Simplifying Development on Old Hardware with Embedded Domain Specific Languages

Rose
University of Huddersfield
Huddersfield, U.K.
rose@empty.town

May 2019

Abstract

In recent years, there has been a strong renewal of interest in retro video games, whether it be the rerelease of classic home consoles in miniature form by the original manufacturers, or the creation of new video games simulating the style of those made during the late 1980s to 1990s. These games often have low-resolution textures and sprites, which are stylistic of the limitations imposed by the hardware of the era.

Because of their primitive nature, these assets are much faster to produce by independent ("indie") game developers today. However, the retro nature of these games is often only skin deep, and in some cases their runtime dependencies, physics implementations and other features are heavily inauthentic to the classic platforms from which they derive inspiration.

At present, creating native software for older, archaic consoles is something which requires an intricate knowledge of the hardware, its quirks, best practices and an existing knowledge of low-level programming and game development.

In this paper, we introduce and describe an *Embedded Domain Specific Language* (EDSL) and its compiler written in Haskell to facilitate code generation for one such console (the Nintendo Game Boy). We explore the potential for compilers to augment development practices for older hardware.

1 Introduction

Within the last few years, there has been an increase in nostalgia for classic (or "retro") video games, as (Wulf, Bowman, Rieger, Velez, & Breuer, 2018) state, players "are starting to return to their initial and past experiences with games". As a consequence of this, new games are being developed, particularly by independent ("indie") developers — due to the decreased workload of making low-resolution assets — which embody traits and stylistic elements of well-loved gaming classics (Bowles, 2017).

At present, options are limited for developers looking to create games which authentically imitate the style of those published for "retro" (1980s-1990s) consumer video games consoles. Currently, there are two routes which can be taken:

1. Utilise a modern engine and target the user's native architecture, merely reproducing the graphical style and/or other limitations of older consoles. The main drawback to this approach is that unless great care is taken, the game may not feel entirely authentic to the platforms from which it derives inspiration.

2. Develop a game which targets the native architecture of an old games console. Naturally, this adheres most accurately to the constraints of old consoles. The game can then be packaged with an emulator to provide a means of running it on modern hardware such as a personal computer or mobile device. The primary caveat here is that it requires the developer to intricately understand the hardware being targeted, as well as its limitations, quirks and idioms. Furthermore, as is the case with the platform the compiler outlined in this project covers, tooling can be outdated, unmaintained and incomplete.

These two development strategies exist with little middle ground. Ideally, a developer would have the prospect of using a toolset they are familiar with to create software which natively targets an antiquated platform. This is not currently achievable today.

This project introduces an *Embedded Domain Specific Language* (hereafter referred to as "the EDSL" or "the compiler", or by name as "Lazyboy" to disambiguate) with the purpose of allowing developers with some familiarity with the host language (Haskell) a method of writing efficient code which is compiled to the Nintendo Game Boy's machine code architecture.

The EDSL takes the form of a library and contains both high and low level function constructs for manipulating the state of the Game Boy. These can be as primitive as loading immediate values into registers, or as complex as displaying text or images onscreen.

For those not familiar with Haskell already, the benefits of a greater range of resources, active community support and better tooling make Haskell a much more approachable and practical investment to learn than Game Boy machine code itself.

In relation to this, a project in encouraging students to learn to program through the development of homebrew for the *Nintendo Game Boy Advance console* concluded that "the class was successful in motivating the students to learn about programming" (Kacmarcik & Kacmarcik, 2009), enough so that the course was continued and expanded upon following completion of the article.

2 Background Research

2.1 Justification for the Target Platform

The Nintendo Game Boy is the third best-selling video game console and the second best-selling handheld console of all time (Various, 2019) with a reported 118.69 million units sold (Nintendo Co. Ltd., 2016) between its iterations (Game Boy, Game Boy Light, Game Boy Pocket, Game Boy Color) which all feature the same core hardware (with the slight exception of the color, which has new hardware capablities whilst retaining backwards compatibility with the original model).

Due to its success and popularity, the Game Boy is widely known in popular culture as a symbol of vintage gaming. The only handheld console to outsell it is the Nintendo DS (Various, 2019), which due to having been released in 2004, does not evoke the same kind of nostalgic, vintage vibe in the public consciousness.

The Game Boy hardware is suitably limited for exploring the titular theme of this project, with its 4.194Mhz Z80-like processor, 8K of RAM and tiny native resolution of 160x144 (Fayzullin et al., n.d.), it poses an interesting technical challenge to take modern programming practices in a high-level language and provide an efficient means of representing these on an archaic instruction set from over 20 years prior.

2.2 Justification for Haskell as a Host Language

Haskell is a language which is well-suited to building EDSLs, and *Domain Specific Embedded Compilers* (Leijen & Meijer, 1999) identifies several benefits of EDSLs (Leijen & Meijer, 1999), with a focus on Haskell, such as the diminished need for the user to know two programming languages, the opportunity to present host language libraries to the user, the benefit of producing only syntactically correct machine code and leveraging host language abstraction features.

Haskell's rich ecosystem has spawned a bevy of such languages, with their application domains ranging from generating static web pages (Van der Jeugt, n.d.) to rendering and composing vector shapes with a variety of backends (Yorgey, n.d.). This project is thus not out of place among the swathes of other DSLs in the Haskell ecosystem today. Furthermore, due to the ubiquitous syntax style of these DSLs, not only do users with existing Haskell experience have a greater predisposition to the kind of constructs which will be present in the compiler, but other DSLs which are of relevance to the project can be integrated without any semblance of disjointedness.

In *Renegade Drive: Usage of today's technology in creating authentic 8-bit and 16-bit video game experiences* (Bowles, 2017), *Renegade Drive* is introduced with the assertion that "the possibility of producing a nostalgic retro gaming experience will become a reality" (Bowles, 2017, p. xv), and after examining in detail the technical specifications of various pieces of classical consumer video game hardware, it is promised the set of design specifications outlined in the

paper will allow for the development of games that "are intended to look, feel and function identically to the ones that were produced 20 to 30 years ago" (Bowles, 2017, p. 115).

However, the methods through which this is achieved are through naive limitations imposed upon a *YoYo Games Game Maker* project, such as using a fixed output resolution and strict use of a palette. In effect, the proposal for *Renegade Drive* is similar if not identical to a fantasy console such as the *Lexaloffle PICO-8* (White, 2019).

While games for such consoles, when well defined, can capture the aesthetic and other factors of retro-style games, unless they go to the effort of emulating a constant clock speed, fixed RAM and ROM sizes et al., the function and thus the experience cannot be considered truly comparable to that of retro games consoles.

Furthermore, should developers adhering to these arbitrary specifications fail either to limit the clock speed, or to take into account the practice of the application of *delta time* to regulate the speed at which motion takes place in their games, they may be left with end products that provide different experiences depending on the player's own hardware.

In contrast, developing new software using modern principles and tooling, but targeting retro hardware itself, as Lazyboy does, naturally carries over all of the features and constraints of the platform — it can truly be said that any software or game built now which runs on the original hardware would've been entirely possible to produce at the height of its popularity. Not only does this encourage authenticity, it strictly *enforces* it.

In relation again to Lazyboy's use of Haskell, students at the University of Gothenburg were tasked with developing optimizing implementations of their own languages, with parsers, which could be compiled to native Game Boy machine code (Kögel et al., 2018).

The common association of Haskell as a language suited to compiler development is highlighted in the Gothenburg project, as it can be noted that while 11 students began development in a myriad of languages (with only 1 choosing Haskell initially), by the termination of the project, 3 students in total had ended up using Haskell for their parser construction, making it the second most popular language among the students after Java (used by 6 students).

(Kögel et al., 2018) strongly advises students "against writing their parsers by hand", in relation to two students constructing parsers for their languages from scratch. We see that one student dropped out and the other switched from C++ to Haskell with the *Parsec* library of parser combinators.

Lazyboy avoids the workload of parser implementation altogether via one of the key benefits of its status as an EDSL — it does not need a parser as the syntax of user programs is valid Haskell and is compiled along with the library.

2.3 Existing Alternatives

Today, the most widely used means of developing new software for the Game Boy is the Rednex Game Boy Development System (RGBDS) with its components

RGBASM (an assembler, of particular significance to this project as it is the assembly language that the compiler emits), RGBLINK (a linker), RGBFIX (checksum and header fixer) and RGBGFX (a utility to convert PNG images to the two-bits-per-pixel graphics format of the Game Boy). RGBASM has some useful features ("RGBASM(5)", 2018), such as automatically deducing which read-only memory bank to place data in when one is not specified, and, of particular use to this compiler, aliasing string labels to memory addresses, which is a common feature of assemblers.

RGBASM is a very mature (Bentley, 2019) assembler, with its ancestor *AS-Motor* first being able to output the Game Boy blend of Z80 assembly circa 1999. As of 2019, it is still actively maintained. In contrast, there exists the Game Boy Development Kit (GBDK), another compilation of tools, which notably includes a C compiler. The project was discontinued in 2002 due to the maintainer no longer having time for it (Hope, 2001), although projects which make use of it continue to be produced today, possibly due to the lack of alternatives for development for the Game Boy using the C programming language.

Because of this lack of serious competition and the clear triumph of RGBDS as the *de facto* Game Boy assembler, the first priority for Lazyboy will be to output assembler code which RGBDS can assemble and link.

2.4 Legal, Social, Ethical and Professional Issues

The primary issues which this project presents are legal in nature, with all rights concerning the hardware design, system software, and other aspects of the Game Boy console belonging solely to Nintendo.

The compiler does not make use of or redistribute any copyrighted software owned by Nintendo, nor does it encourage software piracy or the reverse engineering of Nintendo's intellectual property. We do not claim any affiliation with Nintendo.

This project solely provides a means of generating original, *homebrew* software for the purpose of study, which can be run in a software emulation of the Game Boy hardware or on the hardware itself via the use of an external, third-party device known as a flashcart.

The texts used as reference to implement this functionality are publicly accessible and have been acquired by external individuals through clean-room implementation and observation. The intricacies of the legality of those practices are beyond the scope of this paper, it should be noted that this project is not unique in the sense of touching on the topic of the inner workings of retro games, and others such as the project *Mappy* (Osborn, Summerville, & Mateas, 2017) explore concepts tangentially related to this paper.

A professional issue raised by this project is the reference of documents of dubious and sometimes anonymous or pseudonymous origin. These documents cannot be proven to be accurate or to contain only information acquired through scrupulous means. While care has been taken to find sources which are reliable and accountable for their claims, it should be emphasized that this is an area

in which subject knowledge is relatively decentralized and requires a reasonable degree of investigative effort.

3 Core

3.1 Instructions and Types

The first step in the development of the compiler was to determine the standards which the minimum viable program that can pass validation and run on the Game Boy must conform to. This involved constructing a basic program in assembly language to assemble with RGBASM and test on an emulator. The emulator of choice for the duration of this project was *mGBA*, which features integrated debugging and introspective utilities, which proved useful in testing the outcomes of programs generated by the compiler.

The instruction set of the Game Boy's CPU (officially known as the Sharp LR35902) is relatively well documented, and bears many similarities to the Zilog Z80, but also to the Intel 8080. (Fayzullin, Felber, Robson, & Korth, 2001)

The first and most crucial step of implementing the compiler involved creating type-level representations of the CPU instruction set and associated characteristics such as registers.

This was accomplished in a manner which allows for compile-time checking of the parameters of some instructions. We introduce a `Register8` type which represents a single 8-bit register, and has corresponding named variants.

```
data Register8 = A | B | C | D | E | H | L
  deriving (Read, Show, Eq)
```

The benefit of using these over a simple `Char` parameter is that the type constructors serve as a primary means of validation - these are the only valid 8 bit registers, and attempting to provide anything else to an instruction that operates on an 8-bit register will raise a compile time error. This is one of many ways in which we assert that generated assembly is valid.

Further to this, we also define a series of 16 bit registers in the same style.

```
data Register16 = BC | DE | HL | AF | SP | PC
  deriving (Read, Show, Eq)
```

Note that we also include the Stack Pointer (SP) and Program Counter (PC) among these, which, while being special cases insofar as being invalid in some instructions (such as attempting to load a 16-bit immediate value into the program counter (Fayzullin et al., 2001)), are not felt to warrant their own type, and can instead be handled elegantly with pattern matching, as we will later see.

We define a type `Instruction` to represent a single instruction, with variants for different invocations and their constructors parameterized with types that correspond as closely as possible to the original opcodes themselves.

```
data Instruction =
    LDrr Register8 Register8
  | LDrn Register8 Word8
  | LDAnn Word16
   ...
```

Where possible, care has been taken to remove ambiguity from these names, although that is not always possible or easy. For instance, while `LDrn` loads the 8-bit literal value it is provided into the 8-bit register it is also provided, `LDAnn` loads the 8-bit value stored at the 16-bit address value it is provided into the register A. This can be slightly ambiguous at times, and in future a better naming pattern may be implemented, if thought of.

3.2 Modelling Code Generation

The core of the compiler is the utilization of the `Writer` monad, which ultimately emits a "log", taking the form of a list containing elements of the type `Instruction`.

`Writer` is implemented in terms of two type parameters. The first, *w*, refers to the type of the accumulative output log, and internally uses *mappend* to append to the final result (Gill & Paterson, 2019). The latter type parameter, *a*, is used to return a result from each computation.

For a while, the definition of the `Lazyboy` type alias for the `Writer` monad looked as follows:

```
type Lazyboy a = Writer [Instruction] a
```

This would change upon the implementation of sequential label names in the output assembly, which were observed to be a common trait among other compilers, notably GCC (the GNU Compiler Collection). An example of this behaviour was reproduced with GCC version 8.3, targeting x86-64 and with the flags `-O2 -fno-unroll-loops` using the compiler explorer. The input C program is as follows:

```
int main(int argc, const char** argv){
    while (1)
        for (int i = 0; i < 10; ++i) asm("nop");
    return 0;
}
```

This program results in the following output from GCC, It can be noted that it elects to name labels as a sequential series of integers.

```
main:
.L3:
        mov     eax, 10
.L2:
```

```
nop
sub     eax, 1
jne     .L2
jmp     .L3
```

Initially, it was planned to make Lazyboy completely stateless, which was to involve generating labels pseudo-randomly, as opposed to sequentially.

Two methods of generation for them were pursued — UUIDs (V4) and simple alphabetical strings — however, it soon became apparent that this would require a more complex *monad transformer stack* to accommodate, as introducing randomness is an effectful computation. This would've involved parameterizing the `WriterT` monad transformer with `IO` as the inner monad, at the time its more primitive form `Writer` was being used, and it was concluded that if we needed to modify the monad transformer stack, it would be worth switching to one which can encapsulate state too, as then implementing an incremental counter would be possible.

This course of action was ultimately undertaken, as another drawback of using randomized labels would've been the potential of collision (slim in the case of UUIDs, but possible).

At present, the core of the compiler is implemented using an implementation of the `RWS` monad — which incorporates the functionality of the monads `Reader`, `Writer` and `State`.

The `RWS` monad adds two additional type parameters over `Writer` to its definition, which predictably encapsulate the functionality of `Reader` and `State` respectively. We do not make use of `Reader`, so we assign it the unit type.

Our definition of the type alias for Lazyboy is now:

```
type Lazyboy a = RWS () [Instruction] Integer a
```

Note the introduction of the third parameter `Integer`, which is an arbitrary-precision integral type. This forms the `State` component of our stack, models the label name counter and is the only mutable state which we make use of.

3.3 Control Flow and Do Notation

Haskell's support for *do* notation as syntax sugar for sequences of monadic computations enables us to represent procedural programming in a block-based way while maintaining a typesafe awareness of the effectful implications.

The most crucial function we make use of, which is associated with the `Writer` monad, is `tell`. It has a type signature as follows.

```
tell :: Monad m ⇒ w → WriterT w m ()
```

In praxis, the function simply appends its argument `w` to the log when invoked by `bind` / `>>=`. We can then use it to express a series of instructions using `tell` as follows:

```
-- | Loads an 8-bit immediate value into a 16-bit memory address
write :: Word16 → Word8 → Lazyboy ()
write addr val = tell [LDrrnn HL addr, LDHLn val]
```

Treating the code generation as a monad enables us to structure individual sequences of instructions into compound operations. These operations (such as `write`) can then be built upon to implement more complex functionality. This is the core philosophy of Lazyboy, and working like this provides us the benefits of both high-level and low-level control over the hardware.

3.4 Optimizations

When implementing primitive operations such as the previously-covered `write`, we consult documentation on the number of cycles each instruction takes (Fayzullin et al., 2001).

Our goal is to implement the functionality in as few cycles as possible. For example, possible ways in which `write` can be implemented are as follows. In these examples we use the value `97` and the address `0xC0D0`, but these have no significance.

1.
```
write:
    ld HL, $C0D0
    ld [HL], 97
```

$12 + 12 =$ **24 total cycles**

2.
```
write:
    ld A, 97
    ld $C0D0, A
```

$8 + 16 =$ **24 total cycles**

3.
```
write:
    ld HL, $C0CF
    ld A, 97
    ld [HL+], A
```

$12 + 8 + 8 =$ **28 total cycles**

In this case, two of the methods are the most efficient ways of expressing this functionality, so there is no single most efficient solution, but in some cases there likely will be. Taking these optimizations into account helps to ensure

there are fewer wasted CPU cycles in the output program, directly impacting its performance.

We are able to check that values provided to instructions are correct at compile time. A good example of this is the instruction `RST` which is used to call a restart vector. There are only certain valid arguments for this instruction, so it's desirable to provide feedback at compile time as to whether the user has made a mistake or not.

The way this is implemented is very straightforward, we use Haskell's pattern-matching functionality over the instruction type and have variants depending on the provided value, as we can see below.

```
show (RST 0x00) = printf "RST␣$00"
show (RST 0x08) = printf "RST␣$08"
show (RST 0x10) = printf "RST␣$10"
show (RST 0x18) = printf "RST␣$18"
show (RST 0x20) = printf "RST␣$20"
show (RST 0x28) = printf "RST␣$28"
show (RST 0x30) = printf "RST␣$30"
show (RST 0x38) = printf "RST␣$38"
show (RST _) = error "Invalid␣RST␣vector␣specified!"
```

Note that `Text.Printf.printf` in Haskell here functions like `sprintf` in C — it does not actually perform IO but instead formats to a string. A future augmentation to the compiler would be to provide more detailed error messages, as at present they yield little information other than what caused the error, so in a large project it is minimally useful for finding the source of the problem.

Another example of how pattern matching is used to provide compile-time assertions about program validity can be observed in the bit-manipulation instructions.

For an instruction, which checks the status of an individual bit in an 8 bit register and sets condition flags accordingly, defined as follows:

```
data Instruction =
     ...
    | BITnr Word8 Register8
```

An entry is added to format the instruction as assembly:

```
show (BITnr v r1)
    | v ≥ 0 && v ≤ 7 = printf "bit␣%d,␣%s" v r1
    | otherwise      = error "integer␣constant␣exceeds␣limit"
```

Predictably, constructing a `BITnr` value with a value argument which is greater than 3 bits results in an error (edited here for concision).

4 Product & Project Evaluation

A loose implementation of the spiral development model was chosen for the compiler, as it allowed for iterative improvement of the core functionality. This approach went hand-in-hand with the use of Git for source control.

Perhaps the initial outline of the deliverable product was overly ambitious, as by the deadline for submission of this report, there are not yet user-friendly abstracted methods for graphics rendering. This was thought up as a flashy and marketable way of introducing people to the utility of the project. There is no doubt that Lazyboy is and will be capable of reaching this point in the near future, and work on the project will continue until it can serve as a satisfactory alternative to the other solutions available to developers at present.

One area in which the project is lacking substantially is in the area of testing — this has been carried out during the project, as part of the spiral model of development, but is not trivially documentable. The way in which this was carried out was by inspecting a halted program generated by the compiler in the mGBA emulator.

If an alteration could be made to how this project was conducted, then a clearer idea and better exploration of testing methods available would be a high priority. It may be necessary to find or modify an emulator to produce traces of how values change over time, and draft "plans" for a given input which can be compared against the output programmatically.

Certain areas of the project are implemented more cleanly and efficiently than once was thought possible, for instance the use of only integral state for label management is very elegant and easy to keep track of.

The suitability of Haskell and its strengths for this line of work are exemplified by the ease with which compile-time verification of instructions are carried out. The language was a great choice for this project.

5 Conclusions

In conclusion, the target of this project has been met in the creation of an EDSL which can target the Game Boy, featuring abstractions in the form of composable actions and integrating deeply into Haskell idioms.

We have explored how authenticity in retro video games is a desirable outcome, the options that developers have at the moment to produce retro games and what an ideal development scenario might look like. We have also looked at related areas of study such as other specifications for enforcing authenticity, the applicability of homebrew retro games programming to fostering learning and justified our choices in implementation.

The compiler produced by this project has been released as free & open source software under the terms of the BSD3 license (*Lazyboy*, 2019). This opens up the possibility of scrutiny, contribution and testing by others, all of which are beneficial to bolstering the stability, functionality and maturity of the project.

While perhaps not suitable for those uninitiated to the world of Game Boy internals right now, the project has a roadmap of features that are to be implemented as well as avenues to explore to assess the viability of new ideas (*Lazyboy Roadmap*, 2019).

Potential improvements and further developments are outlined below:

- Experimentation with creating a live-programming environment in the browser that consists of an editor for code and an emulator alongside. When the code is edited, the ROM could be dynamically rebuilt and displayed in real time in the emulator. This could be augmented with a tutorial, and could serve as an excellent introduction for new users.
- The implementation of an interface to easily make use of the serial port ("Link Cable") of the Game Boy, and exploration of potential modern uses for it.
- Improved tracing and error reporting.
- A means of compile-time checking of memory instructions. For instance, we shouldn't be writing past address 0x9FFF in a VRAM update.

To summarize, the project has demonstrated that Haskell is a viable and fertile ground for experimental compilers targeting older hardware. It seems that making one retargetable might be gravely difficult due to the quirks, intricacies and needs of each platform, but perhaps with sufficient abstraction this may be shown possible in time.

References

Bentley, A. (2019, April 26th). rednex/rgbds: Rednex game boy development system [Computer software manual]. Retrieved from `https://github.com/rednex/rgbds#3-history`

Bowles, C. G. (2017). *Renegade drive: Usage of today's technology in creating authentic 8-bit and 16-bit video game experiences* (Master's thesis). doi: 10211.3/203076

Fayzullin, M., Felber, P., Robson, P., & Korth, M. (2001, October). Everything you always wanted to know about gameboy [Computer software manual]. Retrieved from `http://bgb.bircd.org/pandocs.htm#cpuinstructionset`

Fayzullin, M., Felber, P., Robson, P., Korth, M., 'DP', 'Pan of Anthrox', ... 'Bowser' (n.d.). Gameboy cpu manual [Computer software manual]. Retrieved from `http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf`

Gill, A., & Paterson, R. (2019). Control.monad.trans.writer.lazy [Computer software manual]. Retrieved from `https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Writer-Lazy.html#g:1`

Hope, M. (2001, February 28th). *Gbdk.* Retrieved from http://gbdk.sourceforge.net/

Kacmarcik, G., & Kacmarcik, S. G. (2009, March 4th). Introducing computer programming via gameboy advance homebrew. *SIGCSE '09 Proceedings of the 40th ACM technical symposium on Computer science education*. doi: 10.1145/1508865.1508969

Kögel, S., Stegmaier, M., Groner, R., Tichy, M., Götz, S., & Rechenberger, S. (2018, May). Developing an optimizing compiler for the game boy as a software engineering project. In *2018 ieee/acm 40th international conference on software engineering: Software engineering education and training (icse-seet)*, Gothenburg, Sweden. Retrieved from https://ieeexplore-ieee-org.libaccess.hud.ac.uk/document/8445176/authors#authors

Lazyboy. (2019). Retrieved from https://github.com/ix/lazyboy

Lazyboy roadmap. (2019). Retrieved from https://github.com/ix/lazyboy/issues?q=is%3Aissue+is%3Aopen+label%3Aroadmap

Leijen, D., & Meijer, E. (1999). Domain specific embedded compilers. In *In proceedings of dsl'99: The 2nd conference on domain-specific languages*, Austin, TX. Retrieved from https://www.usenix.org/legacy/publications/library/proceedings/dsl99/full_papers/leijen/leijen.pdf

Nintendo Co. Ltd. (2016, April 26th). *Consolidated sales transition by region* (Tech. Rep.). Retrieved from https://www.webcitation.org/6hBWlh2lb?url=https://www.nintendo.co.jp/ir/library/historical_data/pdf/consolidated_sales_e1603.pdf

Osborn, J., Summerville, A., & Mateas, M. (2017, August). Automatic mapping of nes games with mappy. *Proceedings of the 12th International Conference on the Foundations of Digital Games*. Retrieved from https://dl.acm.org/citation.cfm?doid=3102071.3110576 doi: 10.1145/3102071.3110576

Rgbasm(5) [Computer software manual]. (2018, March 13th). Retrieved from https://rednex.github.io/rgbds/rgbasm.5.html

Van der Jeugt, J. (n.d.). *Hakyll.* Retrieved from https://jaspervdj.be/hakyll/

Various. (2019). *List of best-selling game consoles.* Retrieved from https://en.wikipedia.org/wiki/List_of_best-selling_game_consoles

White, J. (2019). *Pico-8 fantasy console.* Retrieved from https://www.lexaloffle.com/pico-8.php

Wulf, T., Bowman, N. D., Rieger, D., Velez, J. A., & Breuer, J. (2018, June 7th). Video games as time machines: Video game nostalgia and the success of retro gaming. *Games Matter? Current Theories and Studies on Digital Games*. doi: http://dx.doi.org/10.17645/mac.v6i2.1317

Yorgey, B. (n.d.). *Diagrams.* Retrieved from https://diagrams.github.io/

6 Appendix

6.1 Compiler Source

The source code can be found at `https://github.com/ix/lazyboy`.

6.2 Ethical Review Form

See Ethical Review Form document enclosed in ZIP file.