen Quadrates gezeigt:

- *GDSiiBOUNDARY[{{0,0},{1,0},{1,1},{0,1},{0,0}}]*

Wie an Hand der zuvor gelisteten Implementierung ersichtlich, bewirkt der Aufruf der Funktion seinerseits die Aufrufe der notwendigen *GDSi[ ]* Funktionen, welche jeweils genau einen GDSII-RECORD generieren. In der Implementierung werden die einzelnen RECORD’s zu einer Mathematica-*Sequence* zusammengefasst und stehen als solche in weiterer Folge bereit, um zusammen mit allfälligen weiteren GDSII-RECORD’s eine Gesamt-Liste aller GDSII-RECORD’s des zu erstellenden Datensatzes zu bilden.

Ergebnis des Funktionsaufrufes für das angegeben Beispiel ist also die folgende *Sequence* von GDSII-RECORD’s mitsamt deren zugehörigen Nutzdaten:

Funktionsaufruf	*GDSiiBOUNDARY[{{0,0},{1,0},{1,1},{0,1},{0,0}}]*
Erzeugte GDSII-RECORD’s	BOUNDARY LAYER 0 DATATYPE 0 XY {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}} ENDEL

Die implementierten Funktionen ermöglichen es nun eine erste vollständige GDSII-Datei zu erzeugen. Im folgenden Codelisting ist zu erkennen, wie die Exportfunktionen eingesetzt werden. Wie bereits erwähnt, wird die zu erstellende GDSii-Datei zunächst als Mathematica-Liste angelegt, jedes Listenelement steht hierbei für einen zu generierenden GDSII-RECORD. Die RECORD's bzw. Listenelemente werden durch Aufruf der jeweiligen low- bzw. high-level GDSII-Funktionen erzeugt.

```
GDSiiTestData = {
        GDSiiBGNLIB[ ],

        GDSiiBGNSTR["UnitSquare"],
                GDSiiBOUNDARY[{{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}],
        GDSii["ENDSTR"],

        GDSii["ENDLIB"]
};
```

Tabelle 12 „Ein einfaches Quadrat"

Durch Ausführung des obigen Code-Listings durch den Mathematica Kernel, wird dem als Variable verwendeten Symbol *GDSiiTestData*, eine Liste von GDSII-RECORD's zugewiesen. Die Listenelemente entstehen hierbei einerseits einzeln durch Aufrufe der low-level Funktion, im Beispiel sind das: *GDSii["ENDSTR"]* sowie *GDSii["ENDLIB"]*, andererseits entstehen ganze Sequenzen von Listelementen durch Aufruf der higher-level Funktionen, im Beispiel sind das: *GDSiiBGNLIB[ ]*, *GDSiiBGNSTR[ ]* und *GDSiiBOUNDARY[ ]*.

Wie bereits beschrieben werden die GDSII-RECORD's in zwei Formen erstellt, der textuellen und der hexadezimalen Form. Um einen Eindruck von den erstellten RECORD's zu bekommen, kann deren textuelle Form extrahiert werden.

Die Extraktion erfolgt in dem die Funktion *GDSiiTextual[ ]* angewandt wird. Die so gewonnene textuelle Form kann mittels der Mathematica Std.-Funktion *TableForm* in Tabellenform dargestellt werden:

```
GDSiiTestData//GDSiiTextual//TableForm
```

Tabelle 13 „GDSiiTextual[ ]"

Die obige Code-Zeile führt zur Generierung der gewünschten GDSII-RECORD's und der textuellen Anzeige der insgesamt 13 generierten RECORD's wie nachfolgend gezeigt:

```
HEADER 600
BGNLIB
LIBNAME
UNITS 1.e-6 1.

BGNSTR
STRNAME UnitSquare
BOUNDARY
  LAYER 0
  DATATYPE 0
  XY {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}
  ENDEL
ENDSTR
ENDLIB
```

Tabelle 14 „Textuelle GDSII Form“

Eine gültige GDSII-Datei zu erzeugen gelingt letztendlich mit folgendem Mathematica-Code:

```
GDSiiTestData //
GDSiiHexadezimal //
GDSiiBinary //
BinaryWrite["SquareTest.GDS",#]& //
Close
```

Tabelle 15 „Erstellen einer binären GDSII-Datei“

Es werden im obigen Codestück aus den generierten GDSII-Daten *GDSiiTestData* zunächst mittels *GDSiiHexadezimal[ ]* die hexadezimalen Formen der enthaltenen GDSII-RECORD’s extrahiert, diese werden dann mittels *GDSiiBinary[ ]* in Binärform gewandelt und schließlich mittels *BinaryWrite[ ]* (Mathematica Std.-Funktion) als gültiges GDSII-File Namens „SquareTest.GDS“ gespeichert, welches abschliessend ordnungsgemäß mittels *Close[ ]* (Mathematica Std.-Funktion) geschlossen wird. Die obige Abfolge von Funktionsaufrufen im Mathematica Postfix-Operator Stil ist natürlich in einer weiteren Funktion zusammengefasst:

- *GDSiiExport[ ]*

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiExport[GDSiiRecordList, GDSiiFile]	*GDSiiFile*

Diese Funktion erwartet zwei Parameter, der erste ist eine gültige Liste von GDSII-RECORD’s wie sie mit den low- bzw. high-level Funktionen erstellt worden ist. Der zweite Parameter ist der Dateiname für die zu erstellende GDSII-Datei. Für Diagnosezwecke liefert die Funktion als Rückgabewert wieder den Dateinamen zurück, nachdem diese erfolgreich geöffnet, geschrieben und abschließend geschlossen wurde.

Mit Abschluss des Kapitels wird gezeigt, wie kompakt mit der Funktion eine GDSii-Datei, mit Namen „SquareTest.GDS“ mit dem bereits vertrauten Quadrat als Inhalt erzeugt werden kann:

```
GDSiiExport[
{
        GDSiiBGNLIB[ ],

        GDSiiBGNSTR["UnitSquare"],
                GDSiiBOUNDARY[{{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}],
        GDSii["ENDSTR"],

        GDSii["ENDLIB"]
},

"SquareTest.GDS";
];
```

Tabelle 16 „Ein GDSII Quadrat“

In Abbildung 6 „Export-Processflow“ werden die Schritte für den GDSII-Export zusammengefasst dargestellt. Der Vorgang für den Export besteht aus zwei Teilen. Zunächst definiert der Nutzer im ersten Schritt unter Zuhilfenahme der vielfältigen GDSii-RECORD-Funktionen eine Liste von GDSII-RECORD’s. Im zweiten Schritt erfolgt entweder die textuelle Darstellung der generierten GDSII-RECORD’s mittels der *GDSiiTextual[ ]* Funktion, oder die RECORD’s werden mit Hilfe der *GDSiiExport[ ]* Funktion in ein GDSII-File exportiert.

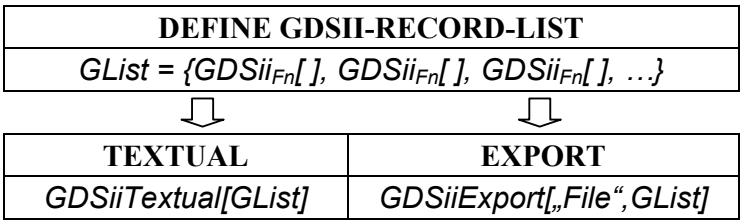

Abbildung 6 „Export-Processflow“

3.5 Import und Analyse von GDSII-Daten

Wie bereits erwähnt, ist das Problem des Imports de facto unbekannter Daten, eine deutlich komplexere Aufgabe als im Vergleich dazu die Erstellung und der Export von Daten nach bestimmten Vorstellungen eines Nutzers. Daten, welche mit Hilfe der Importfunktionen, eingelesen werden können, sind vorab im Detail zwar unbekannt, doch folgen sie, sofern kein Datenfehler vorliegt, den Spezifikationen des GDSII-Formates. Damit folgen die Inhalte korrekter GDSII-Dateien auch der GDSII-EBNF. [7], [8], [9].

Wie schon früher dargelegt, beschreibt die GDSII-EBNF eine formale Sprache. Somit können korrekte GDSII-Daten als korrekte Sätze dieser Sprache angesehen werden. Der Vorgang der Analyse eines Satzes einer formalen Sprache, wird in der Informatik als „Parsen“ bezeichnet. Ein Werkzeug für einen Parsvorgang, wird folglich als Parser bezeichnet. Das bisher gesagte legt nahe, dass ein Parser für das GDSII-Format einen zentralen Bestandteil der Importfunktionen darstellen wird.

Nun verhält es sich mit Parsern in der Regel so, dass diese die vorliegenden Daten, in unserem Fall binäre GDSII-Daten, nicht direkt analysieren können. Um einen Parser einsetzen zu können, müssen aus dem vorliegenden Datensatz erst die so genannten Worte, auch Token genannt, aus dem Satz extrahiert werden. Die Informatik bezeichnet ein Werkzeug, welches die Extraktion dieser Token bewerkstelligt, als Lexical-Analyzer oder kurz Lexer genannt. Die Token die ein Lexer aus den Daten extrahiert, sind genau die Terminale wie sie in der EBNF spezifiziert sind. In unserem Fall des GDSII-Format sind die Token, die einzelnen GDSII-RECORD’s, welche in den zu analysierenden Daten enthalten sind. Folglich wird die Aufgabe des Lexer’s jene sein, die Abfolge der einzelnen GDSII-RECORD’s aus den binären Daten zu extrahieren. Die so gewonnene Abfolge der GDSII-RECORD’s oder Token wird dann vom Parser in weiterer Folge analysiert. Leider ist mit der oben angedeuteten Analyse durch den Parser die Aufgabe noch nicht beendet, mit der Analyse wurde lediglich Kenntnis über die Struktur und Korrektheit der zu analysierenden Daten bezüglich des GDSII-Formates erlangt.

Neben Parser und Lexer wird noch eine dritte wichtige Komponente benötigt um die Importfunktionen erfolgreich implementieren zu können. Die noch ausständige Komponente ist ein Softwarewerkzeug, welches in der Informatik als Codegenerator bezeichnet wird. Der Codegenerator ist jene Komponente, die letztendlich das Ergebnis der Analyse in der entsprechend gewünschten übersetzten Form liefert. In unserem Fall wird das gewünschte Ergebnis des Codegenerators die Abbildung der analysierten GDSII-Daten auf entsprechende Mathemati-

ca-Graphics Object's sein. Diese Graphics Object's können mit dem Mathematica-System grafisch dargestellt oder aber auch weiterverarbeitet und manipuliert werden.

An dieser Stelle sei festgestellt, dass die Tools: Lexer, Parser und Codegenerator in der Informatik zusammengefasst als Compiler, auch Übersetzer genannt, bezeichnet werden. Für Importfunktionen steht also die Aufgabe an, einen Compiler für die Übersetzung vom GDSII-Format nach Mathematica-Graphics zu implementieren.

Nachfolgend wird die Implementierung der notwendigen GDSII-Compiler Komponenten vorgestellt.

3.5.1 Implementierung eines GDSII-Lexer

Die Aufgabe des Lexer ist es, wie bereits erwähnt, die Zerlegung des vollständigen Eingangsdatenstromes (formaler Satz) in einzelne Token (formale Wörter des Satzes) durchzuführen, um sie zur weiteren Analyse an den Parser weiterzureichen. Der zu implementierende GDSII-Lexer muss also aus den GDSII-Eingangsdaten die einzelnen Token, in unserem Fall GDSII-RECORD's, extrahieren und in geeigneter Form dem Parser als Token zur Verfügung stellen. Zur Erinnerung sei das Format der GDSII-Daten noch einmal kurz angerissen: GDSII-Daten bestehen aus einer Abfolge von GDSII-RECORD's von variabler Länge. Der binäre Aufbau eines GDSII-RECORD's, ist immer wie folgt:

GDSII-RECORD			
Byte[0] Byte[1]	Byte[2]	Byte[3]	Byte[4..RecordLength-1]
RecordLength	RecordType	DataType	RecordData

Tabelle 17 „Format der GDSII-RECORD's"

Die binären Daten haben dabei die folgende Bedeutung:

- RecordLength Gesamtlänge des GDSII-RECORD's
- RecordType Angabe des GDSII-RECORD Type
- RecordType Angabe des GDSII-RECORD Daten-Type
- RecordData Die Nutzdaten des GDSII-RECORD's

Die gewählte Implementierung des Lexer führt folgende Aktionen der Reihe nach aus:

- Lesen von Byte[0] und Byte[1] des RECORD's und Interpretation dieser Byte's als 16BIT Integer RecordLength welche die Gesamtlänge des RECORD's in Byte angibt.
- Lesen des Byte[2], dieses liefert den RecordType.
- Lesen des Byte[3], dieses liefert den DataType.
- Lesen von Byte[4] .. Byte[RecordLength-1], diese liefern die RecordData. Zu beachten ist hierbei, dass manche RECORD's keine RecordData enthalten, diese RECORD-Typen besitzen alle die RecordLength = 4 und stellen die kleinsten GDSII-RECORD's dar.
- Aufbau des Token aus zwei Teilen. Teil eins besteht aus dem textuellen RecordType-Namen, welcher aus der Interpretation des binären RecordType gewonnen wird. Teil zwei wird aus RecordData nach entsprechender Interpretation anhand des DataType gewonnen.
- Vorbereitung des Lesevorganges für den nächsten GDSII-RECORD.
- Übergabe des Token als Rückgabe des Lexer an den Parser.

Durch die geschilderte Vorgangsweise bereitet der Lexer die notwendigen Informationen, bestehend aus dem Namen des RecordType sowie den zum jeweiligen RECORD gehörigen Daten, in für den Parser geeigneter Form, auf.

Der Lexer wird in der Implementierung als die Funktion mit Namen *GDSiiRECORDRead[ ]* realisiert. Die Funktion erwartet beim Aufruf einen Parameter, es handelt sich hierbei um das zu analysierende GDSII-File. Als Rückgabewert liefert die Funktion den aktuell gelesenen RECORD als zweiteiliges Token in Form einer zwei-elementigen Mathematica-Liste. Um ihre Aufgabe erfüllen zu können ruft die Funktion ihrerseits unter anderem die Mathematica Std.-Funktion *BinaryRead[ ]* auf um die verschiedenen GDSII-RECORD Bestandteile wie z.B. RecordLength, einzulesen und diesem Fall entsprechend als 16BIT Integerzahl zu interpretieren.

- *GDSiiRECORDRead[ ]*

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiRECORDRead[GDSiiFile]	*{RecordTypeName, RecordData}*

3.5.2 Implementierung eines GDSII-Parsers

Die Theoretische Informatik kennt eine Reihe von Techniken zur Realisierung eines Parsers. Einen verhältnismäßig einfachen Zugang bietet hier wohl der so genannte Recursive-Descent-Parser, kurz RDP genannt. Dieser Parser Type baut auf der formellen Definition der Grammatik der zu übersetzenden Sprache auf. In unserem Fall ist das die GDSII-EBNF. Die Implementierung eines RDP geschieht nach im folgend beschriebenem Muster.

Die Meta-Syntax einer EBNF besteht aus einer Sammlung von Ableitungsregeln. Jede Ableitungsregel hat die Form: $a = B$. Hierbei spezifizieren a ein Nonterminal und B eine Ersetzungsregel für a. Die Ersetzung B besteht hierbei aus einer gemischten Abfolge von weiteren Nonterminalen oder Terminalen. Ausgehend von der Startableitungsregel werden sukzessive alle vorhandenen Nonterminale durch Anwendung jeweils geeigneter Ableitungsregeln ersetzt. Auf diese Art und Weise kann jeder gültige Satz der durch die EBNF definierten Sprache abgeleitet bzw. generiert werden. In unserem Fall der GDSII-BNF kann so jeder gültige GDSII-Datensatz generiert werden. Eng angelehnt an dieses Muster wird eine ganze Reihe von Funktionen implementiert, welche das Regelwerk der GDSII-EBNF nachbilden. Bei der Implementierung wird für jedes der Nonterminale der GDSII-EBNF jeweils eine Ableitungsfunktion erstellt. Aufgabe der jeweiligen Ableitungsfunktion ist es, das zugehörige Nonterminal solange abzuleiten, bis alle Nonterminale aufgelöst sind. Eine Ableitungsfunktion leitet das betreffende Nonterminal ab, indem es selbiges durch die rechten Seiten der betreffenden Ableitungsregel ersetzt. Hierbei produzieren Nonterminale, welche in den rechten Seiten der Ableitungsregeln vorkommen, ihrerseits wechselweise Funktionsaufrufe auf weitere Ableitungsfunktionen. Durch diese wechselseitigen Aufrufe, kann es im Allgemeinen auch zu rekursiven Aufrufen kommen. Terminale in den rechten Seiten der Ableitungsregeln veranlassen den Parser dazu, mit Hilfe des Lexer das nächste Token aus den Daten zu lesen und abzuarbeiten.

Die Funktionsweise des GDSII-Parsers ist zusammengefasst also wie folgt:

Ausgehend von der Start Ableitungsregel/Ableitungsfunktion versucht der Parser die Regel abzuleiten, indem er die Teile der rechten Seite der Regel in einer Folge von Schritten abarbeitet. Das Abarbeiten der Schritte hat für den Fall, dass der Regelteil ein Terminal ist, zur Folge, dass der Parser versucht, das nächste Token zu lesen. In unserem Fall wird also der nächste in den Daten enthaltene GDSII-RECORD gelesen. Die Regelteile welche Nonterminale spezifizieren, führen ihrerseits zum Aufruf der Ableitungsfunktion für eben dieses Nonterminal. Auf diese Art und Weise wird jedes Mal, wenn in den Ableitungsregeln auf Termi-

nale gestoßen wird, ein weiterer GDSII-RECORD gelesen und verarbeitet. Für den Fall von korrekten GDSII-Daten werden dadurch alle Nonterminale der aufgerufenen Ableitungsfunktionen aufgelöst und damit einhergehend alle in den Daten enthaltenen GDSII-RECORD's gelesen und abgearbeitet. Ausgehend von der Start Ableitungsregel/Funktion kommt es so zum möglicherweise rekursiven Abstieg in die weiteren Ableitungsregeln/Funktionen. Dieses Verhalten ist letztlich für die Namensgebung des RDP verantwortlich.

Für die Implementierung des Parsers werden hauptsächlich zwei Typen von Funktionen entwickelt. Zum Einem werden das die oben beschriebenen Ableitungsfunktion sein, welche mit *Descent[ ]* benannt werden, diese werden die Nonterminale bedienen. Zum Anderen werden Funktionen benötigt welche die Terminale bedienen, diese werden weiters in die Funktionen *Expect[ ]* bzw. *Accept[ ]* aufgeteilt.

Entwickelt wird also für jedes Nonterminal genau eine *Descent[ ]* Funktion welche einen Parameter besitzt, der das betreffende Nonterminal in Form eines Kleinbuchstaben-String spezifiziert. Genau genommen dient *Descent[ ]* den einfachen obligatorischen Teilen der EBNF. In der GDSII-EBNF kommen aber teilweise auch optionale, sich wiederholende Elemente vor, für diese Varianten werden die Funktionen *DescentZeroOrOnce[ ]* und *DescentZeroOrMulti[ ]* entworfen.

Wie schon bei den Exportfunktionen wird vom Konzept des „Method Overloading" Gebrauch gemacht, in dem unterschiedliche *Descent[ ]* Funktion implementiert werden, welche sich wie bereits erwähnt durch ihren jeweiligen Parameter unterscheiden.

Für die Terminale wird ähnlich verfahren, für jedes Terminal wird jeweils eine *Expect[ ]* bzw. *Accept[ ]* Funktion implementiert, wobei das zutreffende Terminal in Form eines Großbuchstaben String Parameter spezifiziert wird.

Um das gewählte Konzept besser zu veranschaulichen, wird an dieser Stelle der Code der *Descent["streamformat"]* Funktion für die Start Ableitung präsentiert:

```
Descent["streamformat"]:=Module[{},
        Expect["HEADER"];
        Expect["BGNLIB"];
        Expect["LIBNAME"];
        Accept["REFLIBS"];
        Accept["FONTS"];
        Accept["ATTRTABLE"];
        Accept["GENERATIONS"];
        DescentZeroOrOnce["formattype"];
        Expect["UNITS"];
        DescentZeroOrMulti["structure"];
        Expect["ENDLIB"];
];
```

Tabelle 18 „Descent[„streamformat“]“

An Hand der Implementierung sieht man sehr gut, wie die Struktur der GDSII-EBNF durch die Funktion nachgebildet wird. Zu Beachten ist, dass obiges Listing in der gezeigten Form lediglich in der Lage ist, die Daten korrekt zu parsen und damit deren Struktur zu analysieren, es wird hierbei noch kein Code generiert, dazu fehlt noch der früher erwähnte Codegenerator.

An dieser Stelle wird nun genauer dargestellt, welche Verarbeitungsschritte durch den Aufruf der Funktion *Descent["streamformat"]* ausgelöst werden:

1. *Expect["HEADER"]*
   wird aufgerufen welches an dieser Stelle zwingend HEADER als nächsten GDSII-RECORD in den Daten erwartet.
2. *Expect["BGNLIB"]* (analog zu Punkt 1.)
3. *Expect["LIBNAME"]* (analog zu Punkt 1.)
4. *Accept["REFLIBS"]*
   wird aufgerufen welches an dieser Stelle optional REFLIBS als nächsten GDSII-RECORD in den Daten erwartet.
5. *Accept["REFLIBS"]* (analog zu Punkt 4.)
6. *Accept["FONTS"]* (analog zu Punkt 4.)
7. *Accept["ATTRTABLE"]* (analog zu Punkt 4.)
8. *Accept["GENERATIONS"]* (analog zu Punkt 4.)
9. *DescentZeroOrOnce["formattype"]*
   wird aufgerufen, damit „steigt“ der Parser eine Stufe hinab, in dem die Kontrolle über den weiteren Parseablauf an *DescentZeroOrOnce["formattype"]* übergeben wird. Erst wenn *DescentZeroOrOnce["formattype"]* abgeschlossen wird, kehrt die Kontrolle an die rufende Funktion *Descent["streamformat"]* zurück, womit der Parser wieder eine Stufe „zurücksteigt“.
10. *Expect["UNITS"]* (analog zu Punkt 1.)
11. *DescentZeroOrMulti["structure"]* (analog zu Punkt 9.)
12. *Expect["ENDLIB"]* (analog zu Punkt 1.)
13. Die Start Ableitungsfunktion wird beendet, der Parse-Vorgang ist abgeschlossen.

Der Aufbau und die Implementierung der weiteren *Descent[ ]* Ableitungsfunktionen erfolgt analog, in ähnlicher Art und Weise wie jener von *Descent["streamformat"]* und wird an dieser Stelle nicht weiter erläutert.

Stellvertretend für die *Accept[ ]* bzw. *Expect[ ]* Funktionen wird beispielhaft auch der Code einer dieser Funktionen vorgestellt:

```
Accept[accept:"BGNSTR"|"STRCLASS"|"BOUNDARY"|"PATH"|"SREF"|
  "AREF"|"TEXT"|"NODE"|"BOX"|"STRANS"|"MAG"|"ANGLE"]:=
Module[{},
        GetSym[ ];
        If[First@GDSiiSym == accept, True, UngetSym[ ];False]
]
```

Tabelle 19 „Accept[ ]“

Diese Funktion dient zur Bereitstellung von Token für eine ganze Reihe von GDSII-RECORD’s, wie z.B. BGNSTR und anderen. Wird nun beispielhaft *Accept["BGNSTR"]* im Laufe des Pareseprozesses aufgerufen, so geschieht Folgendes: Die Funktion liest zunächst über den Aufruf der Hilfsfunktion *GetSym[ ]* das nächste Token aus den Daten. Anschließend wird das gelesene Token, ein GDSII-RECORD, mit dem im Moment optional erlaubten GDSII-RECORD BGNSTR verglichen. Ergibt der Vergleich, dass genau dieser RECORD gelesen wurde, dann retourniert die Funktion den Wahrheitswert *True* um anzuzeigen: Das optional erlaubte Token BGNSTR, konnte an dieser Stelle erfolgreich gelesen werden. Für den Fall, dass ein anderes Token als BGNSTR gelesen wurde, wird das eben gelesene Token mittels der Hilfsfunktion *UngetSym[ ]* „ungelesen“ gemacht, so dass es für später folgende *GetSym[ ]* Leseversuche erneut zur Verfügung steht. Abschließend retourniert die Funktion den Wahrheitswert *False*, um anzuzeigen, dass an dieser Stelle kein BGNSTR Token gelesen wurde. Der retournierte Wahrheitswert, welcher über das erfolgreiche Lesen des Token informiert, wird in weiterer Folge von der aufrufenden Funktion ausgewertet und dient dort der weiteren Steuerung des Parse-Vorganges.

An dieser Stelle wird der Parse-Vorgang nochmals zusammen gefasst:

Der Vorgang startet durch Aufruf und Ausführung der *Descent["streamformat"]* Funktion, welche die Start EBNF Ableitungsregel nachbildet. Diese Funktion wiederum ruft weitere Funktionen der implementierten *Descent[ ]*, *Accept[ ]* und *Expect[ ]* Funktionsgruppen auf, welche sich wiederum wechselseitig aufrufen. Während dieser Abfolge von Funktionsaufrufen werden durch die *Accept[ ]* und *Expect[ ]* Funktionen jeweils Aufrufe des Lexer getätigt. Dadurch werden alle Token, in unserem Fall alle in den Daten enthaltenen GDSII-RECORD’s, in sequentieller Reihenfolge gelesen.

Im Falle der *Expect[ ]* Funktion ist streng festgelegt, welche Art von Token gerade erwartet, bzw. im Moment erlaubt wird. Für den Fall, dass an so einer Stelle nicht das erwartete Token durch den Lexer geliefert wird, kommt es mit einer Fehlermeldung zum Abbruch des Parse-Vorganges, andernfalls wird der Vorgang fortgesetzt.

Im Falle der *Accept[ ]* Funktionen wird eines aus einer Reihe verschiedener Token erwartet und gelesen. In Abhängigkeit des an so einer Stelle erkannten Token kommt es dann zu unterschiedlichen weiteren Aufrufen der Parse-Funktionen. Genau durch diese, von den jeweils gelesenen Token bzw. GDSII-RECORD's gesteuerte Abfolge von Parse-Funktion Aufrufen, wird die Struktur der gelesen Daten analysiert bzw. erkannt. Jede prinzipiell unterschiedliche, vom Parser gelesene Datei führt also, gesteuert durch die sie beinhaltenden GDSII-RECORD's, zu einer unterschiedlichen Abfolge von Parse-Funktionsaufrufen. Diese möglichen Abfolgen spiegeln die jeweilige Struktur der Daten wieder.

Die entstehende Struktur, welche sich durch den wechselweisen Aufruf der Parse-Funktionen ergibt, lässt sich sehr gut durch deren Ableitungsbaum verdeutlichen. Ein Ableitungsbaum stellt die Struktur der Ableitung eines gültigen Satzes in Form eines baumförmigen Graphen, bestehend aus Knoten, welche mittels Kanten verbunden sind, dar. Hierbei wird der Anfangsknoten auch als Wurzel bezeichnet. Endknoten, welche nur über eine Kante mit vorangestellten Knoten verbunden sind und nicht über weitere Kanten mit nachfolgenden Knoten verbunden sind, bezeichnet man als Blätter. Die übrigen Knoten werden präzisiert als innere Knoten bezeichnet. Der Ableitungsbaum hat seinen Ursprung immer in seiner Wurzel, welche das Start Nonterminal darstellt. Die inneren Knoten stellen alle Nonterminale dar, welche im Laufe der Ableitung benötigt werden. Die Blätter schließlich stellen die Terminale, in unserem Fall die GDSII-RECORD's dar, welche im Zuge der Ableitung aus den Daten gelesen werden.

Als Anschauungsbeispiel wird hierfür Abbildung 7 „Ein Ableitungsbaum zu GDSII-Daten" gezeigt. Der Ableitungsbaum enthält die bereits bekannten GDSII-Beispieldaten, welche eine einzige GDSII-STRUCTURE beinhalten. Im Beispiel sind die inneren Knoten für die Nonterminale alle in Form einfacher Rechtecke dargestellt. Alle Terminale, also die GDSII-RECORD's, werden in Form von abgerundeten Rechtecken gezeigt. Weiterhin wird aus Gründen der Anschaulichkeit, mittels einer gestrichelten Linie der Verlauf des Ableitungsvorganges markiert. Hierbei wird insbesondere veranschaulicht, wie der Parser im Verlauf der Ableitung tiefer in die Ableitungen/Ableitungsfunktionen absteigt und nach Abarbeitung eines Zweiges, wieder aus den Ableitungsfunktionen schrittweise zurückkehrt bzw. aufsteigt, bis letztendlich mit Ende der Ableitung wieder die Wurzel mit der Start Ableitungsfunktion erreicht ist und terminiert. Aus Gründen der Übersichtlichkeit sind die Nutzdaten der GDSII-RECORD's im Ableitungsbaum nicht dargestellt.

Zur Erinnerung, noch einmal die betreffenden Beispiel GDSII-Daten in textueller Form:

```
HEADER 600
BGNLIB
LIBNAME
UNITS 1.e-6 1
BGNSTR
STRNAME UnitSquare
BOUNDARY
  LAYER 0
  DATATYPE 0
  XY {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}
  ENDEL
ENDSTR
ENDLIB
```

Tabelle 20 „Textuelle GDSII Form von Beispieldaten“

Folgend der Ableitungsbaum zu den beispielhaften GDSII-Daten:

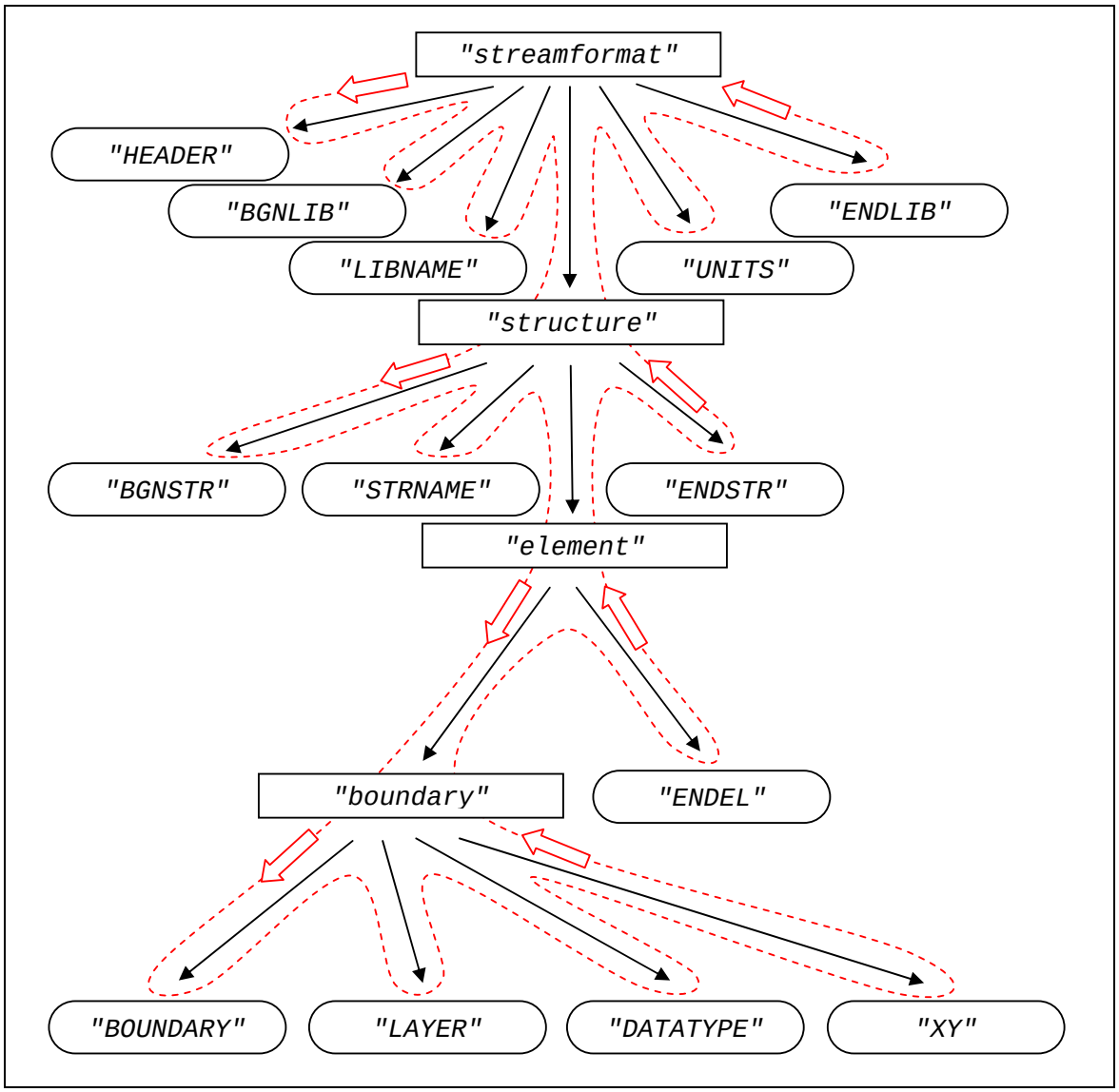

Abbildung 7 „Ein Ableitungsbaum zu GDSII-Daten“

In weiterer Folge verbleibt die Aufgabe, die erkannten bzw. analysierten GDSII-Daten in Mathematica-Grafik Objekte umzuwandeln. Diese Aufgabe übernimmt der nachfolgend beschriebene Codegenerator.

3.5.3 Implementierung eines GDSII-Codegenerators

Wie gezeigt wurde, konnten der Lexer und der Parser, eng angelehnt an ein Standard Schema der Theoretischen Informatik, entwickelt werden. Für den Codegenerator kann diese Vorgangsweise mangels verfügbarer Standardverfahren nicht gewählt werden. Anders als z.B. bei der Compilation eines Programms, welches in einer gängigen Programmiersprache etwa in C++ erstellt ist, existiert für die Umsetzung von GDSII-Format in Mathematica-Graphics, keine etablierte Methode oder Schema für die Codegenerierung.

Für den zu entwickelnden Codegenerator wird demnach jeweils die Implementierung spezieller Mathematica-Codes für die Umsetzung jedes benötigten GDSII Konstruktes in eine entsprechende Mathematica-Graphics Darstellung erforderlich sein.

Des Weiteren ist festzustellen, dass die deutliche Trennung der Implementierungen, wie sie bei Lexer und Parser möglich war, für den Codegenerator nicht mehr in gleichem Umfang gelingen wird. Vielmehr wird die Implementierung des Codegenerators darin bestehen, den Parser dahingehend zu modifizieren, dass an geeigneten Stellen Erweiterungen vorgenommen werden, welche die gewünschte Codegenerator-Funktionalität implementieren. Die geeigneten Stellen sind innerhalb der *Descent[ ]* Funktionen gegeben. Der Grund hierfür ist, dass in diesen Funktionen die Analyse und das Erkennen der GDSII-Daten stattfinden, wodurch genau hier eine passende Umsetzung in Mathematica-Graphics realisiert werden kann.

Die Implementierung des Codegenerators wird nun in Bottom-Up Weise vorgestellt, d.h. es werden zunächst die „tieferen“ Implementierungs-Hierarchien beschrieben, um dann schrittweise folgend die „höheren“ Hierarchiestufen des Codes zu erörtern. Das Konzept der Implementierung wird an dieser Stelle nicht vollständig, sondern nur in den wichtigsten Auszügen vorgestellt.

Dazu wird mit einem der wichtigsten GDSII-Elemente begonnen, es handelt sich hierbei um das *"boundary"* Element, welches für die Darstellung geschlossener Polygone benötigt wird und damit letztendlich alle entscheidenden geometrischen GDSII-Strukturen bereitstellt. Dieses Element besteht, wie bereits bekannt, aus zumindest den folgenden 4 GDSII-RECORD’s: *"BOUNDARY"*, *"LAYER"*, *"DATATYPE"*, und *"XY"*. Bei der Implementierung des Parsers haben wurde die Ableitungsfunktion *Descent["boundary"]* vorgestellt, welche zur Aufgabe

hat, ein *"boundary"* Element zu analysieren. Diese Parse-Funktion liest unter Zuhilfenahme des Lexer die vier zuvor genannten RECORD's und insbesondere auch die jeweiligen zu den RECORD's zugehörigen Nutzdaten. Bei den Nutzdaten handelt es sich hierbei insgesamt um: die Layer-Nummer, die Datatype-Nummer und die X/Y-Koordinaten-Paare des beschriebenen Polygons. Die Implementierung der Codegeneratorfunktionalität sammelt hierfür die ermittelten Nutzdaten in einer Mathematica-Liste von folgender Struktur: *{LayerNr,DatatypeNr,Polygon[{{x1,y1},{x2,y2},…,{xn,yn}}]}*. Diese Liste stellt, bis auf die Layer-Nummer und Datatype-Nummer Einträge, bereits ein gültiges Mathematica-Graphics Objekt dar. Die beiden noch störenden Nummern haben keine direkte Entsprechung in Mathematica-Graphics, welches keine Layer kennt. Dennoch kann die Layer-Nummer gut dazu verwendet werden, das erzeugte Polygon mit einer, der Layer-Nummer entsprechenden, Farbzuordnung einzufärben. Dieser Ersatz der noch störenden Nummern durch z.B. eine Mathematica Farb-Direktive wird jedoch erst zu einem späteren Zeitpunkt durchgeführt und erörtert.

Die beiden GDSII-Elemente *"sref"* und *"aref"* sind ebenfalls von großer Bedeutung für das GDSII-Format und werden durch ihre Ableitungsfunktionen *Descent["sref"]* und *Descent["aref"]* prozessiert. Diese Elemente referenzieren (verweisen), wie schon ihre Namensgebung vermuten lässt, über den Namen ihres zugehörigen SNAME GDSII-RECORD auf eine GDSII-Structure gleichen Namens, welcher an der Stelle ihrer Definition wiederum über den zugehörigen STRNAME GDSII-RECORD identifiziert wird.

Im Falle von *"sref"* handelt es sich um eine Referenz, welche in ihrem zugehörigen XY GDSII-RECORD die Koordinate spezifiziert, an welcher das betreffende Element einzufügen ist.

Im Falle von *"aref"* handelt es sich um eine so genannte Array Referenz, bei welcher das referenzierte Element mehrfach in einer rasterförmigen Anordnung eingefügt wird.

Die referenzierten Elemente können ihrerseits entweder einfache wie z.B. *"boundary"* Elemente darstellen oder aber wiederum vom *"sref"* oder *"aref"* Typ sein, wodurch sich insgesamt der hierarchische Aufbau des GDSII-Formates ergibt. Zu beachten ist hierbei, dass die referenzierten Strukturen an mehr oder weniger beliebiger Stelle in den GDSII-Daten definiert sein können, also insbesondere entweder vor oder nach der Stelle, von welcher aus sie referenziert werden.

Auch die *"sref"* und *"aref"* Elemente werden von den Codegenerierungsteilen der zugehörigen Ableitungsfunktionen solcherart prozessiert, dass eine geeignete Mathematica Struktur

zur Repräsentation der Elemente erzeugt wird. Hierbei ist zu beachten, dass für die Referenzen keine sofortige Umsetzung in Mathematica-Grafiken erfolgen kann, da die referenzierten Strukturen möglicherweise erst an späterer Stelle in den Daten definiert sind und somit erst im weiteren Verlauf des Parse-Vorganges gelesen werden. Somit ist im Allgemeinen erst nach Abschluss des gesamten Parse-Vorganges eine Auflösung aller Referenzen in Mathematica-Grafiken möglich.

Die mittels der Ableitungsfunktionen für die GDSII-Elemente wie z.B. *"boundary"*, *"sref"*, *"aref"*, etc. generierten Mathematica Strukturen werden in weiterer Folge von der in der Hierarchie übergeordneten *Descent["element"]* Funktion verarbeitet.

Die Aufgabe der Parse-Funktion *Descent["element"]* besteht darin, eine Sequenz von aufeinander folgenden GDSII-Elementen mit Hilfe der entsprechenden jeweiligen *Descent[ ]* Funktionen zu parsen, analog wie es für die *Descent["boundary"]*, *Descent["sref"]* und *Descent["aref"]* bereits beschrieben wurde. Jeder dieser Aufrufe generiert jeweils eine Mathematica Struktur mit den entsprechenden Mathematica Anweisungen. Schließlich fasst *Descent["element"]* all diese Mathematica Strukturen in einer Liste zusammen und retourniert diese Liste als ihr Funktionsergebnis.

Nach einem weiteren Schritt nach oben in der Funktionshierarchie wird die Funktion *Descent["structure"]* erreicht. Der Codegenerierungsteil dieser Funktion paart den während des Parse-Vorganges eingelesenen Strukturnamen (gewonnen aus den Nutzdaten des GDSII-RECORD STRNAME) mit der durch die Funktion *Descent["element"]* gewonnen Liste, von allen zu der aktuellen GDSII-Struktur gehörenden GDSII-Elementen, zu einer zweielementigen Mathematica-Liste.

Diesen Vorgang wiederholt die Funktion *Descent["structure"]* für alle in den Daten vorhandenen Strukturen. Diese zwei-elementigen Listen werden nach zwei Kriterien aufgeteilt. Die Aufteilung erfolgt zum einen in Strukturen, welche keinesfalls *"sref"* oder *"aref"* Elemente enthalten und zum anderen in solche, welche mindesten eines der Elemente *"sref"* oder *"aref"* enthalten. Getrennt nach diesen beiden Kriterien werden schließlich alle Strukturen in jeweils einer von zwei Listen gesammelt. Mit Abschluss des geschilderten Vorganges sind nun alle referenzfreien Strukturen der GDSII-Daten in einer Liste gesammelt und alle anderen, nicht referenzfreien Strukturen ebenso in einer weiteren Liste gesammelt.

Letztendlich wird in die oberste Funktionshierarchie aufgestiegen. Die wichtigste Aufgabe der Funktionen dieser Hierarchie ist es, die noch vorhandnen Referenzen aufzulösen. Dazu

wird die Liste mit den nicht referenzfreien Strukturen Listelement für Listelement sequentiell abgearbeitet. Für jedes abzuarbeitende Listelement wird versucht, alle enthaltenen Referenzen aufzulösen, indem die referenzierenden Elemente über deren Namen, in der referenzfreien Liste gesucht werden. Mit Auffinden der referenzierten Struktur in besagter Liste werden die referenzierenden Elemente durch die gefundenen referenzierten Elemente ersetzt.

Zu beachten ist hierbei, dass die Elemente bei der Ersetzung noch gegebenenfalls entsprechend translatiert, rotiert und skaliert werden müssen. Sobald alle referenzierenden Elemente einer Struktur ersetzt worden sind, wird selbige in die Liste der referenzfreien Strukturen verschoben. Dieser Vorgang wird nun solange wiederholt, bis alle nicht referenzfreien Strukturen aufgelöst sind. Auf diese Art und Weise werden alle ursprünglich nicht referenzfreien Strukturen in die Liste der referenzfreien Strukturen verschoben und es existiert schlussendlich nur mehr eine einzige Liste mit allesamt referenzfreien Strukturen.

Die so gewonnene Liste von Strukturen besteht im Wesentlichen nur noch aus Mathematica-Grafik Objekten. Die Behandlung der noch verbliebenen Elemente welche keine Mathematica-Grafik darstellen, folgt im anschließenden Kapitel.

3.5.4 GDSII-Mathematica-Graphics und Feature Filterung

Die bisher prozessierten Daten enthalten neben der Mathematica-Grafik noch die folgenden Daten-Attribute: Layer-Nummern, Datatype-Nummern und Struktur-Namen. Diese Attribute gilt es, als nicht Mathematica-Grafik, entweder zu entfernen, oder aber es wird die darin enthaltene Information in sinnvolle Mathematica-Grafik Anweisungen umgesetzt. Eine mögliche Umsetzung der Attribute ist es z.B. den verschiedenen Layer-Nummern jeweils eine gewünschte Farbe zuzuordnen. Diese Zuordnung erfolgt indem die Layer-Nummer durch eine entsprechende Mathematica-Grafik Farbdirektive ersetzt wird. In analoger Weise kann auch mit den Datatype-Nummern und Struktur-Namen umgegangen werden.

An dieser Stelle ist das Ziel der Bemühungen erreicht. Es ist nun möglich, einen vollständigen GDSII-Datensatz in eine Mathematica-Grafik umzuwandeln. Die so gewonnene Mathematica-Grafik kann nun durch das Mathematica-System in der gewünschten Art und Weise grafisch ausgegeben werden.

Mit Hilfe der vielen Mathematica Standard Funktionen besteht nun auch die Möglichkeit, die importierten GDSII-Daten nach beliebigen Kriterien wie z.B. LAYER, DATATYPE, etc. zu filtern, um so die näher interessierenden Strukturen aus GDSII-Daten zu extrahieren.

3.6 GDSII-Import-Funktionen

Aufbauend auf den vorgestellten Konzepten und implementierten Funktionen können dem Nutzer nun eine Reihe von Funktionen für den GDSII-Import zu Verfügung gestellt werden. Zunächst folgt in Tabelle 21 „GDSII-Import-Nutzerfunktionen“ eine Übersicht aller für den Nutzer verfügbaren Import-Funktionen

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiFParse[GDSiiFile]	*{Resolved, Unresolved}*
FixRef[{ResolvedIn, UnresolvedIn}]	*{ResolvedOut, UnresolvedOut}*
StripData[Data]	*StripedData*
GDSiiFilterSTRNAME[Data, StrList]	*FilteredData*
GDSiiFilterLAYER[Data, LayerList]	*FilteredData*
GDSiiFilterTYPE[Data, TypeList]	*FilteredData*
GDSiiFilter[Data, StrList, LayerList, TypeList]	*FilteredData*
GDSiiMappingSTRNAME[Data]	*MappedData*
GDSiiMappingLAYER[Data]	*MappedData*
GDSiiMappingTYPE[Data]	*MappedData*
GDSiiMappingClear[Data]	*AttributeClearedData*

Tabelle 21 „GDSII-Import-Nutzerfunktionen“

Nachfolgend werden die Import-Funktionen detailliert vorgestellt.

Zu allererst wird die Funktion *GDSiiFParse[ ]* vorgestellt:

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiFParse[GDSiiFile]	*{Resolved, Unresolved}*

Die Funktion erwartet ein gültiges GDSII-File als Eingabeparameter und liefert eine zweielementige Liste als Ergebnis. Das erste Ergebnis-Element enthält dabei eine Liste der vollständig referenzfreien Strukturen des Files, das zweite Element liefert eine Liste der restlichen, Referenzen enthaltenden, Strukturen.

Die nächste Funktion *FixRef[ ]* dient zum Auflösen vorhandener Referenzen:

Funktionsaufruf	**Funktionsrückgabewert**
FixRef[{ResolvedIn, UnresolvedIn}]	*{ResolvedOut, UnresolvedOut}*

Diese Funktion erwartet als Eingabeparameter eine zwei-elementige Liste, wie sie mittels *GDSiiFParse[ ]* gewonnen wird. Wie früher beschrieben, ersetzt die Funktion wiederholt alle Referenzen durch die, durch sie referenzierten Strukturen. Im Falle korrekter GDSII-Daten werden dadurch alle Referenzen aufgelöst. Das Ergebnis dieses Prozesses liefert so genannte flache Daten, in welchen die ursprünglichen Strukturhierarchien aufgelöst sind. Wie früher beschrieben werden im Zuge der Referenzauflösung auch alle notwendigen geometrischen Transformationen auf die Strukturen angewandt. Die Funktion liefert als Ergebnis zwei Listen. Die erste ist die Liste mit den aufgelösten referenzfreien Strukturen. Die zweite Liste enthält, für den Fall nicht Korrekter GDSII-Daten mit fehlenden Referenzen, die verbliebenen Strukturen mit nicht auflösbaren Referenzen. Im Normalfall korrekter Daten wird die genannte zweite Liste als leere Liste *{}* retourniert. Im Allgemeinen wird also die erste der erhaltenen Ergebnislisten, welche alle aufgelösten Strukturen enthält, von weiterem Interesse sein.

Als nächste wird die Funktion StripData*[ ]* vorgestellt:

Funktionsaufruf	**Funktionsrückgabewert**
StripData[Data]	*StripedData*

Diese Funktion entfernt mit Ausnahme der Strukturnamen, alle in den Eingangsdaten enthaltenen Datenelemente wie z.B. Polygonkoordinaten, Layerangaben, etc. Als Eingangsdaten dienen hier z.B. das Resultat eines Aufrufes der Funktion *FixRef[ ]*. Als Ergebnis liefert die Funktion eine Mathematica Expression, welche nun ausschließlich GDSII-STRNAME's, also Strukturnamen, enthält. Die so gewonnene Expression, liefert, wie noch später demonstriert wird, einen guten Überblick der Strukturhierarchie der gelesenen GDSII-Daten.

Die Nächste Funktions-Gruppe dient zum Filtern der GDSII-Daten:

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiFilterSTRNAME[Data, StrList]	*FilteredData*
GDSiiFilterLAYER[Data, LayerList]	*FilteredData*
GDSiiFilterTYPE[Data, TypeList]	*FilteredData*
GDSiiFilter[Data, StrList, LayerList, TypeList]	*FilteredData*

Bei der Analyse vorgegebener GDSII-Daten ist es von großem Wert, selbige nach verschiedenen Kriterien auswählen oder eben filtern zu können. Der Nutzer wird dadurch in die Lage versetzt, für den Betrachtungsmoment, einen logisch begrenzten Ausschnitt der Daten analysieren und betrachten zu können. Die Tools stellen drei Filtermöglichkeiten zur Verfügung: Filterung nach: GDSII-STRNAME, GDSII-LAYER und GDSII-DATATYPE (BOXTYPE, NODETYPE). Dazu werden die 4 zuvor aufgelisteten Funktionen bereit gestellt. Die ersten 3 Funktionen filtern ihrem Namen entsprechend nach: STRNAME, LAYER bzw. DATATYPE. Die 4. Funktion fasst die Filtermöglichkeiten der ersten 3 Funktionen zu einer einzigen Funktion zusammen. Die Funktionen erwarten als Parameter zum einen die zu filternden Daten, zum anderen jeweils eine Liste mit den zu filternden Elementen. Für den Fall der Filterung von LAYER's ist das eine Liste ganzer Zahlen mit den gewünschten Layernummern, für die Filterung von STRNAME's ist das eine Liste von Strings mit den gewünschten Strukturnamen. Ergebnis der Filterung sind Daten welche ausschließlich die spezifizierten Elemente enthalten.

Wie früher erwähnt, enthalten die bis jetzt, trotz eventueller Filterung prozessierten Daten, nach wie vor störende Elemente. Das können z.B. GDSII-LAYER-Nummern und andere sein, die keine gültige Mathematica-Graphics darstellen. Diese noch störenden GDSII-Elemente weisen den geometrischen Objekten, welche durch die Daten spezifiziert sind, bestimmte GDSII-Atribute zu. Die in dieser Arbeit entwickelten Tools ermöglichen es, diese Attribute auf graphische Weise zu veranschaulichen. Dazu wird eine Reihe von Mapping-Funktionen implementiert, welche die GDSII-Attribute der geometrischen Objekte in Mathematica-Graphics Directive umsetzen. Diese Umsetzung kann z.B. als spezifische Farbtongebung für ein bestimmtes GDSII-Attribut durchgeführt werden.

Die folgende Funktionsgruppe dient zum Mapping von GDSII-Attributen:

Funktionsaufruf	**Funktionsrückgabewert**
GDSiiMappingSTRNAME[Data]	*MappedData*
GDSiiMappingLAYER[Data]	*MappedData*
GDSiiMappingTYPE[Data]	*MappedData*
GDSiiMappingClear[Data]	*MappedData*

Die Mapping-Funktionen erwarten allesamt als Parameter die Eingangsdaten mit den zu mappenden Elementen. Des Weiteren akzeptieren die Funktionen die *Option*: *MappingFunction* sowie die *Option Clear*. Die *Option MappingFunction* bewerkstelligt das eigentliche Mapping, als Default wird eine Funktion verwendet, welche ein Farbtonmapping der jeweiligen Attribute durchführt. Die *Clear Option* hat den Defaultwert *True*, mit der Bedeutung, dass alle weiteren, vom Mapping nicht erfassten Attribute gelöscht werden.

Mit den Mapping Funktionen ist es nun möglich, Objekte z.B. anhand ihrer GDSII-LAYER-Nummer einzufärben. Nachdem alle gewünschten Mappings durchgeführt wurden, müssen letztendlich noch alle eventuell verbliebenen nicht gemappten Attribute entfernt werden. Dies geschieht mit Hilfe der oben aufgelisteten Funktion: *GDSiiMappingClear[ ]*, welche noch verbliebene GDSII-Attribute ersatzlos aus den Daten entfernt. Die Funktion *GDSiiMapping-Clear[ ]* wird nur benötigt, falls die Mapping-Funktionen mit der Option Clear→False verwendet wurden.

Die Abbildung 8 „Import-Processflow“ zeigt zusammengefasst den typischen Ablauf des GDSII-Daten-Imports.

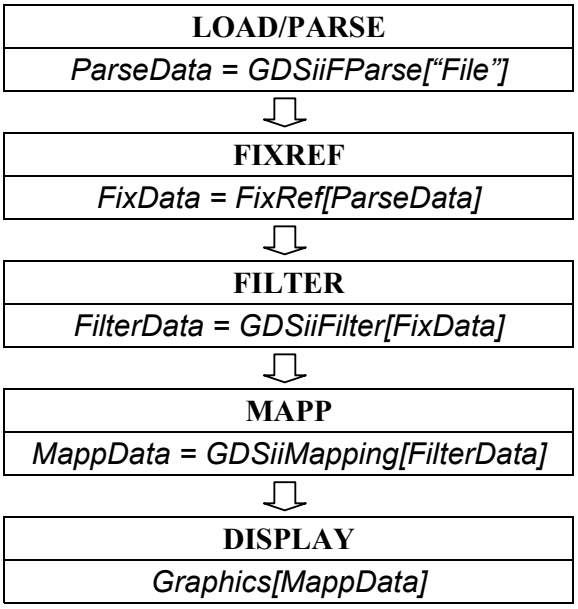

Abbildung 8 „Import-Processflow“

Zum Abschluss dieses Kapitels wird noch eine typische Beispiel Sequenz von Mathematica-Aufrufen, welche als Ergebnis eine Grafik Bildschirmausgabe bewirken, gezeigt. Die gezeigte Sequenz wird in Mathematica Postfix Operator Form realisiert. Zur Erinnerung: Wie früher bereits erläutert wird in dieser Form jeweils ein Funktionsaufruf mit einem Parameter, nämlich dem Ergebnis des jeweils vorhergehenden Aufrufes getätigt, wobei als allererster Parameter geeignete Eingangsdaten Verwendung finden. Seien die Eingangsdaten mit *e* und Funktionen mit *f1, f2, …fn* bezeichnet so hat eine Postfix Aufrufsequenz folgende Form: *e//f1//f2//…fn*. Dies ist gleichwertig mit der ansonsten üblicheren Form: *fn[….f2[f1[e]]]*, hat aber den Vorteil der besseren Lesbarkeit, auch wird die sequentielle Bearbeitungskette dadurch verdeutlicht. Der nachfolgende Mathematica Code: Tabelle 22 „Mathematica Postfix Aufrufsequenz“ prozessiert ein GDSII-File mit dem Inhalt, wie in Tabelle 56 „Leute“ ersichtlich und stellt es grafisch dar wie in Abbildung 19 „Leute gezeigt:

```
GDSiiFParse["Leute.gds"]//
FixRef//
First//
GDSiiFilter[#,"Leute"]&//
GDSiiMappingLAYER//
Graphics
```

Tabelle 22 „Mathematica Postfix Aufrufsequenz“

Das obige 6-zeilige Listing aus Tabelle 22 „Mathematica Postfix Aufrufsequenz“ führt Zeile für Zeile die folgenden Schritte durch:

1. Lesen und parsen des GDSII-Files mit Namen: „Leute.gds“
2. Auflösen der hierarchischen Referenzen
3. Extrahieren der vollständig aufgelösten referenzfreien Strukturen
4. Filtern der Struktur mit Namen: „Leute“
5. Ersetzen der GDSII-LAYER Attribute durch eine default Farbton Zuweisung
6. Löschen eventuell noch verbliebener GDSII-Attribute
7. Generieren einer Mathematica-Grafik und Bildschirmausgabe selbiger

Die tatsächlich generierte Grafik-Bildschirmausgabe des Beispiels wird im folgenden Kapitel gezeigt.

4 Praktische Anwendung und Demonstration der Tools

In diesem Kapitel werden die erarbeiteten Funktionen des implementierten Tools anhand von Beispielen angewandt und demonstriert. Dazu werden zunächst mit Hilfe der Export Tools GDSII-Strukturen erzeugt und als GDSII-File gespeichert. Anschließend werden mit dem frei erhältlichen Programm KLAYOUT, die generierten GDSII-Files geladen und grafisch dargestellt, um ein Bild von den erzeugten Strukturen zu verschaffen.

4.1 Demonstration der GDSII-Exportfunktionen

Zunächst werden die Tools für den GDSII-Export getestet. Hierfür wird das Softwareprogramm KLAYOUT als Hilfsmittel eingesetzt. Bei diesem Programm handelt es sich um einen GDSII-Editor/Viewer welcher, unter der GNU Lizenz, frei verfügbar ist. Der Viewer wird benutzt werden, um die Ergebnisse der Export Bemühungen der GDSII-Tools zu kontrollieren.

Bei den Tests wird der hierarchische Charakter von GDSII verdeutlicht werden. Insbesondere wird auch die Umsetzung der Hierarchien mit Hilfe der Tools demonstriert. Als GDSII-Testobjekt dient nun eine Ansammlung symbolischer Gesichter. Zunächst wird das Testobjekt mit Hilfe der GDSII-Tools aufgebaut. Dazu wird der folgende Mathematica Code betrachtet:

```
SetDirectory[NotebookDirectory[ ];
<<GDSii.m
GDSiiSetOverallScale[10]
```

Tabelle 23 „Laden des GDSII-Package“

Der Code aus Tabelle 23 „Laden des GDSII-Package“ bewirkt zunächst einen Wechsel in jenes Directory, in welchem das Notebook gespeichert ist. Anschließend werden die GDSII-Tools, welche in Form eines Mathematica Package mit Namen „GDSii.m“ vorliegen, geladen. Als nächstes wird ein, für das Beispiel praktischer, Gesamtmaßstab von 10 gesetzt. Nun folgt in Tabelle 55 „LeuteLayout“ welches auf Grund seines Umfanges im Anhang zu finden ist, der weitere Code für den Aufbau der gewünschten GDSII-Strukturen.

Hierbei ist zunächst die Definition einer GDSII-Polygon-Struktur mit Namen Pupille zu sehen. Diese Struktur wird anschließend, von der ihr im Code nachfolgenden Struktur Augen zweimal referenziert. Die Struktur besteht also aus zwei Augen in Form einfacher Polygone, sowie jeweils einer Pupille, welche ja bereits zuvor als Polygon definiert wurde. Anschließend definiert der Code eine einfache Nase in Form eines Dreieckes, sowie weiters einen ebenfalls einfachen, dreieckigen Mund. Der nächste Codeabschnitt definiert die symbolische, 8-eckige Kontur eines Gesichtes in Form eines GDSII-Path. Bekanntlich handelt es sich bei GDSII-Path um einen offnen Polygonzug mit einer gewissen Liniestärke. Als letzte einfache

Struktur wird im Code ein symbolischer Bart in Form eines Trapezes definiert. Als nächstes folgt die Definition der Struktur Gesicht, welches die Strukturen Augen, Nase, Mund und Kontur referenziert. In weitere Folge werden Strukturen mit Namen Mutter, Vater und Kind definiert, welche jeweils die Struktur Gesicht referenzieren. Diese Referenzen werden hierbei mit unterschiedlichem Maßstab verwendet, um verschieden große Gesichter zu erstellen. Die Struktur Vater referenziert zusätzlich noch einen Bart. Der hierarchische Aufbau findet seine Fortsetzung, in dem je eine Mutter ein Kind und ein Vater von der Struktur Familie referenziert werden. Die Familie Struktur wird schlussendlich in Form einer 3x3 Arrayreferenz von der Struktur Leute referenziert. Bei dieser Gelegenheit werden zu Demonstrationszwecken die mit Familie benannten Strukturen um 5° gedreht und auch noch um die X-Achse gespiegelt.

Der weitere Code in Tabelle 24 „Textuelle Form“ dient dazu, die textuelle Darstellung der generierten GDSII-Daten auszugeben

```
LeuteLayout//GDSiiTextual
```

Tabelle 24 „Textuelle Form“

Die generierte textuelle Darstellung ist im Anhang unter Tabelle 56 „Leute“ zu finden.

Schließlich dient der Code aus Tabelle 25 „Export“ dazu, die definierten GDSII-Daten, welche dem Mathematica Symbol LeuteLayout zugewiesen worden sind, in eine GDSII-Datei, mit Namen „Leute.GDS“ zu exportieren.

```
LeuteLayout//GDSiiExport[#,"Leute.GDS"]&
```

Tabelle 25 „Export“

Damit wurde nun erfolgreich eine GDSII-Datei mit hierarchischen Strukturen erzeugt. KLAYOUT hilft dabei diese Hierarchie übersichtlich, wie in Abbildung 9 „Hierarchiestruktur ersichtlich, anzuzeigen. Dazu wird KLAYOUT.EXE gestartet und über das File-Menü des Programms das GDSII-File: „Leute.GDS“ geöffnet. Das Programm stellt die Hierarchie der geladenen GDSII-Strukturen im linken Fensterbereich, wie in Abbildung 9 „Hierarchiestruktur“ gezeigt dar. Durch einen „Rechte Maustaste Klick“ kann man dem Menüpunkt „Show A New Top“ wählen, worauf die so angewählte Struktur im rechten Fensterbereich grafisch dargestellt wird. InAbbildung 10 „Augen“, Abbildung 11 „Gesicht“, Abbildung 12 „Familie“ und Abbildung 13 „Leute“ wird die grafische Ausgabe von KLAYOUT exemplarisch für einige der im File „Leute.GDS“ enthaltenen Strukturen gezeigt. Dabei fällt, nebenbei erwähnt, auf, dass die Spiegelung um die X-Achse für die Struktur „Leute“ von KLAYOUT nicht

durchgeführt wird. Dieses Verhalten von KLAYOUT entspricht nicht der GDSII-Spezifikation!

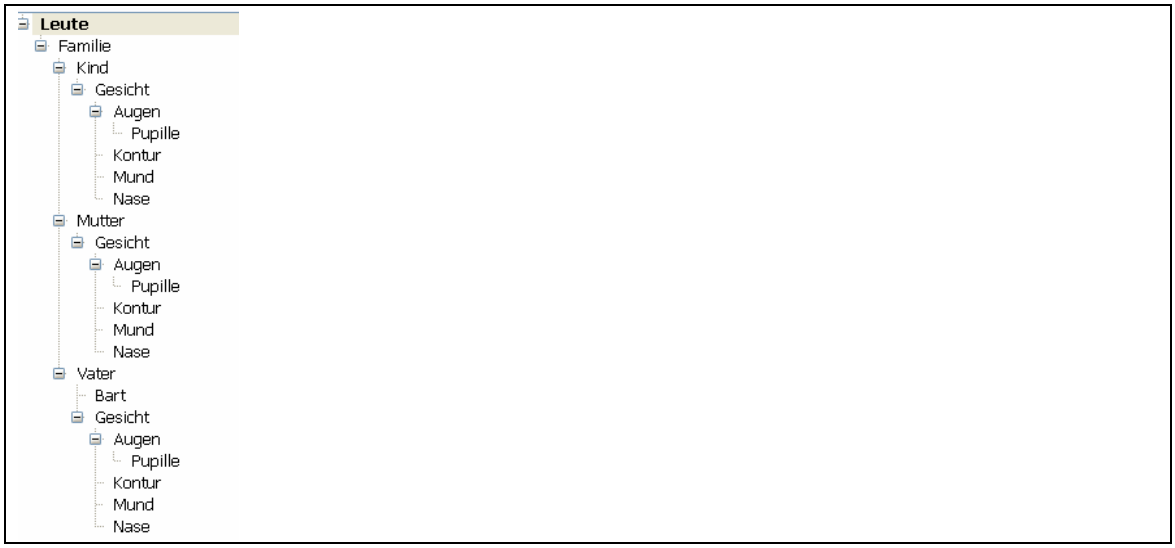

Abbildung 9 „Hierarchiestruktur“

Abbildung 10 „Augen“

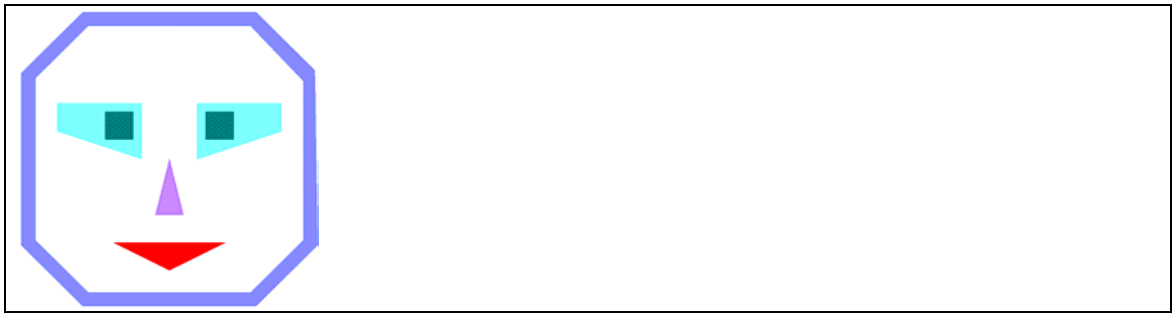

Abbildung 11 „Gesicht“

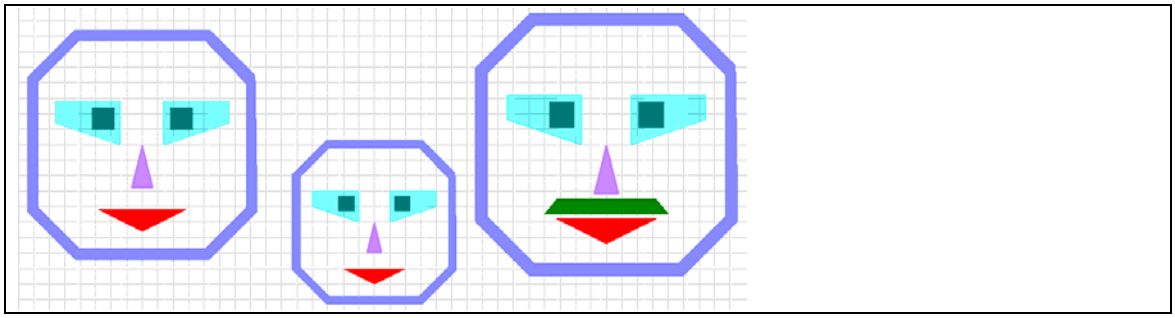

Abbildung 12 „Familie“

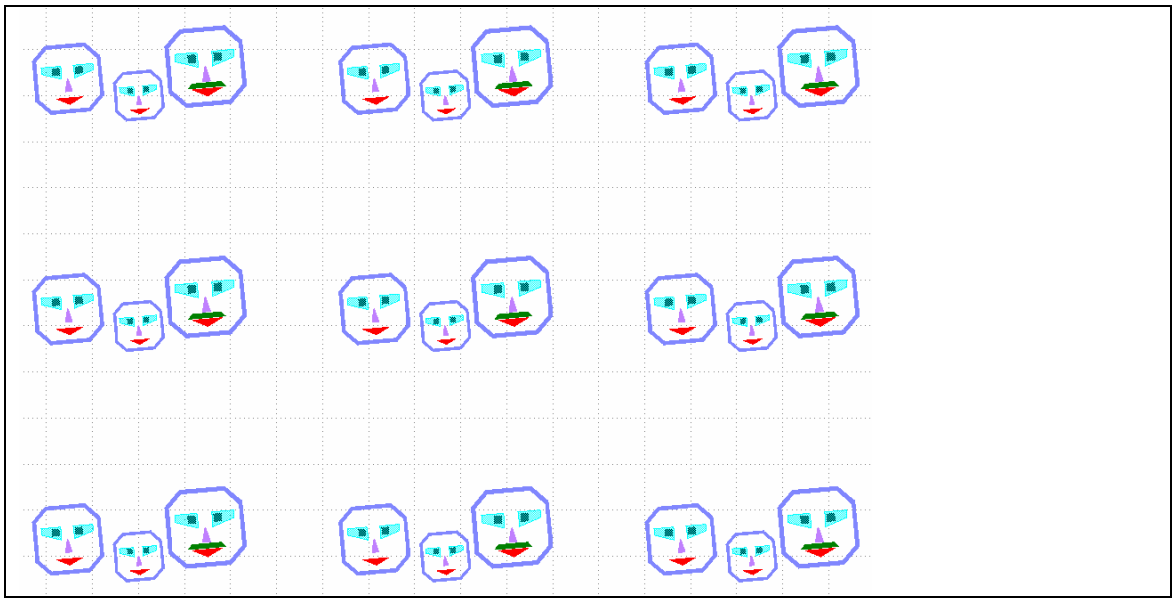

Abbildung 13 „Leute“

Nachdem die GDSII-Exportfunktionen erfolgreich an einem hinreichend komplexen Beispiel demonstriert wurden, werden in weiterer Folge die GDSII-Importfunktionen, erörtert.

4.2 Demonstration der GDSII-Importfunktionen

Bei der Demonstration der GDSII-Exportfunktionen wurde im vorangehenden Abschnitt das Programm KLAYOUT bemüht, um die mit den Funktionen generierten GDSII-Strukturen zu visualisieren. Dank der entwickelten GDSII-Importfunktionen, können die Strukturen nun auch ohne Zuhilfenahme dieses Programms dargestellt werden. So wie zuvor mit Hilfe von KLAYOUT können nun, mittels der entwickelten GDSII-Importfunktionen, die Strukturen geladen und visualisiert werden. KLAYOUT bleibt dennoch, auch in diesem Fall, wertvoll, es dient nun als Referenz um die Korrektheit der implementierten GDSII-Importfunktionen zu verifizieren.

Für die Demonstration der GDSII-Importfunktionen dient der Mathematica Code aus Tabelle 26 „Laden der GDSiiStructs“. Hierbei wird das GDSII-File „Leute.GDS“, welches zuvor mit Hilfe der Exportfunktionen erstellt wurde, prozessiert. Dazu werden zunächst die als Mathematica-Package vorliegenden GDSII-Tools geladen. Anschließend werden der Variablen *GDSiiStructs,* alle aus dem GDSII-File gelesenen und aufgelösten Strukturen zugewiesen.

```
<<GDSii.m
GDSiiStructs =
  GDSiiFParse["Leute.gds"]//
  FixRef//
  First//
```

Tabelle 26 „Laden der GDSiiStructs“

Mit Hilfe der Tools und weiterer Mathematica Standard Funktionen kann nun die Analyse der GDSII-Strukturen erfolgen. Zunächst wird ein Überblick über die enthaltenen Strukturen und deren hierarchischen Aufbau benötigt. Dazu wird die Toolfunktion *GetTopLevel[ ]* bemüht, welche alle Strukturen der obersten Hierarchieebene liefert, sowie die ToolFunktion *StripData[ ]*, welche alle Elemente mit Ausnahme der Strukturnamen aus den Daten entfernt. Abschließend wird die Standard Mathematica Function *TreeForm[ ]* angewandt, um die Strukturhierarchie darzustellen. In Tabelle 27 „GDSII Treeform“ sind die notwendigen Mathematica Anweisungen ersichtlich, welche für den gewünschten Überblick notwendig sind. Das Mathematica-System reagiert auf die Eingabe der Anweisungen aus Tabelle 27 „GDSII Treeform“ mit der gewünschten baumförmigen Ausgabe wie in Abbildung 9 „Hierarchiestruktur“ zu sehen ist. Sofort erkennen wir, dass die Daten genau eine Toplevel-Struktur mit Namen „Leute“ enthalten, der hierarchische Aufbau dieser Toplevel-Struktur ist ebenso anschaulich dargestellt.

```
GDSiiStructs//GetTopLevel//StripData//TreeForm
```

Tabelle 27 „GDSII Treeform“

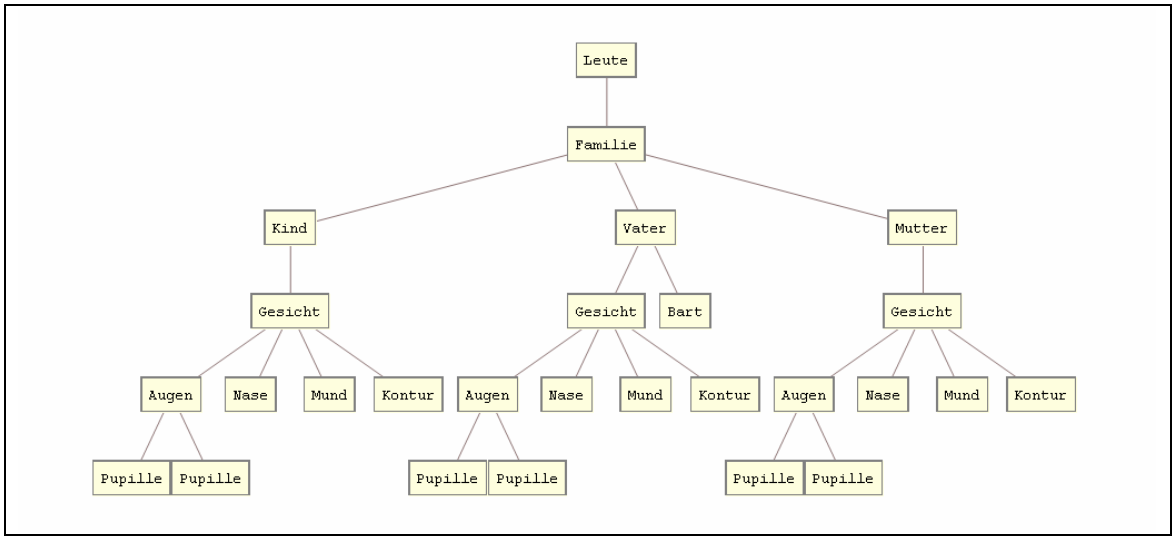

Abbildung 14 „Strukturhierarchie“

Im zuvor gezeigten Beispiel waren 6 Hierarchieebenen enthalten. Im Falle noch komplexrer Strukturen kann es hilfreich sein, nur ausgewählte Substrukturen zu inspizieren. Wenn der

Augenmerk z.B. auf die Substrukturen „Mutter“ und „Vater“ gelegt werden soll, so gelingt das mit der passenden Filterfunktion der implementierten Tools, wie in Tabelle 28 „Filter STRNAME“ und der zugehörigen Ausgabe in Abbildung 15 „Vater, Mutter Baumstruktur“ gezeigt wird

```
GDSiiStructs//
GDSiiFilterSTRNAME[#,{"Vater","Mutter"}]&//
StripData//
TreeForm
```

Tabelle 28 „Filter STRNAME“

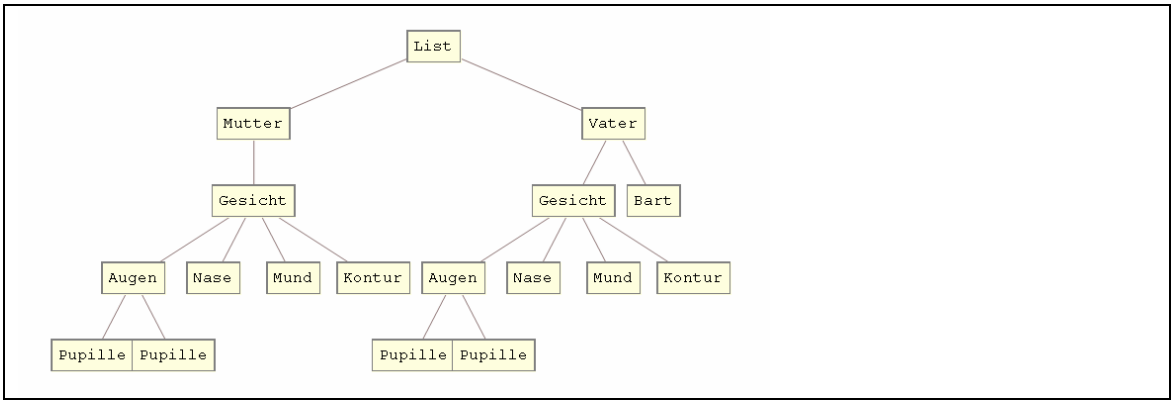

Abbildung 15 „Vater, Mutter Baumstruktur“

Nun werden ausgewählte GDSII-Strukturen mit Hilfe der entwickelten Tools und dem Mathematica-System grafisch dargestellt. Dazu werden die gleichen Substrukturen wie zuvor mittels des Programms KLAYOUT betrachtet. In Tabelle 29 „Augen“ sind die notwendigen Mathematica Anweisungen ersichtlich, Abbildung 16 „Augen“ zeigt die entsprechende Ausgabe. Durch den Vergleich mit Abbildung 16 „Augen“ welches mittels KLAYOUT erzeugt wurde, wird das korrekte Verhalten der Toolimplementierung ersichtlich.

```
GDSiiStructs//
GDSiiFilter[#,"Augen"]&//
GDSiiMappingLAYER//
Graphics
```

Tabelle 29 „Augen“

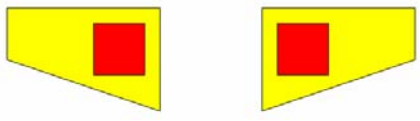

Abbildung 16 „Augen“

Wie zuvor mit KLAYOUT, werden auch noch die Strukturen „Gesicht“ und „Familie“ dargestellt wie in Abbildung 17 „Gesicht“ und Abbildung 18 „Familie“ zu sehen ist.

```
GDSiiStructs//
GDSiiFilter[#,"Gesicht"]&//
GDSiiMappingLAYER//
Graphics
```

Tabelle 30 „Gesicht“

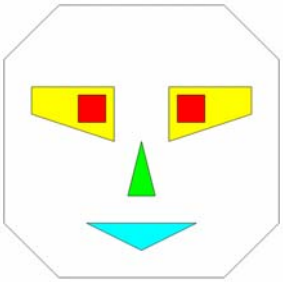

Abbildung 17 „Gesicht“

```
GDSiiStructs//
GDSiiFilter[#,{"Familie"}]&//
GDSiiMappingSTRNAME//
GDSiiMappingClear//
Graphics
```

Tabelle 31 „Familie“

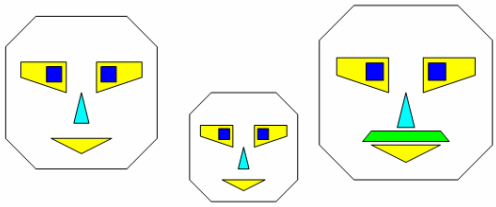

Abbildung 18 „Familie“

Abschließend wird noch die Toplevel Struktur „Leute“ gezeigt. Wie zuvor erwähnt und in Abbildung 13 „Leute“ ersichtlich, ist die Darstellung durch KLAYOUT, aufgrund der fehlenden Spiegelung um die X-Achse nicht korrekt. Die Prozessierung der Struktur mit Hilfe des GDSII-Tool-Codes nach Tabelle 32 „Leute“ liefert wie in Abbildung 19 „Leute“ zu sehen ist eine korrekte (um die X-Achse gespiegelte) Darstellung.

```
GDSiiStructs//
GetTopLevel//
GDSiiMappingLAYER//
Graphics
```

Tabelle 32 „Leute“

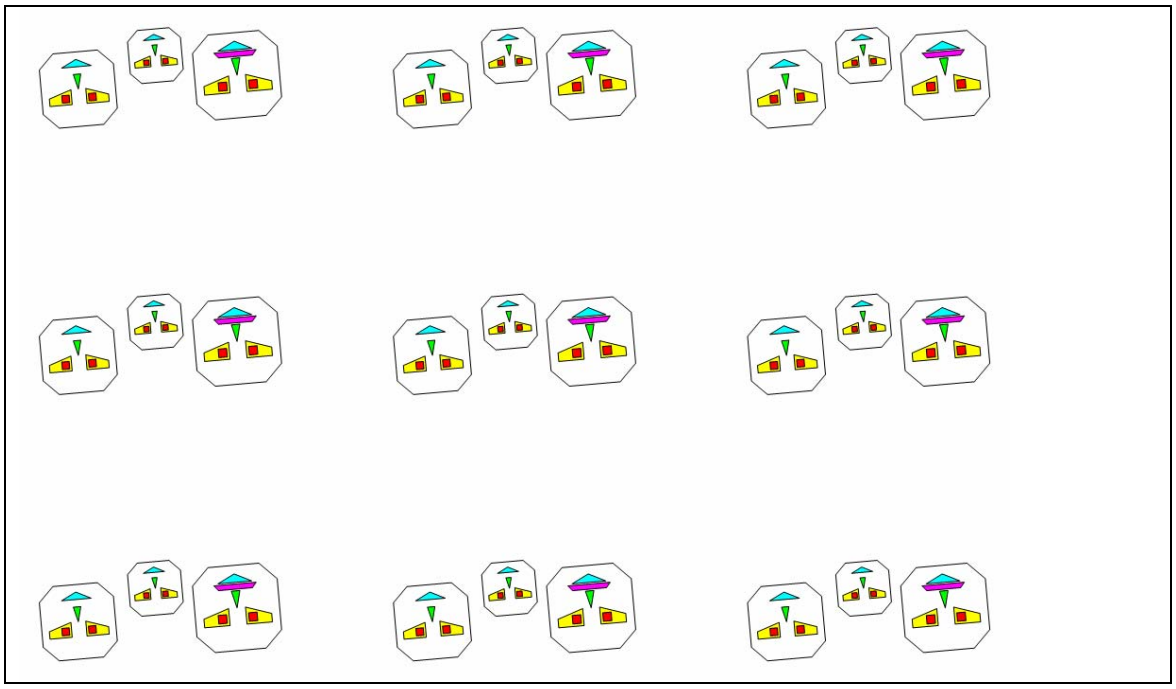

Abbildung 19 „Leute“

Mit dem Beispiel „Leute“ wurden in anschaulicher Weise die hierarchischen Aspekte des GDSII-Formates, sowie der Umgang der GDSII-Tools mit selbigen demonstriert.

- **Beispiel NAND-Gatter**

Im nun folgenden Beispiel „NAND-Gatter“ werden insbesondere der Umgang der Tools mit den GDSII-LAYER’s demonstriert.

Dazu wird zunächst die Geometrie eines NAND-Gatters spezifiziert. Zu diesem Zweck werden die Polygonkoordinaten der verschiedenen Gatter-Bestandteile entsprechenden Mathematica-Variablen zugewiesen, wie in Tabelle 57 „NAND-Gatter Polygon-Koordinaten“ ersichtlich ist. Der Mathematica-Code aus Tabelle 58 „NAND-Gatter GDSII“ erzeugt aus den Koordinaten das gewünschte GDSII-NAND-Gatter. Das so erzeugte Gatter wird mittels *Nand-Layout//GDSiiExport[#,"Nand.GDS"]&* in eine GDSII-Datei exportiert.

Die eben generierte Datei „Nand.GDS“ kann nun mit Hilfe der Import-Funktionen geladen und analysiert werden

Der Code *{ResGDS,UnresGDS}=GDSiiFParse["Nand.gds"]*lädt das File mit dem NAND-Gatter, anschließend werden alle Referenzen mit dem Code: *GDSiiStructs = First@FixRef[{ResGDS,UnresGDS}]*; aufgelöst.

Das NAND-Gatter steht nun als Mathematica Variable *GDSiiStructs* zur Verfügung. Wir untersuchen nun die Struktur der geladenen Daten mit dem Code:

GDSiiStructs//GetTopLevel//StripData//TreeForm und erhalten einen anschaulichen Überblick wie in Abbildung 20 „NAND Aufbau“ ersichtlich ist.

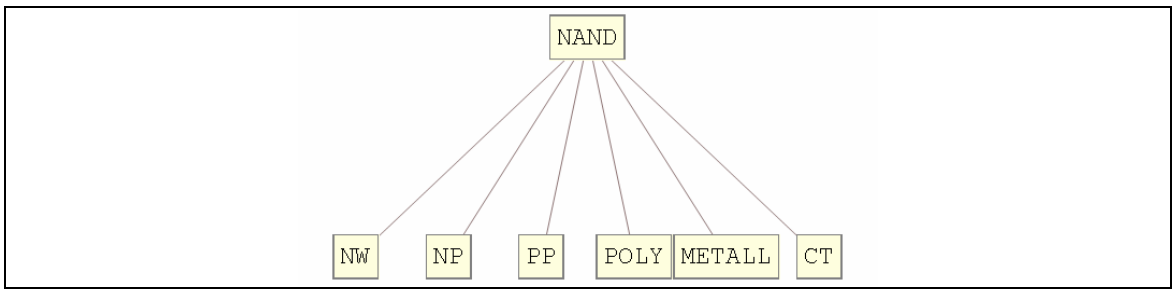

Abbildung 20 „NAND Aufbau“

In den Daten ist genau eine „Topl-Level“ Struktur mit Namen „NAND“ enthalten. Der Code: *NandStruct = GDSiiStructs//GDSiiFilterSTRNAME[#,{"NAND"}]&;* filtert uns diese Struktur aus den Daten heraus und weist sie der Variablen *NandStruct* zu. Um die Gatter-Teile anhand ihres LAYER’s entsprechend einzufärben und grafisch darzustellen wird der Code *NandStruct//GDSiiMappingLAYER//Graphics* bemüht, welcher die Ausgabe Abbildung 21 „NAND-Gatter LAYER-Colorierung“ liefert.

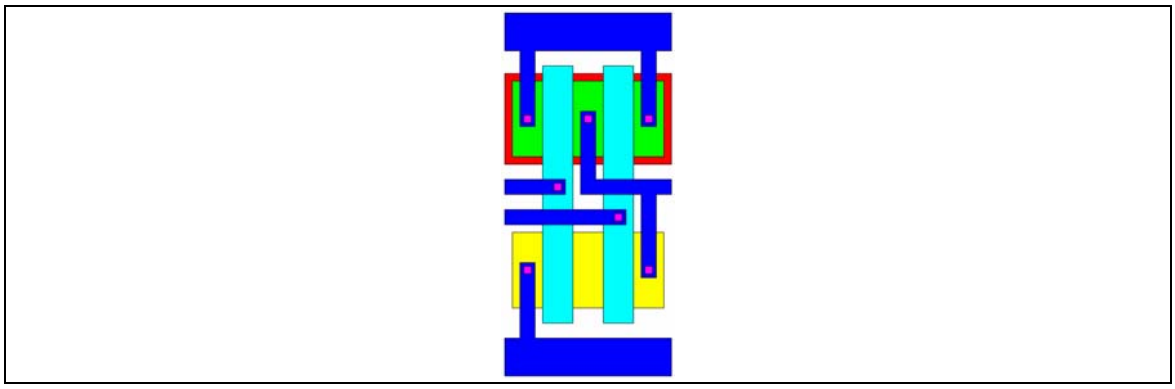

Abbildung 21 „NAND-Gatter LAYER-Colorierung“

Soll das Augenmerk z.B. auf die LAYER 1, 2, 4, 5 gelegt werden, welche den Metall, Kontakt, p+ und n+ Halbleiter-Schichten entsprechen, so leistet das der Code aus Tabelle 33 „Layer Filterung“. Die entsprechende Ausgabe ist in Abbildung 22 „NAND-LAYER Filterung“ zu sehen.

```
NandStruct//
GDSiiFilterLAYER[#,{1,2,4,5}]&//
GDSiiMappingLAYER//
Graphics
```

Tabelle 33 „Layer Filterung“

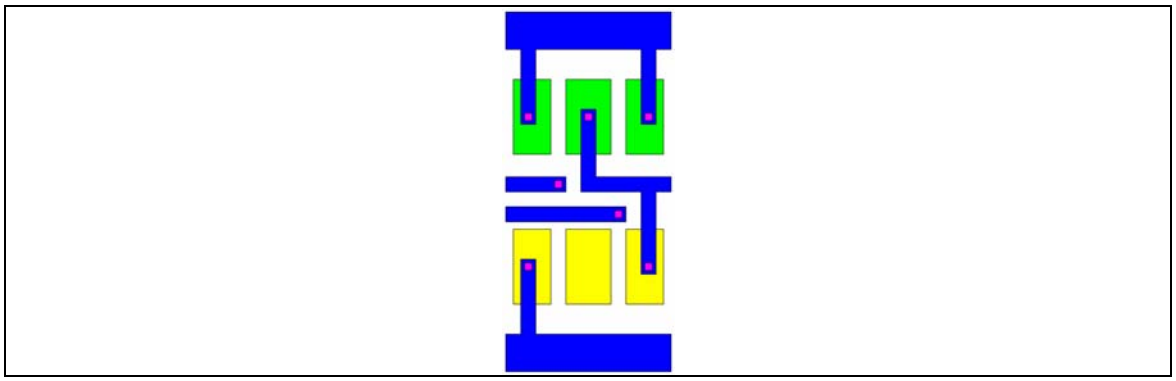

Abbildung 22 „NAND-LAYER Filterung“

Einen besonders guten Eindruck vom Halbleiter-Schichtaufbau liefert die iterative Filterung nach den LAYERN: 0, 0-1, 0-2, …, 0-5. Das Ergebnis dieser Filterung ist in Abbildung 23 „Halbleiter Schichtaufbau“ zu sehen.

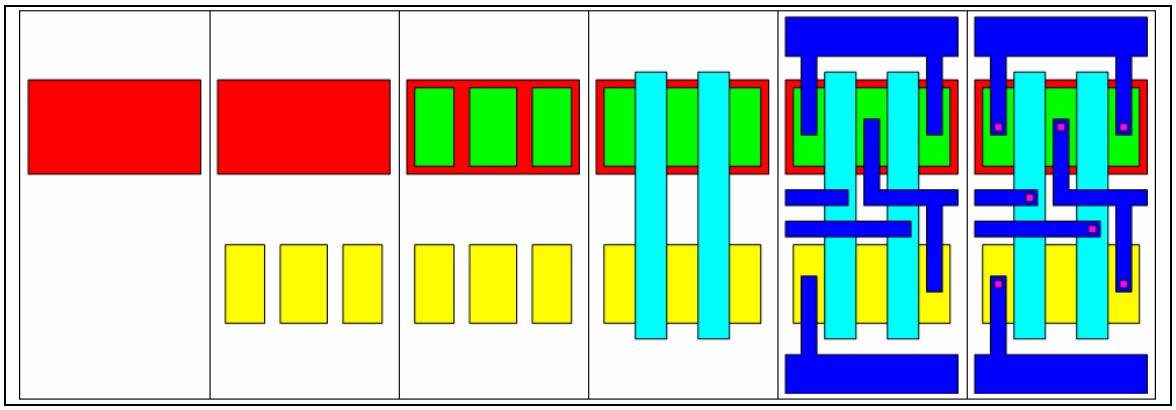

Abbildung 23 „Halbleiter Schichtaufbau“

Mit der erfolgten Demonstration der Import-Tools an Hand des gezeigten NAND-Gatters wird das Kapitel abgeschlossen.

5 Zusammenfassung, Status und Ausblick

Ziel der Diplomarbeit war es, dem potentiellen Nutzer eine Sammlung von Mathematica Softwaretools für die Datenkonversion von GDSII Layoutdaten zur Verfügung zu stellen. In der Arbeit wurde zunächst in die Grundlagen des GDSII-Formates sowie in die Mathematica Programmierung eingeführt. In weiterer Folge wurden Methoden und Konzepte für den Import/Export von GDSII-Daten in und aus Mathematica vorgestellt bzw. erarbeitet. Mit Hilfe der erarbeiteten Methoden wurden eine Reihe von Import/Exportfunktionen in der Mathematica Programmiersprache implementiert. In der Arbeit wurde der Implementierungscode an geeigneten Stellen auszugsweise präsentiert und seine Funktionsweise erläutert. Der Code steht nun, mit Abschluss der Implementierungsarbeiten als Mathematica-Package, dem Anwender zur Verfügung. Im Kapitel: „Praktische Anwendung und Demonstration der Tools“ werden die Tools an einem hinreichend komplexen Beispiel demonstriert. Nach Durchsicht des genannten Kapitels sollte dem geübten Mathematica Nutzer welcher über Grundkenntnisse des GDSII-Formates verfügt, die praktische Anwendung der Tools nicht weiter schwer fallen.

- **Status:**

Zum Erstellungszeitpunkt dieser Arbeit kann erfreulicherweise berichtet werden, dass der Exportteil der Tool-Funktionen bereits erfolgreich für die Generierung von Teststrukturen Verwendung findet.

- **Ausblick:**

Die Zukunft wird zeigen inwieweit die entwickelten Tools von den potentiellen Nutzern akzeptiert werden. Als nächster Schritt in Richtung weiterer Nutzung der Tools ist die Übergabe der fertig gestellten Import-Funktionen an die Nutzer vorgesehen.

Die Erfahrungen und das Feedback beim Umgang mit den Tools werden bei Bedarf, zu einem späteren Zeitpunkt in eine Verbesserung bzw. Erweiterung der Tools einfließen.

An dieser Stelle sei darauf hingewiesen, dass mit dem OASIS-Format möglicherweise eine Ablösung von GDSII bevorsteht. Somit könnte eine künftige Erweiterung der Tools in Richtung einer Unterstützung von OASIS, eine lohnenswerte Aufgabe darstellen.

6 Beschriftungsverzeichnis

7 Literaturverzeichnis und Internetquellen

[1] P. Rai-Choudhury: SPIE Handbook of Microlithography, Micromachining and Microfabrication Volume 1: Microlithography, Appendix 2.9: GDSII Stream Format, SPIE Publications 1997

[2] Klaas Holwerda: GDSII Format, http://boolean.klaasholwerda.nl/interface/bnf/gdsformat.html, 6/2009

[3] Rainer Minixhofer: Integrating Technology Simulation into the Semiconductor Manufacturing Environment, B.2 Calma GDS II Stream Format, http://www.iue.tuwien.ac.at/phd/minixhofer/node52.html, 6/2009

[4] Jim Buchanan: The GDSII Stream Format, http://www.buchanan1.net/stream_description.shtml, 6/2009

[5] Wolfram Research: Wolfram Mathematica Documentation Center, http://reference.wolfram.com/mathematica/guide/Mathematica.html, 6/2009

[6] Stephen Wolfram: The Mathematica Book, 5th Edition, Wolfram Research 2004

[7] Niklaus Wirth: Compilerbau, 3. Auflage, Teubner 1984

[8] Alfred V. Aho; Ravi Sethi; Jeffrey D. Ullman: Compilers Principles, Techniques, and Tools, Addison-Wesley 1986

[9] Wikipedia: Recursive Descent Parser, http://en.wikipedia.org/wiki/Recursive_descent_parser, 6/2009

Hinweis: Datumsangaben der Internetquellen beziehen sich auf den Zeitpunkt der Verfügbarkeit.

8 Anhang: Das GDSII-Format

8.1 Die wichtigsten GDSII-RECORD's

- **HEADER, BGNLIB, LIBNAME, UNITS, ENDLIB**

Jeder GDSII Stream enthält genau einen HEADER-RECORD welcher gleichzeitig den ersten RECORD des Streams bildet. Der HEADER beinhaltet genau ein INTEGER_2 als Nutzdaten in welchem die GDSII Versionsnummer abgelegt ist, die Gesamtlänge des HEADER RECORD's beträgt 6 BYTE.

Unmittelbar anschließend an den HEADER folgt obligatorisch der einzige BGNLIB-RECORD des Streams, dieser markiert den Anfang der Library und enthält als Nutzdaten jeweils das Datum der letzten Modifikation und jenes des letzten Zugriffes auf die Library codiert in 12 INTEGER_2. Die Gesamtlänge beträgt immer 0x1C = 28 BYTE.

Auf BGNLIB folgt obligatorisch der einzige LIBNAME-RECORD des Streams, dieser enthält den Library Namen des Streams als ASCII STRING, welcher den UNIX Konventionen für Filenamen entsprechen muss. Die Gesamtlänge des RECORD's ist hierbei von der Stringlänge des Namens abhängig.

An den LIBNAME anschließend folgen einige optionale GDSII-RECORD's, gefolgt von einem obligatorischen UNITS-RECORD, welcher als Nutzdaten 2 REAL_8 enthält. Die beiden REAL_8 sind USER_UNIT, gefolgt von DATABASE_UNIT. Die Gesamtlänge des RECORD's beträgt 0x14 = 20 BYTE.

An UNITS schließt eine beliebige Anzahl von *structure's* (ebenfalls in Form von GDSII-RECORD's) an, gefolgt vom einzigen und obligatorischen ENDLIB-RECORD, wobei dieser den letzten RECORD im Stream bildet.

- **HEADER:**

0x0006	
0x00	0x02
0x0006	

RECORDLENGTH	
HEADER	INTEGER_2
VERSIONNUMMER	

Tabelle 34 „GDSII-HEADER-RECORD"

- **BGNLIB:**

0x001C	
0x01	0x02
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	

RECORDLENGTH	
BGNLIB	INTEGER_2
YEAR (modification)	
MONTH (modification)	
DAY (modification)	
HOUR (modification)	
MINUTE (modification)	
SECOND (modification)	
YEAR (access)	
MONTH (access)	
DAY (access)	
HOUR (access)	
MINUTE (access)	
SECOND (access)	

Tabelle 35 GDSII-BGNLIB-RECORD"

- **LIBNAME:**

0x….	
0x02	0x06
.	
.	
.	

RECORDLENGTH	
LIBNAME	STRING
ASCII CHARACTER	
ASCII CHARACTER	
ASCII CHARACTER	

Tabelle 36 „GDSII-LIBNAME-RECORD"

- **UNITS:**

0x0014	
0x03	0x05
.	
.	
.	
.	
.	
.	
.	
.	

RECORDLENGTH	
UNITS	REAL_8
USER_UNIT highest WORD	
USER_UNIT	
USER_UNIT	
USER_UNIT lowest WORD	
DATABASE_UNIT highest W	
DATABASE_UNIT	
DATABASE_UNIT	
DATABASE_UNIT lowest W	

Tabelle 37 „GDSII-UNITS-RECORD"

- **ENDLIB:**

0x0004	
0x04	0x00

RECORDLENGTH	
ENDLIB	NO_DATA

Tabelle 38 „GDSII-ENDLIB-RECORD"

- **BGNSTR, STRNAME, ENDSTR**

Die eigentlichen GDSII Geometriedaten werden innerhalb von *structure's* spezifiziert, diese werden immer mit einem BGNSTR-RECORD eingeleitet. Dieser RECORD markiert den Beginn einer *structure*, die Nutzdaten sind gleich aufgebaut wie beim BGNLIB-RECORD.

Unmittelbar auf ein BGNSTR muss obligatorisch ein STRNAME-RECORD folgen, dieser RECORD beinhaltet einen ASCII String welcher die *structure* unter anderem für spätere Referenzzwecke benennt.

STRNAME wird von einer beliebigen Anzahl von *elment's structure's* (selbstverständlich auch in Form von GDSII-RECORD's) gefolgt, welche ihrerseits von einem die *structure* abschließenden obligatorischen ENDSTR-RECORD gefolgt werden.

- **BGNSTR:**

0x001C	
0x05	0x02
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	

RECORDLENGTH	
BGNSTR	INTEGER_2
YEAR (modification)	
MONTH (modification)	
DAY (modification)	
HOUR (modification)	
MINUTE (modification)	
SECOND (modification)	
YEAR (access)	
MONTH (access)	
DAY (access)	
HOUR (access)	
MINUTE (access)	
SECOND (access)	

Tabelle 39 „GDSII-BGNSTR-RECORD"

- **STRNAME:**

0x….	
0x06	0x06
.	
.	
.	

RECORDLENGTH	
STRNAME	STRING
ASCII CHARACTER	
ASCII CHARACTER	
ASCII CHARACTER	

Tabelle 40 „GDSII-STRNAME-RECORD"

- **ENDSTR:**

0x0004	
0x07	0x00

RECORDLENGTH	
ENDSTR	NO_DATA

Tabelle 41 „GDSII-ENDSTR-RECORD"

- **BOUNDARY, PATH, SREF, AREF, SNAME, ENDEL**

Diese RECORD's dienen zum Aufbau von *element's*. Die *element's* sind in variabler Anzahl in den einzelnen *structure's* enthalten, sie spezifizieren die gewünschte Geometrie.

Der BOUNDARY-RECORD leitet die Definition einer geschlossenen Polygonfläche ein und ist somit einer der wichtigsten RECORD's zur Geometrie-Beschreibung. Dieser RECORD beinhaltet keine weiteren Nutzdaten.

Der PATH-RECORD leitet die Definition eines Linienzuges ein, er beinhaltet ebenfalls keine weiteren Nutzdaten.

Der SREF-RECORD dient mit Hilfe des SNAME-RECORD's der Referenzierung von *structure's*. Eine im Stream enthaltene *structure* wird über den ASCII STRING ihres STRNAME referenziert in dem ein SNAME mit übereinstimmenden ASCII STRING's angegeben wird.

Der AREF RECORD dient ebenso wie der SREF RECORD der Referenzierung von *structure's*, wobei hier die Referenz mehrfach, in Form eines zu spezifizierbaren Arrays, eingefügt wird.

Der ENDEL-RECORD ist für den Abschluss eines jeden *element* obligatorisch erforderlich.

- **BOUNDARY:**

0x0004	
0x08	0x00

RECORDLENGTH	
BOUNDARY	NO_DATA

Tabelle 42 „GDSII-BOUNDARY-RECORD"

- **PATH:**

0x0004	
0x09	0x00

RECORDLENGTH	
PATH	NO_DATA

Tabelle 43 „GDSII-PATH-RECORD"

- **SREF:**

0x0004	
0x0A	0x00

RECORDLENGTH	
SREF	NO_DATA

Tabelle 44 „GDSII-SREF-RECORD"

- **AREF:**

0x0004	
0x0B	0x00

RECORDLENGTH	
AREF	NO_DATA

Tabelle 45 „GDSII-AREF-RECORD"

- **SNAME:**

0x….	
0x12	0x06
.	
.	
.	

RECORDLENGTH	
SNAME	STRING
ASCII CHARACTER	
ASCII CHARACTER	
ASCII CHARACTER	

Tabelle 46 „GDSII-SNAME-RECORD“

- **ENDEL:**

0x0004	
0x11	0x00

RECORDLENGTH	
ENDEL	NO_DATA

Tabelle 47 „GDSII-ENDEL-RECORD“

- **LAYER, XY, STRANS, MAG, ANGLE, COLROW**

Jedes *element* einer *structure* muss auf genau einem LAYER platziert werden, dies ist die Aufgabe des LAYER-RECORD’s.

Der XY-RECORD dient zur Definition von Listen von x/y Koordinatenpaaren wie sie z.B. für die Spezifikation von Polygon-Eckpunkten bei BOUNDARY-RECORD’s benötigt werden. In Tabelle 49 „GDSII-XY-RECORD“ ist beispielhaft ein RECORD mit einer Liste von 2 Koordinatenpaaren, {(x[1]/y[1]), (x[2]/y[2])} = {(0/0), (3/5)} dargestellt. Aus der Gesamtlänge des RECORDLENGTH in BYTE, im Beispiel ist das 0x0014 = 20, wird auf die Anzahl der Koordinatenpaare geschlossen. Dazu werden von der Gesamtlänge 4 BYTE (2 BYTE für RECORDLENGTH, 1 BYTE RECORDTYPE, 1BYTE DATATYPPE ergibt zusammen 4 BYTE) abgezogen. Es bleiben im Beispiel also 20 – 4 = 16 BYTE für die Nutzdaten. Mit dem DATATYPE INTEGER_4 beansprucht jedes Koordinatenpaar 8 BYTE. Somit ergeben sich für das Beispiel 16/8 = 2 als Anzahl für die im RECORD enthaltenen Koordinatenpaare.

Der STRANS-RECORD enthält als Nutzdaten ein INTEGER_2 welches 16 BIT FLAG’s beinhaltet. Die FLAG’s steuern gewisse Transformationen des betreffenden *element*.

8. FLAG [00] Reflexion der x-Koordinaten aktiviert.
9. FLAG [13] MAG, Vergrößerung aktiviert.
10. FLAG [14] ANGLE, Rotation aktiviert.

Die sonstige FLAG’s sind unbenutzt.

Der MAG-RECORD sowie der ANGLE-RECORD enthalten als Nutzdaten jeweils einen REAL_8 Wert, welche die Vergrößerung bew. Rotation eines *element* spezifizieren.

Der COLROW-RECORD spezifiziert in Zusammenhang mit dem ARFE-RECORD die Anzahl der Spalten und Zeilen aus denen die Array Referenz bestehen soll, im Beispiel ein Array mit 3 Spalten, 5 Zeilen.

- **LAYER:**

0x0006	
0x0D	0x02
0x0006	

RECORDLENGTH	
HEADER	INTEGER_2
LAYERNUMBER	

Tabelle 48 „GDSII-LAYER-RECORD"

- **XY:**

0x0014	
0x10	0x03
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	
0x0003	
0x0000	
0x0005	

RECORDLENGTH	
XY	INTEGER_4
x[1] High WORD	
x[1] Low WORD	
y[1] High WORD	
y[1] Low WORD	
x[2] High WORD	
x[2] Low WORD	
y[2] High WORD	
y[2] Low WORD	

Tabelle 49 „GDSII-XY-RECORD"

- **STRANS:**

0x0006	
0x1A	0x01
0x….	

RECORDLENGTH	
STRANS	INTEGER_2
FLAG's	

Tabelle 50 „GDSII-STRANS-RECORD"

- **MAG:**

0x000C	
0x1B	0x05
.	
.	
.	
.	

RECORDLENGTH	
MAG	REAL_8
MAG highest WORD	
MAG	
MAG	
MAG lowest WORD	

Tabelle 51 „GDSII-MAG-RECORD"

- **ANGLE:**

0x000C	
0x1C	0x05
.	
.	
.	
.	

RECORDLENGTH	
ANGLE	REAL_8
ANGLE highest WORD	
ANGLE	
ANGLE	
ANGLE lowest WORD	

Tabelle 52 „GDSII-ANGLE-RECORD“

- **COLROW:**

0x0006	
0x13	0x02
0x0003	
0x0005	

RECORDLENGTH	
COLROW	INTEGER_2
COL	
ROW	

Tabelle 53 „GDSII-COLROW-RECORD“

8.2 Tabelle der GDSII-RECORD-Typen

Folgende Tabelle 54 „GDSII-RECORD’s zeigt eine Auflistung aller im GDSII-Format verfügbaren GDSII-RECORD-Typen. Jeder Tabelleneintrag enthält hierbei die RECORDTYPE-Nummer, den symbolischen RECORDTYPE-Namen, die DATATYPE-Nummer, sowie den symbolischen DATATYPE-Namen.

RECORDTYPE-NR.	RECORDTYPE	DATATYPE-NR.	DATATYPE
0x00	HEADER	0x02	INTEGER_2
0x01	BGNLIB	0x02	INTEGER_2
0x02	LIBNAME	0x06	STRING
0x03	UNITS	0x05	REAL_8
0x04	ENDLIB	0x00	NO_DATA
0x05	BGNSTR	0x02	INTEGER_2
0x06	STRNAME	0x06	STRING
0x07	ENDSTR	0x00	NO_DATA
0x08	BOUNDARY	0x00	NO_DATA
0x09	PATH	0x00	NO_DATA
0x0a	SREF	0x00	NO_DATA
0x0b	AREF	0x00	NO_DATA
0x0c	TEXT	0x00	NO_DATA
0x0d	LAYER	0x02	INTEGER_2
0x0e	DATATYPE	0x02	INTEGER_2
0x0f	WIDTH	0x03	INTEGER_4
0x10	XY	0x03	INTEGER_4
0x11	ENDEL	0x00	NO_DATA
0x12	SNAME	0x06	STRING
0x13	COLROW	0x02	INTEGER_2
0x14	TEXTNODE	0x00	NO_DATA
0x15	NODE	0x00	NO_DATA
0x16	TEXTTYPE	0x02	INTEGER_2
0x17	PRESENTATION	0x01	BIT_ARRAY
0x19	STRING	0x06	STRING
0x1a	STRANS	0x01	BIT_ARRAY
0x1b	MAG	0x05	REAL_8
0x1c	ANGLE	0x05	REAL_8
0x1f	REFLIBS	0x06	STRING
0x20	FONTS	0x06	STRING
0x21	PATHTYPE	0x02	INTEGER_2
0x22	GENERATIONS	0x02	INTEGER_2
0x23	ATTRTABLE	0x06	STRING
0x24	STYPTABLE	0x06	STRING
0x25	STRTYPE	0x02	INTEGER_2
0x26	ELFLAGS	0x01	BIT_ARRAY
0x27	ELKEY	0x03	INTEGER_4
0x2a	NODETYPE	0x02	INTEGER_2
0x2b	PROPATTR	0x02	INTEGER_2
0x2c	PROPVALUE	0x06	STRING
0x2d	BOX	0x00	NO_DATA
0x2e	BOXTYPE	0x02	INTEGER_2
0x2f	PLEX	0x03	INTEGER_4
0x30	BGNEXTN	0x03	INTEGER_4
0x31	ENDTEXTN	0x03	INTEGER_4
0x32	TAPENUM	0x02	INTEGER_2
0x33	TAPECODE	0x02	INTEGER_2
0x34	STRCLASS	0x01	BIT_ARRAY
0x35	RESERVED	0x03	INTEGER_4
0x36	FORMAT	0x02	INTEGER_2
0x37	MASK	0x06	STRING
0x38	ENDMASKS	0x00	NO_DATA
0x39	LIBDIRSIZE	0x02	INTEGER_2
0x3a	SRFNAME	0x06	STRING
0x3b	LIBSECUR	0x02	INTEGER_2

Tabelle 54 „GDSII-RECORD’s“

8.3 Beispiel GDSII-Layout

Der folgende Mathematica-Code generiert mit Hilfe der GDSII-Mathematica-Tools einen GDSII-Datensatz, welcher eine Ansammlung einfacher Formen in einer hierarchischen Ordnung darstellt. Bei den Formen handelt es sich um einfache symbolische Gesichter.

```
GDSiiSetOverallScale[10];

LeuteLayout = {
GDSiiBGNLIB[ ],

GDSiiBGNSTR["Pupille"],
    GDSiiBOUNDARY[{{-1,-1},{-1,1},{1,1},{1,-1},
{-1,-1}},0],
GDSii["ENDSTR"],

GDSiiBGNSTR["Augen"],
    GDSiiBOUNDARY[{{1,0},{4,1},{4,2},{1,2},{1,0}},1],
    GDSiiSREF["Pupille",1+.75,2-.75,.5],
    GDSiiBOUNDARY[{{-1,0},{-4,1},{-4,2},{-1,2},
{-1,0}},1],
    GDSiiSREF["Pupille",-1-.75,2-.75,.5],
GDSii["ENDSTR"],

GDSiiBGNSTR["Nase"],
    GDSiiBOUNDARY[{{0,0},{.5,-2},{-.5,-2},{0,0}},2],
GDSii["ENDSTR"],

GDSiiBGNSTR["Mund"],
    GDSiiBOUNDARY[{{-2,-3},{2,-3},{0,-4},{-2,-3}},3],
GDSii["ENDSTR"],

GDSiiBGNSTR["Kontur"],
    GDSiiPATH[{{5,3},{3,5},{-3,5},{-5,3},{-5,-3},
{-3,-5},{3,-5},{5,-3},{5,3}},4,.5,1],
GDSii["ENDSTR"],

GDSiiBGNSTR["Bart"],
    GDSiiBOUNDARY[{{-2,-2.2},{2,-2.2},{2.5,-2.8},
{-2.5,-2.8},{-2,-2.2}},5],
GDSii["ENDSTR"],

GDSiiBGNSTR["Gesicht"],
    GDSiiSREF["Augen"],
    GDSiiSREF["Nase"],
    GDSiiSREF["Mund"],
    GDSiiSREF["Kontur"],
GDSii["ENDSTR"],

GDSiiBGNSTR["Mutter"],
    GDSiiSREF["Gesicht",0,0,1.4],
GDSii["ENDSTR"],

GDSiiBGNSTR["Vater"],
    GDSiiSREF["Gesicht",0,0,1.6],
    GDSiiSREF["Bart",0,0,1.6],
GDSii["ENDSTR"],

GDSiiBGNSTR["Kind"],
    GDSiiSREF["Gesicht"],
GDSii["ENDSTR"],

GDSiiBGNSTR["Familie"],
    GDSiiSREF["Kind"],
    GDSiiSREF["Vater",15,5],
    GDSiiSREF["Mutter",-15,5],
GDSii["ENDSTR"],

GDSiiBGNSTR["Leute"],
    GDSiiAREF["Familie",3,3,0,0,200,0,0,150,1,5,True],
GDSii["ENDSTR"],

GDSii["ENDLIB"]
};
```

Tabelle 55 „LeuteLayout“

Der Mathematica-Code aus Tabelle 55 „LeuteLayout“ erzeugt den nachfolgenden GDSII-Datensatz Tabelle 56 „Leute“.

```
HEADER 600
BGNLIB
LIBNAME
UNITS 1. 1.e-6

BGNSTR
STRNAME Pupille
BOUNDARY
   LAYER 0
   DATATYPE 0
   XY {{-10, -10},
{-10, 10},{10, 10},
{10, -10}, {-10, -10}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Augen
BOUNDARY
   LAYER 1
   DATATYPE 0
   XY {{10, 0}, {40, 10},
{40, 20}, {10, 20},
{10, 0}}
   ENDEL
SREF
   SNAME Pupille
   STRANS Mag
   MAG 0.5
   XY {{18, 12}}
   ENDEL
BOUNDARY
   LAYER 1
   DATATYPE 0
   XY {{-10, 0}, {-40, 10},
{-40, 20}, {-10, 20},
{-10, 0}}
   ENDEL
SREF
   SNAME Pupille
   STRANS Mag
   MAG 0.5
   XY {{-18, 12}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Nase
BOUNDARY
   LAYER 2
   DATATYPE 0
   XY {{0, 0}, {5, -20},
{-5, -20}, {0, 0}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Mund
BOUNDARY
   LAYER 3
   DATATYPE 0
   XY {{-20, -30},
{20, -30}, {0, -40},
{-20, -30}}
   ENDEL
ENDSTR
```

```
BGNSTR
STRNAME Kontur
PATH
   LAYER 4
   DATATYPE 0
   PATHTYPE 1
   WIDTH 5
   XY {{50, 30}, {30, 50},
{-30, 50}, {-50, 30},
{-50, -30}, {-30, -50},
{30, -50}, {50, -30},
{50, 30}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Bart
BOUNDARY
   LAYER 5
   DATATYPE 0
   XY {{-20, -22},
{20, -22}, {25, -28},
{-25, -28}, {-20, -22}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Gesicht
SREF
   SNAME Augen
   XY {{0, 0}}
   ENDEL
SREF
   SNAME Nase
   XY {{0, 0}}
   ENDEL
SREF
   SNAME Mund
   XY {{0, 0}}
   ENDEL
SREF
   SNAME Kontur
   XY {{0, 0}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Mutter
SREF
   SNAME Gesicht
   STRANS Mag
   MAG 1.4
   XY {{0, 0}}
   ENDEL
ENDSTR
```

```
BGNSTR
STRNAME Vater
SREF
   SNAME Gesicht
   STRANS Mag
   MAG 1.6
   XY {{0, 0}}
   ENDEL
SREF
   SNAME Bart
   STRANS Mag
   MAG 1.6
   XY {{0, 0}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Kind
SREF
   SNAME Gesicht
   XY {{0, 0}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Familie
SREF
   SNAME Kind
   XY {{0, 0}}
   ENDEL
SREF
   SNAME Vater
   XY {{150, 50}}
   ENDEL
SREF
   SNAME Mutter
   XY {{-150, 50}}
   ENDEL
ENDSTR

BGNSTR
STRNAME Leute
AREF
   SNAME Familie
   STRANS ReflX Rot
   ANGLE 5
   COLROW 3 3
   XY {{0, 0},
{2000, 0}, {0, 1500}}
   ENDEL
ENDSTR

ENDLIB
```

Tabelle 56 „Leute“

8.4 Ein GDSII Nand-Gatter

Für die Konstruktion eines NAND-Gatters benötigen wir zunächst die Koordinaten der Geometrie der Gatter-Bestandteile: „n+“, „p+“, „n Wanne“, „Polysilizium“, „Metall“, „Kontakt“. Die notwendigen Koordinaten werden jeweils einer entsprechenden Mathematica-Variablen zugewiesen, wie aus Tabelle 57 „NAND-Gatter Polygon-Koordinaten“ ersichtlich ist.

```
pt =
{{60,160},{60,360},{140,360},{220,360},{220,160},{180,230},{100,270}};

ct =
Function[{p},p+#&/@(5{{-1,-1},{-1,1},{1,1},{1,-1}})]/@pt;

pp =
{{{40,310},{40,410},{90,410},{90,310}},{{110,410},{170,410},{170,310},{110,310}},
{{190,410},{240,410},{240,310},{190,310}}};

nw =
{{30,300},{30,420},{250,420},{250,300}};

np =
{{{40,110},{40,210},{90,210},{90,110}},{{110,210},{170,210},{170,110},{110,110}},
{{190,210},{240,210},{240,110},{190,110}}};

poly =
{{{80,430},{120,430},{120,90},{80,90}},{{160,430},{200,430},{200,90},{160,90}}};

metall =
{{{30,20},{30,70},{50,70},{50,170},{70,170},{70,70},{250,70},{250,20}},
{{210,150},{210,260},{130,260},{130,370},{150,370},{150,280},{250,280},{250,260},
{230,260},{230,150}},{{50,350},{50,450},{30,450},{30,500},{250,500},{250,450},
{230,450},{230,350},{210,350},{210,450},{70,450},{70,350}},
{{30,220},{190,220},{190,240},{30,240}},{{30,260},{110,260},{110,280},{30,280}}};
```

Tabelle 57 „NAND-Gatter Polygon-Koordinaten“

Das NAND-Gatter wird als eine Mathematica-Liste von GDSII-RECORD's mit Hilfe der GDSii-Funktionen aufgebaut, wie in Tabelle 58 „NAND-Gatter GDSII" ersichtlich. Hierbei werden die zuvor definierten Polygon-Koordinaten aus Tabelle 57 „NAND-Gatter Polygon-Koordinaten" in Aufrufen der *GDSiiBOUNDARY[ ]* Funktion, verwendet um die notwendigen geometrischen Figuren für das NAND zu erzeugen.

```
NandLayout = {

GDSiiBGNLIB[ ],

GDSiiBGNSTR["NW"],
    Sequence@@(GDSiiBOUNDARY[#,0,0]&/@{nw}),
GDSii["ENDSTR"],

GDSiiBGNSTR["NP"],
    Sequence@@(GDSiiBOUNDARY[#,1,1]&/@np),
GDSii["ENDSTR"],

GDSiiBGNSTR["PP"],
    Sequence@@(GDSiiBOUNDARY[#,2,0]&/@pp),
GDSii["ENDSTR"],

GDSiiBGNSTR["POLY"],
    Sequence@@(GDSiiBOUNDARY[#,3,2]&/@poly),
GDSii["ENDSTR"],

GDSiiBGNSTR["METALL"],
    Sequence@@(GDSiiBOUNDARY[#,4,3]&/@metall),
GDSii["ENDSTR"],

GDSiiBGNSTR["CT"],
    Sequence@@(GDSiiBOUNDARY[#,5,3]&/@ct),
GDSii["ENDSTR"],

GDSiiBGNSTR["NAND"],
    GDSiiSREF["NW"],
    GDSiiSREF["NP"],
    GDSiiSREF["PP"],
    GDSiiSREF["POLY"],
    GDSiiSREF["METALL"],
    GDSiiSREF["CT"],
GDSii["ENDSTR"],

GDSii["ENDLIB"]
};
```

Tabelle 58 „NAND-Gatter GDSII"

9 Anhang: Das Mathematica Software System

9.1 Ursprung und Historie des Mathematica-Systems

Mathematica ist ein Softwarepaket welches bei einer Vielzahl von wissenschaftlichen Aufgabenstellungen Anwendung findet. Oft wird das Programm in die Kategorie der CAS (Computeralgebrasystem) eingeordnet, wobei diese einschränkende Klassifizierung den vielfältigen Möglichkeiten des Systems nicht gerecht wird. Die Mathematica Software besteht aus zwei Komponenten: Einerseits aus dem Kernel welcher alle angeforderten Berechnungen bzw. Evaluationen durchführt und andererseits aus dem Notebookinterface mit welchem dem Benutzer ein GUI (Graphical User Interface) für die Erstellung wissenschaftlicher Dokumente sowie die Schnittstelle für die Kommunikation mit dem Kernel zur Verfügung gestellt wird.

Die historische Entwicklung der Mathematica-Software wird in der nachfolgenden Übersicht zusammengestellt:

1988 Mathematica 1.0 wird von Wolfram Research für den Apple Macintosh vorgestellt.

1996 Mathematica 3.0 erhält ein neues Notebookinterface welches nun überwiegend auf Mathematica Code basiert.

2003 Mathematica 5.0 bringt einen Performanceschub für numerische Berechungen, so dass die Geschwindigkeit mindestens vergleichbar mit jener von numerischer spezial Software wird.

2007 Mathematica 6.0 bringt die vollständig einheitliche Behandlung von Graphikobjekten, Soundobjekten, etc. mit allen weiteren Mathematica Objekten. Eine neue interaktive dynamische Funktionalität wird realisiert.

2008 Mathematica 7.0 stellt die derzeit aktuelle Version des Programmpaketes dar.

Aktuell ist Mathematica auf folgenden Plattformen verfügbar:
Microsoft Windows, Apple Mac OS X, Linux, Sun Solaris.

9.2 Einführung in das Mathematica-System

9.2.1 Der Mathematica Kernel

Der Mathematica Kernel ist als separates Programm im System realisiert, als solches ist er auch eigenständig lauffähig. Die Aufgabe des Kernel ist es, alle geforderten Auswertungen und Berechnungen, welche in Form der Mathematica Programmiersprache an ihn herangebracht werden, durchzuführen.

Sobald der Kernel gestartet ist läuft er in einer REP loop (Read Evaluate Print), d.h. der Kernel wartet auf Eingabedaten, liest selbige, evaluiert die Eingabe und gibt das zugehörige Ergebnis aus, um anschließend einen weiteren REP Zyklus usw. durchzuführen.

Im einfachsten Fall kommt die Eingabe von der Tatstatur in eine Konsole und die Ausgabe erfolgt ebenso in über die Konsole. Ein einfaches Beispiel verdeutlicht diesen Umstand, hierbei wird zunächst der Kernel math.exe aufgerufen, die Eingabe „*1+1*“ über die Tastatur eingegeben worauf der Kernel mit „*2*“ antwortet. Erwähnenswert ist an dieser Stelle, dass alle Eingaben an den Kernel, also auch die n’te Eingabe im System, unter *In[n]* abgelegt werden und zur Verfügung stehen, gleiches gilt analog auch für Ausgaben mit *Out[n]*.

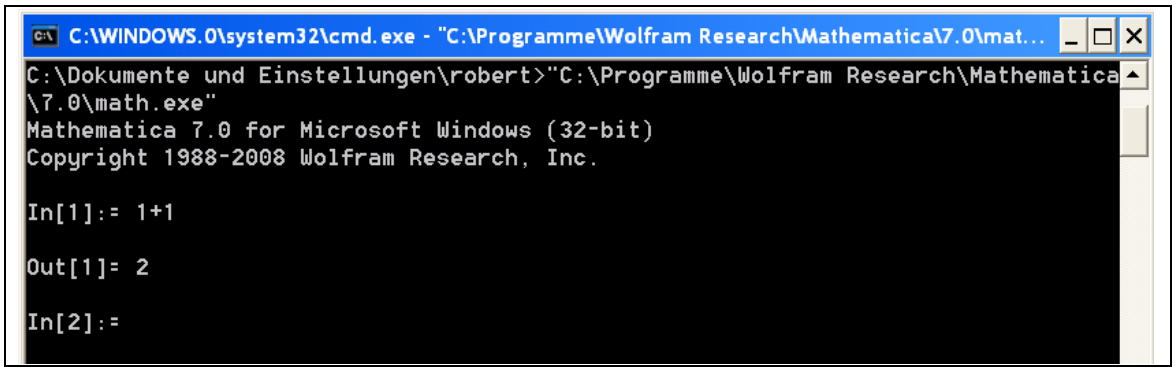

Abbildung 24 „Mathematica Kernel in einer Konsole“

9.2.2 Das Mathematica Frontend/Notebookinterface

Üblicherweise bedient man den Mathematica Kernel über ein Frontend; das so genannte Notebookinterface. Es handelt es sich hierbei um ein GUI Programm zur Erstellung wissenschaftlicher Dokumente (Notebooks) und um eine Schnittstelle zum Mathematica Kernel. Das Frontend wird mittels des Programms Mathematica.exe gestartet, worauf zwei GUI Fenster auf der graphischen Oberfläche des Betriebssystems erscheinen. Zum einen erhält man ein Fenster mit einer Menüstruktur wie man es ähnlich von typischen GUI Programmen gewohnt ist.

Abbildung 25 „Hauptmenue des Mathematica GUI Notebookinterface“

Dieses dient wie üblich zum Laden, Speichern, Ausdrucken, etc. von Dokumenten, sowie für sonstige für Mathematica spezifische Aktionen und Einstellungen.

Zum anderen erhält man ein Fenster, welches einen (graphischen) Editor für das zu erstellende Dokument (Notebook) bereitstellt, für dieses Fenster wird üblicherweise auch das Synonym Notebook verwendet. Dieses Notebook kann nun vom Benutzer editiert werden wie im nachfolgenden Beispiel ersichtlich ist. Hierbei ist eine Eigenheit von Mathematica zu beachten:

Das Betätigen der <RETURN> Taste aus dem Tatstaturhauptfeld fügt im Notebook einen Zeilenumbruch ein. Die Betätigung von <SHIFT RETURN> oder von <ENTER> des Tastaturnummernblockes hingegen bewirkt die Evaluierung, der letzten Eingabe, durch den Kernel sowie die Ausgabe des entsprechenden Ergebnisses im Notebook. Die Ausgabe des Ergebnisses kann mit einem Semikolon unterdrückt werden. Man beachte, dass Leerzeichen in Mathematica, wie in der Mathematik üblich, als Multiplikationsoperator interpretiert werden. Zu bemerken ist weiterhin, dass sämtliche Eingaben wie auch Ausgaben im Notebook zu Cell’s (Zellen) zusammen gefasst werden. Diese Gruppierung findet hierarchisch statt, so dass im nachfolgenden Beispiel die zugehörigen Ein-Ausgabe Cell’s wiederum zu einer übergeordneten Cell gruppiert werden. Graphisch werden die Cell’s am rechten Rand des Notebooks als (rechte) eckige Klammern dargestellt.

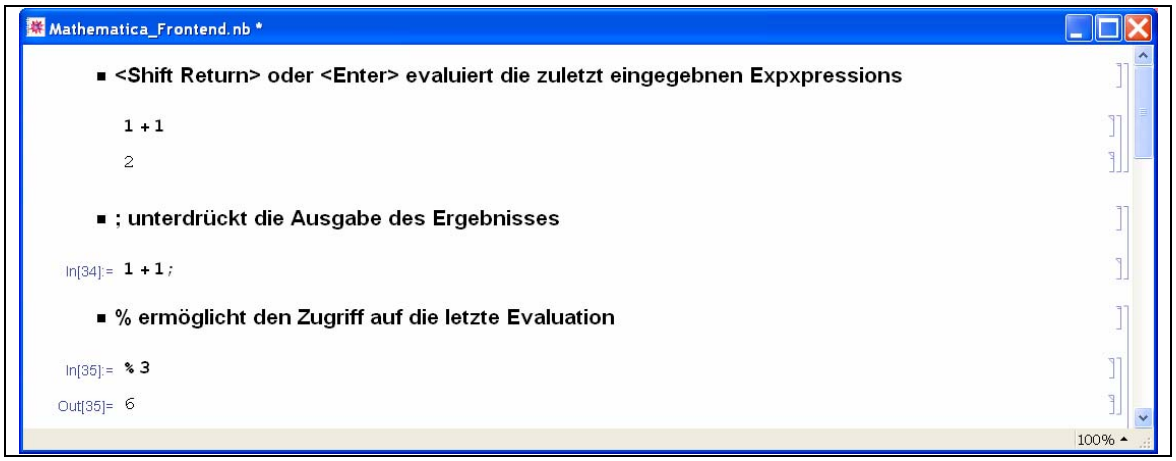

Abbildung 26 „Evaluierung von Mathematica Expression’s“

Um einen ersten Eindruck von der Mächtigkeit des Mathematica-Systems zu vermitteln, sei an dieser Stelle ohne tiefer greifende Erklärungen ein kleiner Mathematica „Einzeiler“ präsen-

tiert , welcher sin(n x) mit einer interaktiven Verstellmöglichkeit von n graphisch darstellt. Die Eingabe des Einzeiler-Codes wird durch den Kernel mit der Ausgabe des entsprechenden Plot's beantwortet. Die solchermaßen generierte Grafik enthält, wie man sehen kann, einen Slider (Schieber) für den Parameter n welcher bei interaktiver Betätigung mit der Maus ein frisches rendern (zeichnen) des Plot's in Realtime bewirkt.

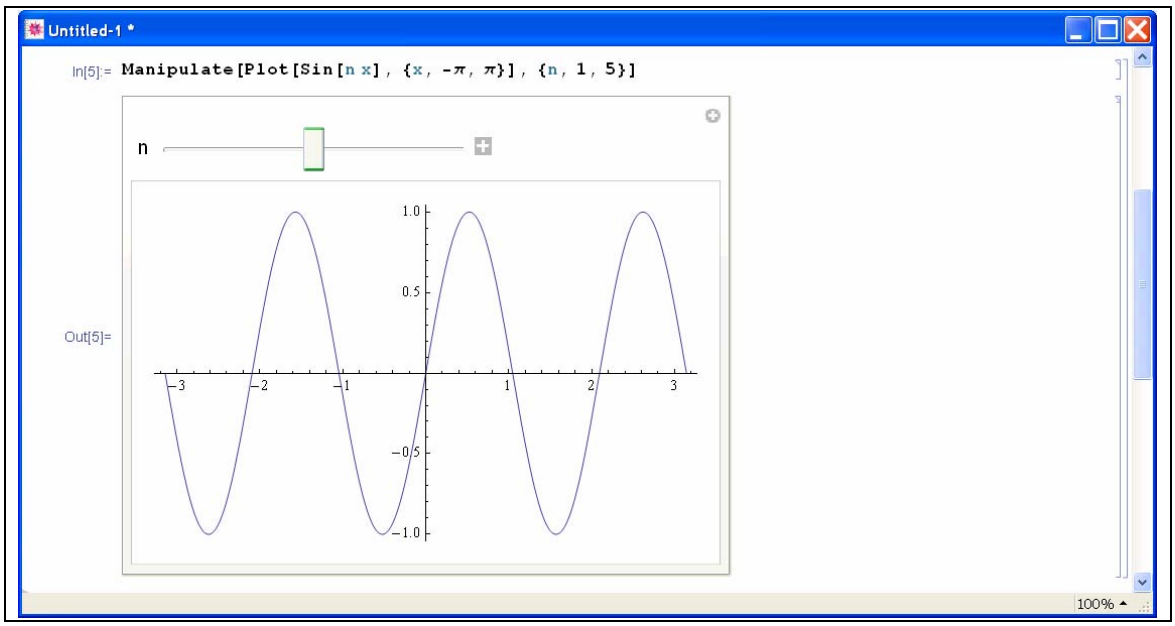

Abbildung 27 „Ein mächtiger Mathematica Einzeiler"

An dieser Stelle sei angemerkt, dass nicht nur Plots oder Grafiken, sondern sämtliche Mathematica Objekte, Ausdrücke, Ausgaben etc. sich in gleicher simpler Weise interaktiv manipulieren lassen.

9.3 Einführung in die Mathematica-Programmierung

Um die Implementierung der vorgesehnen Tools nachvollziehen und verstehen zu können, ist ein Verständnis der Mathematica-Programmierung unverzichtbar. Zu eben diesem Verständnis sollen die folgenden Kapitel dem geschätzten Leser und potentiellen Nutzer der Tools verhelfen.

9.3.1 Prinzipien der Mathematica-Programmiersprache

Die Mathematica Programmiersprache unterstützt in erster Linie Prinzipien der Funktionalen Programmierung, der Prozeduralen Programmierung, dem Patternmatching sowie Konzepte der Objektorientiertheit. Dabei flossen vielfältige Ideen aus verschiedenen Programmiersprachen in Mathematica ein.

Folgende Programmiersprachen dienten teilweise als Vorbild bzw. verfügen teilweise über ähnliche Features wie Mathematica:

- LISP Funktionale Programmierung, Listenverarbeitung
- PROLOG Patternmatching
- APL Operatoren für die Manipulation von Strukturen
- C++ Gewisse Aspekte der Objektorientiertheit
- PASCAL, C Aspekte der Prozeduralen Programmierung

Die folgenden Abschnitte beinhalten zahlreiche Einfügungen aus Präsentationsfolien welche mit Hilfe der Mathematica „Slideshow" Funktionalität erstellt wurden, dies erübrigt die Benutzung eines eigenen Präsentationsprogramms. Damit wird ein weiteres Mal die Universalität des Mathematica-Systems unterstrichen.

9.3.2 Mathematica Expression's

In Mathematica stellen Expression's die fundamentale Datenstruktur dar. Dementsprechend lautet das oberste Prinzip in Mathematica: „Alles ist eine Expression".

Diese Aussage bezieht sich ausnahmslos auf alle Mathematica Objekte wie z.B. Daten, Funktionen, Grafiken, etc. Eine Expression kann hierbei entweder ein „atomares" Objekt wie z.B. eine Zahl, ein String, ein Symbol, oder aber eine zusammengesetzte Expression sein. Letztere Expression verfügt über einen Kopf, genannt *Head*, sowie innerhalb eines eckigen Klammern-Paares, über eine Auflistung, genannt *Sequence*, von Elementen, welche ihrerseits entweder atomar oder zusammengesetzt sein können. In Mathematica stehen eine Vielzahl von Funktionen (welche selbstverständlich auch Expression's sind) zur Manipulation von Expression's zur Verfügung. So liefert die Mathematica Funktion *Head* angewandt auf eine Expression eben den *Head* der Expression, genauso wie das „applizieren" mittels @@ (Erläuterung folgt später) von *Sequence* auf die Expression die zugehörige *Sequence* der Elemente der Expression liefert.

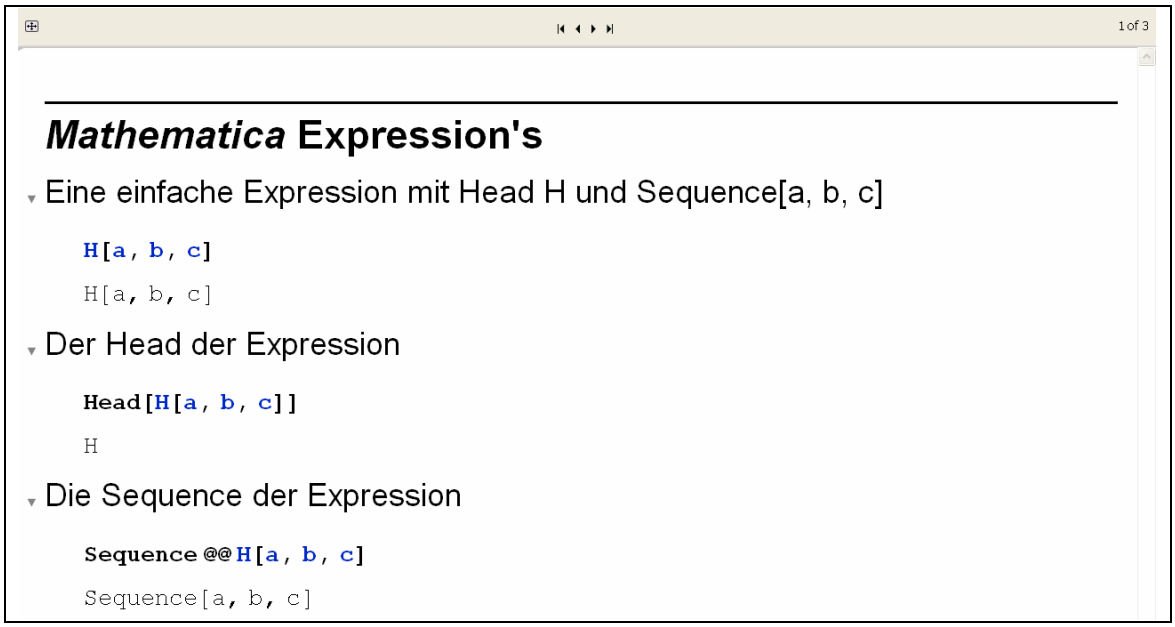

Abbildung 28 „Mathematica Expression's"

Weiters sei *AtomQ* vorgestellt, dies ist eine Test-Funktion welche, angewandt auf eine nicht weiter strukturierte Expression, den Wahrheitswert *True* liefert, ansonsten im Falle einer allgemeinen nicht atomaren Expression den Wert *False*.

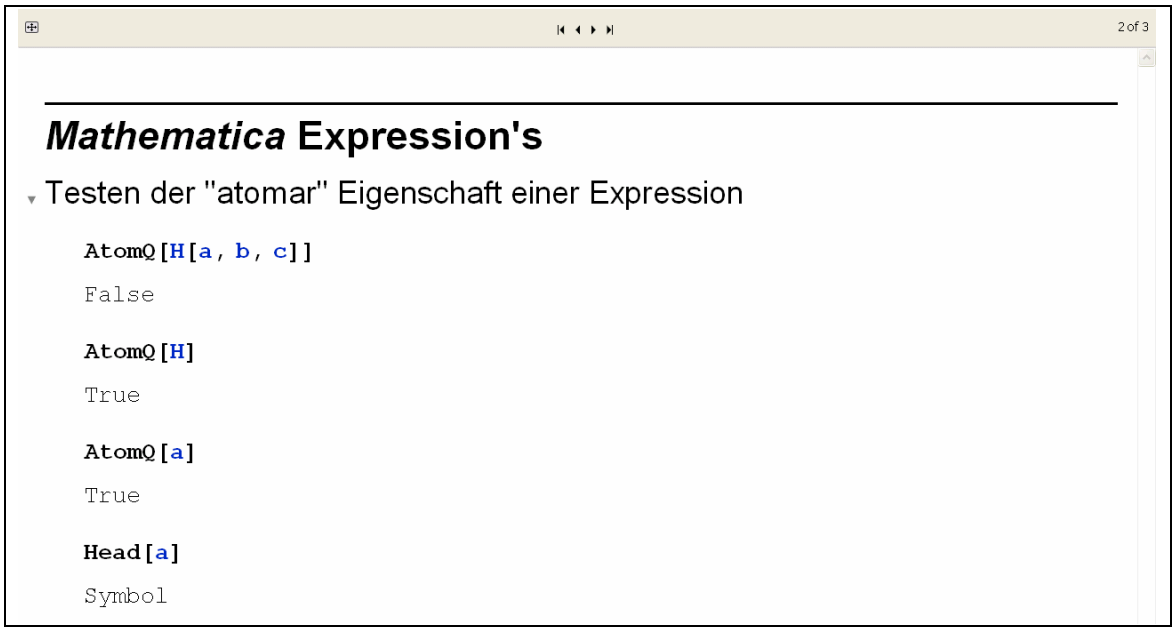

Abbildung 29 „Atomare Expression's"

Mathematica Symbol's sind atomare Expression's. Symbol's können Werte zugewiesen bekommen und können so als „klassische" Variablen dienen. Alle Zuweisungen an ein Symbol können auch wieder gelöscht werden.

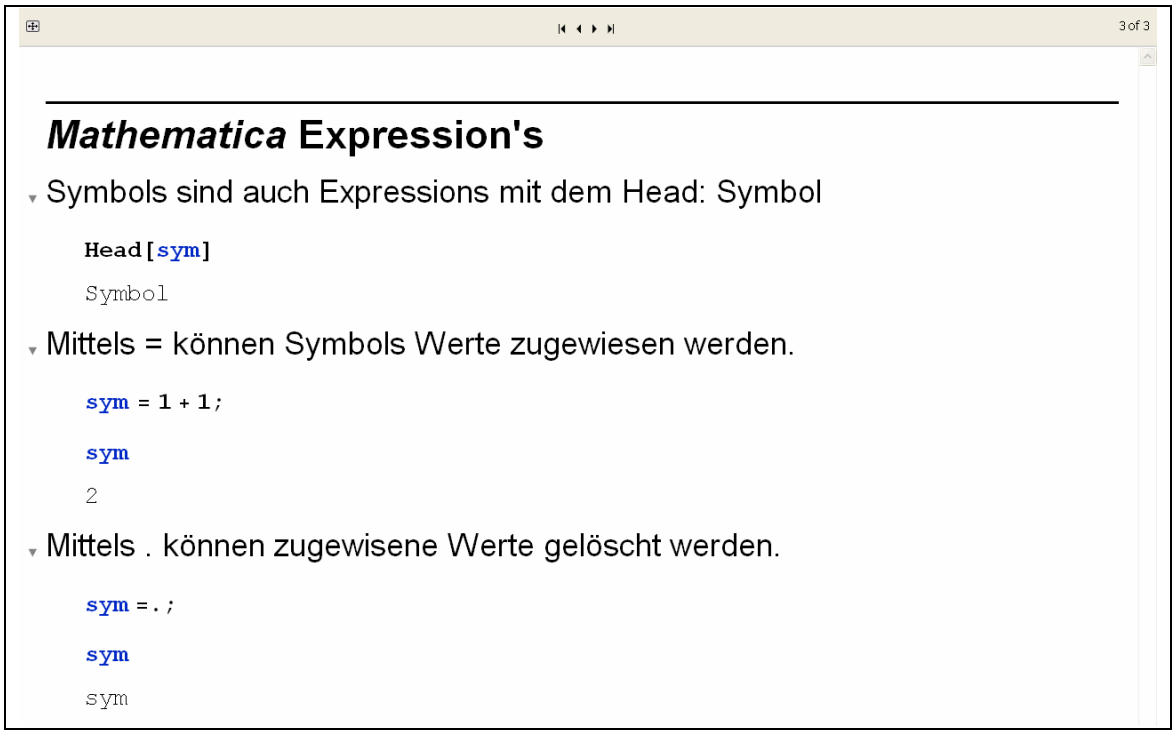

Abbildung 30 „Mathematica Symbol's"

9.3.3 Die Evaluation von Expression's

Die Evaluation einer Expression geschieht (vereinfacht) wie folgt dargestellt:

- Der *Head* der Expression (auch eine Expression) wird evaluiert, gefolgt von der Evaluation der Elemente (auch Expression's) der zugehörigen *Sequence*.
- Ist eine Expression ein „atomares" Objekt (z.B. *Integer*, *String*, "freies" Symbol, etc.) wird die Evaluation beendet, die Expression wird nicht weiter verändert.
- Ist eine Expression kein „atomares" Objekt, so wird diese durch passende, dem System bekannte, Transformationsregeln modifiziert.
- Die obigen Regeln werden so oft wiederholt angewendet, bis sich das Ergebnis der Evaluation der Expression nicht weiter verändert.
- Mittels *Trace* lassen sich die Evaluations-Abläufe verfolgen.

9.3.4 Die wichtigste Mathematica Datenstruktur, die Liste

Mathematica nutzt für das ablegen strukturierter Daten im Allgemeinen die Liste als Datenstruktur. Eine Mathematica-Liste besteht immer aus einer Sequenz von beliebig vielen Elementen. Diese Sequenz von Elementen wird von einem eckigen Klammernpaar eingeschlossen, welchem das Symbol *List* als Mathematica-Expression-Head vorangestellt wird.

Der einfachste Gebrauch einer Mathematica-Liste ist jener, bei welchem eine beliebige Anzahl von gleich gearteten Elementen in einer Liste zusammengefasst wird. Dies entspricht in

etwa dem Gebrauch von Arrays in klassischen Programmiersprachen. Üblicherweise wird für Mathematica-Listen eine ansprechendere Notation gewählt. Bei dieser Notation wird eine Sequenz von Elementen von einem geschwungenen Klammernpaar umschlossen. Diese Notation ist gleichwertig zu der eingangs erläuterten Form als Expression mit dem *Head*: *List*.

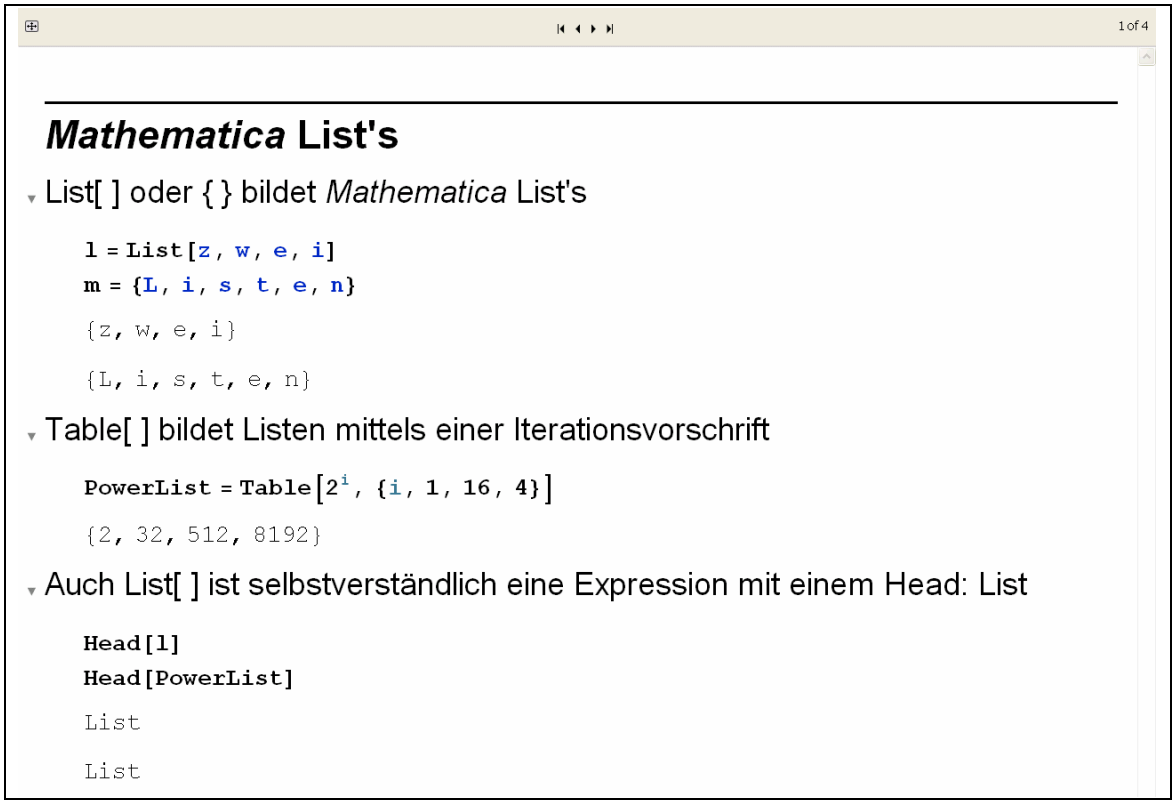

Abbildung 31 „Mathematica List's"

Zur Manipulation von Mathematica List's existiert eine Vielzahl von Funktionen. Der indizierte Zugriff auf Listenteile erfolgt mittels einem doppelten eckigen Klammernpaar *[[ ]]*

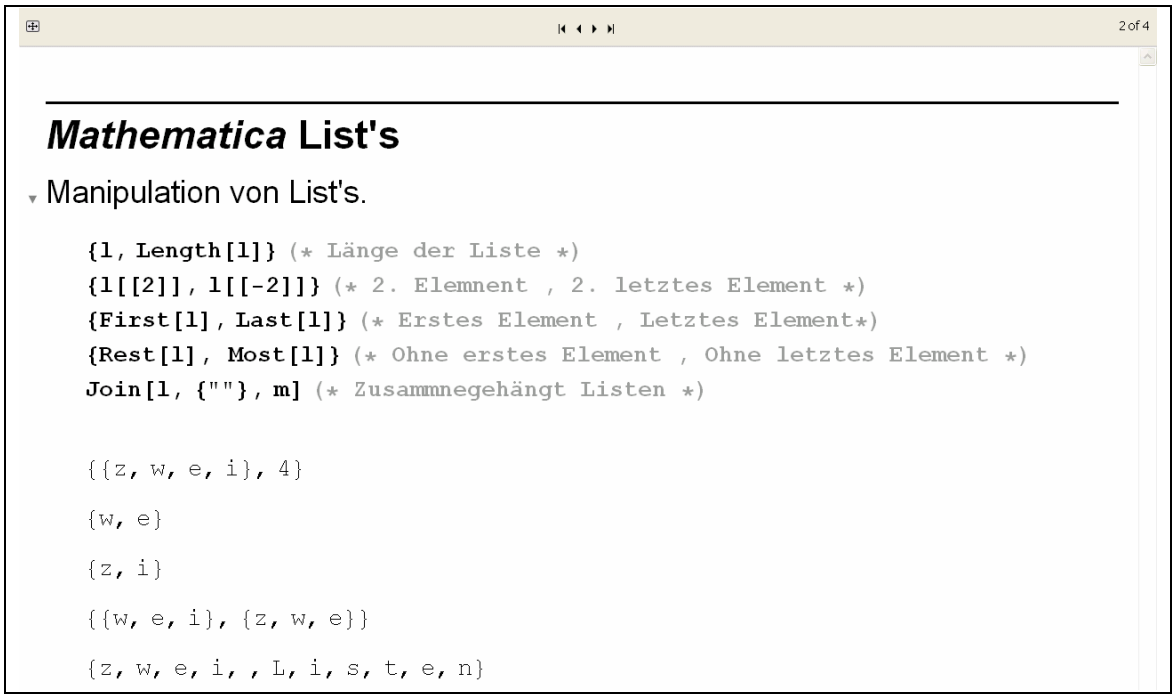

Abbildung 32 „Manipulation von List's

Listen können selbstverständlich mehrdimensional sein und auch ansprechend formatiert dargestellt werden. Zweidimensionale Listen können als Matrizen interpretiert werden, hierfür gibt es eine Vielzahl von Matrixfunktionen. Listen müssen nicht homogen sein (unterschiedliche Elementtypen sind möglich).

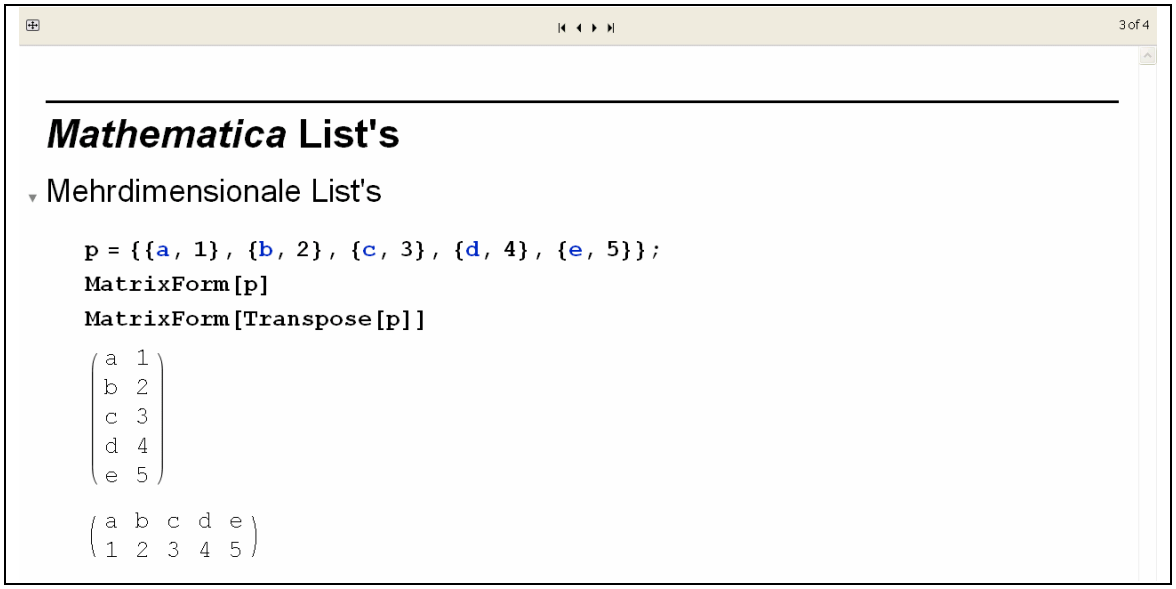

Abbildung 33 „Mehrdimensionale List's"

Sublisten können unterschiedlich lang sein. Damit sind Baumstrukturen möglich, welche sich anschaulich mittels *TreeForm* darstellen lassen.

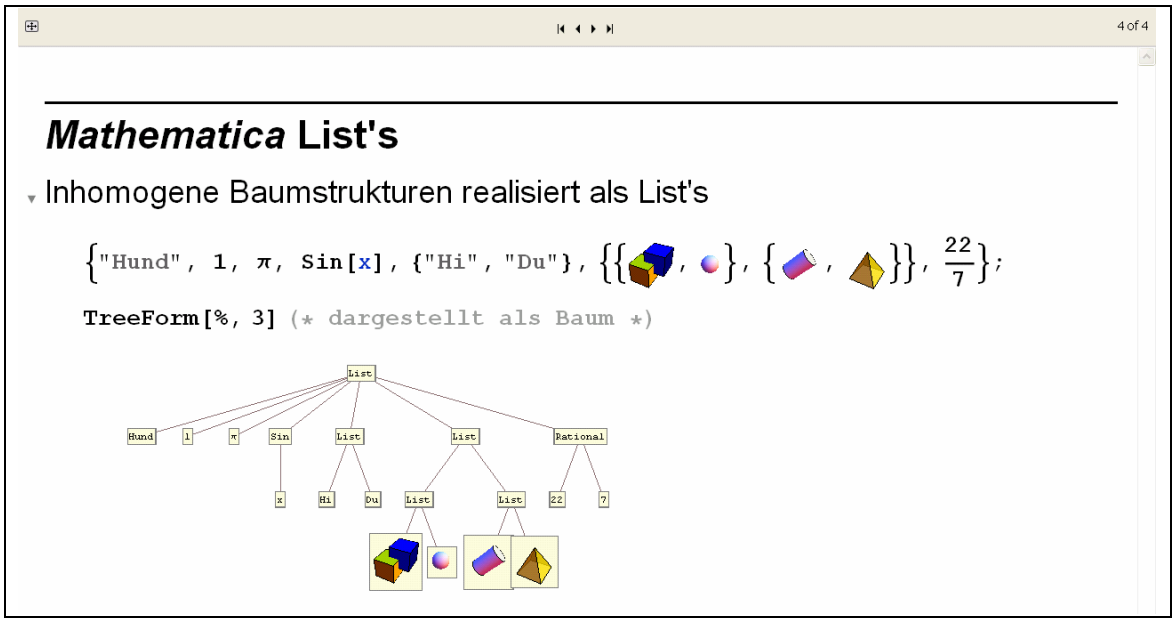

Abbildung 34 „Baumstrukturen mit Mathematica“

9.3.5 Mathematica Operator Formen

Das Anwenden von Funktionen ist in Mathematica in verschiedenen Formen möglich. Einerseits ist es möglich, einen „klassischen“ Funktionsaufruf zu tätigen, andererseits können die Operator-Formen Prefix, Infix, Postfix verwendet werden. All diese Möglichkeiten sind gleichwertig und dienen, bei überlegtem Einsatz, der besseren Lesbarkeit von Mathematica Code.

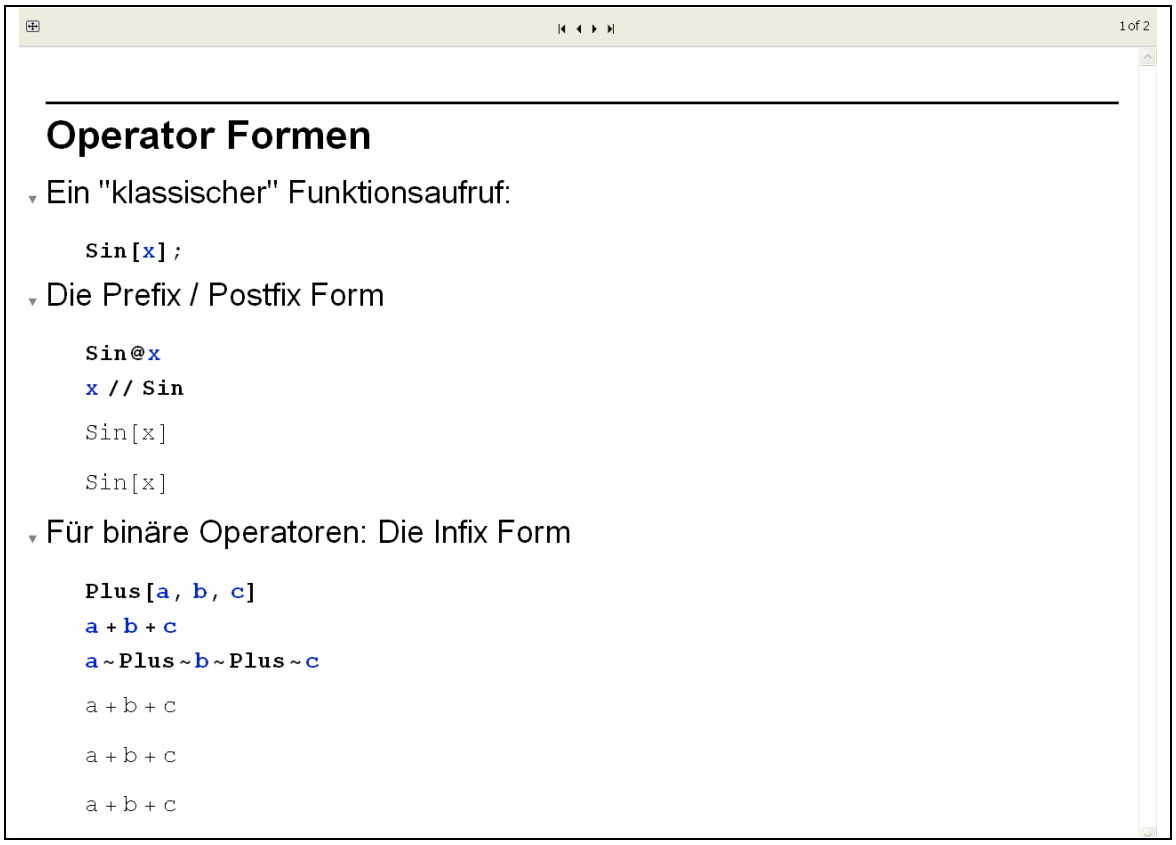

Abbildung 35 „Mathematica Operator Formen“

Mathematica stellt noch weitere, spezielle Operator-Formen zur Verfügung. Die wichtigsten dieser Formen sind *Map* und *Apply*. *Map* wird bei Listen angewendet und bewirkt, dass die zu „mappende“ Funktion sequentiell auf jedes Listenelement angewendet wird. *Apply* ist eine Hilfsfunktion welche es ermöglicht, Funktionen, die ursprünglich eine *Sequence* von mehreren Parametern erwarten, auf eine einzelne Liste, eben dieser Parameter, anzuwenden. Für *Map* existiert die Operator Kurzform /@. Für *Apply* gibt es die Abkürzung @@.

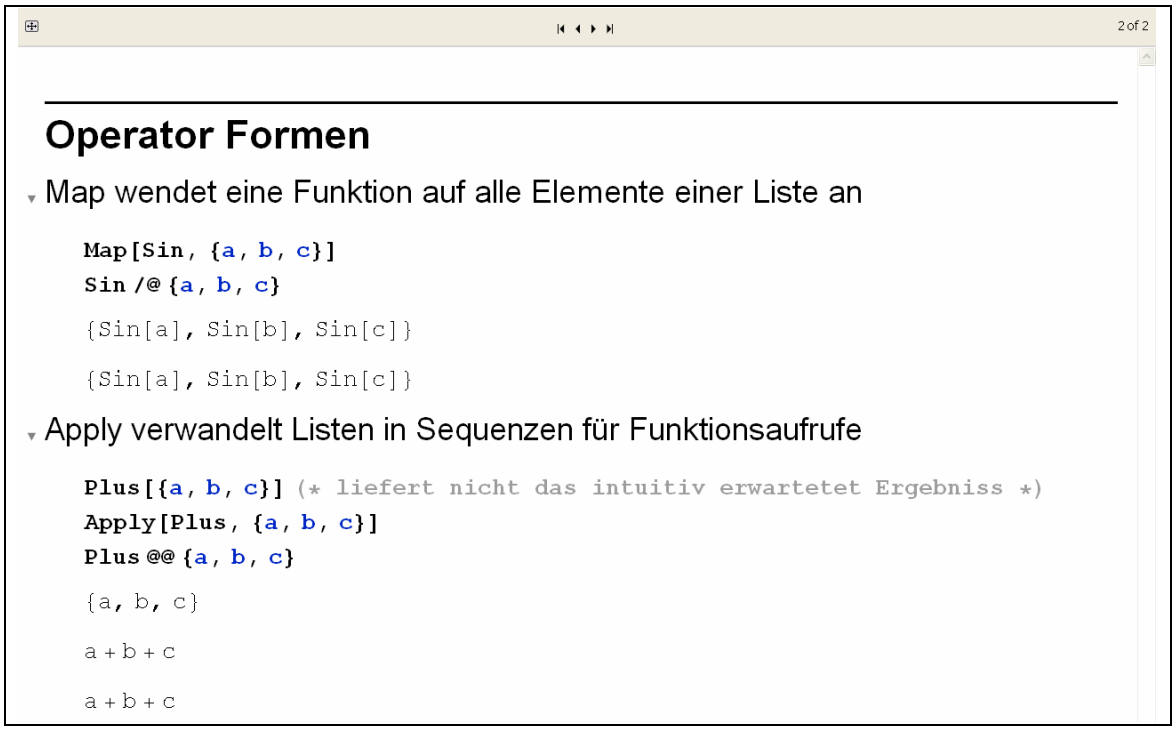

Abbildung 36 „Mathematica Map und Apply Operatoren“

9.3.6 Mathematica Ersetzungs-Regeln, Options

Rule[ ] mit seiner Operator Form -> sowie *ReplaceAll* mit der Operator Form */.* ermöglichen es, beliebige Mathematica Expression’s zu manipulieren, indem Teile der selbigen damit ersetzt werden können. Einige Mathematica Funktionen liefern als Ergebnis Listen von Ersetzungsregeln.

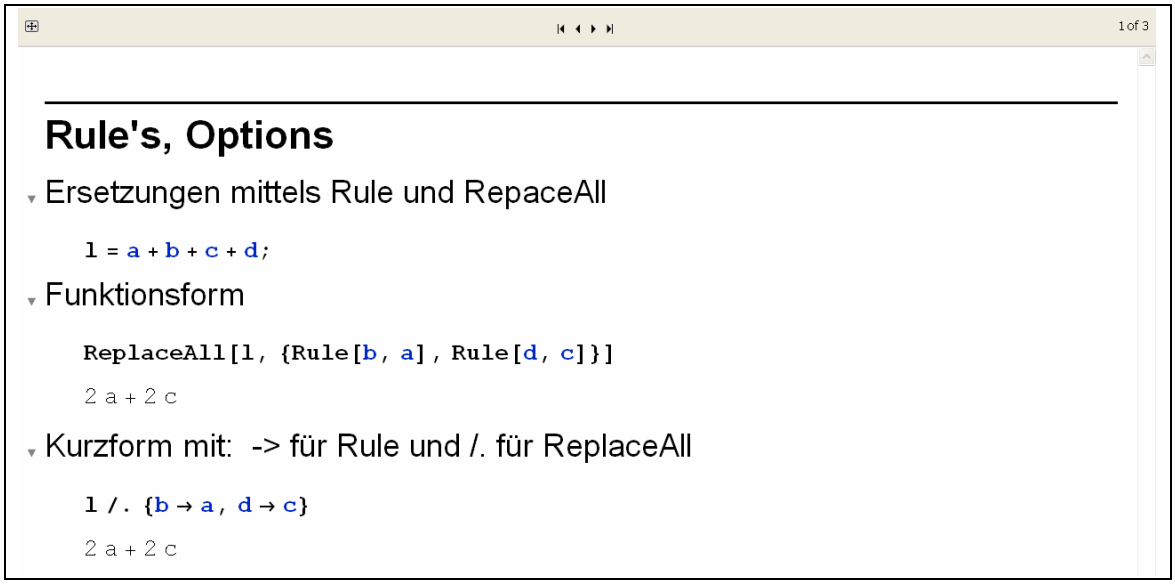

Abbildung 37 „Mathematica Rule’s und Options“

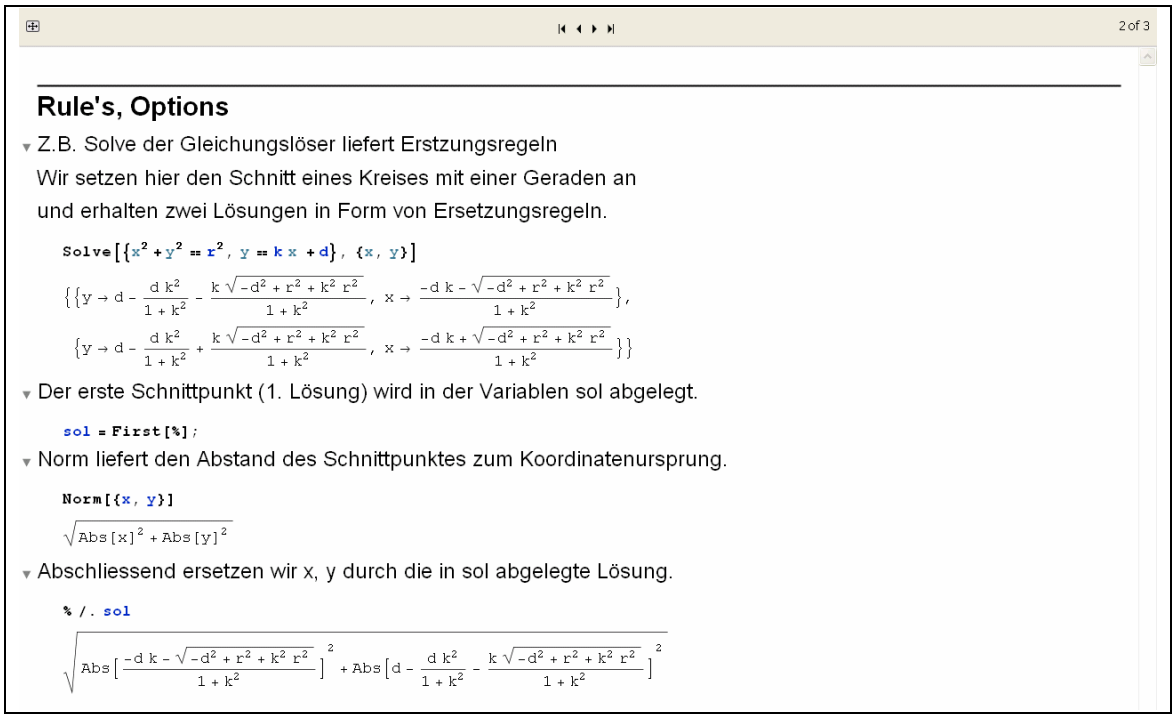

Abbildung 38 „Rule's als Ergebnis vom Mathematica Standardfunktionen“

Viele Mathematica Funktionen akzeptieren optionale Parameter, genannt *Options*. Diese *Options* werden als Ersetzungsregeln (*Rule*) übergeben.

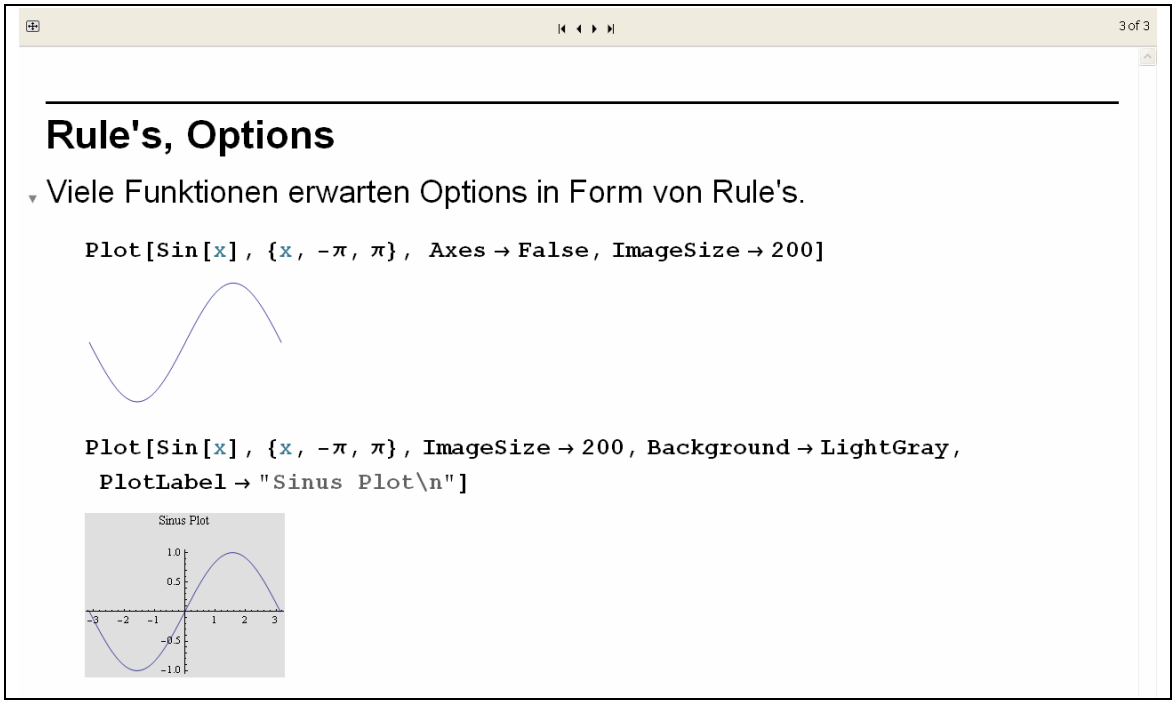

Abbildung 39 „Rule's für Optionale Parameter“

9.3.7 Mathematica Kontrollstrukturen, Funktionsdefinitionen

„Klassische“ Kontrollstrukturen wie z.B. Fallabfragen, Schleifen, etc. sind in Mathematica als Funktionen realisiert. Dieser Umstand unterstreicht den funktionalen Aspekt der Mathematica Programmierung.

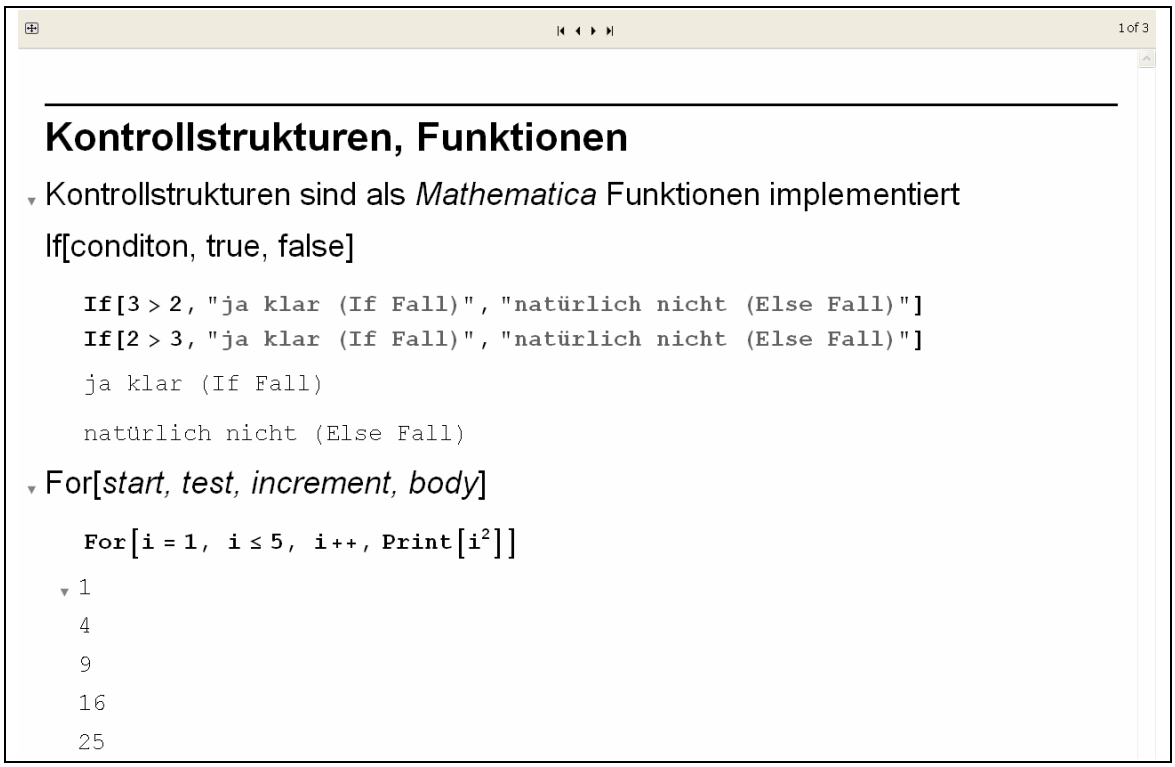

Abbildung 40 „Klassische Kontrollstrukturen in Mathematica“

Funktionen mit Parametern und lokalen Variablen werden mittels Module definiert.

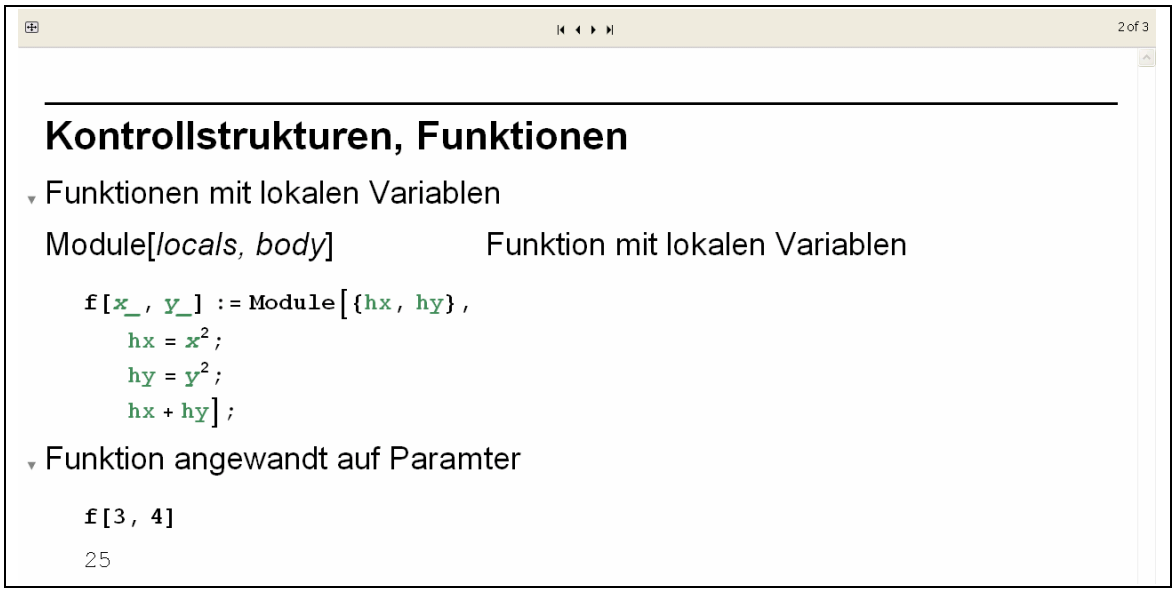

Abbildung 41 „Funktionen in Mathematica“

Angelehnt an das Lambda Kalkül der Theoretischen Informatik sind in Mathematica auch so genannte „Anonyme Funktionen“ mittels *Function[ ]* bzw. der Kurzform *&* implementiert. Bei der Kurzform dient *#1* als erster, *#2* als zweiter Parameter, usw. der anonymen Funktion, *&* schließt die Funktionsdefinition ab.

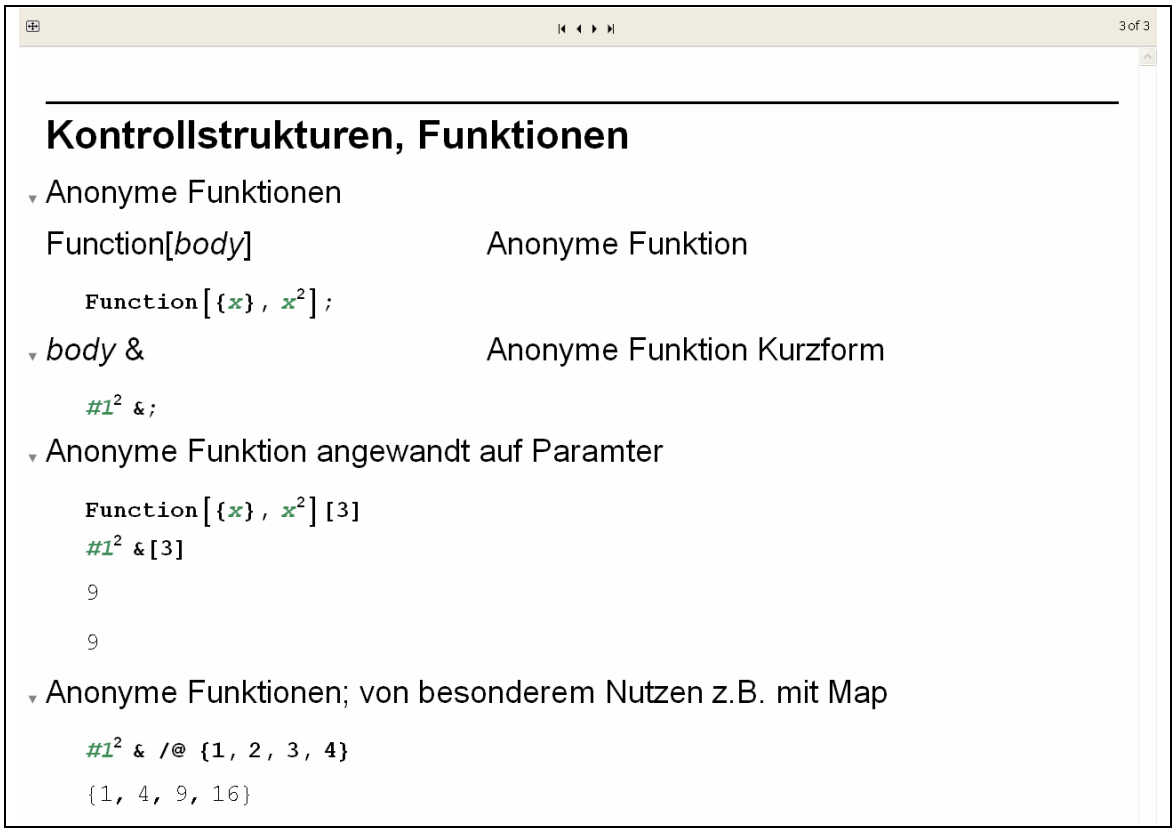

Abbildung 42 „Anonyme Funktionen“

9.4 Das Mathematica (2D) Graphics System

9.4.1 Graphics primitives und Directives

Alle Mathematica-Grafiken werden aus so genannten *Primitives*, wie z.B. *Point*, *Line*, *Polygon*, *Text*, *Circle* etc. für die Repräsentation geometrischer Objekte, aufgebaut. Des Weiteren können die grafischen Objekte mittels so genannter *Directives* formatiert werden. Die *Directive's* können z.B. *PointSize*, *Thickness*, *RGBColor*, *Red*, *Blue*, *FontSize*, etc. für eine entsprechend gewünschte Formatierung sein. Die *Primitives* und *Directives* werden in *List's* (linear oder hierarchisch) organisiert. So eine Mathematica Liste, folgend *gList* genannt, wird von *Graphics[ ]* zu einer Expression umschlossen: Eine Struktur der Form *Graphics[gList]* (eine Expression) wird in Mathematica auch als (2D) *Graphics-Object* bezeichnet (*Graphics3D[ ]* repräsentiert 3D Grafiken, welche für diese Arbeit nicht weiter von Bedeutung sind). Die Aufgabe der *Primitives* ist es also, Grundelemente wie z.B. Punkte, Linien, Poly-

gone, Texte, Kreise, etc. zur Verfügung zu stellen. Die Aufgabe der *Directives* ist es, bestimmte Eigenschaften der Grundelemente wie z.B. Farbe, Strichstärke, Schriftgröße, etc. festzulegen. Ein *Directive* gilt für alle sequentiell nachfolgenden bzw. hierarchisch untergeordneten *Primitives*, solange bis es durch ein nachfolgendes Directive mit unterschiedlicher Parametrisierung abgelöst wird. Alle eingebauten higher-level Grafik Funktionen wie z.B. *Plot[ ]* liefern als Ergebnis eine Expression in Form eines *Graphics-Object* wie oben beschrieben. Die Liste *gList* welche die Grafik spezifiziert, kann im allgemeinen Fall auch als Baumstruktur aufgebaut sein. So ist es möglich, Gruppen von *Primitives* zu bilden, deren gemeinsame Eigenschaften mittels zugehöriger *Directives* hierarchisch spezifiziert werden können.

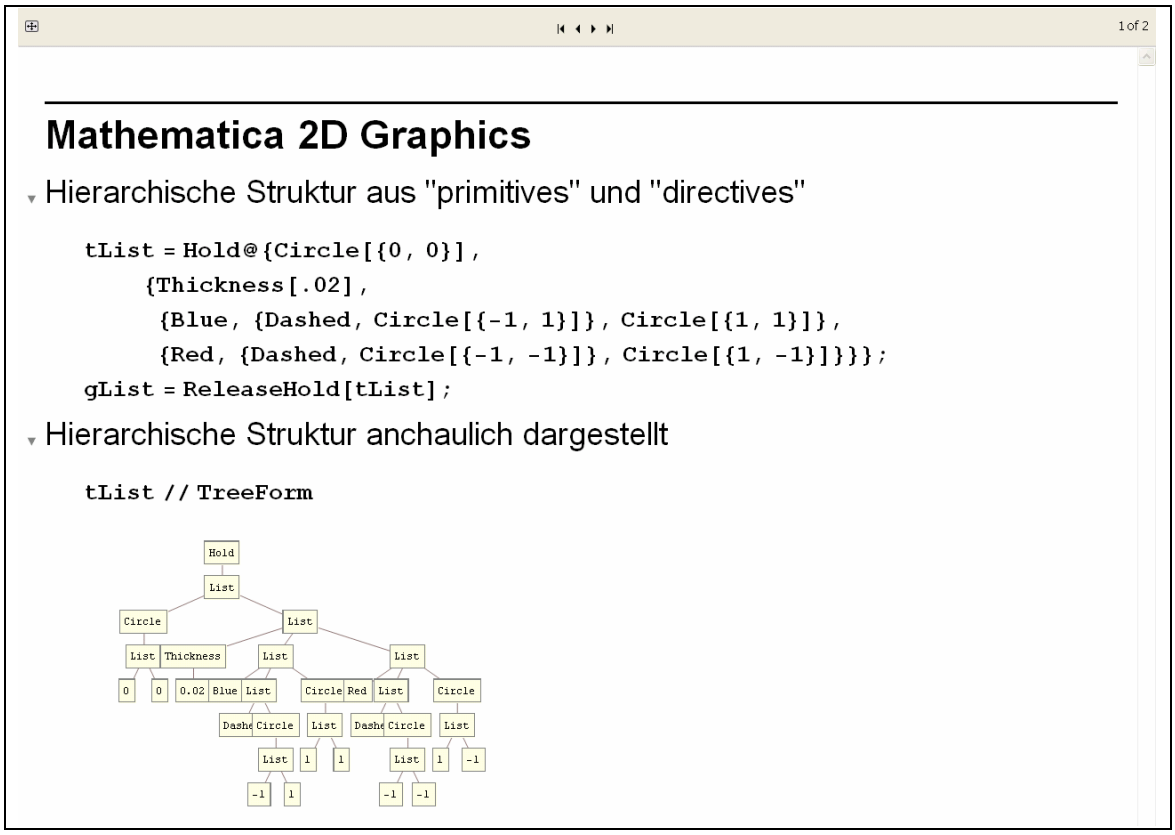

Abbildung 43 „Struktur und Aufbau von Mathematica-Grafiken“

Wird dem Mathematica Frontend ein *Graphics-Object* in der Form: *Graphics[gList]* übergeben, so wird defaultmäßig die mittels *gList* repräsentierte Grafik vom System gerendert und ausgegeben.

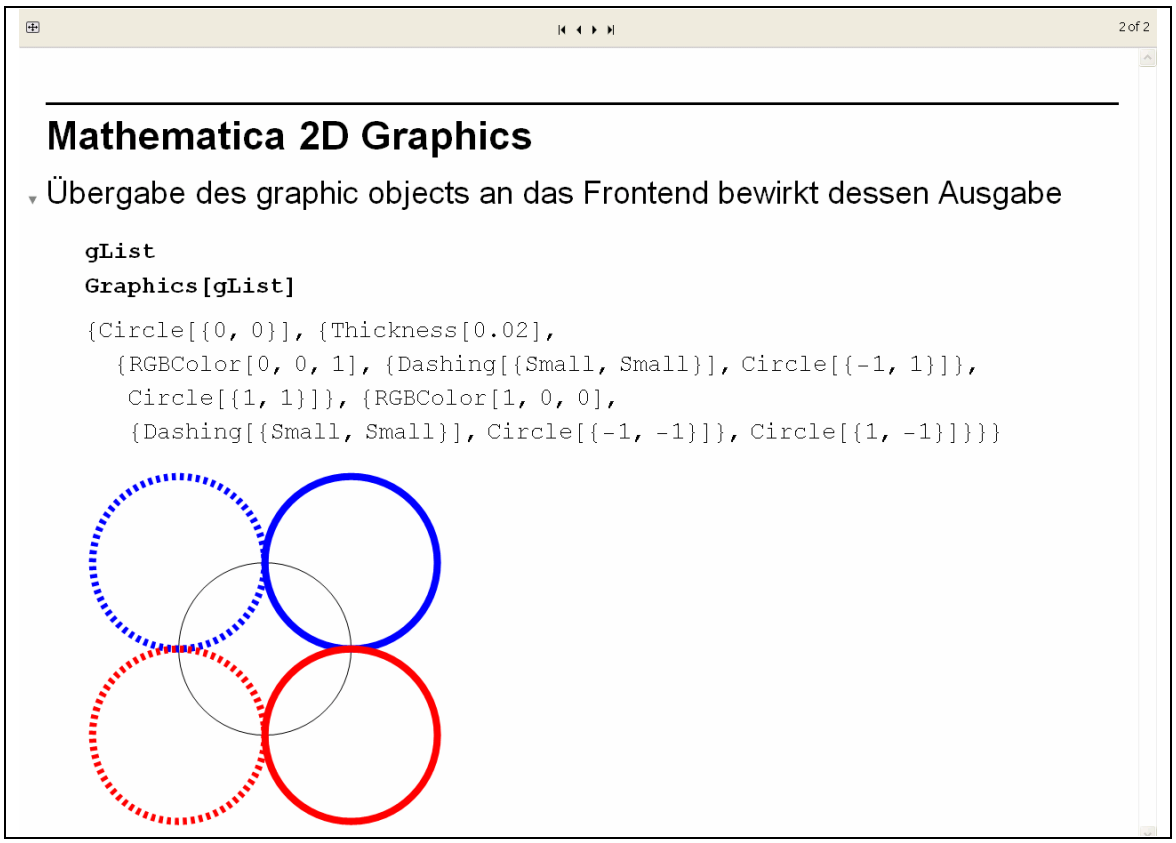

Abbildung 44 „Ausgabe von Grafiken durch Mathematica“

9.4.2 Die wichtigsten Graphics Primitive

- **Line, Polygon**

Wichtige „primitive“ im Zusammenhang mit dieser Arbeit sind *Line* und *Polygon*. In Mathematica repräsentiert *Line[{p1, p2, …pm}]* m Stück Linienzüge wobei die einzelnen Linienzüge *pi* jeweils als Listen von n Stück *{xi/yi}* Koordinatenpaaren spezifiziert werden: *pi: {{x1,y1},{x2,y2},…{xn,yn}}*. Mittels *Polygon[{p1, p2, …pm}]* werden m Stück Polygone spezifiziert, wobei die Polygonpunkten wie bei *Line* als Listen von *{x,y}* Koordinatenpaaren spezifiziert werden. Polygone werden mit der default Füllfarbe eingefärbt und sind implizit geschlossen, d.h. der erste Punkt dient gleichzeitig auch als letzter Punkt und muss nicht explizit wiederholt werden.

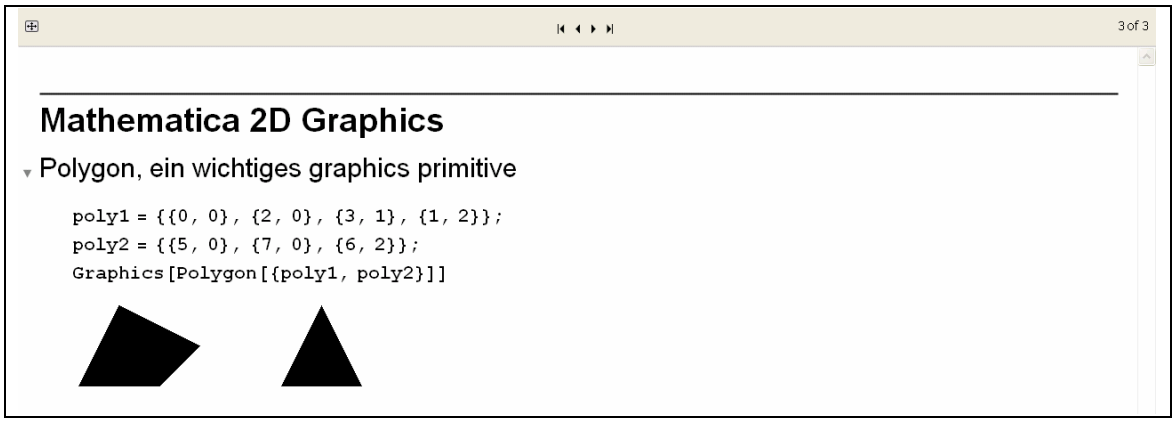

Abbildung 45 „Mathematica-Graphics Polygon“

9.5 Mathematica Export/Import Standard Funktionen

Das Mathematica-System unterstützt standardmäßig eine große Anzahl von Dateiformaten. Die unterstützten Formate stammen unter anderem aus folgenden Bereichen:

- Rastergrafik
- Vektorgrafik
- 3D Grafik
- Audio
- Multimedia
- Tabellenkalkulationsformate
- Datenbankformate
- WEB - und Dokumentformate
- XML - Formate
- Wissenschaft, Medizin
- Geodäsie
- Kompressionsformate
- Binärformate

Wie in der Einleitung bereits festgestellt, befindet sich GDSII nicht unter den von Mathematica unterstützten Formaten.

Die Import/Exportfunktionen fungieren als Bindeglied zwischen der internen Mathematica eigenen Repräsentation von Daten in entsprechenden Expression’s und den externen Daten, welche in unterschiedlichsten Formaten in Dateien oder noch allgemeiner als URL’s vorlie-

gen. Dabei erlaubt der Mathematica Import/Export „Elements“ Mechanismus eine detaillierte Spezifikation des Zugriffes auf die externen Daten.

- **Export**

Export[file, expr, elems] dient zum Export von Daten in verschiedensten Formaten. Hierbei wird eine Mathematica Expression *expr* z.B. ein *Graphics-Object* in einem spezifiziertem Grafikformat (JPG, GIF, PNG, etc.) in einer spezifizierten Datei *file* abgespeichert. Mittels des optionalen Parameters *elems* erfolgt, falls gewünscht, eine detaillierte Spezifikation von Parametern des gewählten Formates. Ziel dieser Arbeit ist es, unter anderem, eine Exportfunktion auch für das standardmäßig nicht unterstützte GDSII Format zu implementieren.

- **Import**

Import dient zum Import von Daten. *Import[file, elems]* importiert, sofern es sich um ein Mathematica bekanntes Format handelt, die in *file* enthaltenen Daten, oder durch Angabe des optionalen Parameters *elems* Datenteile, Metadaten, Formatparameter, etc. des entsprechenden Formates. Die Daten werden beim Import in entsprechende Mathematica Expression’s umgewandelt und als solche vom System anschließend dargestellt. Im beispielhaften Fall einer „*.jpg“ Grafik-Datei wird die enthaltene Grafik gerendert und anschließend vom System dargestellt. Auch die Importfunktionalität wird im Rahmen dieser Arbeit für das GDSII Format implementiert.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Robert Nowak
Wien, Juli 2009