FLOORPLAN: The language of the future

Peter Hebden* peterhebden6@gmail.com
Anna Williams† hello@anna-maths.xyz
Sofia Wolf‡ sofiakwolf@gmail.com

Abstract

The biggest problem in the architectural and building engineering sectors is an overabundance of underutilised floor plans. The biggest problem in software development is the underabundance of programs to overutilise. We propose that these problems may be solved by a common solution; a solution which also serves to make redundant all other programming languages (especially `FORTRAN`). We present a language for the now, and for the future. We present: `FLOORPLAN`.

1 Introduction

Since its first appearance in 1957, `FORTRAN` has experienced an enduring popularity unparalleled by any other language. After its last stable release in 2023, it has only *increased* in popularity, re-entering the TIOBE top 10 [5]. However, it is also certainly not mainstream. As of 2025, its primary use cases are in banks and nuclear reactors [3]. What do those two things have in common? It's not the use of safety-critical legacy codebases—it's floor plans. Whether you like it or not, both banks and nuclear reactors are physical locations with a layout of rooms and such. Taking a leaf out of `FORTRAN`'s book, it was vital to consider these use cases in a language of the future. But, equally, floor plans are every day. Quotidienne. Domestic, even. Most (if not all) houses have floor plans. Places of work generally do, even if it's rather labyrinthine. Perhaps, uniting these concepts was what needed to happen. As it is, floor plans are the daily bread of architects and civil engineers, but are rarely used after or outside of these instances. This is, quite frankly, wasteful, especially in our modern world.

We present a solution addressing both of these challenges—in one fell swoop—in the obvious way: by making floor plans executable. We argue that this presents a number of distinct advantages over traditional programming, such as:

- **Increased sustainability in the architectural and building engineering sectors.** Billions of floor plans already exist and are just rotting, waiting to be run.
- **Obvious program structure.** Floor plans are designed to present a layout, and so they make structure obvious. Programmers have been fighting for decades to make their program structure more obvious; in roundabout ways such as through indentation and syntax highlighting.[1] Ultimately, programs are multi-dimensional, and we need more than just the free monoid on ASCII characters to represent them.

To these ends we present `FLOORPLAN` [2], a Haskell implementation of an interpreter of floor plans. `FLOORPLAN` works by exploiting the computational connotations we find to be inherent in floor plans, which will be discussed in Section 3. We show how to compute factorial in your office, or Euclidean division with a castle.

*University of Birmingham, UK
†University of Birmingham, UK
‡University of Birmingham, UK

[1] Although it has been suggested that the syntax highlighting actually encodes the program *semantics* [1].

A note for the yanks. We are aware that, due to geographical issues, many attending SIGBOVIK may be from across the Atlantic. As such, we've included this note to set the record straight and clear up any possible confusion that may arise in reading this paper:

- A lift is an elevator,
- The word "labour" has a u in it,
- The first floor of a building sits above its ground floor.

2 Syntax

Programs are floor plans.

3 Connotational Semantics

We are grateful to have received feedback that the connotations inherent in floor plans are less intuitive to some than they were to the authors, and so we make some effort to elucidate them in this section. First, we have to introduce you to: *The Guy*.

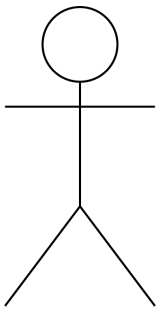


Figure 1: The Guy

The Guy also comes equipped with an arbitrarily large notepad.

Figure 2: His Notepad[2]

[2]There is also a human hand in this image. That is not a part of the notepad.

The basic premise is that a floor plan connotes a specific journey that The Guy may take inside it with his notepad. To run each program, we imagine that we give some initial value to The Guy, who then scribbles it down on his notepad before entering the building. As The Guy makes his way through the floor he interacts with the objects in each room, which may alter the value of his notepad or influence the path he takes. Each time The Guy enters a room he runs through his checklist of things to do before passing forward to the next room. His checklist is as follows:

1. The *secret first thing* (spoilers),
2. Interact with the objects in the room (shredder, photocopier, etc), scanning row by row from the North-West corner of the room to the South-East corner,
3. Interact with any stairs or lifts in the room,
4. If there is a window, show what you've written on the notepad to any outside observers,
5. If there is a door leading out, exit the room,
6. If there are no doors leading out, you are done! Your notepad tells you the value of the floor.

Object-oriented programming

We first introduce the simple objects; namely the shredder, the wastepaper basket, and the photocopier.

- When The Guy comes across the shredder, he first checks what the current value is on his notepad. He then tears the page out of his notepad and copies down a new value, based on the type of data he has before then shredding the paper. If he has a string, he copies down all but the first value, if he has a boolean, he copies down the negation of the original value, and if he has an natural number, he copies down the predecessor of that number.
- When The Guy comes across the wastepaper basket, he checks his current value for "usefulness". A useless value corresponds to the empty string for strings, false for booleans, and 0 for numbers. If he has one of these non useful values, he tears out the page from his notepad and puts it in the bin, before leaving by the leftmost[3] exit in search of a new value. Otherwise he keeps the current value and leaves by the second leftmost exit.
- When The Guy comes across the photocopier, he places his notepad inside and meticulously copies each page, which he then paperclips onto the front.

One important case is when there are no objects in a room. If the room has no name, he doesn't do anything. If it does have a name, The Guy is so excited by finding a rare room with a name that he forgets his duties and overwrites the current value with the name of the room.

Connotational denotations. If you are wondering about the precise nature of these connotations, we can go deeper and reason about the denotations of our connotations. The set V of things The Guy is capable of writing is given by the disjoint union of $\{\texttt{true}, \texttt{false}\}$, S the Kleene closure on ASCII characters, and the natural numbers $\mathbb{N}$ (regrettably with 0).

However, we must also consider that the guy is capable of photocopying and shredding pages. We denote the notepad (Figure 2) by the freely generated (non-unital, non-associative) magma on V.[4] Writing the magma composition as $\langle \cdot, \cdot \rangle$, the action of the photocopier on The Guy's notepad is to take n to $\langle n, n \rangle$. Should The Guy encounter a shredder, he only shreds when the denotation of his current notepad lies in V—otherwise he would jam the shredder. This is a very mindful Guy.

[3] NB: Up is more left than down.

[4] Colloquially known as a *cons pair*.

Isn't this all too much for just one guy?

So picture this, The Guy enters a room and there are two exits. What a dilemma! What could he possibly do? Of course, the answer is clear—he simply splits himself in twain, creating a *clone* of himself, and splits the current value between the two halves to match. Now he can go through both doors, no problem. If he doesn't have enough information in his notebook for two people, The Guy is forced to just choose a door and go it alone. The natural choice is of course the leftmost door.

(More) connotational denotations. If the denotation of the current notebook looks like $\langle n, n' \rangle$, the clone who exits via the leftmost door does so with a notebook denoted by n. The other clone exits via the rightmost door with a notebook denoted by n'. In all other cases, The Guy continues alone.

Reaching new heights

Floor plans don't often come in isolation. Often, a building may have multiple floors, linked by stairs and lifts. So The Guy has just gotten to a lift or set of stairs, but how is he supposed to go up those *and* find his way back down? These buildings are practically mazes! The only logical solution is to send an exact clone of yourself—notepad and all—into the new floor. This fresh clone can then do all the work for you on the next floor, and then call back down the stairs whatever ends up on his notepad.

It may be becoming clear that there are a good few ways to create clones while visiting the tangled towers and hallowed halls of `FLOORPLAN` buildings—from photocopiers to lifts—which may raise questions such as, *"isn't that a lot of clones?"*. The answer to that question is, yes. To keep ethics committees happy, our interpreter reports at the end of an evaluation just how many clones' labour was used in the computation. It's always good to acknowledge the unseen labour of many-universe clones.

The secret first thing (and team-building exercises)

Now that we have discussed clones, we can talk about rooms with two entrances. If The Guy gets to a room with two entrances, he has to wait for one of his clone buddies to show before continuing the computation. Why be lonely when you can wait for a friend, right? When his pal shows up, the two clones merge together and collate their values into one.[5] For example, rooms with a team-building exercise often have two entrances. It takes two to tango, after all. A team building exercise sees the values that the clones just put together be *merged* somehow. For example, numbers are added and strings are concatenated.

It's important that the clones are highly coordinated. After assessing a room, they should wait until all the other clones involved in assessing that floor have assessed their own rooms before proceeding into the next one. In effect the clones work together to do a breadth-first sweep of the floor.

Addressing the controversies. One may notice that throughout this discussion we assume that it is a guy doing all this work for us (and his many clones). This is because it is more fun to imagine men going through hell.

[5] If the clone who entered from the leftmost door has a notepad denoted by n, and the clone who entered from the rightmost door has a notepad denoted by n', they merge into a clone with a notepad with denotation $\langle n, n' \rangle$.

4 Example buildings

Hello World. The simplest building is the one with one floor and one room. Add an entrance, a window, and a label, and `FLOORPLAN` sees the famous "Hello World" program (Figure 3).

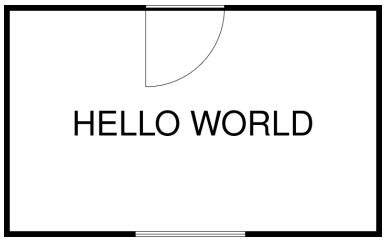


Figure 3: Hello world floor plan

Yes-No bungalow. The Yes-No bungalow (Figure 4) represents one step up from Hello World. It connotes a notepad man who enters carrying a notepad with something written on it. Encountering the wastepaper basket in the first room, he determines whether or not what he has written down is useless. If it is, he goes left and shows "YES" to the world. Otherwise, he goes right and shows "NO".

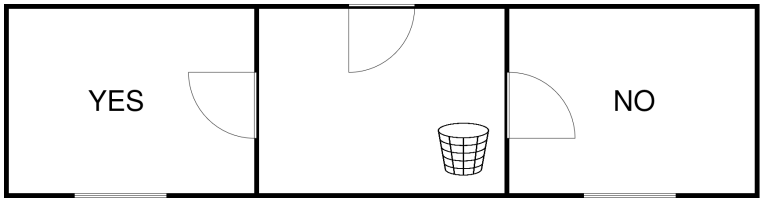


Figure 4: The Yes-No bungalow

Adder Condo. The Adder Condo (Figure 5) is the first example where we have to use clones. It is a little contrived as the same thing could be achieved with just one room, but it's worth stepping through. We assume The Guy enters the building with two numbers in his notepad. In the first room, he creates a clone of himself, and gives one of the numbers to his clone. They part ways and enter the NE and SW rooms. Immediately they each enter the final room, where they pair their values together back into one notepad, and add them together during the team-building exercise.

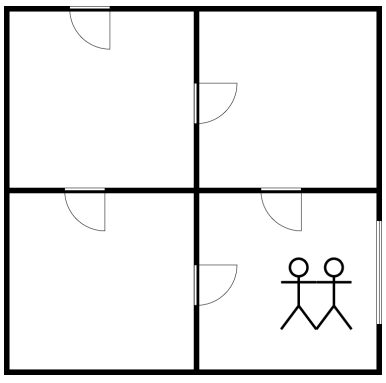


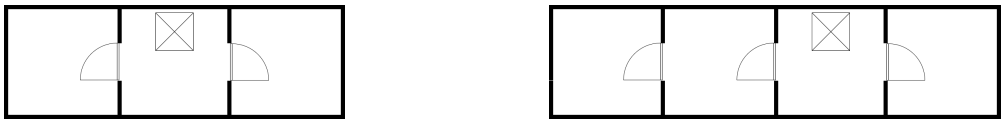
Figure 5: The Adder Condo. It's worth mentioning explicitly that the two stickmen in the SE room are *not* instances of The Guy. They represent the team-building exercise. The Guy is connotational, not syntactical, so this is consistent.

First & Second Projections

The task of projecting the first and second values from a pair can be approached in two different ways. These strategies enumerate different ways in which clones can be culled, such that we can drop their values. Hey, take it up with the ethics committee.

The *Rigged Footrace* approach. The first strategy requires setting up a footrace (the winner, of course, escapes being culled). We do this as follows: The Guy exits a lift with the pair of values we wish to project from on his notepad. He then enters a room with two exits and clones himself, taking the first pair value from the notepad and entering the leftmost door. The clone takes the second pair value and enters the second-leftmost door, as we know. The race is now on. In the case of the first projection, The Guy wins, dealing with his room and realising there is no exit first, thus calling out his value to the clone waiting below. In doing so, he unfortunately dooms the second-leftmost clone to cease to exist—and to never return his value.

This can of course be rigged such that we obtain the second projection, by simply adding another exit to the leftmost room such that the second-leftmost clone finishes first (also calling out his value first).



(a) The first projection (b) The second projection

Figure 6: Projections via the footrace method

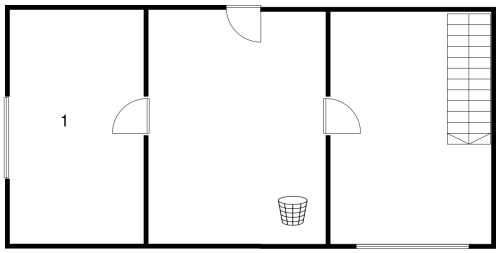
The *Eternally Lonely* approach. The second strategy involves setting up a situation in which clones end up eternally waiting for their buddy to arrive. This approach has its advantages—it doesn't require a lift as above, and can occur in-floor (ideal for situations where buildings are limited in height). To set up this HR violation of a floor, The Guy walks into the first room with the pair on his notepad. This room has two exits, so The Guy clones himself, as before, walking through the leftmost door with the first value of the pair. This leads to a room with two entrances, so he starts waiting for someone to join him. Unfortunately, due to how one of the rooms connected to this one has no entrance, he will wait forever. This gives the clone as much time as he pleases to do anything he would like, and then return the end value.

(a) The first projection (b) The second projection

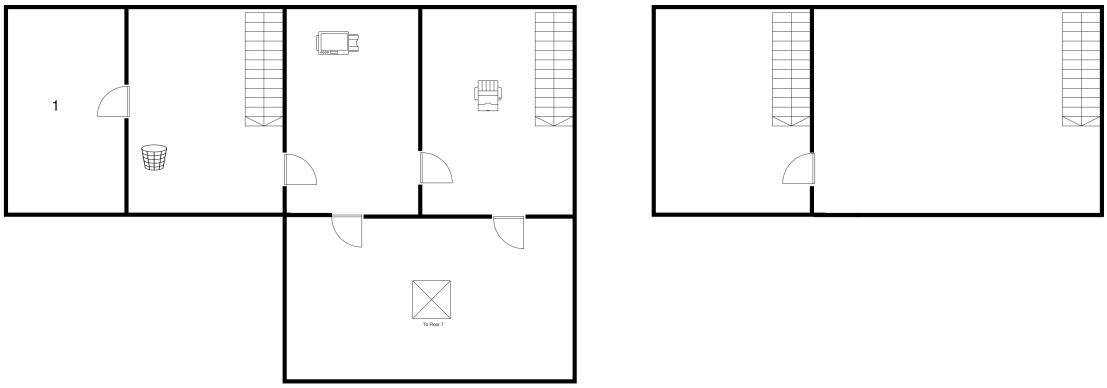
Figure 7: Projections via the eternally lonely method

Factorial building. We present the factorial building (Figure 8) but we refuse (with union-backing) to do a full walk-through of it. This building has been described as "malware" by some detractors. We do not agree with this designation, but please only run factorial 12 or higher if you have a lot of RAM and/or are willing to lose anything unsaved on your computer. If you do choose to do so, please inform us of how long it took to run and the memory usage. 16GB was not enough and produced concerning noises.[6]

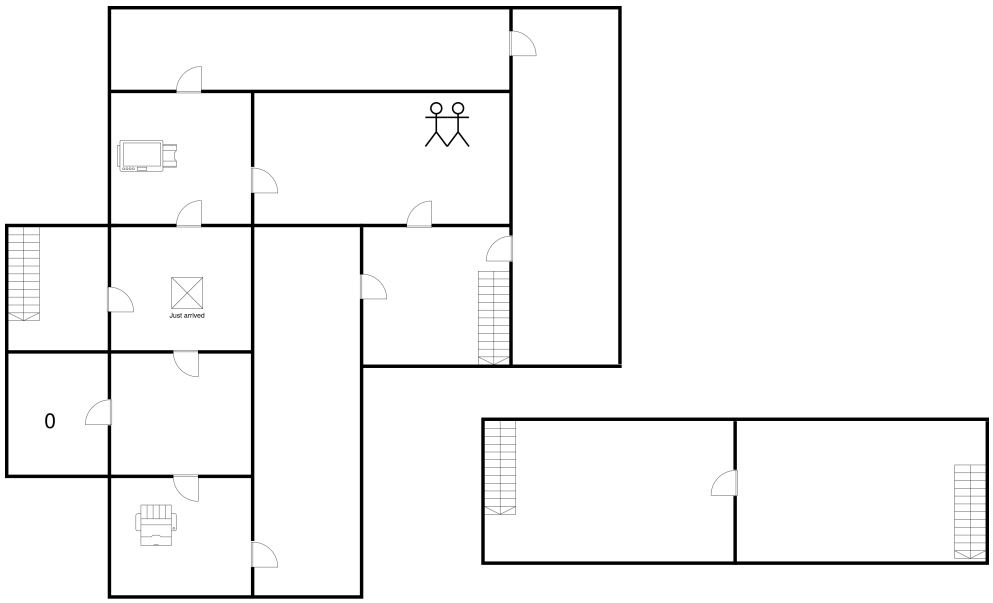
[6]Unfortunately, we accidentally optimised the interpreter and now this isn't really true. We can do factorial 1000 in about six seconds with floor plans.



(a) The ground floor of the factorial building is responsible for showing the final result.



(b) Floors 1 and 2 (shown left and right, respectively) compute $n \cdot (n-1)$, with multiplication performed by sending a clone up the lift to floor 7.



(c) Floors 7 and 8 (shown left and right, respectively) compute multiplication of x and y by recursively evaluating $y + (x-1) \cdot y$.

Figure 8: The factorial building

The DivMod Castle. DivMod castle is one of the more complex examples, it boasts three lifts and eight total floors. It also uses three standard library functions, these being minus, less than and second.

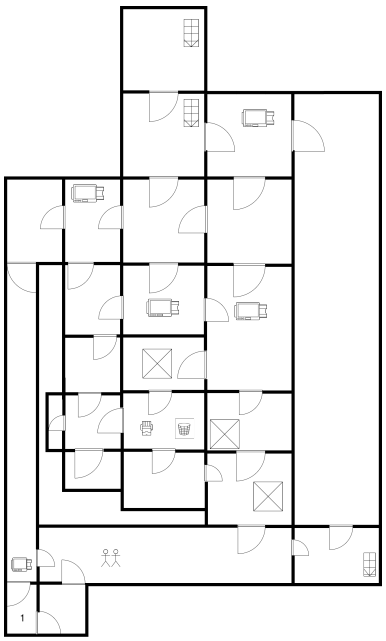


Figure 9: The first floor of divmod castle

FizzBuzz Mansion. The final example we will mention is the colossal FizzBuzz mansion. We will not present any of its floors here; but it includes the divmod castle within it. In total it boasts 16 floors, but proves that `FLOORPLAN` is useful for practical problems such as leetcode interviews.

5 Implementation

All of the programs described above may be found in the repository [2] and can be run by our interpreter. In this section we describe some of the details of the our implementation.

Parsing floor plans

We simplify the problem of parsing floor plans—which are generally presented as unwieldy vector graphics (see above)—to parsing UTF-8 encodings of floor plans. Our best friend is the *Box Drawing* block (02500–0257F) as well as some quarter circles from *Geometric Shapes* (025DC–025DF). Objects with significant connotations must therefore be mapped to unicode characters, which led to its own set of challenges. ≅ provided an excellent representation of a photocopier were one to squint just right, and the # symbol is clearly a representation of a wastepaper basket, but we were at a loss for the team-building exercise. In the end our implementation used the *Doubled Female Sign*[7], or codepoint U+026A2, as it looks somewhat like two people standing next to each other. While lesbians are not represented in the connotational semantics of floor plans, they are at least an implementation detail, and as such we have chosen to license the software under the *Gay Agenda License* (GAL-1.0).[8,9]

[7] If you've ever tried to load a unicode font into LaTeX you will understand why we have opted to not.

[8] Originally written for a previous SIGBOVIK submission [4].

[9] This license may have superpowers. Already at least one known-transphobe failed to compile `FLOORPLAN` after selecting the wrong GHC version.

A buildings-first approach

We must have humility and accept that floor plans are first-and-foremost plans for building layouts, and any computation they represent is incidental.

This leads to some important implications for `FLOORPLAN` programmers—programs must not merely be valid programs to run, but valid *buildings*. First of all, it's vitally important that your staircases and lifts line up correctly between floors—it's possible that we could have inferred a correct alignment, but then we'd be interpreting an invalid building; a cardinal sin.

6 Conclusions and Future Work

We have solved most of the pressing problems in both software architecture and civil engineering, but there are still a few points where progress could be made, which are:

1. It doesn't work on Microsoft Windows.[10]

Acknowledgements. `FLOORPLAN` was originally created (with SIGBOVIK in mind) in 24 hours from 12pm on March 22 to 12pm on March 23 for *birmingHack*, the first Hackathon of its kind hosted by the University of Birmingham School of Computer Science and Computer Science Society. We are grateful to the organisers of the event for providing us the opportunity to step away from our usual work in order to focus on the real problems of computer science.

We also thank our friend Fern Warwick for discovering the genius *Eternally Lonely* approach to pair projections, and Todd Waugh Ambridge for proofreading our draft on extremely short notice.[11]

Anti-Acknowledgements. We would like to un-thank the organisers of birmingHack for the no-sleep suffered by the authors on the night of March 22.

Co-Acknowledgements. On behalf of the banks/nuclear reactors/&c. who have been separately maintaining both floor plans and `FORTRAN` programs for all these decades, we would like to thank *ourselves* for having halved their workload. We expect to see a rapid adoption of `FLOORPLAN` from these sectors in the coming years.

References

[1] William Gunther and Brian Kell. "WysiScript: Programming via direct syntax highlighting". In: *a record of the proceedings of SIGBOVIK 2017*. Apr. 2017, pp. 119–127.

[2] Peter Hebden, Anna Williams, and Sofia Wolf. *FLOORPLAN*. `https://codeberg.org/Piturnah/birminghack-floorplan`. 2025.

[3] *This seems true and probably is*.

[4] Nicole Tietz-Sokolskaya. "I'm going to Hurl". In: *a record of the proceedings of SIGBOVIK 2024*. Apr. 2024, pp. 226–229.

[5] TIOBE. *TIOBE Index for March 2025*. 2025. URL: `https://www.tiobe.com/tiobe-index/` (visited on 03/26/2025).

[10] *Microsoft Windows* is a niche operating system most often used for video games and possibly accounting.

[11] And basically on his birthday.