

Paul Tarvydas

CEPTRE -WALKTHROUGH OF THE DUNGEON CRAWLER EXAMPLE

Based on the paper: Ceptre: A Language for
Modeling Generative Interactive Systems

THE PAPER

<https://www.cs.cmu.edu/~cmartens/ceptr.pdf>

Source code for a more involved version of RPG:

<https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg.cep>

AGENDA

Ceptre: A Language for Modeling Generative Interactive Systems

Chris Martens
Carnegie Mellon University
cmm2m@cs.cmu.edu

18.500

Introduction

Today, game designers and developers have a wealth of tools available for creating executable prototypes. Readily-available engines such as Unity, Unreal, Patcherific, and Lab42 provide a range of features for prototyping and rapid development of games. On the other hand, to avoid interesting novel mechanics in any of these tools typically requires the use of off-specified, game-specific programming languages. For example, Unity's C# scripting language is designed to support game development, but it is not well-suited for prototyping a language such as C# or F#. In addition, Unity's constraints are often too strict for prototyping, such as requiring a specific grid size. Patcherific and Lab42 provide a more general-purpose programming language, and Patcherific simply ignores the author from going outside the well-specified 10x10 grid boundaries.

The concept of operational dynamics (Meadow and Winslade 2005), the underlying substrate of meaningful change, happens at step which mechanics are built, was recently highlighted by Koenig et al. (2017) in their study of game design and analysis. As most prototyping tools, there is a

1

2

Cv

on a stage of 10x10.

The player must then try to switch the direction of the ball.

The development of this example has been extremely slow due to the need to learn how to use the engine while making non-player-controlled stages (like Figure 1) switch stages to follow player-controlled stages (like Figure 2). In fact, we have had to learn how to make the ball move in a straight line, as well as how to make it move in a circle. We could negotiate the two rules in the 1st stage to be determined by the player's movement, but we will talk about this later.

Figure 1 - A Stage 1 with a Bright Ball

 A screenshot of a game stage with a light blue background. A bright yellow ball is positioned in the center-left. It has a small trail of yellow light behind it, indicating it is moving to the right. The ball is surrounded by a faint circular glow. The stage is bounded by black lines forming a rectangle.

Figure 2 - A Stage 2 with a Bright Ball

 A screenshot of a game stage with a light blue background. A bright yellow ball is positioned in the center-left, similar to Figure 1. However, its trail forms a complete circle around it, indicating it is rotating clockwise. The ball is surrounded by a faint circular glow. The stage is bounded by black lines forming a rectangle.

We have been working on this stage, then we got another assignment which required us to make the player's death.

So, we have to make the ball move in a way of loops in Cogito to prevent certain instructions before multiple computational and hence decision processes in a strategy game setting.

Related Work

Lisence has been used previously to study games, particularly in the interactive storytelling paradigm (Zheng et al., 2011; Zheng, Chittenden, & Gaver, 2012), and in serious games (Carroll and Carroll 2010). For instance, Zheng et al. model story choices as linear logic propositions and use a modal logic to reason about the possible outcomes of each choice. Related formalisms, such as Petri nets (Aoyagi and Renge 2009) and state transition diagrams (Zheng et al. 2011), have been used to similar ends. These formalisms are all limited to *unstructured* processes, however, meaning they do not encode information about the game's story character and location to make use of it. In contrast, our approach is to encode game-specific knowledge and rules directly into the game grammar itself, such that the game can easily reduce the grammar to a first-order logic, which we have included in Cogito (but can support in other languages as well). This makes our approach unique, as no other system to date has attempted to do so without tying up game semantics.

Some formalisms have been proposed to represent the semantics of general (video game) descriptions and the execution of game logic. For example, the Game Logic language family, as listed in the Dagostino paper proposing VIGOR, is a game logic transcription (Dagostino et al. 2012). Another example is the Game Logic Language, which is based on objects in 2D space. Gottschall, Deneve, Green, and McDaniel (2010) propose a game logic language for describing puzzle states, but they do not provide a logical semantics for their language. In contrast, we at IARCS have investigated computer aided authoring of games through such means as the Game Logic Language (Gottschall, Deneve, Green, and McDaniel 2010).

Smith 2012), which shares with our approach a use of logic programming and Prolog. In their work, predicates are treated as nested data objects (i.e., the indices of predicate over which program rules are written, whereas they do not have to be in our work). They also discuss the semantics of the logic programs themselves (as opposed to formal analysis as distinct from the previous article (although they are concerned semantically)). We believe that our approach is more appropriate for game design, as it is much easier to implement logic programs in Prolog than to implement logic programs in Prolog-like languages (such as the one used in the previous article).

Finally, several game creation tools have gained popularity in recent years, such as Construct 2 (Construct 2 2012), which attract to us one of striking a balance between expressiveness and ease of use. In addition, we have used Koda (Koda 2012) and Puzzlyfication (Lawrie 2012). We have carried out detailed evaluations of these tools and found them to be useful and are hopeful that, despite the power sacrificed for generality, Cogito can provide the game-specific, efficient and flexible tools in a way that may be more appropriate for game design.

Conclusion

We have presented a logic-based framework for authoring and reasoning about game logic designs, and an initial implementation of this framework in a game creation language, which models programs as prolog search trees. The language is designed to be easy to learn and to use, and to support game design. It has already inspired many stories and a diverse range of applications. We believe the use of this language as a generic system to which game designers can add domain specific knowledge will allow us to extend the language to support new application domains. We are actively developing the language in the open, and welcome contributions from members of the game design community at large.

Given the success of Cogito, we are currently working on Cogito for game applications. We have in progress several candidate algorithms for checking program-specified conditions in game logic programs, such as the detection of cycles and contradictions. These tools put us in the position of the game designer: game creators can now experiment with Cogito to detect errors in their logic programs and to experiment with logic analysis (Smith and Whitehead 2012), for example.

Given the potential of Cogito for game design, we see many opportunities, such as memory management, concurrency, game theory, and quantum physics. To conclude (but very briefly), we believe that the most interesting aspect of the name "gameformalism" is in computer science, robust to all kinds of games, and that the most interesting aspect of what we formalise is that it is both logically and computationally equivalent to video game designs. By providing a logic-based language for game design, we hope that game designers will be able to author games more easily and all less expensively. To game designers, we say to you: we are here to provide a tool that makes the incomprehensibility of game specifications in the language brief.

Acknowledgments

This work was partially funded by grants NPF07-011-120, from the Queen Silvia Research Fund in memory of the Queen Silvia (Sweden). The statements made herein are solely the responsibility of the authors.

References

Aspinwall, M., and Rogers, D.: 2000, 'Modeling games with pure nests: Breaking New Ground in Game Theory', *Play Practice and Theory* 20(2), 209–230.

Bonacina, M.: 2007, 'A note on the complexity of the logic of linear logic for non-linear encodings', *ICALP Proceedings in Informal Semantics and Logic*, 1–10.

Chapman, R., Podelski, A., and Rybalchenko, A.: 2009, 'Syntax building based on local methods and adaptive semantics', *PLDI* (USA), 1–12.

Deng, K., Gao, S., Guo, J., Chongraphan, R., and Rybalchenko, A.: 2011, 'A formal framework for reasoning about programs', *PLDI* (USA), 1–12.

Dengrao, J.: 2011, 'Reasoning methods to study logic games in games'. In: *Artificial Intelligence in the Game Development Industry*, 1–10.

Elmer, M., Lutzow, J., Lutze, M., Schubert, T., Thompson, T., and Togelius, J.: 2013, 'Towards a viable game description language for hybrid deductive and computational intelligence', *Games* 4(6), 1–20.

Gurev, J. Y.: 1987, 'Linear logic: Theoretical Computer Science', *Journal of Symbolic Computation* 3(1), 1–18.

Hansen, R., Howell, F., and Scott, M. O.: 2011, 'Toward supporting writers with potentially generated game worlds', *Computational Intelligence and Games* (CIG), IEEE, 1–8.

Holte, R. S., and Milne, D.: 1999, 'Logic programming as a fragment of linear logic: Approximation and computation', *ACM SIGART Newsletter* 12(2), 157–160.

Jurjens, H., Korten, S., Sulikova, M., and Synta, K.: 2009, 'A formal framework of reasoning systems: In software engineering (FROPS). Proceedings. 13th International Conference on', 63–72, IEEE.

Korten, S., and Synta, K.: 2009, 'A formal framework of reasoning systems: In software engineering (FROPS). Proceedings. 13th International Conference on', 63–72, IEEE.

Lavelle, S.: 2013, 'PostgreSQL game engine', <http://www.postgresql.org/reports/10/pg.html>. Accessed 2013-05-31.

MacLennan, B. S.: 2011, 'The design of Kappa: A toy model of computation', *Computational Intelligence and Games* (CIG), IEEE, 243–248.

Makinson, D., and Soberón, A., and Scott, M. O.: 2014, 'Generative early worlds as logic programs', In: *Artificial Intelligence in Game Development*, 1–10.

Milne, D., and Winter-Pica, M.: 2009, 'Defining game-based logic', *Logics Games Research Association (LGRA)*.

McCoy, J., Thomas, M., Samuel, B., Nixon, B., Moran, M., and Winter-Pica, M.: 2013, 'Composing a fully automatic model for socially-oriented gameplay', In: *Proceedings of the Artificial Intelligence for Interactive Media Technologies (AI3) Workshop*, 10–15.

Moshe, B., and Margalit, B.: 2008, 'Schele: A tool for generating game engines', *Software: Practice and Experience* 38(2), 179–198.

Nelson, G.: 2000, *The Usenet Designer's Manual*. Prentice Hall, NJ.

Graven, J.: 2008, *IEEE Transactions on Professional Communication* 51(4), 742–746.

Obregon, J. C., Gómez, A., and Matos, M.: 2013, 'Mobile computational critics for games', In: *ASIDE*.

Perez, J., Cimino, M., and Charles, T.: 2010, 'Applying game theory to the design of AI opponents: Using state constraints', *ACM Trans. Game Dev.* 7(1), 1–20.

Schack-Nielsen, A., and Schmidmeier, C.: 2008, 'Call - a logical framework for deduction and consequence relations', *Proceedings of the 2008 International Conference on Automated Reasoning (IJCAR'08)*, 520–528, Springer LNCS 5015.

Reiter, E., and Scherl, R.: 2001, 'Expanding the expressive range of a logic program', In: *Proceedings of the 2000 Workshop on Procedural Content Generation in Games*, 4–10, ACM.

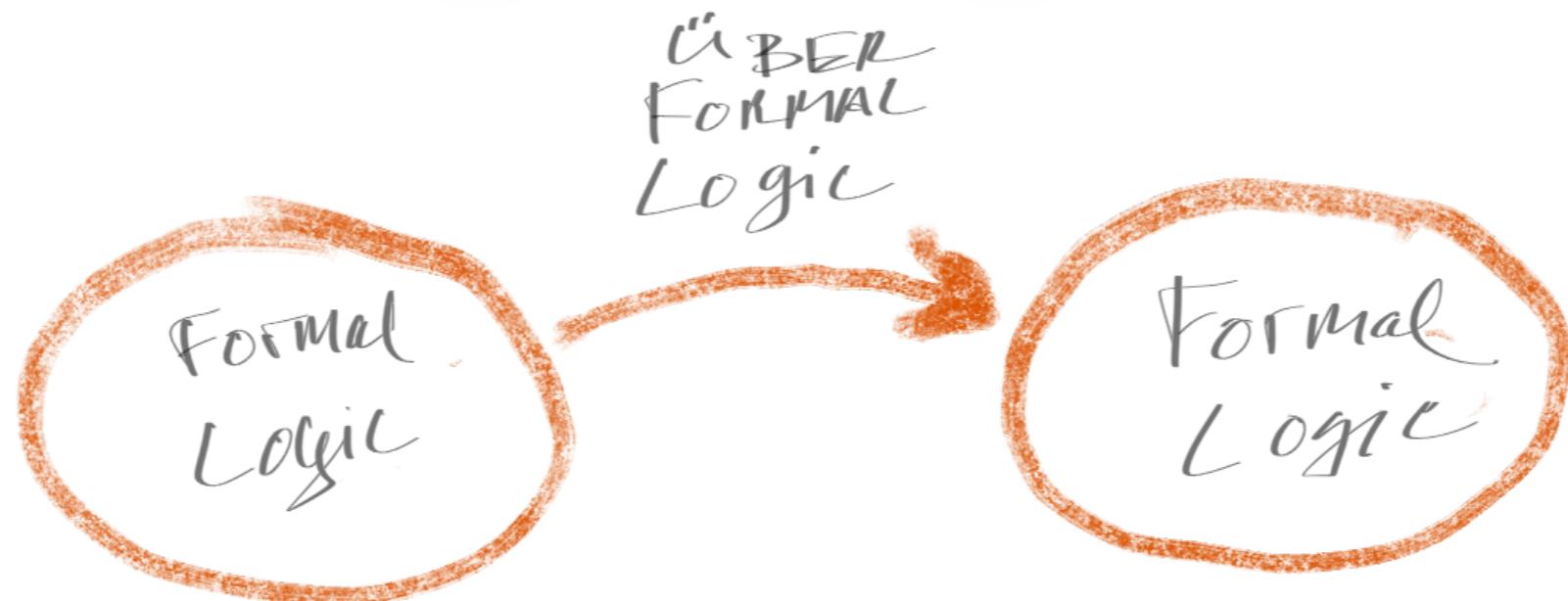
Smith, A. M., Nelson, M. J., and Matos, M.: 2009, 'Computational critics for play testing game sketches', In: *ASIDE*, 1–6.

Smith, A. M., and Matos, M.: 2010, 'Automatic game design', PhD Dissertation, University of California, Santa Cruz.

Zach, A., and Reid, M. O.: 2008, 'Automatic game generation for mobile devices', In: *Proceedings of the 2008 AAAI Conference on Artificial Intelligence*.

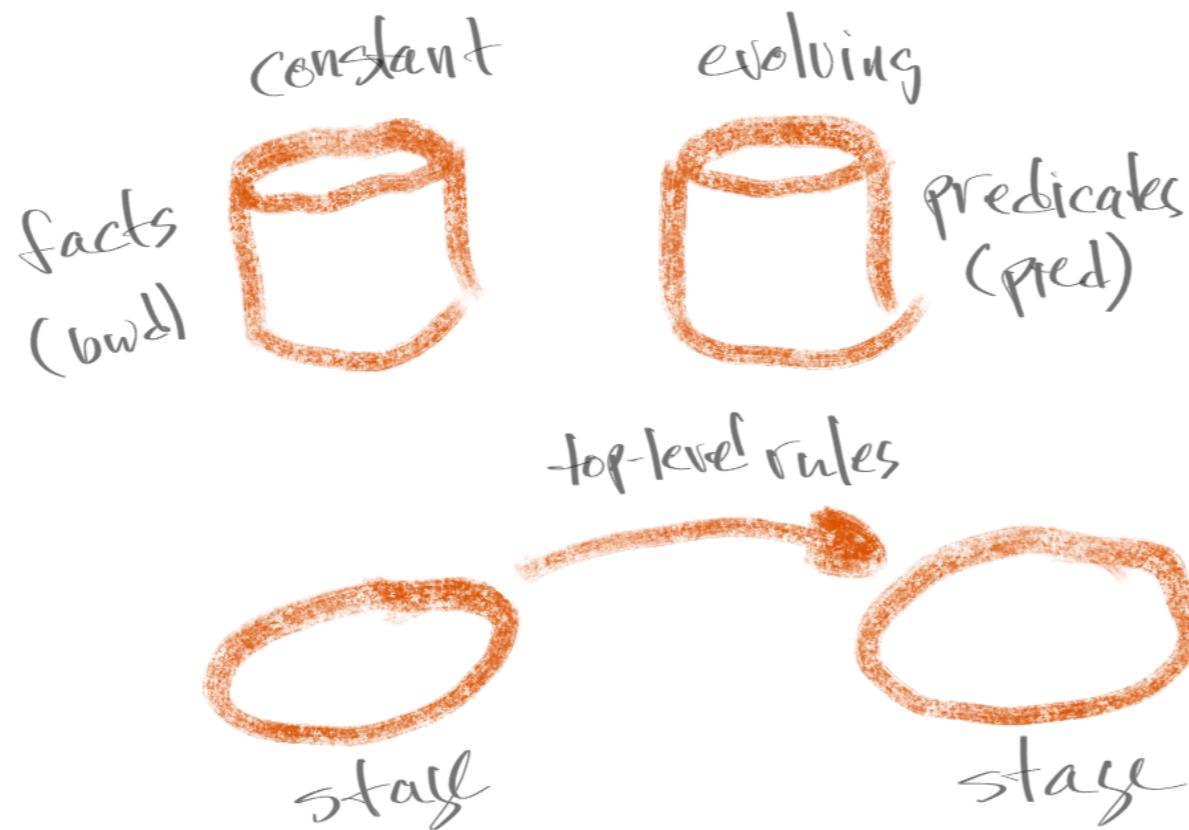
I view the code from the perspective of thinking of Ceptre as a language for writing practical gaming software, instead of as a way to formally analyze the required operations.

CEPTRE



Ceptre uses formal logic ("linear logic", "multiset rewriting") to describe low-level operations and higher-level operations.

Ceptre uses the same syntax throughout.



Ceptre operates on one stage at a time.

It finds enabled rules, then applies *one* rule to the factbase.

The choice of which rule to apply is *nondeterministic*. If the rule is marked “interactive”, then the player gets to choose which rule to apply, else, the system chooses a rule randomly.

This process repeat until no rules are enabled, i.e. the stage is “quiescent”.

Higher level rules can match for a quiescent stage using the *qui* operator.

Initially

minimum value as a predicate option.

read as A plus B capped at Cap is C. we omit the definition of the predicate for brevity, but make use of it in later rules. We also use backward-chaining predicates to define a few constants, such as the player's maximum health and the damage and cost of various weapons:

```
max_hp 10. damage sword 4. cost sword 10.
```

We then define a initial context and an initial stage that sets up the game's starting state:

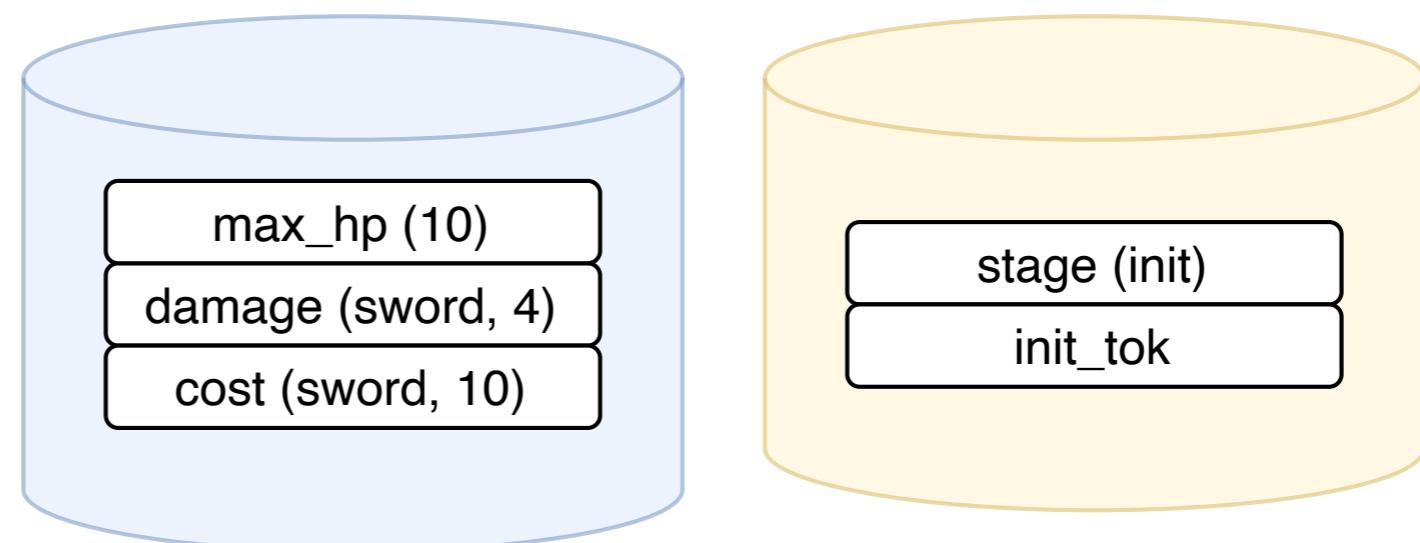
```
context init_ctx = {init_tok}.

stage init = {
    i : init_tok * max_hp N
    -o health N * treasure z * ndays z * weapon_damage 4.
}
```

We define the rest of the game using a “screen” idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

```
qui * stage init -o stage main * m

stage main = {
```



Caption

also use backward-chaining predicates to define a few constants, such as the player's maximum health and the damage and cost of various weapons:

```
max_hp 10. damage sword 4. cost sword 10.
```

We then define a initial context and an initial stage that sets up the game's starting state:

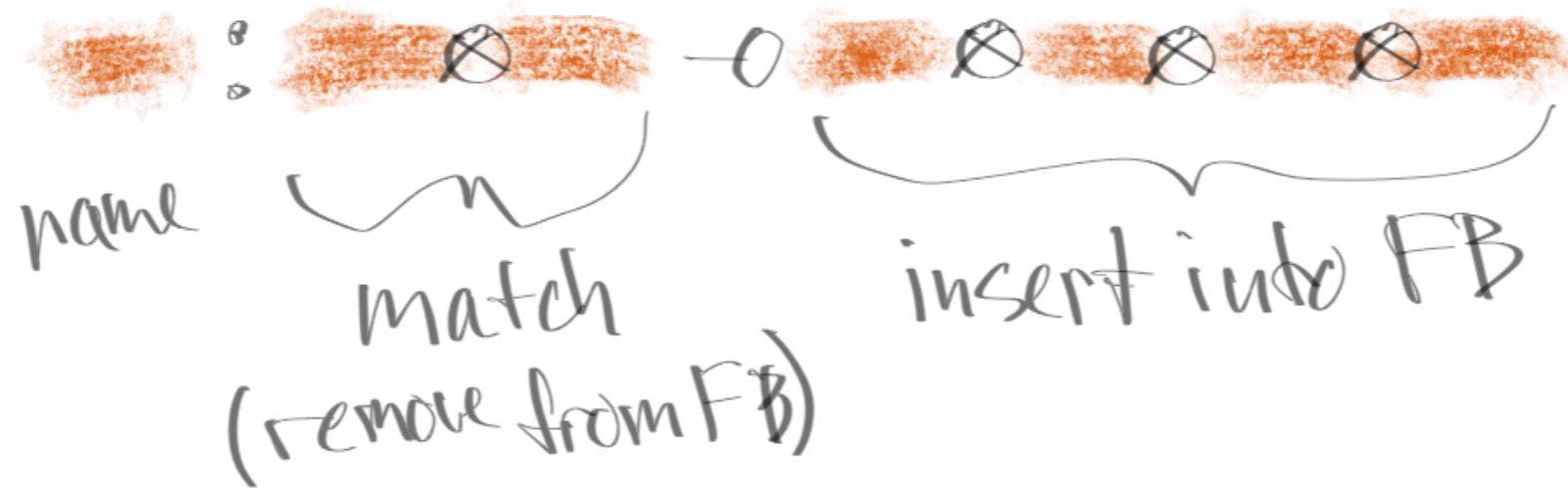
```
context init_ctxt = {init_tok}.
stage init = {
  i : init_tok * max_hp N
  -o health N * treasure z * ndays z * weapon_damage 4.
}
```

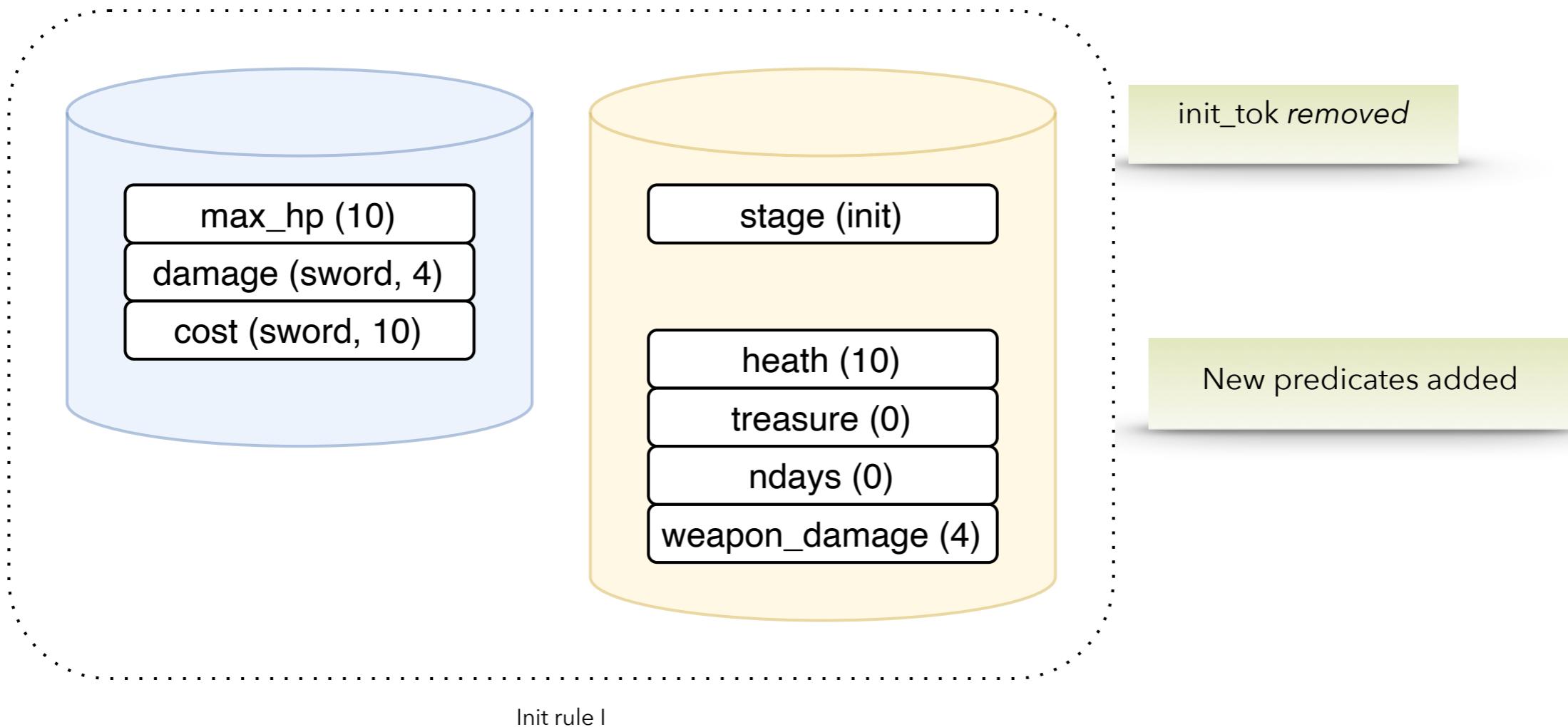
We define the rest of the game using a “screen” idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

```
qui * stage init -o stage main * main_screen.

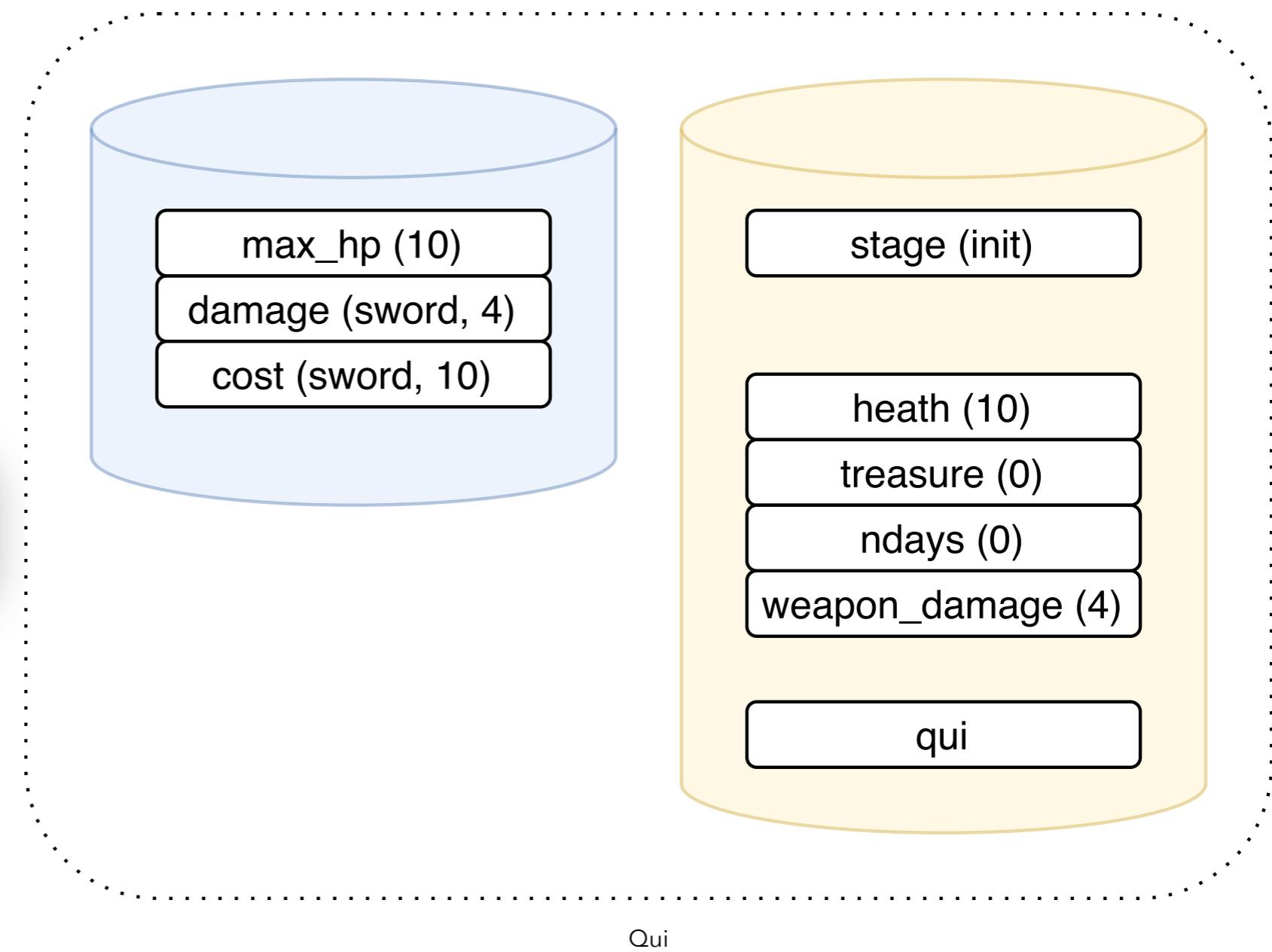
stage main = {
  do/rest : main_screen -o rest_screen.
  do/adventure : main_screen -o adventure_screen.
  do/shop : main_screen -o shop_screen.
```

Ceptre code stage init

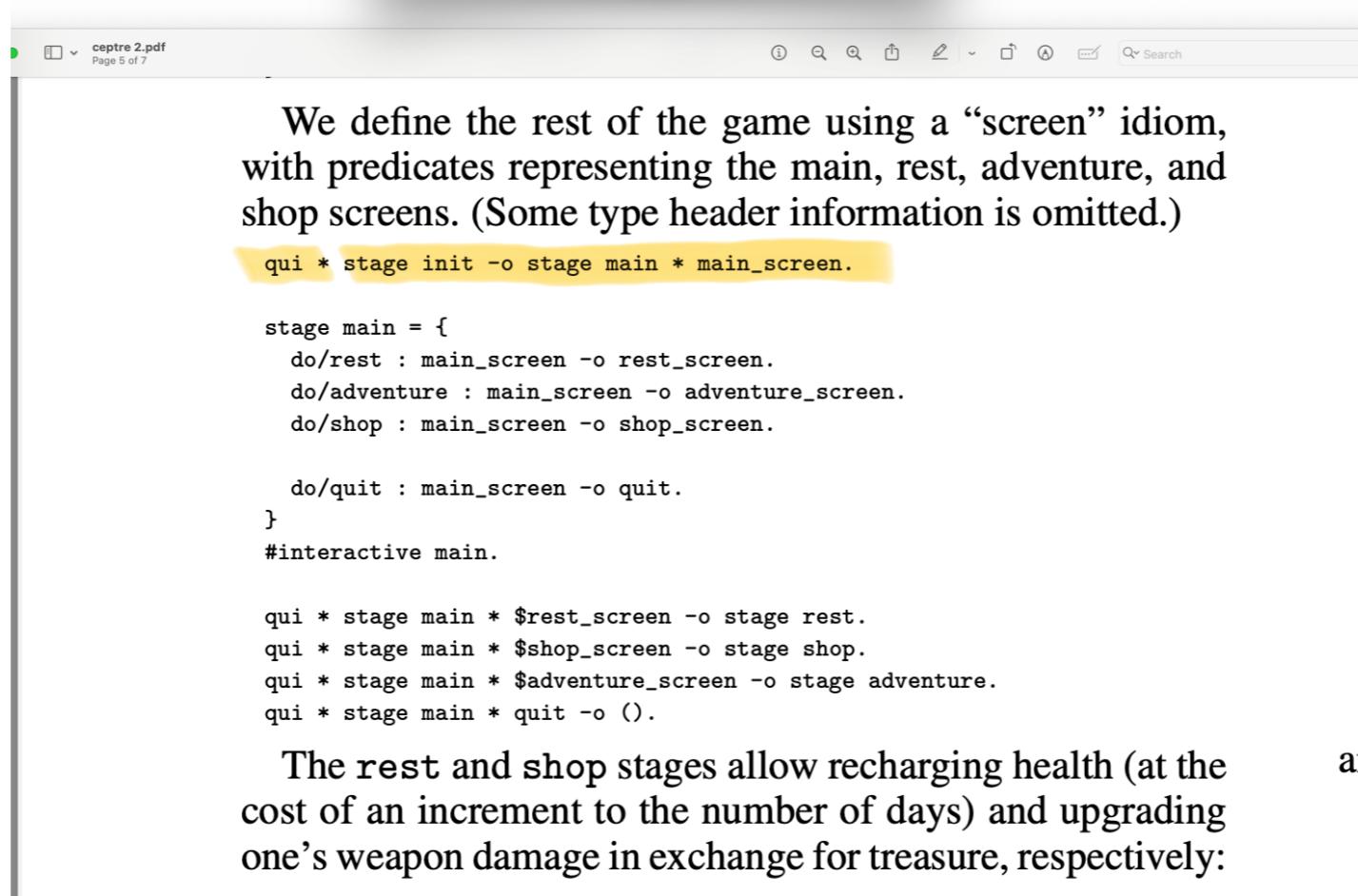




When no more rules can be enabled
(matched), insert *qui* into the factbase.



First Transition



The **rest** and **shop** stages allow recharging health (at the cost of an increment to the number of days) and upgrading one's weapon damage in exchange for treasure, respectively:

Caption

IF state (*init*) IS quiescent then {

 Remove stage (*init*) from FB

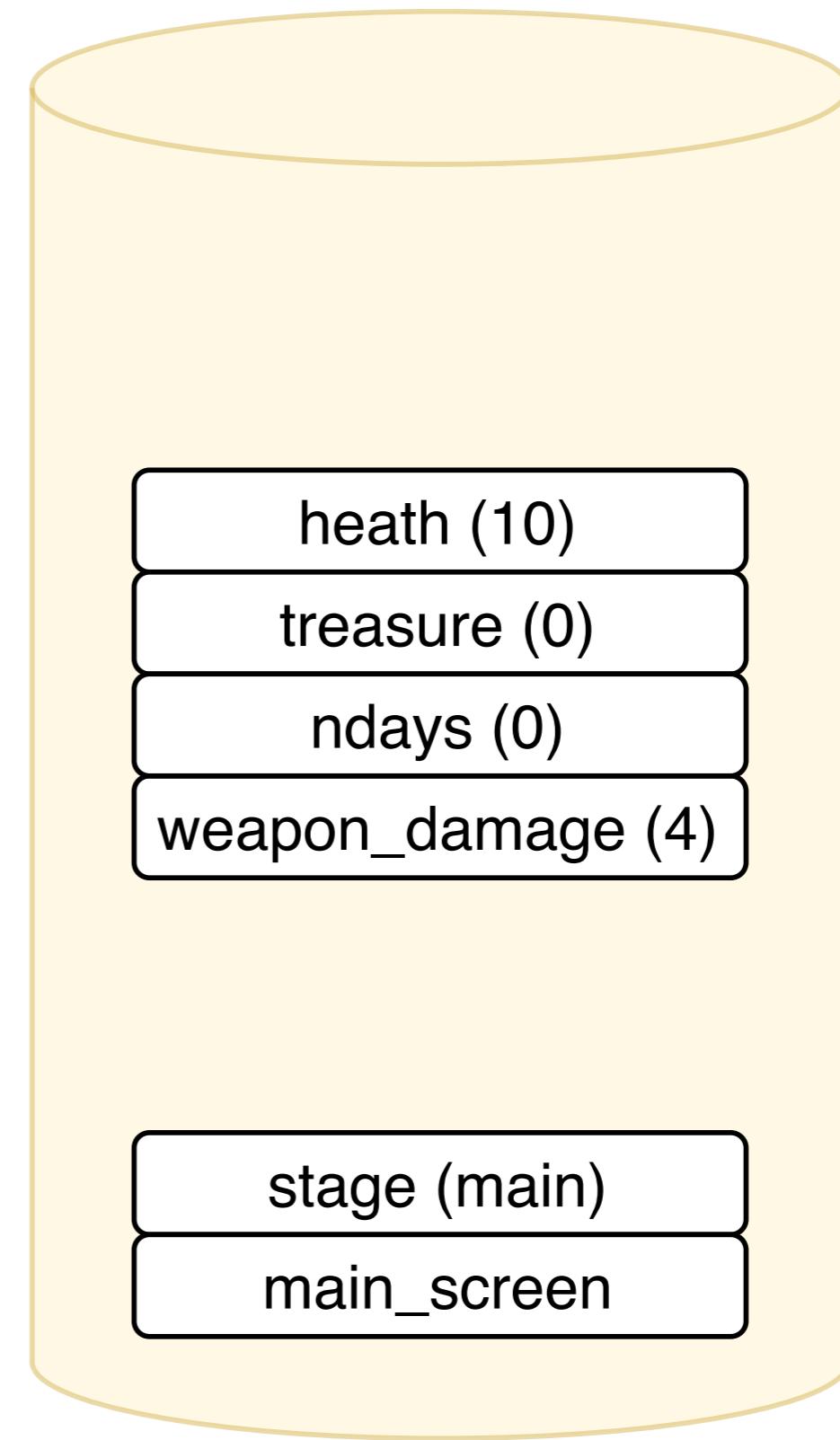
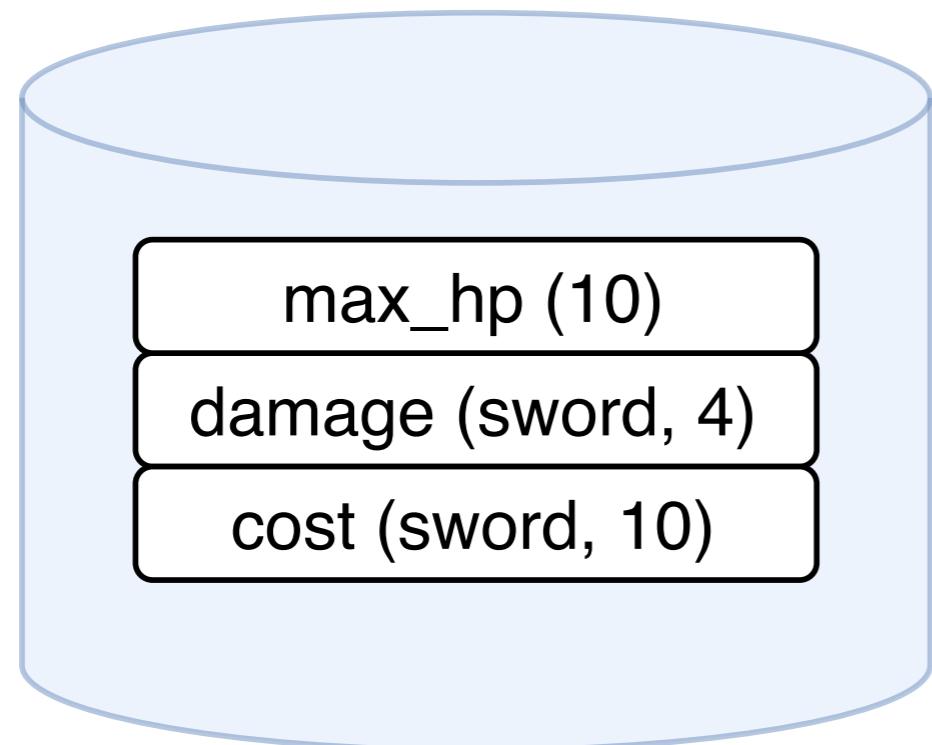
 Remove *qui* from FB

 AND

 Insert stage (*main*) into FB

 Insert *main_screen ()* into FB

}



Caption

GAME DESIGN INTENT

The game used in this example is quite simple, as far as games are concerned. The intent behind this example is to show various techniques for designing games..

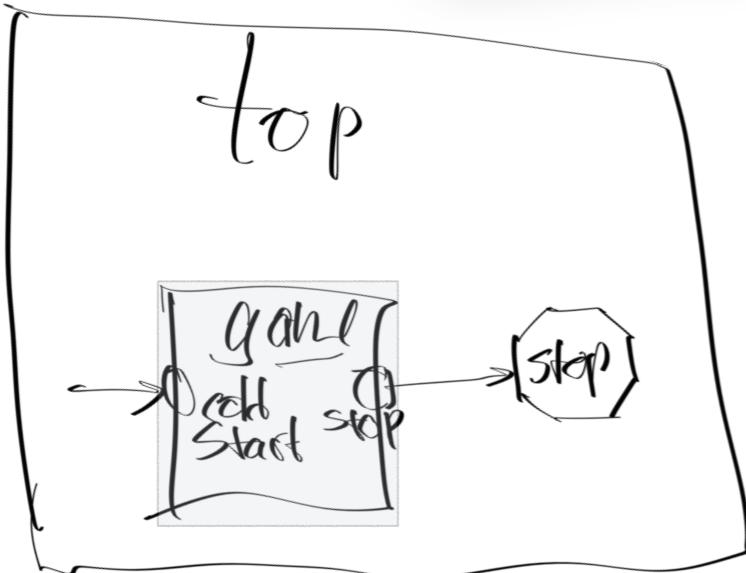
The following pages are freehand sketches that give my best understanding of the design of the game. I choose to explain the design using a set of layers.

Later, I will describe the Ceptre code in terms of these sketches.

Legend:

- p.ok is the player's health, when it drops to 0, the player dies and the game can be restarted afresh, referred to as the predicate 'health N' in the Ceptre code
- p.\$ is the player's money, referred to as the predicate 'treasure N' in the Ceptre code
- p.pwr is the player's weapon's power, referred to as the predicate 'weapon_damage N' in the Ceptre code
- spoils is the \$'s awarded to the player for defeating a monster,
- ndays is a score - the "number of days" that have transpired in the game
- m.ok is the monster's health in a fight, when it drops to 0, the player wins the fight, referred to as the predicate 'health N' in the Ceptre code
- m.pwr is the amount of damage a monster can inflict in one blow to the player in a fight, referred to as the predicate 'monster Size' in the Ceptre code
- m.\$ is the amount rewarded to the player when the monster m has been vanquished, referred to as the predicate 'drop_amount Size Drop' in the Ceptre code
- z is zero (Church numerals are used in some of the bwd predicates in the larger rpg.cep code).

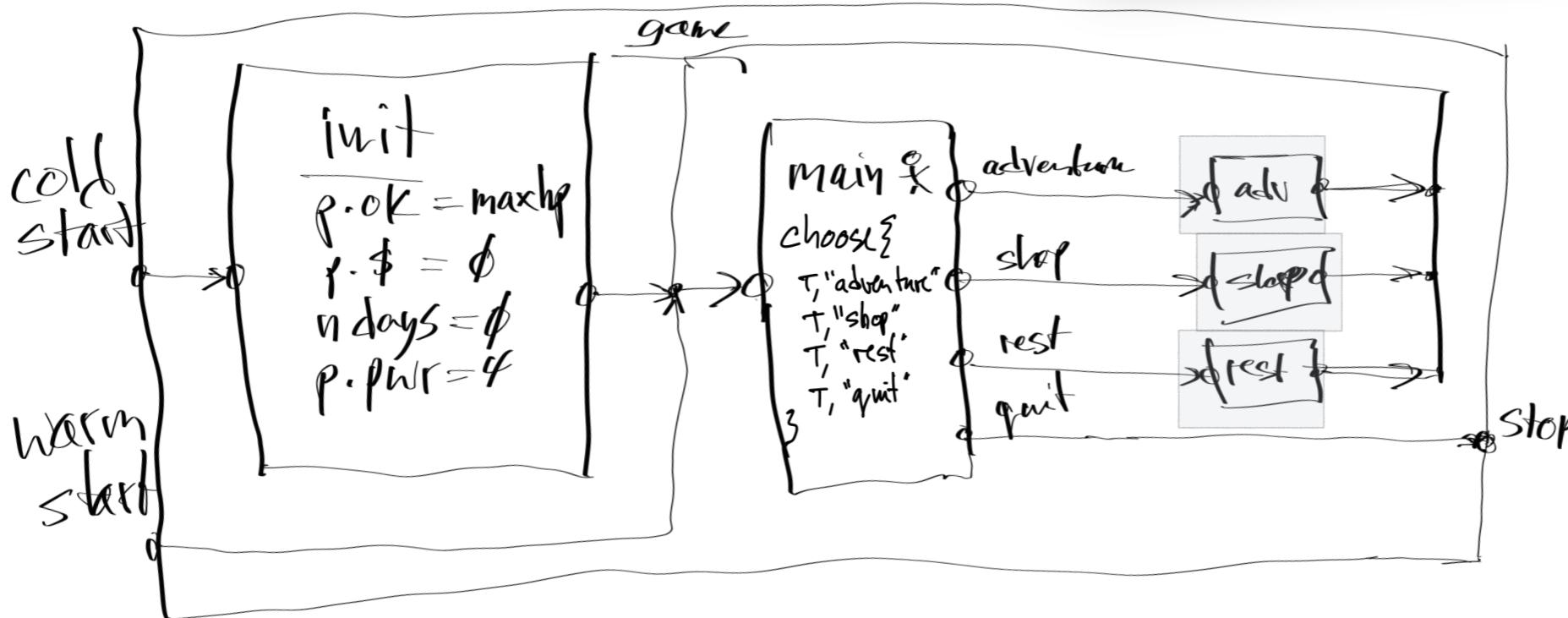
Layers Init and Main



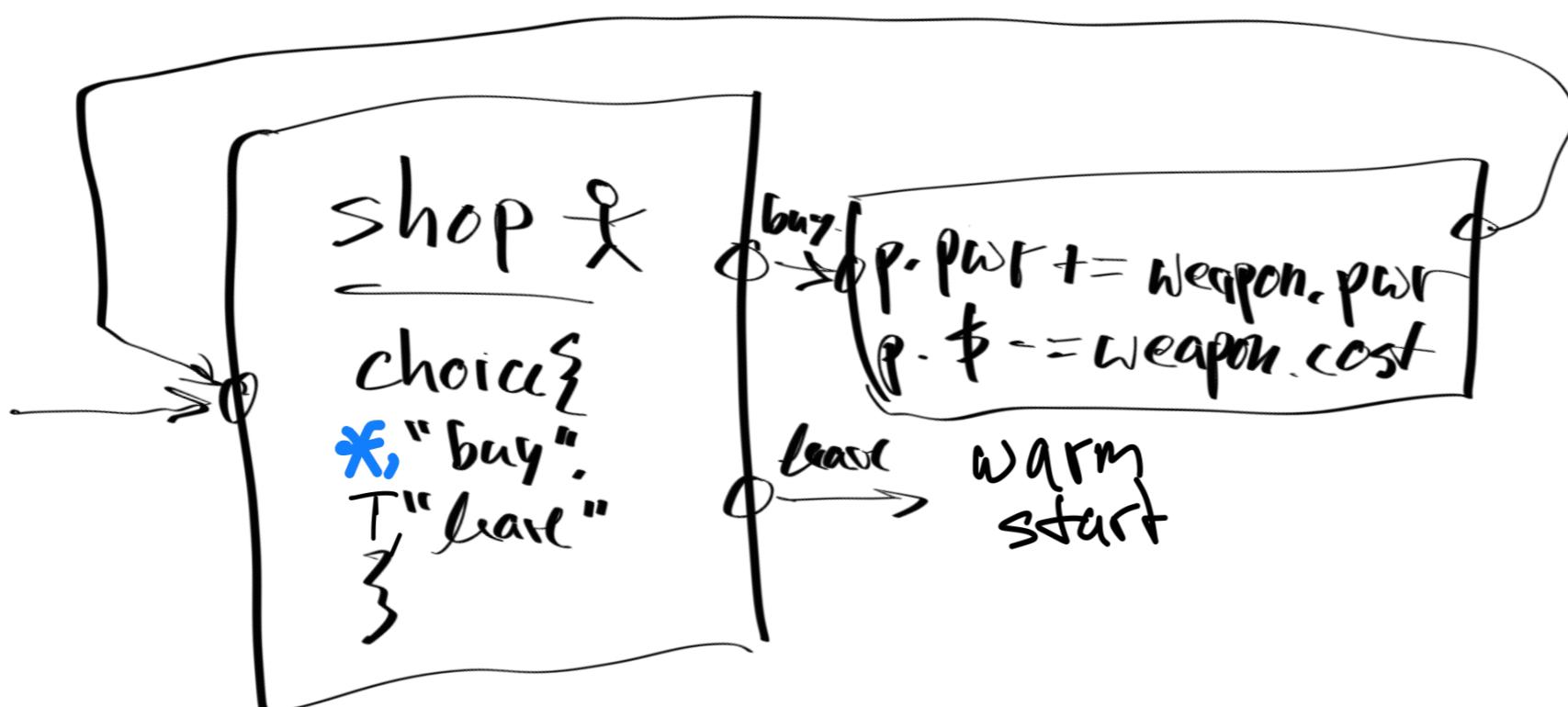
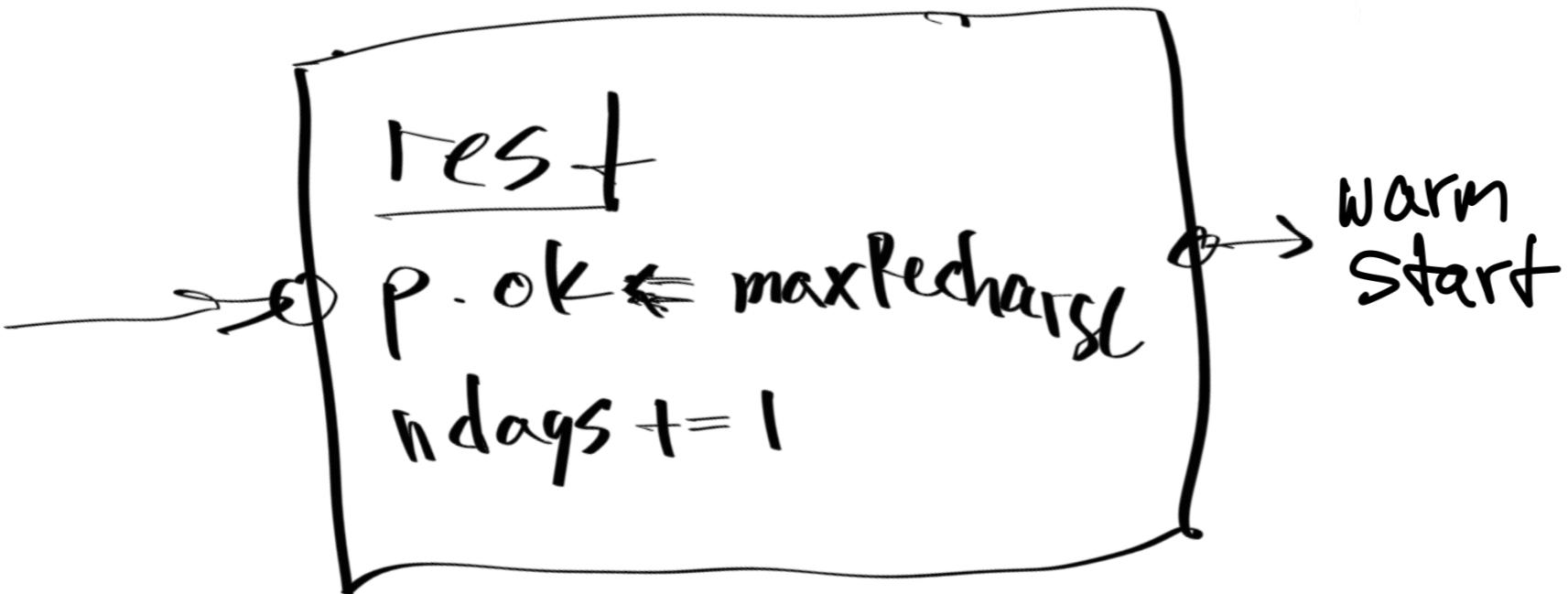
The game begins with a cold start - some variables are initialized, then the main loop is entered.

The player must choose from 4 courses of action "adventure", "shop", "rest", "quit".

When the player wins or dies, the game begins again with a cold start, and, in all other cases, the loop repeats without reinitializing the top-level variables.



Layers Rest and Shop

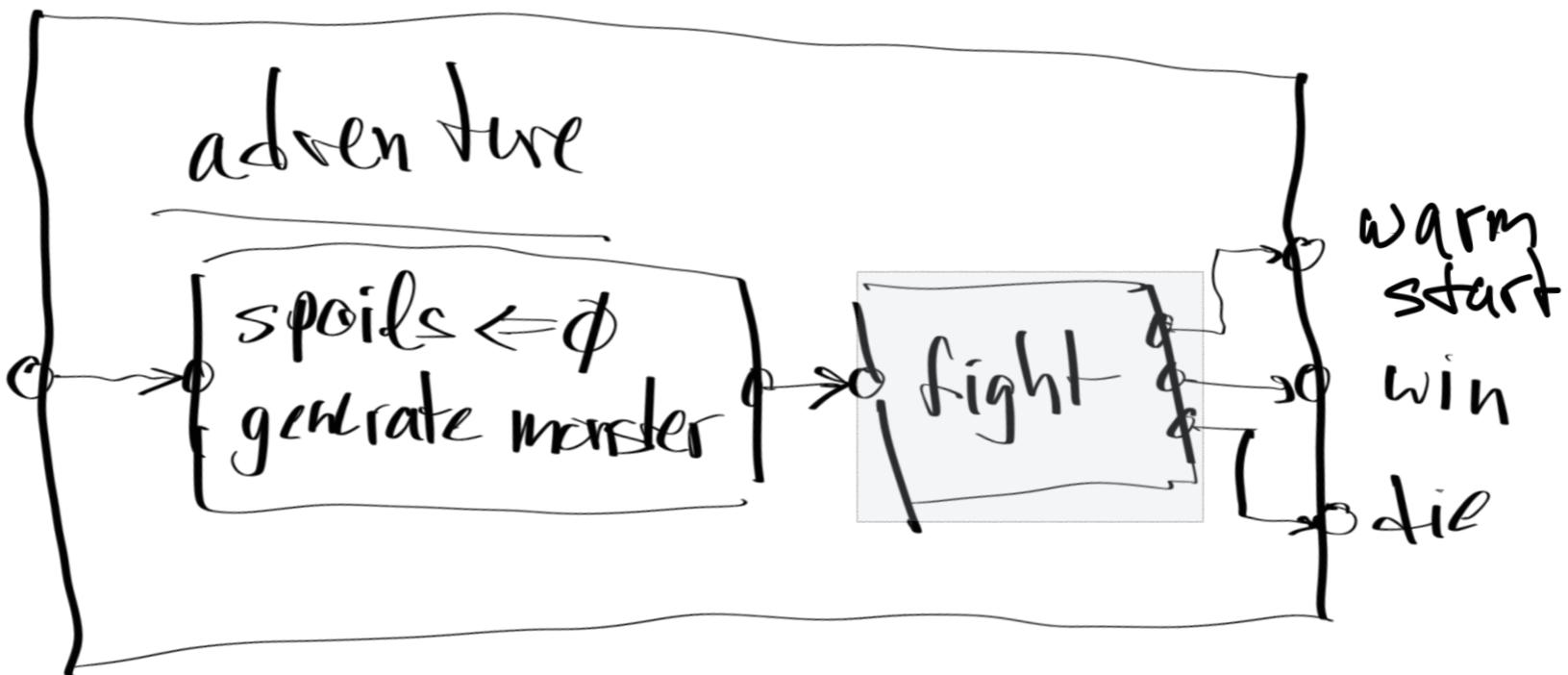


If the player chooses "rest", the player's health is restored to a maximum value, but, 1 day is lost.

If the player chooses "shop", the player is given the option to exchange some \$'s for more powerful weapons, if the player has enough \$. The player can leave the shop at any time.

* $P.\$ \geq \text{weapon.cost}$

Layer Adventure



Adventure kicks off a round of fighting. The amount of \$'s to be collected when the player wins a battle, is reset and a new monster is generated, then the fight loop is entered.

The adventure can end in one of 3 ways

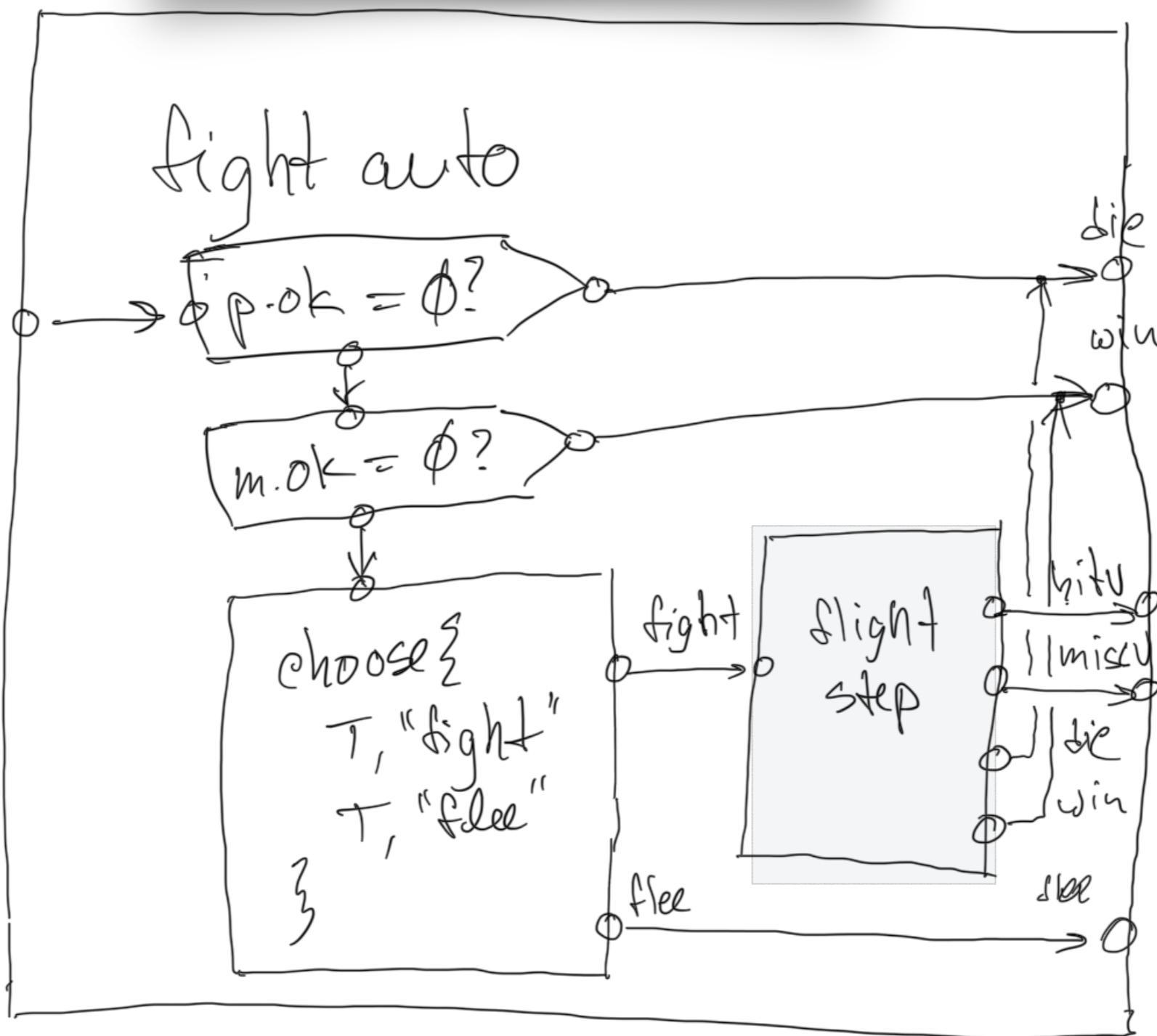
- The player wins the game (all monsters have been vanquished)
- The player loses the game ('dies')
- The player flees from the current battle and foregoes all winnings in a given round of fighting, the game continues with a warm start.

Layer Fight



Fight runs the fight loop by repeatedly calling invoking "fight auto" and performing health decrements as required.

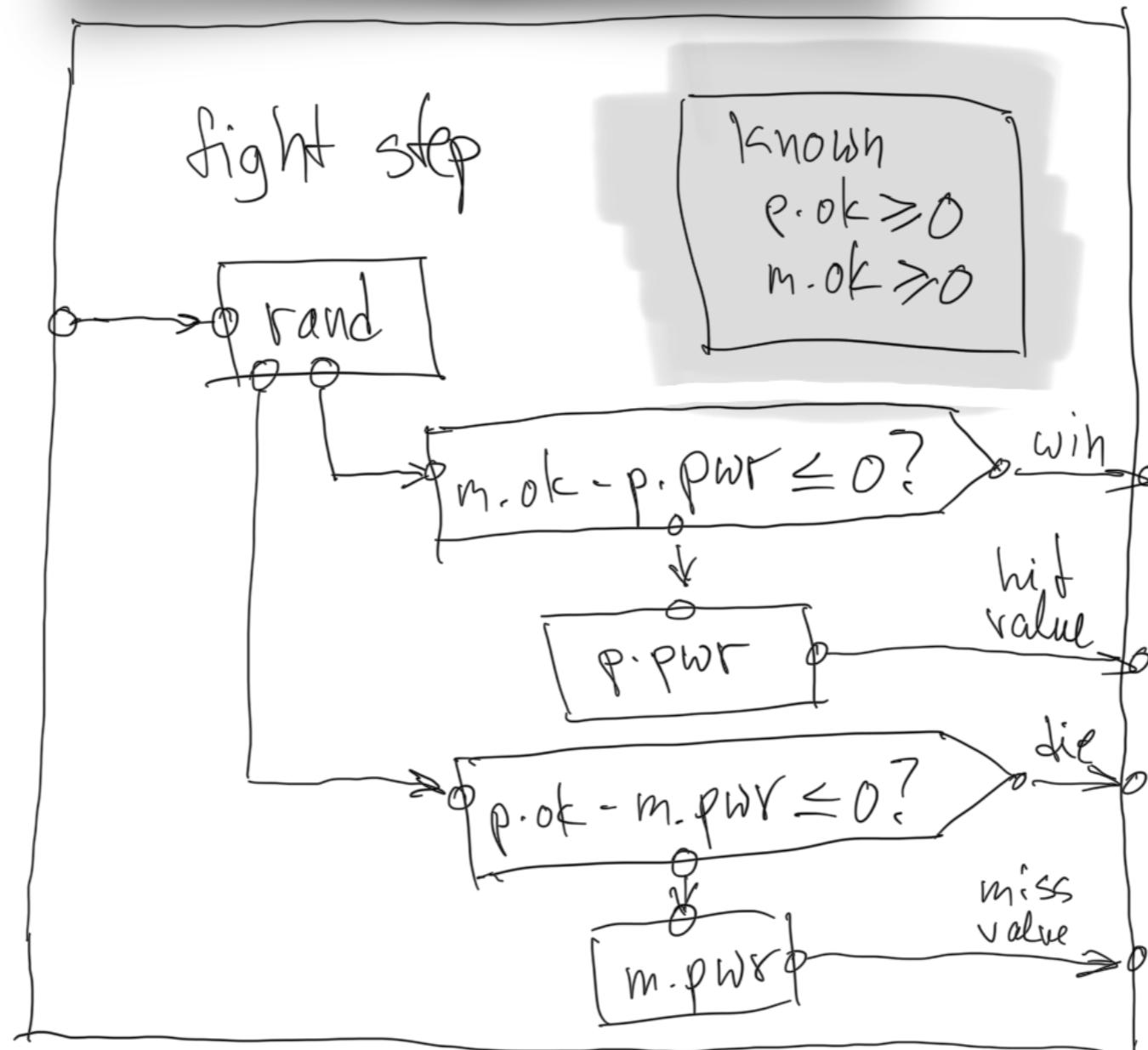
Layer Fight Auto



Fight auto runs the fight loop.

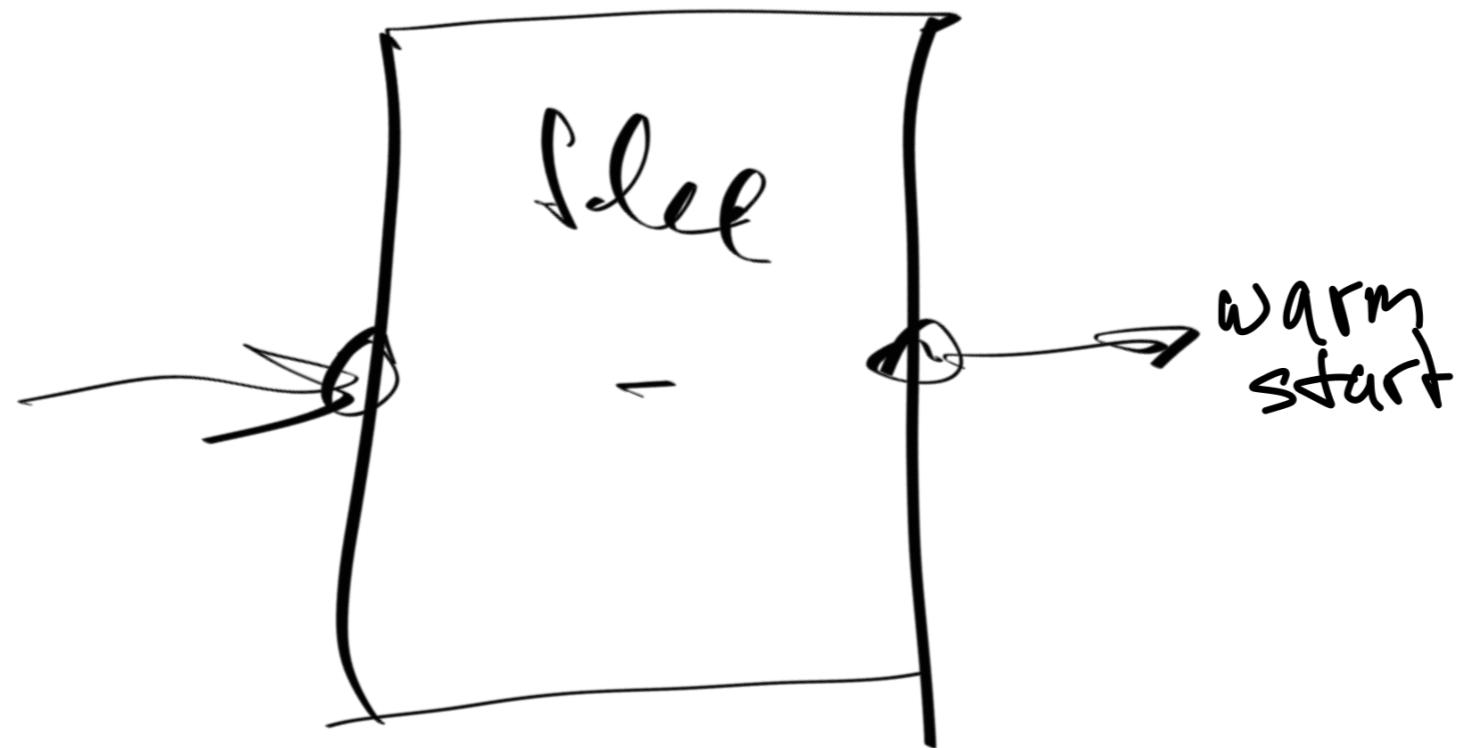
Fight auto returns 1 of 5 possibilities: a hit value, a miss value, a die event, a win event, a flee event.

Layer Fight Step



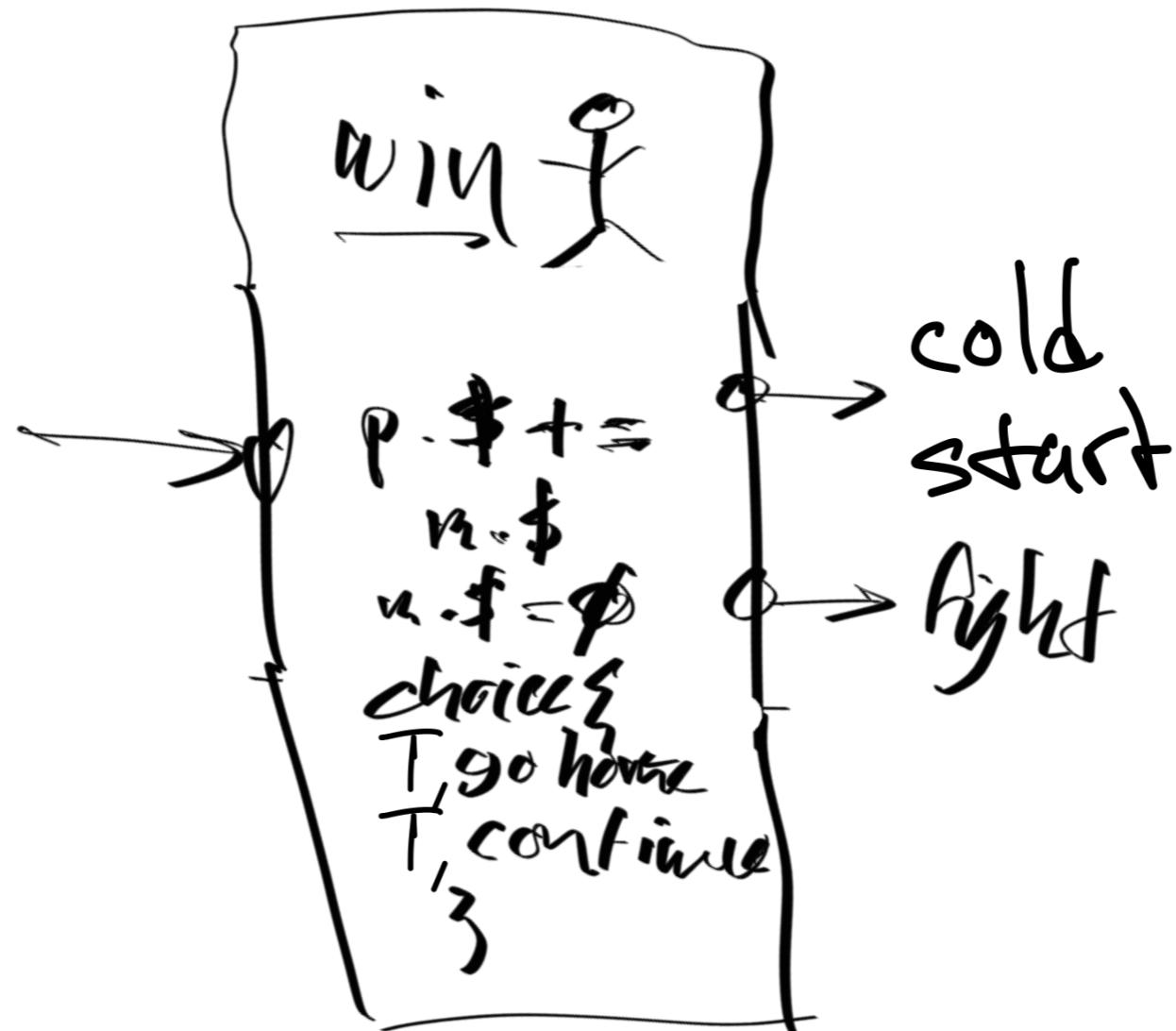
Fight step picks a hit or a miss at random, then produces a "win" or "hit value" event, or, produces a "die" or "miss value" event.

Layer Flee



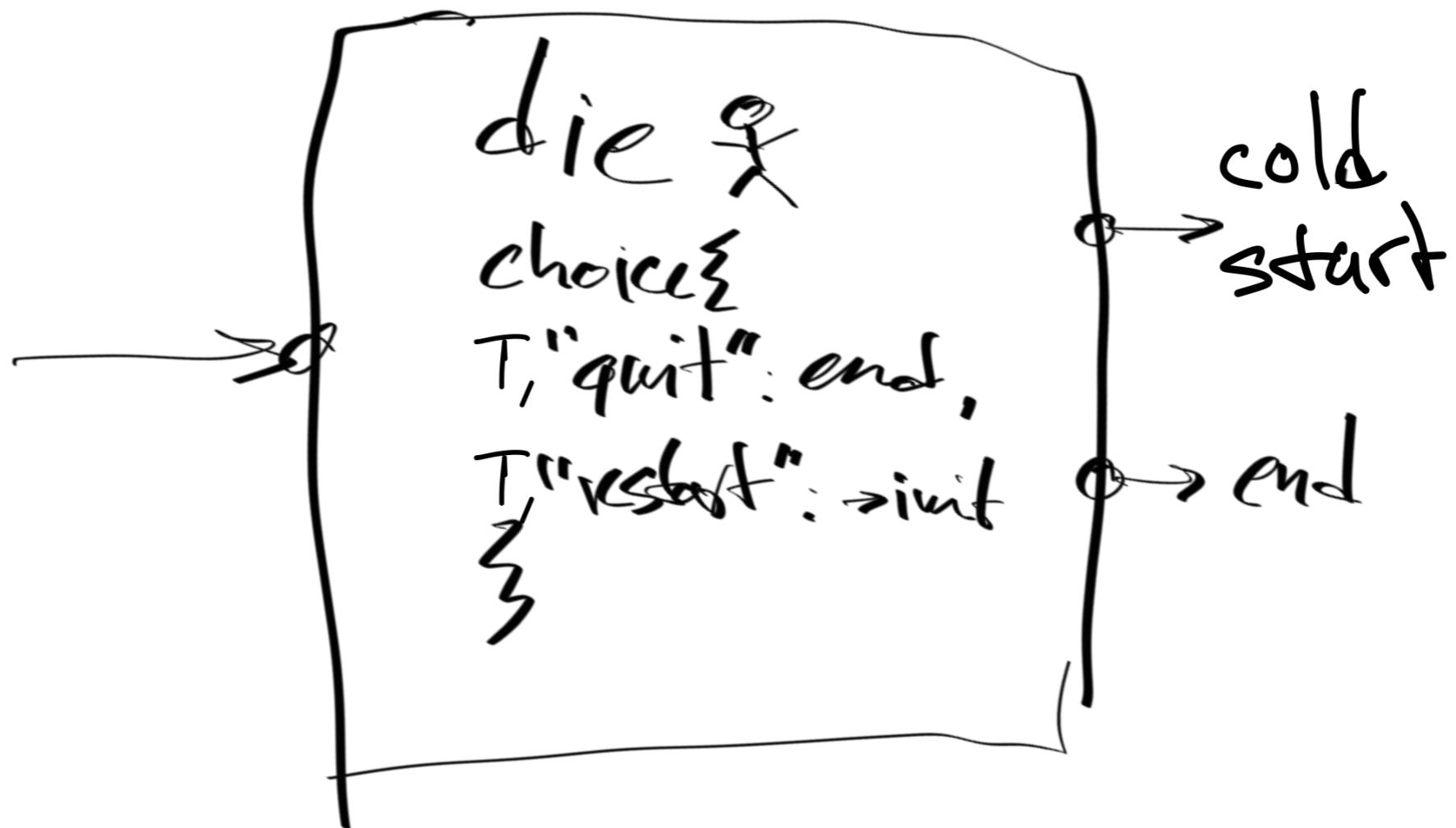
The player foregoes all \$'s dropped by the monster and goes back to the main loop.

Layer Win



The player has slayed one monster and the player is given a choice of going back to the main loop or fighting another monster.

Layer Die

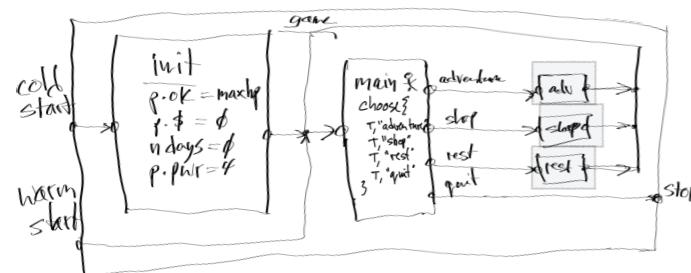
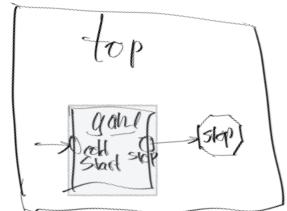


The player lost the game (player's health has dropped to 0). The player can play again or quit the game.

CEPTRE CODE

The following section examines the actual code presented in the paper and compares it with the Design Intent.

Code Layers Init and Main



```
qui * stage init -o stage main * main_screen.  
  
stage main = {  
    do/rest : main_screen -o rest_screen.  
    do/adventure : main_screen -o adventure_screen.  
    do/shop : main_screen -o shop_screen.  
  
    do/quit : main_screen -o quit.  
}  
#interactive main.
```

In the main loop of the game, four rules are (always) enabled, since each rule consumes a `main_screen` predicate.

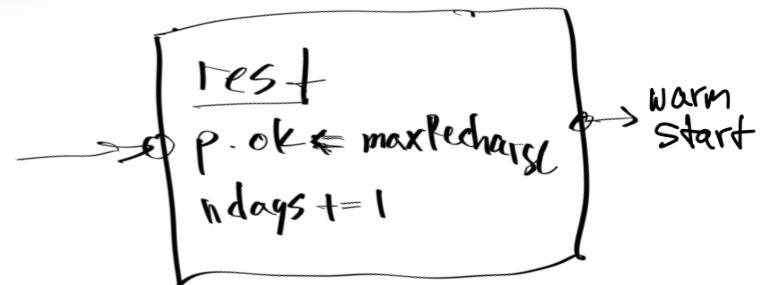
The player is asked to pick one of the rules.

The rules simply drop “screen” predicates into the FB (factbase). The stage becomes quiescent. Upper-level logic then determines which transition to take.

Code Layer Rest Stage

In stage *rest*, player's health recharges, but, ndays is incremented

```
stage rest = {
    recharge : rest_screen
        * health HP * max_hp Max * recharge_hp Recharge
        * cplus HP Recharge Max N
        * ndays NDAYS
    -o health N * ndays (NDAYS + 1).
}
qui * stage rest -o stage main * main_screen.
```



Logic Variables

HP
Max
Recharge
N
NDAYS

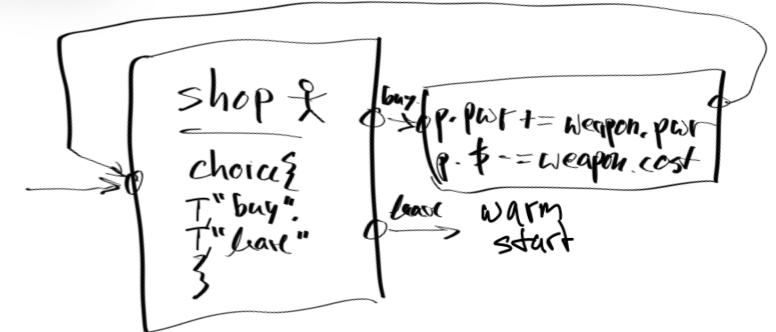
N := max (Max, HP + Recharge)
FB += health (N)
FB += ndays (NDAYS + 1)

backward-chaining predicates. For instance, we can define the arithmetic operation of addition capped at a certain maximum value as a predicate **cplus A B Cap C** which can be read as **A plus B capped at Cap is C**. We omit the definition of the predicate for brevity, but make use of it in later rules. We also use backward-chaining predicates to define a few con-

Cplus predicate

Code Layer Shop Stage

In stage *shop*, the player can buy more weapon power, if the player has enough \$.



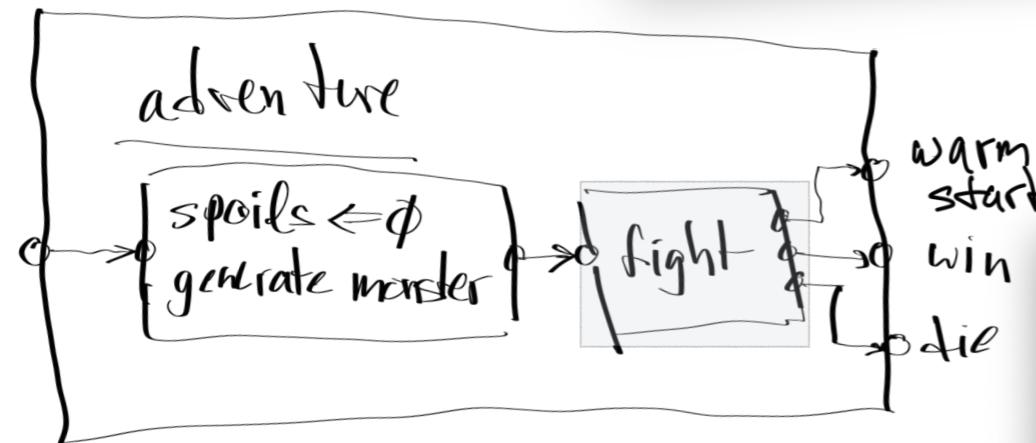
```
stage shop = {
    leave : shop_screen -o main_screen.
    buy : treasure T * cost W C * damage_of W D * weapon_damage _ 
        * subtract T C (some T')
        -o treasure T' * weapon_damage D.
}
#interactive shop.
qui * stage shop * $main_screen -o stage main.
```

```
let T = p.$ in
let W = any weapon (retryable)
let C = cost of the given weapon
let D = power of the given weapon
If a weapon can be found that satisfies  $T - C \geq 0$  then
enable {
    prompt "buy"
    remove treasure predicate with value T
    remove cost predicate with values W and C
    remove damage_of predicate with values W and D
    remove weapon_damage predicate without regard for its value
    insert treasure T' where  $T' = T - C$ 
    insert weapon_damage D (effectively overriding previous weapon_damage)
}
```

Definition of subtract {
 $T - C = +ve\ integer\ or\ 0$
 $T - C = \text{none}\ otherwise$
}

The rule 'buy' is enabled only if
the player has enough \$, ie.
 $(\text{some } T')$ is not none.

Code Layer Adventure



```
stage adventure = {
    init : adventure_screen -o spoils z.
}

qui * stage adventure -o stage fight_init * fight_screen.
```

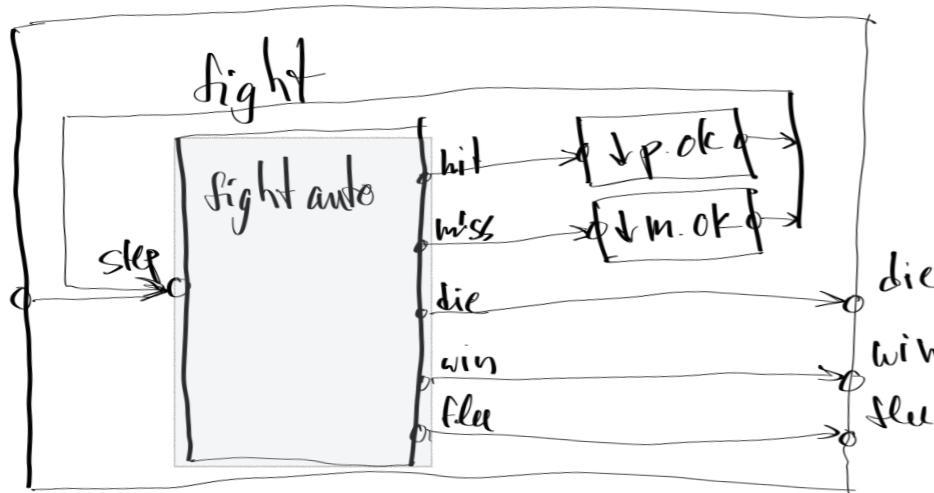
```
stage fight_init = {
    init : fight_screen -o gen_monster * fight_in_progress.
    gen_a_monster : gen_monster * monster_size Size
        -o monster Size * monster_hp Size.
}
```

The predicate `spoils` is initialized to 0 (z).

A monster is generated and inserted into the FB.

Then, we enter the fight loop .

Code Layer Fight Loop



```
stage fight_auto = {
    fight/hit
        : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
            * subtract MHP D (some MHP') -o monster_hp MHP'.
    win
        : fight_in_progress * monster_hp MHP * $weapon_damage D
            * subtract MHP D none -o win_screen.
    fight/miss
        : try_fight * $fight_in_progress * $monster Size * health HP
            * subtract HP Size (some HP') -o health HP'.
    die_from_damages
        : health z * fight_in_progress -o die_screen.
    fight/die
        : try_fight * fight_in_progress * monster Size * health HP
            * subtract HP Size none -o die_screen.
}
```

```
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

stage fight =
    do_fight : choice * $fight_in_progress -o try_fight.
    do_flee  : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.
```

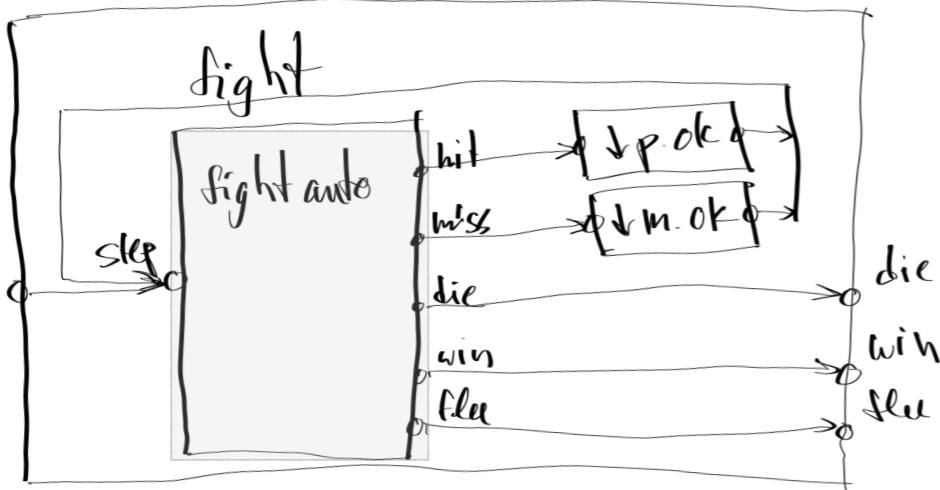
Reverse-engineering this code is overly difficult.

The code conflates several issues.

I've broken this down into 3 layers.

Structured programming emphasizes “narrow waist” and “narrow neck” (1 in, 1 out).

This code is like unstructured GOTO programming, spraying control-flow logic across several layers using “flags” (predicates).



Here, we have 1 message in, 1 reaction out. (N.B. not one *datum* out, but one *reaction* out. A reaction might be composed of 0 or more output events).

In the above sketch, it looks like “fight auto” has 5 outputs, but, only one of them fires in response to an input. In this case, one *reaction* is composed of one *event* (message).

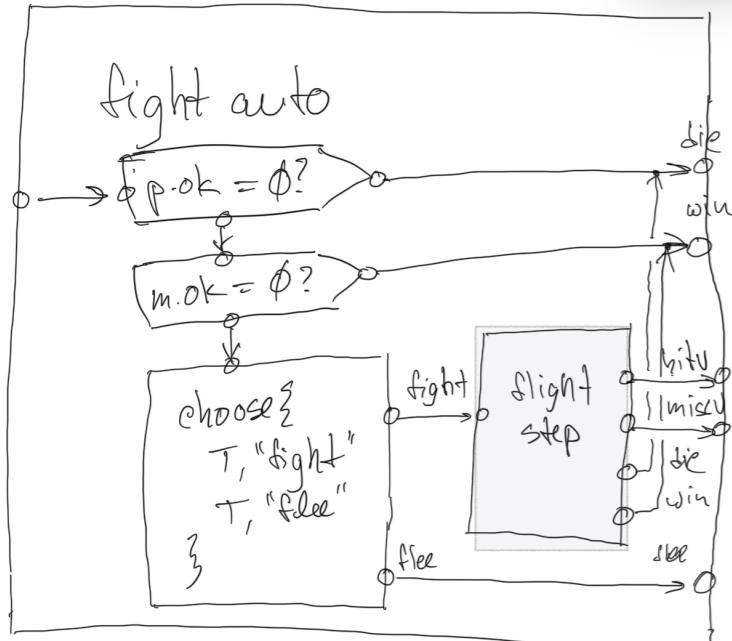
N.B. When 1 output fires, the other 4 outputs produce *nothing*. Not *nil*, not *false* - nothing, no event whatsoever.

When “hit” is fired, the monster’s health is reduced and we loop back for more fighting.

When “miss” is fired, the player’s health is reduced and we loop back for more fighting.

In the other 3 cases, the fighting loop is terminated and we restart (cold or warm appropriately).

Code Layer Fight Auto



```

stage fight_auto = {
    fight/hit
        : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D (some MHP') -o monster_hp MHP'.
    win
        : fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D none -o win_screen.
    fight/miss
        : try_fight * $fight_in_progress * $monster Size * health HP
        * subtract HP Size (some HP') -o health HP'.
    die_from_damages
        : health z * fight_in_progress -o die_screen.
    fight/die
        : try_fight * fight_in_progress * monster Size * health HP
        * subtract HP Size none -o die_screen.
}

```

```

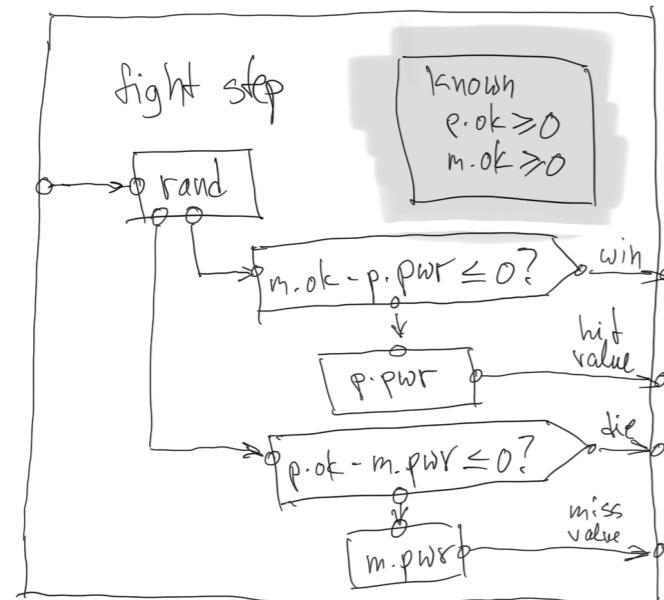
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

stage fight = {
    do_fight : choice * $fight_in_progress -o try_fight.
    do_flee  : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.

```

"Fight auto" weeds out the low-hanging fruit - player dead, monster dead, then asks the user how to proceed and punts to "fight step".

Code Layer Fight Step



stage fight_auto = {
 fight/hit
 : try_fight * \$fight_in_progress * monster_hp MHP * \$weapon_damage D
 * subtract MHP D (some MHP') -o monster_hp MHP'.
 win
 : fight_in_progress * monster_hp MHP * \$weapon_damage D
 * subtract MHP D none -o win_screen.
 fight/miss
 : try_fight * \$fight_in_progress * \$monster Size * health HP
 * subtract HP Size (some HP') -o health HP'.
 die_from_damages
 : health z * fight_in_progress -o die_screen.
 fight/die
 : try_fight * fight_in_progress * monster Size * health HP
 * subtract HP Size none -o die_screen.
}

qui * stage fight_auto * \$fight_in_progress -o stage fight * choice.
qui * stage fight_auto * \$win_screen -o stage win.
qui * stage fight_auto * \$die_screen -o stage die.

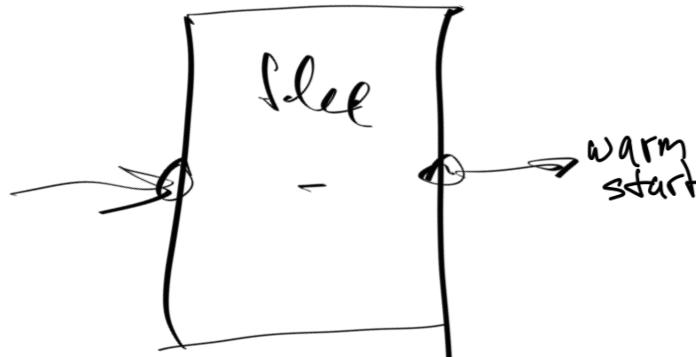
```

stage fight = {
  do_fight : choice * $fight_in_progress -o try_fight.
  do_flee   : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.
  
```

"Fight step" figures out one step in the fight loop.

We pick a hit or a miss at random, then determine if this hit kills the monster (player wins), or just weakens the monster, or, if the miss kills the player or just weakens the player.

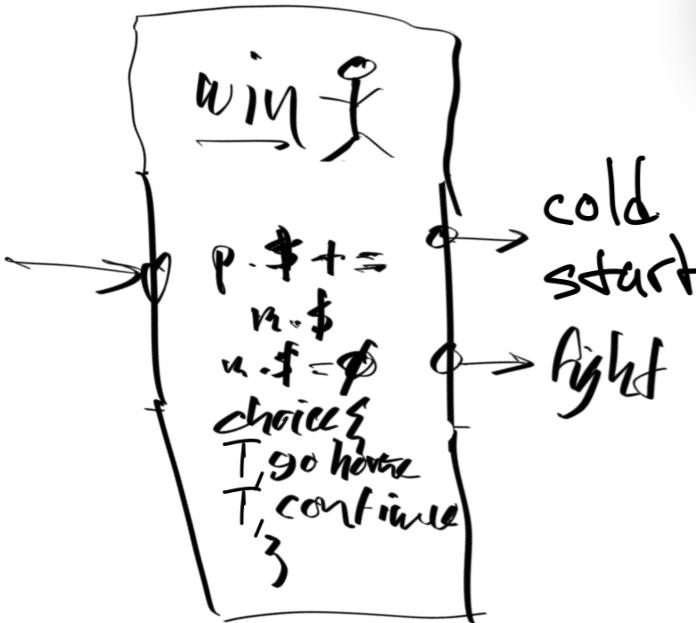
Code Layer Flee



stage flee = {
 % lose spoils
 do/flee : flee_screen * spoils X * monster _ * monster_hp _
 -o () .
 }
 qui * stage flee -o stage main * main_screen.

If the player flees the fight without conquering the monster, all spoils are removed, and, the monster is removed before going back to the main loop.

Code Layer Win

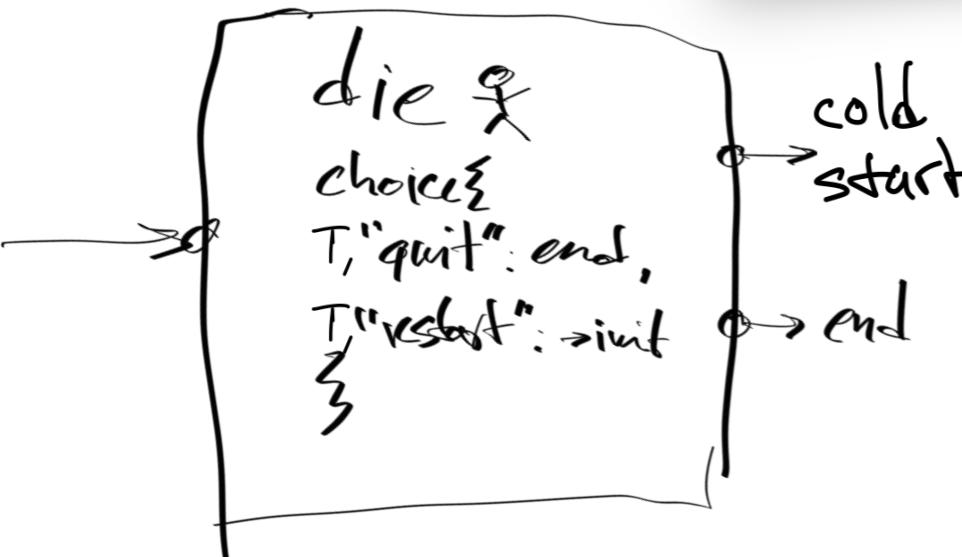


```
go_home_or_continue : pred.  
stage win = {  
    win : win_screen * monster Size * drop_amount Size Drop  
        -o drop Drop.  
    collect_spoils : drop X * spoils Y * plus X Y Z  
        -o spoils Z * go_home_or_continue.  
    go_home : go_home_or_continue  
        * spoils X * treasure Y * plus X Y Z  
        -o treasure Z * main_screen.  
    continue : go_home_or_continue -o fight_screen.  
}  
#interactive win.  
qui * stage win * $main_screen -o stage main.  
qui * stage win * $fight_screen -o stage fight_init.
```

The player is required to choose 3 times. (1) "win", (2) "collect_spoils", then (3) "go_home" or "continue".

If the player chooses "go_home", the player's \$ (treasure) is calculated and inserted into the FB before restarting.

Code Layer Die



ceptre 2.pdf
Page 5 of 7

```
stage die = {
    quit : die_screen -o end.
    restart : die_screen * monster_hp -
        * spoils _ * ndays _ * treasure _ 
        * weapon_damage _ -o init_tok.
}
#interactive die.
```

The player gets to choose "quit" or "restart" (both rules are always enabled at the same time)

If the player chooses "restart", then we delete several predicates - die-screen, monster_hp, spoils, ndays, treasure, weapon_damage - and do a cold start.

paultarvydas@gmail.com

<https://discord.gg/TnzEtPeAzN> (Programming Simplicity Discord (everyone welcome))

blog: <https://guitarvydas.github.io/>

blog (2022-2023): <https://publish.obsidian.md/programmingsimplicity/>