Entwicklung eines Compilers für die Sprache „GB-J" für den Game Boy

Julius Clasen

31.3.2019

Informatik (IF7), Jgst. 12, Herr Faßbender

Inhaltsverzeichnis

1 Einleitung **3**

2 Die Sprache „GB-J“ **3**
- 2.1 Sektionen und Klassen . . . 3
- 2.2 Importe . . . 4
- 2.3 Datentypen . . . 5
- 2.4 Arrays . . . 5
- 2.5 Kontrollstrukturen . . . 6
- 2.6 Objekte . . . 7
- 2.7 Einbinden von Assemblercode . . . 8
- 2.8 Standardbibliothek . . . 9
  - 2.8.1 Sprites.asm . . . 9
  - 2.8.2 Tiles.asm . . . 10
  - 2.8.3 LCD.asm . . . 11
  - 2.8.4 Utils.asm . . . 11
  - 2.8.5 GBPrinter.asm . . . 12
  - 2.8.6 ScreenPrinter.asm . . . 13
- 2.9 Benötigte Methoden . . . 14

3 Lexer **14**

4 Preliminary Parser **15**

5 Parser **15**

6 Code-Generator **17**
- 6.1 StdlibCallGenerator . . . 18

7 Optimizer **19**

8 Ausführung von RGBDS **19**

9 Verwendung des Compilers **19**

10 Der Game Boy **19**
- 10.1 Hardware . . . 19
- 10.2 Besonderheiten beim Programmieren . . . 21
  - 10.2.1 Grafik . . . 21
  - 10.2.2 Speicher . . . 22
- 10.3 GB Printer . . . 23
- 10.4 Der Emulator BGB . . . 24
  - 10.4.1 Steuerung . . . 26

11 Beispielprogramm **26**

12 Fazit 26

13 Quellenverzeichnis 27

14 Versicherung der selbstständigen Erarbeitung 29

15 Einverständniserklärung zur digitalen Veröffentlichung dieses Dokumentes auf der Webseite des Städt. Gymnasiums Rheinbach 29

1 Einleitung

Als ich von der Möglichkeit einer zusätzlichen Lernleistung erfuhr und mich mit Herrn Faßbender darüber unterhielt, war mir relativ schnell klar, dass ich ein Projekt mit dem Game Boy machen wollte.

Herr Faßbender schlug zu Anfang einen Compiler für den J.O.H.N.N.Y.-Simulator vor, aber leider existierte dazu kein richtiges Gerät, und der Simulator war relativ limitiert. Als Alternative schlug ich daraufhin den Game Boy vor, da ich bereits eine Weile damit gearbeitet hatte und auch sehr genaue Emulatoren verfügbar sind, die zudem noch mehr Debug-Features als der J.O.H.N.N.Y.-Simulator bieten.

2 Die Sprache „GB-J“

„GB-J“ ist die Programmiersprache, die der Compiler übersetzt. Sie ist an Java angelehnt, weist aber einige Unterschiede auf.

Ich entschied mich, eine eigene Sprache zu entwickeln, da C oder Java sowohl zu umfangreich als auch zu restriktiv in der Implementation waren. So konnte ich mir beispielsweise selbst aussuchen, welche Strukturen ich einbaute oder wie groß die einzelnen Datentypen waren. Zudem ließen sich Konzepte wie die verschiedenen Speicherbänke so einfacher in GB-J implementieren.

2.1 Sektionen und Klassen

In GB-J gibt es zwei Arten von Organisationsstrukturen: *Sektionen*, die statisch allozierte Variablen und darauf zugreifenden Code enthalten, und *Klassen*, aus denen Objekte instanziiert werden können. Hierbei wird der Speicher zur Laufzeit alloziert.
Sektionen sind deutlich schneller als Klassen, da Speicherzugriffe direkt ausgeführt werden können, anstatt erst langwierig Adressen zu berechnen.
Zur Veranschaulichung zwei Beispiele zur Deklaration einer Sektion und einer Klasse:

Listing 1: Beispiel einer Sektion

```
section Test
{
    package "ROM0";

    void method()
    {
        //Kommentar
    }
}
```

Listing 2: Beispiel einer Objektklasse

```
class TestObject
{
    package "ROMX, BANK[2]";

    void method()
    {
        //Kommentar
    }
}
```

Es fällt auf, dass die üblichen Attribute „private“ und „public“ fehlen. Dies liegt daran, dass sie in GB-J nicht existieren. Alle Methoden sind jederzeit über die Referenz auf ein Objekt oder, im Falle einer Sektion, von überall aus aufrufbar.

Die „package“-Deklarationen befinden sich zudem hier in der Sektion/Klasse und verhalten sich etwas anders als die von Java bekannten: Statt einen beliebigen Namen zu verwenden, wird diese Deklaration an den Assembler weitergegeben. Sie bestimmt, wo die Klasse im ROM gespeichert wird, was auch für Bankswitching wichtig ist. Hier sind zwei Varianten zu sehen: „ROM0“, die immer verfügbare Speicherbank, und „ROMX“, welches eine beliebige ROM-Bank (außer Bank 0) bezeichnet.

Die Hauptsektion mit der „main“-Methode muss sich in ROM0 befinden. Beim Hinzufügen von „, BANK[Zahl]“ zu „ROMX“ wird eine bestimmte Bank forciert. Beim Bankswitching muss die Nummer der Bank bekannt sein, weswegen dies durchaus sinnvoll ist.

2.2 Importe

Um in einem Projekt mehrere Dateien verwenden zu können, müssen alle importiert werden. Es ist sowohl möglich, weitere „.gbj“-Quellcodedateien zu importieren als auch Assemblerdateien im RGBDS-Format (deren Methoden müssen allerdings über Inline-Assembly aufgerufen werden). Die Dateierweiterung ist bei Assemblerdateien irrelevant (wobei „.asm“ die allgemein akzeptierte ist; „.z80“ wird [fälschlicherweise, die CPU ist kein Z80] von einigen Exportern verwendet).

Listing 3: Beispiel-Importe

```
import "Tiles.z80";
import "Sprites.asm";
import "Enemies.gbj";
```

Importe sind immer oberhalb einer Klasse oder Sektion zu schreiben (genau wie in Java). Der Compiler überträgt bei Importen von Assemblerdateien den Import direkt in die generierte Assemblerdatei; bei GB-J-Dateien wird vorher die importierte Datei

bearbeitet, so dass Importe von dort ebenfalls gültig sind und die Methoden in der Hauptdatei verwendbar sind.

2.3 Datentypen

In GB-J gibt es drei Datentypen:

Tabelle 1: Datentypen

Name	Größe (Bytes)	Reichweite (unsigned)	Reichweite (signed)
char	1	0 - 255	-128 - 127
int	2	0 - 65535	-32768 - 32767
Objekt	2 (verwendet zusätzlich Heap-Speicher)	/	/

Für primitive Datentypen wird bereits beim Compilevorgang Speicher reserviert; bei Objekten ist dies hingegen nur für die Referenz der Fall. Der eigentliche Speicher (die Größe entspricht der addierten Größe aller Variablen im Objekt) wird erst zur Laufzeit reserviert.

Da dies (im Vergleich zu modernen PCs, wo objektorientierte Programmierung problemlos möglich ist) relativ langsam ist, ist es nicht sinnvoll, viele Objekte zu erzeugen und zu vernichten.

Im Gegensatz zu Java ist es möglich, „signed“- und „unsigned“- Werte zu verwenden. „signed“-Werte haben kein Vorzeichen, weswegen sie bei gleicher Größe eine größere Reichweite haben. Zu beachten ist, dass „signed“ und „unsigned“ sich nur durch die Interpretation unterscheiden. Zur Sicherheit sollte deswegen angenommen werden, dass die Variablen immer „signed“ sind.

2.4 Arrays

Wenn mehrere Objekte eines gleichen Typs gefragt sind (beispielsweise für Gegner in einem Spiel), ist es sinnvoll, diese mit Arrays zu organisieren, besonders wenn der Speicher so limitiert ist wie auf dem Game Boy.

Hier ein Beispiel, um die Verwendung von Arrays zu demonstrieren:

Listing 4: Verwendung von Arrays

```
1 int xyz[8];
2 
3 void test()
4 {
5     int i;
6     i = 2;
7     xyz[i] = 564;
8 }
```

Die Syntax der Sprache orientiert sich an C, da dies im Vergleich mit dem erzeugten Code logischer ist als die Java-Syntax. Es gibt kein „= new int[8]“ wie bei Java, da hier kein neues Objekt erzeugt wird. Stattdessen wird der Speicher bereits beim Compilevorgang reserviert.

GB-J hat keine eingebaute Fehlerbehandlung wie Java, so dass bei falschen Array-Indizes einfach von einer anderen Stelle im Speicher gelesen wird, anstatt eine Exception zu werfen. So ist der generierte Code deutlich schneller, aber der Programmierer muss diese Fälle selbst absichern.

2.5 Kontrollstrukturen

GB-J verfügt über drei Kontrollstrukturen:

Listing 5: Beispiel einer if-Abfrage

```
if(x > 10)
{
    //Do stuff
}
else if(x >= 20)
{
    //Do something else
}
else
{
    //Do something different
}
```

Listing 6: Beispiel einer while-Schleife

```
char x;
x = 0;
while(x < 5)
{
    //Do stuff
    x = x + 1;
}
```

Bei diesen beiden Kontrollstrukturen ist zu beachten, dass ein Vergleich immer nötig ist, d.h. „if(x)“ ist nicht möglich (C würde hier einen Integer-Wert von 0 als „false“ und „nicht 0“ als „true“ interpretieren). Zudem unterstützen (aus zeitökonomischen Gründen) weder „if“ noch „while“ die Operatoren „&&“ sowie „||“. Diese müssen durch Verwendung mehrerer Abfragen ersetzt werden.

GB-J unterstützt keine For-Schleifen, diese können allerdings recht einfach durch while-Schleifen ersetzt werden (die obige Schleife entspricht einer einfachen Zählschleife).

Listing 7: Beispiel eines switch-Statements

```
switch(i):
{
    case 0:
    {
        //Do stuff
    }
    case 1:
    {
        //Do something else
    }
}
```

Bei switch-Statements fällt auf, dass kein „break“ innerhalb eines Cases benötigt wird, GB-J springt nach Abarbeitung eines Cases direkt aus dem gesamten switch-Statement heraus. Auch existiert kein „default“-Case.

Da alle Switch-Statements als Sprungtabelle kompiliert werden, ist es nur sinnvoll, diese bei relativ nah aufeinanderfolgenden Zahlen für die Cases zu verwenden, da ansonsten die Sprungtabelle zu groß wird (die Größe entspricht dem größten Case * 2).

Switch-Cases sind zudem auf Indizes von 8 Bit limitiert, wird eine 16-bit Zahl als Variable angegeben, so werden nur die unteren 8 Bits verarbeitet.

2.6 Objekte

Objekte in GB-J sind denen von Java sehr ähnlich:

Listing 8: Verwendung von Objekten

```
TestObj test;
test = new TestObj(4, 2);
test.method();
test.destroy();
```

Sie werden wie bekannt instanziiert; Methodenaufrufe verwenden ebenfalls die gleiche Syntax. Allerdings muss, damit der Speicher freigegeben wird, ein Destruktor aufgerufen werden. Ein Garbage-Collector nach dem Vorbild von Java würde deutlich mehr Leistung benötigen, als auf dem Game Boy verfügbar ist, weswegen dieser Weg gewählt wurde.

Listing 9: Beispiel einer Klasse

```
class TestObj
{
    char a;
    char b;

    //Konstruktor
    TestObj(char x, char y)
    {
        a = x;
        b = y;
    }

    void method()
    {
        a = b - a;
    }
}
```

Der Destruktor wird innerhalb der Klasse nicht definiert; er ist Teil der Sprache und kann nicht überschrieben werden. Falls vor der Zerstörung des Objekts noch „aufgeräumt“ werden soll, muss dafür eine separate Methode erstellt werden.

2.7 Einbinden von Assemblercode

GB-J erlaubt es, Assemblercode nach RGBDS-Syntax direkt in den Code einzufügen.

Listing 10: Beispiel von eingebettetem Assemblercode

```
Asm
{
    "ld a, 12"
    "add b"
}
```

Dies ist bei besonders geschwindigkeitsrelevanten Passagen hilfreich, die dann von Hand optimiert werden können. Es ist außerdem möglich, direkt auf Assemblercode zuzugreifen, indem man mittels „Asm.labelname“ die Referenz holt. Dies wird beispielsweise beim Laden von Tiles und Paletten benötigt, da diese nicht in GB-J-Code definiert werden können und die meisten gängigen Exporter sowieso Assemblercode exportieren.

Listing 11: Beispiel einer Assembler-Referenz

```
loadPalettes(0, Asm.myCoolPalettes);
```

2.8 Standardbibliothek

Für den Compiler wurde eine umfangreiche Standardbibliothek in Assembler entwickelt. Enthalten sind Methoden zur Ansteuerung der Grafikhardware, des Druckers und zur Verarbeitung von Eingaben, sowie einfache Methoden, um testweise Text anzuzeigen. Für Audio sind keine Methoden enthalten, da mehrere Musikplayer und Sound-Engines existieren und der Nutzer so frei wählen kann, welche er verwenden möchte (da sich alle im Speicherplatz- und Rechenbedarf unterscheiden). Im Detail sehen die Methoden der Standardbibliothek wie folgt aus:

2.8.1 Sprites.asm

Ansteuerung der Sprites.
Speicherort: stdlib/Graphics/Sprites.asm

Name	Argumente	Rückgabe	Beschreibung
setSpritePosition	Sprite-Index, X-Position, Y-Position	/	Setzt die Position des angegebenen Sprites. Zu beachten: 0, 0 ist außerhalb des Bildschirms. Um ein Sprite an der oberen linken Ecke zu platzieren, muss die Position 8, 16 verwendet werden.
setSpriteTile	Sprite-Index, Tile-Index	/	Setzt das Tile eines Sprites.
setSpriteAttributes	Sprite-Index, Sprite-Attribute	/	Setzt die Attribute eines Sprites. Attribute haben folgendes Format (Tabelle aus dem GBDev-Wiki): • Bit 7: Sprite ↔ Hintergrund Priorität (0 = Sprite über dem Hintergrund) • Bit 6: Sprite an der Y-Achse spiegeln (0 = Normal) • Bit 5: Sprite an der X-Achse spiegeln (0 = Normal) • Bit 4: Paletten-Index DMG-Modus • Bit 3: VRAM-Bank des Tiles • Bit 2-0: Paletten-Index CGB-Modus

2.8.2 Tiles.asm

Ansteuerung des Background und Window, zudem Laden von Tiles in den VRAM.
Speicherort: stdlib/Graphics/Tiles.asm

Name	Argumente	Rückgabe	Beschreibung
loadTiles	VRAM-Bank, Startindex, Anzahl der Tiles, Referenz zu Tiledaten	/	Lädt Tiles vom angegebenem Speicherort in den VRAM. Der Startindex gibt an, wo im VRAM die Tiles platziert werden.
setBGTile	VRAM-Bank, Tile-Index, X-Position, Y-Position	/	Setzt ein einzelnes Tile auf der Hintergrund-Map an der angegebenen Position. Wenn VRAM-Bank 1 ausgewählt ist, werden stattdessen Attribute für das Tile gesetzt.
setWinTile	VRAM-Bank, Tile-Index, X-Position, Y-Position	/	Setzt ein einzelnes Tile auf der Window-Map an der angegebenen Position. Wenn VRAM-Bank 1 ausgewählt ist, werden stattdessen Attribute für das Tile gesetzt.
loadBGMap	VRAM-Bank, Referenz zu Map-Daten, X-Position, Y-Position, X-Größe, Y-Größe	/	Lädt eine Hintergrund-Map, startet an der angegebenen Position und verwendet die angegebene Größe. Wenn VRAM-Bank 1 ausgewählt ist, werden stattdessen Attribute gesetzt.
loadWinMap	VRAM-Bank, Referenz zu Map-Daten, X-Position, Y-Position, X-Größe, Y-Größe	/	Lädt eine Window-Map, startet an der angegebenen Position und verwendet die angegebene Größe. Wenn VRAM-Bank 1 ausgewählt ist, werden stattdessen Attribute gesetzt.
setBGScroll	X-Scroll, Y-Scroll	/	Setzt die Scrollregister auf die angegebenen Werte.
setWinPosition	X-Position, Y-Position	/	Setzt das Window an die angegebene Position.

2.8.3 LCD.asm

Ansteuerung des LCD, zudem Laden von GBC-Paletten.
Speicherort: stdlib/Graphics/LCD.asm

Name	Argumente	Rückgabe	Beschreibung
lcdOff	/	/	Schaltet den Bildschirm ab. Danach ist er weiß, und VRAM-Zugriff ist immer möglich.
lcdOn	LCD An/Aus, Window-Map Adresse, Window An/Aus, Background Tiledaten-Adresse, Background-Map Adresse, Spritegröße, Sprites An/Aus, Background An/Aus	/	Schaltet das LCD mit den gegebenen Argumenten an
loadPalettes	Sprite-Palette?, Referenz zur Palette	/	Lädt Paletten in den Grafikspeicher. Nur auf dem GBC verfügbar.

2.8.4 Utils.asm

Abfrage der Eingabehardware.
Speicherort: stdlib/Utils.asm

Name	Argumente	Rückgabe	Beschreibung
readPad	/	/	Liest das D-Pad sowie alle Knöpfe aus und speichert den derzeitigen Status.
isButtonPressed	Index	1 = Knopf gedrückt	Überprüft, ob ein Knopf gedrückt ist. Der Index ist eine Bitmask (s. u.)
wasButtonPressed	Index	1 = Knopf war gedrückt	Überprüft, ob ein Knopf in der letzten Frame (vor der jetzigen) gedrückt war. Der Index ist eine Bitmask (s. u.)
switchSpeed	/	/	Schaltet zwischen Single-Speed- und Double-Speed-Modus um. Nur auf dem GBC verfügbar.

Bitmasks für Eingaben werden wie folgt erzeugt:

- Bit 7: D-Pad Unten
- Bit 6: D-Pad Oben
- Bit 5: D-Pad Links
- Bit 4: D-Pad Rechts
- Bit 3: Start
- Bit 2: Select
- Bit 1: B
- Bit 0: A

Wenn überprüft werden soll, ob mehrere Knöpfe gedrückt sind, ist es möglich, mehrere Bits auf 1 zu setzen.

2.8.5 GBPrinter.asm

Ansteuerung des Game Boy Printer.
Speicherort: stdlib/GBPrinter.asm

Name	Argumente	Rückgabe	Beschreibung
initPrinter	/	1 = Verbindung OK	Sendet ein Initialisierungspaket an den Drucker und wartet auf eine Anwort.
transferData	/	/	Sendet über mehrere Frames den Inhalt des Puffers an den Drucker; der Puffer ist hierbei der obere linke 20×18 Tiles große Teil des Window.
getStatus	/	Druckerstatus	Fragt beim Drucker den Status an und gibt die Antwort zurück.
startPrint	/	/	Versucht, das Drucken zu starten. Funktioniert nur, wenn vorher genügend Daten transferiert wurden.

2.8.6 ScreenPrinter.asm

Ausgabe von Text auf dem Bildschirm, sollte nur zu Testzwecken verwendet werden, da der VRAM mit eigenen Tiles überschrieben wird und die Methoden relativ langsam sind. Speicherort: stdlib/Graphics/ScreenPrinter.asm

Name	Argumente	Rückgabe	Beschreibung
loadCharset	/	/	Lädt den Zeichensatz in VRAM.
print	X-Positon, Y-Position, Referenz zum Text	/	Schreibt Text an der angegebenen Stelle auf den Bildschirm.
printNumber	X-Position, Y-Position, 8-bit-Zahl	/	Schreibt eine Zahl an der angegebenen Stelle auf den Bildschirm.

Zudem gibt es noch einige Methoden, die immer verfügbar sind:

Name	Argumente	Rückgabe	Beschreibung
halt	/	/	Versetzt die CPU in einen Low-Power-Modus, bis der nächste Interrupt eingeht.
waitVBlank	/	/	Wartet auf die VBlank-Phase (hier kann der Grafikspeicher verändert werden).
enableInterrupts	/	/	Schaltet Interrupts ein.
disableInterrupts	/	/	Schaltet Interrupts aus.
softwareBreak	/	/	Fügt ein „ld b, b" in den Code ein, welches von BGB als Breakpoint verwendet werden kann.
switchBank	Bank-Nummer	/	Schaltet Bank X auf die angegebene Bank um. Möglich sind Bänke 0-255 für MBC5, Bänke 1-127 für MBC3 und Bänke 1-127 exklusive Bank 32, 64 und 96 für MBC1.

Intern werden für Objekte noch die Methoden „malloc" und „free" verwendet: Eigentlich entsprechen Objekte eher den „Structs" aus C als Java-Objekten: Mittels „malloc" wird ein Block Speicher aus den 2 KB Heap reserviert; die Größe dafür entspricht 6 Bytes plus allen Inhalten des Objekts. Die 6 Bytes enthalten Header-Informationen: die Größe des Datenblocks sowie einen Zeiger auf den vorherigen und einen auf den nächsten Headerblock. Statisch gespeichert (im RAM) wird der Zeiger auf das Objekt.

Objektmethoden werden statisch im ROM gespeichert und nur einmal pro Klasse; sie erhalten den Zeiger auf den zugewiesenen Speicherblock des zu bearbeitenden Objekts.

2.9 Benötigte Methoden

Jedes GB-J-Programm muss folgende Methoden enthalten:

```
//Hauptmethode jedes Programms
void main() {}
//Wird beim VBlank-Interrupt aufgerufen
void VBlank() {}
```

Die „main"-Methode wird beim Start aufgerufen. Sie sollte niemals verlassen werden, sonst wird aus uninitialisiertem Speicher gelesen und ausgeführt, was fast immer zu einem Softlock („Aufhängen") des Geräts führt. In der „VBlank"-Methode sollte alle Grafikarbeit (bei eingeschaltetem Bildschirm; wenn dieser aus ist, ist es auch möglich, Grafikarbeit außerhalb der VBlank-Phase zu machen) erledigt werden.

Die Methode ist ein Interrupt-Handler und wird 59.73 (der Einfachheit halber werden fast immer 60 angenommen) mal pro Sekunde aufgerufen (der Game Boy läuft mit 59.73 Bildern pro Sekunde).

Zu „Grafikarbeit" werden der DMA-Transfer (wird für Sprites benötigt), das Setzen von Hintergrundtiles sowie das Laden von Paletten und Tiles gezählt.

3 Lexer

Der Lexer hat bei einem Compiler die Aufgabe, den zu übersetzenden Code in Textform in eine Liste von Tokens zu konvertieren. Jedes Token entspricht hierbei einem kleinen Teil des Codes, der an sich eine Einheit ist. Hierzu zählen beispielsweise Zahlen, Keywords, Namen (z.B. für Variablen) und Klammern.

Tokens enthalten sowohl einen Typ als auch einen String, der bei Bedarf weitere Informationen enhält. Ein Semikolon hat keine weiteren Informationen, aber ein Identifier (hierunter fallen Keywords und Namen) muss erhalten bleiben. Zusätzlich enthält jedes Token die Zeilennummer des Quelltextes, aus der es stammt.

Die Funktionsweise des Lexers lässt sich am einfachsten an einem Beispiel demonstrieren:

```
section test
{
    int x;
}
```

wird in folgende Tokens umgewandelt:

```
Token{type=IDENTIFIER, string=section}
Token{type=IDENTIFIER, string=test}
Token{type=LBRACE, string=}
Token{type=IDENTIFIER, string=int}
Token{type=IDENTIFIER, string=x}
```

```
Token{type=SEMICOLON, string=}
Token{type=RBRACE, string=}
```

Diese werden vom Parser später weiterverarbeitet.

Der Lexer enthält einige Funktionen, die theoretisch zum Parser gehören, aber hier deutlich einfacher und sinnvoller umzusetzen sind. Dazu gehört das Parsing von Zahlen, es können sowohl Dezimalzahlen als auch Hexadezimalzahlen (z.B. „$81“) eingelesen und direkt zu einem Token verarbeitet werden, anstatt für jede Ziffer ein separates Token zu erstellen.

4 Preliminary Parser

Da GB-J es ähnlich wie Java (im Gegensatz zu C) erlaubt, Methoden unterhalb eines Aufrufs derselben zu definieren, erzeugt der Preliminary Parser eine Liste der Methoden, mit der später verglichen werden kann, ob eine Methode existiert.

Zudem werden noch Listen von Objekttypen, -methoden, -konstruktoren, -variablen und -instanzen angelegt, die später ebenfalls für Vergleiche verwendet werden.

So kann der Parser ermitteln, ob ein Objekttyp existiert (und ansonsten einen Syntaxfehler ausgeben) oder wie viele Argumente ein Konstruktor nimmt und ob diese im Aufruf gegeben sind.

5 Parser

Der Parser wandelt die Tokenliste in einen Syntaxbaum um. Das gesamte Programm lässt sich in einem solchen Baum abbilden, aus dem später der Code generiert wird.

Hier der Syntaxbaum, der aus den Tokens des vorherigen Beispiels erzeugt wird:

```
└── PROGRAM|
        └── SECTION|test
            └── DECLARATION|int|x
```

Die Generierung des Syntaxbaumes läuft rekursiv ab: Der „PROGRAM“-Baum dient nur als Wurzel des Syntaxbaumes; er wird immer generiert, um alle Klassen und Sektionen anzuhängen.

Bei den Klassen findet der erste richtige Parsingschritt statt: Es wird überprüft, ob die Tokens vom Typ Identifier „class“/„section“ und der Name der Klasse/Sektion sowie ein Token vom Typ „öffnende geschweifte Klammer“ (LBRACE) existieren. Sofern dies der Fall ist, wird die entsprechende Methode zur Erzeugung des Baumes einer Klasse oder Sektion aufgerufen.

Importe werden ebenfalls generiert; sie werden auf der gleichen Ebene wie Klassen/-Sektionen an den „PROGRAM“-Baum angehängt (hier wird nach den Tokens „import“ vom Typ Identifier sowie einem String und einem Semikolon gesucht).

Innerhalb einer Sektion/Klasse werden, entsprechend der Tokens, Methoden für Funktionen, Variablendeklarationen und Arraydeklarationen aufgerufen. Während Deklarationen relativ simpel sind, gibt es in der Methode zur Generierung des Baumes einer Funktion wiederum Aufrufe weiterer Methoden: Zum einen wird ein Unterbaum für Funktionsargumente aus dem Methodenkopf generiert; zum anderen wird der Inhalt der Funktion als „Statement Sequence“ bearbeitet (eine Zusammenfassung, die doppelten Code für if-Statements, Schleifen usw. verhindert).

In der „Statement Sequence“ können nun Deklarationen (auch innerhalb von Funktionen können Variablen deklariert werden) erzeugt werden sowie Rückgaben, Zuweisungen (beispielsweise „i = 2“), Funktionsaufrufe, Schleifen und If-Abfragen.

Bei den letzteren beiden gibt es noch eine Methode für Vergleiche.

Prinzipiell enthält die Parserklasse einen einzelnen Index für die Tokenliste, mit dessen Hilfe alle genannten Methoden arbeiten. So wird sichergestellt, dass keine Tokens doppelt gelesen werden. Einzelne Methoden haben eigene Indizes, die auf Basis des Hauptindex initialisiert werden (beispielsweise für Vergleiche, wo Indizes für den Teil vor dem eigentlichen Vergleichszeichen und danach benötigt werden).

Bei Syntaxfehlern, die im Parser ermittelt werden, kann eine Zeilennummer ausgegeben werden, da die Tokens diese enthalten. Der Parser fängt folgende Fehler ab: Klammerfehler (Fehlen von normalen und geschweifte Klammern), fehlende Semikola sowie falsche Datentypen (z.B. „long“ existiert in GB-J nicht) und unerlaubte Deklarationen (Arrays dürfen nicht in Funktionen deklariert werden).

Mit dem Syntaxbaum können außerdem Fehler im Compiler einfacher erkannt und behoben werden, da exakt die Methode, in der der Fehler auftritt, ermittelt werden kann.

Der Parser lässt sich als Automat beschreiben (Automaten wurden auch im Unterricht behandelt). Aufgrund der Tatsache, dass GB-J die Anzahl an Klammerebenen nicht limitiert, ist die Sprache kontextfrei, und somit muss der Parser einem Kellerautomaten entsprechen:

Da bei jeder weiteren Klammerebene ein rekursiver Aufruf stattfindet (teilweise aufgrund besserer Organisation sogar innerhalb der gleichen Ebene), entsteht eine Art „Stapel“ wie im Kellerautomaten auch:

```
.     parse
..    parseStaticClass
...   parsePackageDeclaration
..
...   parseFunction
....  parseStatementSequence
..... parseReturn
....
...
..
.
```

In diesem Beispiel ist jeder Punkt ein Methodenaufruf, dargestellt als Element des Kellerstapels.

Jede Methode des Parsers entspricht einem Zustand des Kellerautomaten. Einen Endzustand gibt es allerdings nicht, stattdessen wird am Ende der Tokenliste auch das Parsing beendet.

6 Code-Generator

Der Codegenerator übernimmt den Syntaxbaum vom Parser und erzeugt daraus Assemblercode nach RGBDS-Syntax, welcher am Ende in eine Textdatei gespeichert wird.

Der Syntaxbaum wird rekursiv bearbeitet, d.h. für jeden Kindknoten wird eine weitere Methode aufgerufen. So wird das gesamte Programm generiert.

Beim Verarbeiten des vorherigen Beispiels wird zuerst die Methode für Sektionen aufgerufen; es wird eine „SECTION“ für den Code generiert (eine „SECTION“ bestimmt, wo Code im ROM liegt). Auch für die Variablendeklaration wird eine Methode aufgerufen; diese generiert allerdings keinen Code in die vorher erzeugte Sektion, sondern fügt die Variable in die Sektion für den Arbeitsspeicher ein (RGBDS hat die Möglichkeit, Speicher zum Compilezeitpunkt zu reservieren und alle Labels dorthin später aufzulösen).

Für jeden Teil eines Syntaxbaums gibt es also im Codegenerator eine Methode, die letztlich den Code erzeugt. So kann aus den „kleinsten Teilen“ eines Programms schließlich das Ganze erzeugt werden.

Eine sehr effiziente Registerallokation besitzt der Codegenerator nicht, da deren Entwicklung deutlich über den Rahmen dieses Projektes hinausgegangen wäre. Allerdings werden alle 7 Register verwendet (und somit ist zumindest eine gewisse Effizienz gewährleistet im Gegensatz zur Verwendung des Game Boy z.B. als Stackmaschine):

- a: Generelles Register für alle arithmetisch-logischen Operationen und die Verwendung des Arbeitsspeichers
- b, c: Register, in denen der erste Wert einer Operation gespeichert wird (nur c bei einem 8-bit-Wert)
- d, e: Register, in denen der zweite Wert einer Operation gespeichert wird (nur e bei einem 8-bit-Wert)
- h, l: Werden immer zusammen als 16-bit-Register verwendet, als Zeiger. Damit werden Daten vom Stack geholt oder darauf abgelegt. Wird auch bei Arrays verwendet.

Funktionsargumente werden in generiertem Code immer mittels des Stacks übergeben; auch innerhalb von Funktionen definierte Variablen werden auf dem Stack abgelegt. Somit ist Rekursion möglich, auch wenn sie aufgrund der niedrigen Geschwindigkeit nicht empfohlen ist.

6.1 StdlibCallGenerator

Da die Standardbibliothek abseits von selbstgeschriebenem Assemblercode die einzige Möglichkeit ist, mit der Hardware zu interagieren, ist es wichtig, dass diese möglichst schnell ist.

Deswegen wird für die Standardbibliothek die normale Calling Convention (Argumente mittels des Stacks übergeben) missachtet, und alle Methoden werden „manuell“ von dieser Klasse behandelt.

Die normale Calling Convention übergibt Argumente mittels des Stacks; hier werden sie in Registern übergeben. Der Unterschied lässt sich recht einfach demonstrieren:

Normale Calling Convention	Cycles	Argumente in Registern	Cycles
ld hl, 10	3	ld a, 10	2
push hl	4		
ld hl, 56	3	ld b, 56	2
push hl	4		
call setWinPosition	6	call setWinPosition	6
add sp, 4	4		
	= 24		= 10

Die optimierte Variante ist mehr als doppelt so schnell, und zusätzlich gibt es auch Ersparnisse in der Methode selbst:

Normale Calling Convention	Cycles	Argumente in Registern	Cycles
setWinPosition:		setWinPosition:	
ld hl, sp + 4	3		
ld a, [hl]	2		
ld [rWX], a	4	ld [rWX], a	4
ld hl, sp + 2	3		
ld a, [hl]	2	ld a, b	1
ld [rWY], a	4	ld [rWY], a	4
ret	4	ret	4
	= 22		= 13

So können bei häufig verwendeten Methoden, die z.B. während der kurzen VBlank-Phase eingesetzt werden, massiv Cycles gespart werden. Dadurch können die Methoden öfter aufgerufen werden, ohne zu lange zu brauchen.

Im Endeffekt führt das dazu, dass beispielsweise in einem Programm mehr Gegner dargestellt werden können.

7 Optimizer

Der Optimizer ist relativ simpel gehalten; er fällt in die Kategorie eines „Peephole Optimizer“. Er erhält den vom Code-Generator erstellten Code und ersetzt dann einzelne Teile, die ineffizient sind. Hierunter fallen sowohl einfache Änderungen wie „ld a, 0“ → „xor a“, die auf diese Weise CPU-Cycles sparen, als auch kompliziertere: beispielsweise das Optimieren von Vergleichen, wo auf diese Weise die Verwendung eines ganzen Registers und mehrere Cycles eingespart werden, was gerade bei Schleifen (sie enhalten als Bedingung immer einen Vergleich) sinnvoll ist.

Der Optimizer gibt am Ende noch die Anzahl der gesparten Cycles und Bytes (einige Änderungen benötigen auch weniger Speicher als die ineffizientere Variante) aus.

8 Ausführung von RGBDS

Da der Code-Generator Assemblercode in Textform erzeugt, muss dieser noch assembliert werden, um ausführbar zu sein. Dafür wird der Assembler RGBDS (kurz für „Rednex Game Boy Development System“) verwendet. Es ist der am weitesten verbreitete Assembler für den Game Boy.

Der GB-J Compiler speichert nach Erzeugung des Codes diesen als Textdatei und führt zuerst den Assembler (übersetzt den Assemblercode im Textformat in die tatsächlichen Opcodes), dann den Linker (berechnet Adressen und fügt alle Dateien zu einem ROM zusammen) und als letztes noch „rgbfix“ aus.

„rgbfix“ berechnet die Prüfsumme und setzt einige Flags sowie das Nintendo-Logo im ROM-Header. Ohne würde der Game Boy (und auch genaue Emulatoren) das ROM nicht starten. Nintendo setzte das Logo als Schutz gegen unlizensierte Spiele ein, da man Entwickler bei Verwendung des Logos ohne Erlaubnis verklagen konnte. Findige Entwickler fanden allerdings bereits in der Hochphase des Game Boy Möglichkeiten, diesen Schutz zu umgehen (das bekannteste Beispiel dürfte Argonaut Games sein).

9 Verwendung des Compilers

Der Compiler wird mittels des Befehls

```
java -jar GB-J-Compiler.jar
```

ausgeführt. In diesem Modus (ohne Argumente) wird nur auf die Hilfe verwiesen.

Mittels des Arguments „-help“ werden alle unterstützten Argumente sowie deren Bedeutung angegeben.

10 Der Game Boy

10.1 Hardware

Der Game Boy verwendet eine Custom-CPU (vermutlich eine Variante des Sharp SM83), die beim Originalmodell mit 4MHz und beim GBC mit 4MHz/8MHz getaktet ist. Die CPU lässt sich als Hybrid zwischen dem Intel 8080 und dem Zilog Z80 beschreiben.

Die Bildschirmauflösung beträgt 160×144 Pixel, aufteilbar auf 20×18 „Tiles". Die gesamte Grafikhardware ist tilebasiert, d.h. alles ist aus 8×8 Pixel großen Blöcken zusammengesetzt. Es gibt zwei Layer: Den Hintergrund, der 32×32 Tiles groß ist und sich scrollen lässt, sowie das ebenfalls 32×32 Tiles große Window, welches über dem Hintergrund liegt und sich nur positionieren lässt. Zusätzlich dazu sind 40 Sprites verfügbar, die man pixelgenau an jeder Stelle des Bildschirms platzieren kann, wobei nur 10 Sprites auf der gleichen Zeile des Bildschirms erlaubt sind.

Als Eingabehardware enthält der Game Boy 8 Knöpfe; 4 davon sind als Steuerkreuz angeordnet, die anderen vier mit A, B, Start und Select bezeichnet.

Abbildung 1: Die Platine eines Game Boy Color.

Da Spiele auch Ton benötigen, existieren ein Lautsprecher und eine APU (Audio Processing Unit), die vom Prozessor direkt angesteuert werden kann. Es gibt 4 Kanäle: Zwei Square-Wave-Kanäle, einen White-Noise-Kanal sowie einen Wave-Kanal, der theoretisch sogar das Abspielen von Audiodateien erlaubt, allerdings nur mit einer Samplerate von 4 bit.

Als Speicher werden austauschbare Cartridges verwendet, auf denen sich die Programme befinden. Adressierbar sind 32KB. Bei Spielen, die größer als 32KB sind, werden sogenannte „Memory Bank Controller", kurz: MBCs verwendet. Hier wird nun der Speicher wie folgt aufgeteilt: Die ersten 16KB sind immer Bank 0, die anderen 16KB lassen sich auf jede andere Bank „umswitchen".

Somit ist es möglich, mittels des MBC5 (das als das Beste angesehene MBC) bis zu 8MB ROM zu verwenden.

Cartridges können zudem auch weiteren Arbeitsspeicher enthalten (8KB sind addressierbar), der mittels Banking auf 32KB aufgestockt werden kann. Dieser Speicher kann entweder mittels einer Batterie am Leben erhalten werden und für Speicherstände dienen, oder einfach als Erweiterungsspeicher für die Konsole.

10.2 Besonderheiten beim Programmieren

10.2.1 Grafik

Das LCD hat drei Modi, und auf VRAM, OAM und die Paletten kann nicht immer zugegriffen werden. Zur Verdeutlichung hier ein Diagramm:

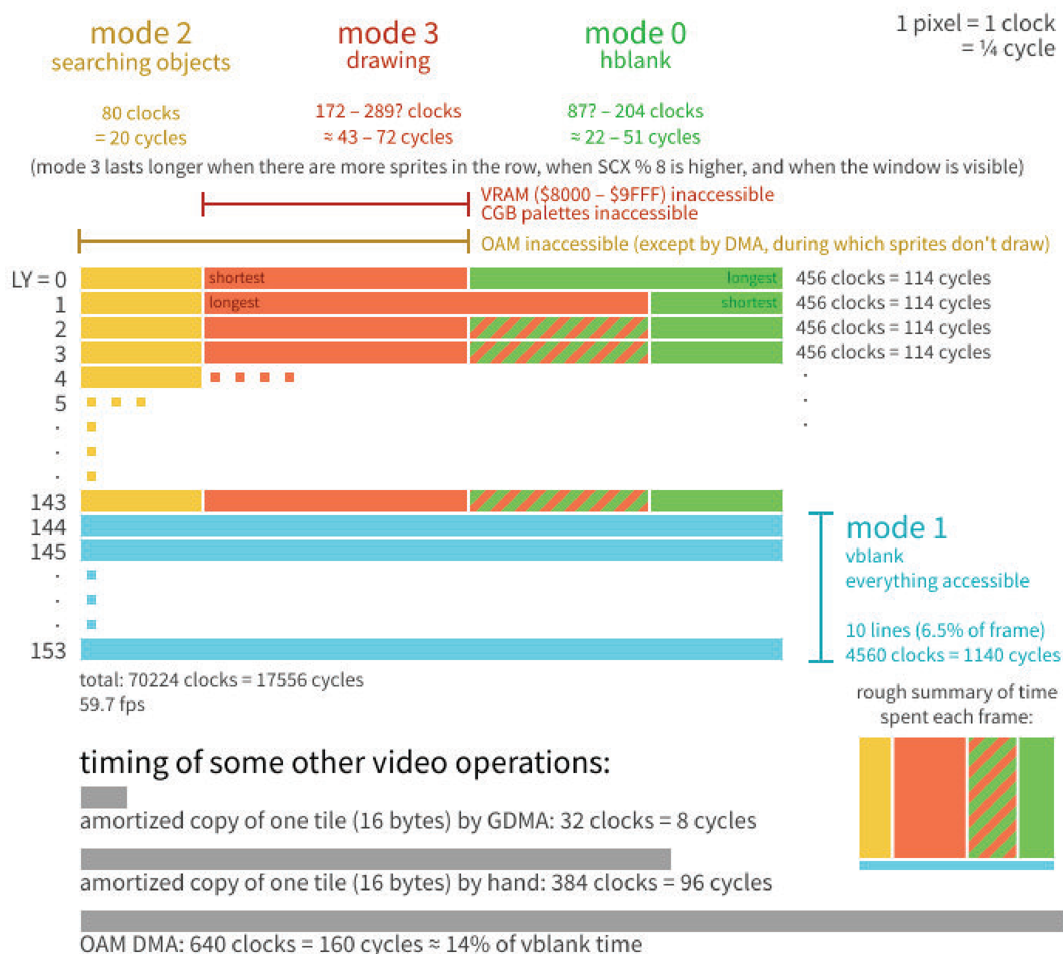


Abbildung 2: Diagramm von eevee. Dargestellt sind alle LCD-Modi inklusive Zeiten und Zugriffsmöglichkeiten auf VRAM, OAM und Paletten.

Man muss also darauf achten, Arbeiten an der Grafikhardware (VRAM, OAM, Paletten) nur in der richtigen Phase durchzuführen; dazu enthält GB-J auch den VBlank-Interrupthandler (da VBlank die meistgenutzte Phase ist, HBlank ist zu kurz, und für Mode 2 gibt es keinen Interrupt). Sprites können allerdings mittels eines Tricks immer modifiziert werden (diesen setzt GB-J auch ein): Anstatt direkt in den OAM („Object Attribute Memory“, hier sind alle Attribute von Sprites gespeichert) zu schreiben, wird eine Kopie im Arbeitsspeicher verwendet (wo logischerweise immer geschrieben und gelesen werden kann).

Während der VBlank-Phase sollte dann die Methode

```
dmaTransfer();
```

aufgerufen werden, um mittels eines „Direct Memory Access“ (kurz DMA) die OAM-Kopie im Arbeitsspeicher in den tatsächlichen OAM zu übertragen. Um diese zu verwenden, muss noch am Anfang der Main-Methode

```
copyDMARoutine();
clearOAM();
```

aufgerufen werden. Die erste Methode wird benötigt, um den DMA-Transfer überhaupt durchführen zu können. Die zweite sorgt dafür, dass im OAM alle Werte auf null stehen, da die Hardware nach dem Einschalten zufällige Werte im Speicher enthält.

10.2.2 Speicher

Aufgrund des begrenzten Adressbereiches des Game Boy (16 bit, daher können 65535 Bytes adressiert werden) greifen Spiele oft auf eine Technik namens „Banking“ zurück.

Dabei werden die 32 KiB, die zum Zugriff auf das ROM der Cartridge reserviert sind, in zwei 16 KiB große Speicherbänke geteilt:

Die ersten 16 KiB adressieren immer den gleichen Bereich des Speicherchips; hier werden die am häufigsten benötigten Routinen abgelegt. Die anderen 16 KiB können auf einen anderen Teil des Speicherchips umgeschaltet werden. So können je nach MBC bis zu 4 MiB verwendet werden. Das MBC ist der „Memory Bank Controller“, ein weiterer Chip auf der Cartidge. Er dient zur Umschaltung der 16 KiB auf einen Teil des eigentlichen ROM-Chips.

Theoretisch ist auch ein beliebig großes ROM denkbar, wenn ein entsprechendes MBC entwickelt wird (in der gbdev-Community existiert ein MBC mit Kapazitäten für bis zu 1 GiB).

Abbildung 3: Eine Cartridge mit dem MBC5, hier Pokémon Gelb.

Wichtig für das Konzept der Speicherbänke ist noch, dass jeglicher Code, der einen Wechsel verursacht, in Bank 0 (der nicht wechselbaren) liegen muss. GB-J stellt für die Verwendung der Speicherbänke dem Programmierer die package-Deklaration (bestimmt, in welcher Bank eine Sektion oder Klasse platziert wird) sowie die Methode

```
switchBank(x);
```

zur Verfügung. Der Programmierer muss sich selbst entscheiden, welcher Code in welche Bank kommt und wann ein Bankwechsel sinnvoll ist. Es ist allerdings auch möglich, für kleinere Projekte innerhalb des Rahmens von 32 KiB zu bleiben und die Technik des „Bankswitching“ nicht zu verwenden.

10.3 GB Printer

Der Game Boy Printer wurde 1998 veröffentlicht und erlaubt es, auf 38mm breitem Thermopapier vom Game Boy aus zu drucken, mit einer horizontalen Auflösung von 160 Pixeln (dies entspricht der Bildschirmauflösung des Game Boys).

Der Game Boy kommuniziert mit dem Drucker über den Link Port, der eingebauten seriellen Schnittstelle.

Abbildung 4: Der Game Boy Printer.

Aufgrund spärlicher und teils falscher Dokumentation war die Entwicklung des Druckertreibers für die Standardbibliothek nicht einfach.

In der Theorie ist die Ansteuerung simpel: Zuerst sendet man ein Datenpaket, um den Drucker in eine Art Bereitschaftsmodus zu versetzen. Wenn dieser antwortet, dass er normal funktionstüchtig sei, kann der Game Boy anfangen, Grafikdaten im normalen Tileformat zu senden.Hier werden für einen normalen Bildschirm (20×18 Tiles) mindestens 9 Pakete benötigt.

Danach wird ein Paket mit dem Auftrag zum Drucken abgeschickt, woraufhin der Printer den Druckprozess beginnt und, sobald er fertig ist, auf Statusanfragen wieder mit „Bereit“ statt mit „Busy“ antwortet.

Tatsächlich gab es dann beim Schreiben des Druckertreibers hauptsächlich Timingprobleme, da der Drucker sowohl einen dokumentierten Timeout zwischen Datenpaketen enthält (hier wird nach kurzer Zeit ohne Erhalt neuer Daten alles erhaltene gelöscht), wie auch einen undokumentierten Timeout zwischen den einzelnen Bytes eines Paketes. Nach ausgiebigem Testen fand sich schließlich eine Lösung, wie die Bytes schnell genug an den Drucker zu senden waren (sodass der Timeout nicht anspricht): Anstatt immer ein Paket bis zur maximalen Größe zu füllen (in mehreren VBlank-Interrupts) und dann erst abzuschicken, wird nun in einigen HBlank-Interrupts ein kleines Paket abgeschickt. Die Gesamtzahl vergrößert sich, aber insgesamt läuft der Transfer sogar schneller (VBlank tritt nur einmal pro Bild auf, HBlank 144 Mal).

10.4 Der Emulator BGB

BGB ist nach SameBoy der genaueste Emulator für den Game Boy, der momentan existiert. Er wird aufgrund seiner exzellenten Debugfunktionen deutlich häufiger verwendet als SameBoy, der nur in der MacOS-Version eine Debug-GUI besitzt. Zudem ist die extrem hohe Genauigkeit von SameBoy oft nicht notwendig, um Spiele zu entwickeln.

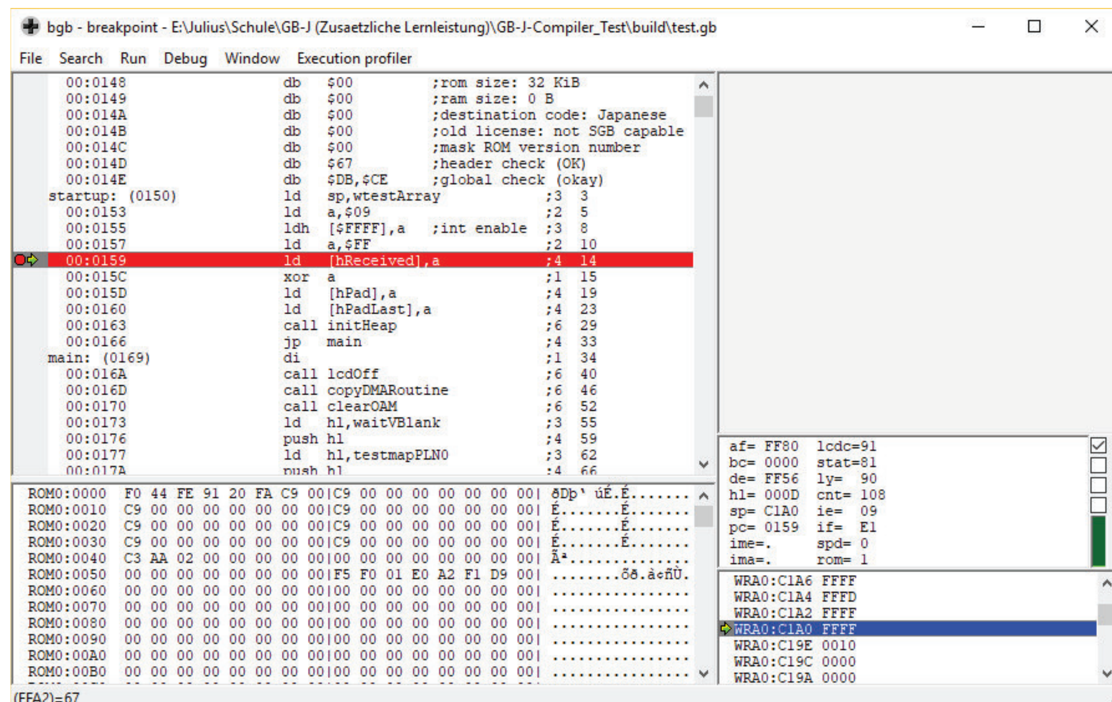
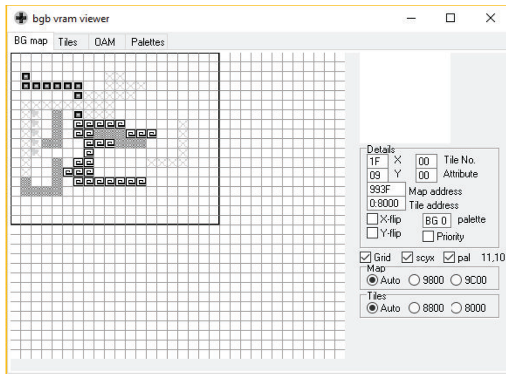
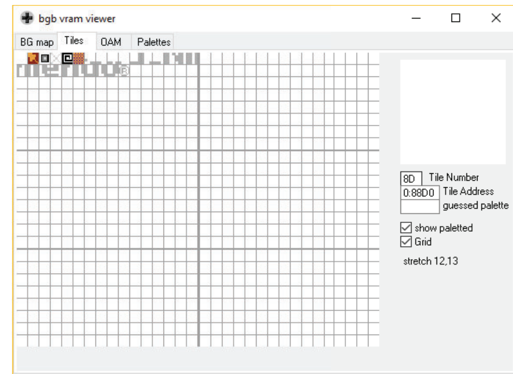
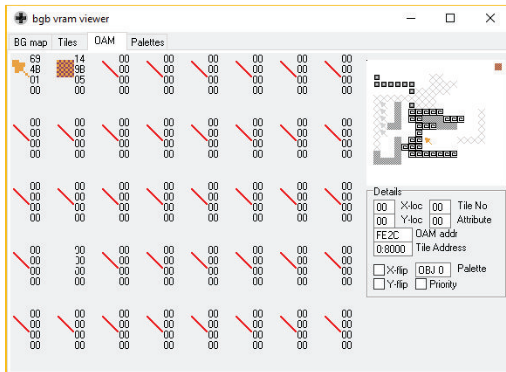


Abbildung 5: Das Hauptfenster des Emulators BGB. Zu sehen sind der Disassembler mit Step-through-Debugger (oben links), die Speicheransicht in Hexadezimal und ASCII (unten links), die Zweit-Speicheransicht mit Stack-Pointer(unten rechts), die Registeranzeige sowie CPU-Auslastung (mitte rechts) und der Monitor des emulierten Game Boy (oben rechts).







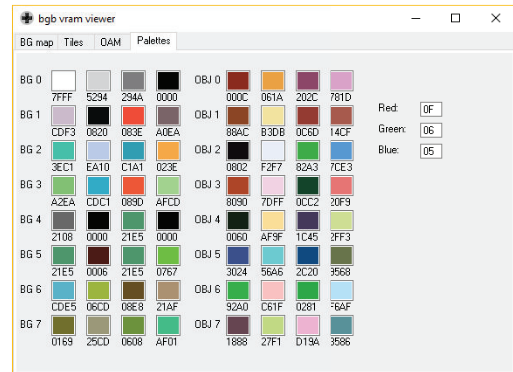


Abbildung 6: Der VRAM-Viewer zeigt alle Bestandteile des Grafikspeichers an: Die Hintergrund- und Window-Map, den Tile-Speicher, den OAM (Object Attribute Memory) sowie die Paletten

Dank der mächtigen Debuggingwerkzeuge kann der Lauf eines Programms vollständig nachvollzogen werden. Da BGB auch einer der genauesten Emulatoren ist, kann davon ausgegangen werden, dass ein Programm, welches unter BGB läuft, auch auf der Hardware funktioniert. Dazu habe ich ein Flashcart – eine Cartridge, auf die ROMs vom PC aus aufgespielt werden können.

Die Druckerbibliothek wurde sogar zum größten Teil mithilfe des Flashcarts entwickelt, da BGB den Drucker nicht emuliert und andere Emulatoren zu ungenau waren: *Max* und *LIJI32* aus der GBDev-Community führten meine Test-ROMs auf ihren Emulatoren aus. Auf beiden sprach der Drucker an, in Hardware tat er es nicht.

Abbildung 7: Die EMS-64M Cartridge.

10.4.1 Steuerung

Die auf der CD mitgelieferte Version von BGB ist so eingestellt, dass die Pfeiltasten als Steuerkreuz des emulierten Geräts verwendet werden, A und S entsprechen den A- und B-Tasten, M und N entsprechen Start und Select.

11 Beispielprogramm

Um die Möglichkeiten des Compilers zu demonstrieren, ist noch ein relativ kleines Beispielprogramm beigefügt: ein „Zeichenprogramm“, mit dem verschiedene Tiles auf dem Bildschirm platziert werden können und das Gezeichnete ausgedruckt werden kann.

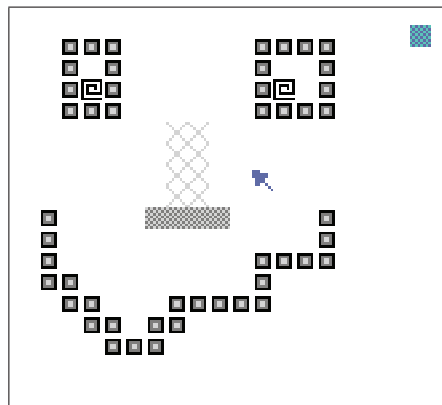


Abbildung 8: Screenshot des Zeichenprogrammes. Die Sprite-Paletten werden nicht initialisiert; deswegen sind die Farben bei jedem Start anders.

12 Fazit

Insgesamt war die Entwicklung des GB-J-Compilers ein auf vielen Ebenen interessantes Projekt: Ich lernte mehr über die Game Boy-Hardware und ihre Besonderheiten; gleichzeitig konnte ich von Grund auf einen Compiler schreiben, der nun, trotz einiger Limitationen, auch gut funktioniert.

Hier möchte ich mich auch bei der hilfreichen GBDev-Community bedanken, die mir bei Fragen über die Hardware und bei Problemen immer freundlich zur Seite gestanden hat.

Besonders danke ich *ISSOtm*, der immer meinen Assemblercode korrekturgelesen und generell viele hilfreiche Tipps gegeben hat; *DaKnig*, der interessante Denkanstöße gab; sowie *Max* und *LIJI32*, die mir bei der Druckerimplementation geholfen haben, indem sie Test-ROMs auf ihren Emulatoren laufen ließen. Außerdem *nameless*, der den Compiler getestet hat.

Abbildung 9: Ich sagte, dass Fehler nur zeilen- und nicht spaltenweise gemeldet werden.

13 Quellenverzeichnis

- CPU-Instruktionsset nach RGBDS-Syntax:
  https://rednex.github.io/rgbds/gbz80.7.html (abgerufen am 3.9.2018)
- Offizielle „Game Boy Programming Manual“:
  http://www.chrisantonellis.com/files/Game Boy/gb-programming-manual.pdf (abgerufen am 3.9.2018)
- Dokumentation der Videohardware im GBDev-Wiki:
  http://gbdev.gg8.se/wiki/articles/Video_Display (abgerufen am 5.9.2018)
- Dokumentation des Datentransfers über den Link-Port im GBDev-Wiki:
  http://gbdev.gg8.se/wiki/articles/Serial_Data_Transfer_(Link_Cable) (abgerufen am 6.10.2018)
- Dokumentation der MBCs im GBDev-Wiki:
  http://gbdev.gg8.se/wiki/articles/MBC1

http://gbdev.gg8.se/wiki/articles/MBC3
http://gbdev.gg8.se/wiki/articles/MBC5
(abgerufen am 5.10.2018)

- Game Boy Cribsheet:
  http://gbdev.gg8.se/files/docs/GBCribSheet000129.pdf (abgerufen am 10.10.2018)

- Divisionsroutinen:
  http://wikiti.brandonw.net/index.php?title=Z80_Routines:Math:Division
  https://github.com/NovaSquirrel/Game BoyFALSE/blob/master/game.z80
  (abgerufen am 6.10.2018)

- Multiplikationsroutinen:
  http://wikiti.brandonw.net/index.php?title=Z80_Routines:Math:Multiplication
  (abgerufen am 6.10.2018)

- Basics der Compilerprogrammierung:
  https://norasandler.com/2017/11/29/Write-a-Compiler.html
  http://www.cse.chalmers.se/edu/year/2015/course/DAT150/lectures/proglang-04.html
  https://www.youtube.com/watch?v=TG0qRDrUPpA
  (abgerufen am 6.10.2018)

- Titelbild:
  https://www.vecteezy.com/vector-art/76824-game-boy-color
  (abgerufen am 6.10.2018)

- Foto Game Boy Printer:
  https://upload.wikimedia.org/wikipedia/commons/thumb/7/74/Game_Boy_Printer.jpg/220px-Game_Boy_Printer.jpg
  (abgerufen am 11.10.2018)

- Foto Platine Game Boy Color:
  https://gbhwdb.gekkio.fi/static/cgb/C10342618_03_pcb_front.jpg
  (abgerufen am 11.10.2018)

- Foto Platine Pokémon Gelb:
  https://gbhwdb.gekkio.fi/static/DMG-APSE-0/gekkio-1_02_pcb_front.jpg
  (abgerufen am 10.1.2019)

- Diagramm zu LCD-Modi: eevee (Webseite: https://eev.ee)

- Foto EMS-64M Cartridge:
  https://www.dmgs-r-us.de/shop/gb-usb-smart-card-64m/
  (abgerufen am 10.1.2019)

14 Versicherung der selbstständigen Erarbeitung

Ich versichere, dass ich die vorliegende Arbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen, Darstellungen u.ä.m. selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Rheinbach, den 1.4.2019: ______________________________

Julius Clasen

15 Einverständniserklärung zur digitalen Veröffentlichung dieses Dokumentes auf der Webseite des Städt. Gymnasiums Rheinbach

Ich erkläre mich damit einverstanden, dass dieses Dokument in digitaler Form auf der Webseite des Städt. Gymnasiums Rheinbach (www.sg-rheinbach.de) veröffentlicht wird.

Rheinbach, den 1.4.2019: ______________________________

Julius Clasen