

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
cmartens@cs.cmu.edu

18.500

We present a simple language, called *Cognis*, to enable rapid prototyping of reasoning systems. The language is designed to be used in combination with a graphical interface for understanding programs based on image sequences. It is also designed to be used as a programming language for specifying programs and proofs. In this paper we introduce the language and its basic features. In Cognis, we introduce the ability to define and reason about sets of objects and relations. We show that the language can support pure deductive reasoning. We also show how Cognis can be used for the construction of rule-based, probabilistic, and hybrid systems. Cognis is a graphical language, one from which it is easy to generate programs for a variety of applications.

Introduction

Today game designers and developers have a wealth of tools available for creating executable prototypes. Prototype engines such as Unity, Unreal, Pencilsoft, and Flash have built-in support for prototyping and testing all these subfunctions of games. On the other hand, to invest in investing, novel mechanics in any of these tools typically requires significant expertise. In the case of Unity, for example, state registries dip into JavaScript, adding a language such as C# or JavaScript, learning how to use them, and then learning how to use them, and low-level programming languages, and Pencilsoft simply prevents the author from going outside the well-specified 3D grid system.

The concern of operational logic (Meadow and Witten 2006), the underlying substrate of meaningful character interaction, is often overlooked. This concern was present on a number of critical commentaries of game design and analysis. In most prototyping tools, there is a

1

2

3

4

South 2012), which shares with our approach a set of logic programs as its foundations. In their setting, predicate names are treated as zero-level objects (i.e., in the indices of predicate names) while their meanings are left undefined, rather than being directly associated with them. This treatment separates the semantics of the logic programs itself from the semantics of the formal analysis in disjoint from the predicate symbols (although they are connected semantically). We do not propose that this is the best way to handle the semantics of logic programs, but part the flexible handling of domains, and we perceive some merit in the approach taken by South et al. (2012), which can be directly mapped onto general operational semantics.

Finally, we have also been inspired by the work of Koenig and Lutz (2009) and others who have proposed general purpose logics for knowledge representation and reasoning. These attract us to our goal of striking a balance between expressiveness and tractability. The approach of Koenig and Lutz, inspired by many world and e-damages creator-inspired games, is based on a logic of action and observation, and it is not clear how one may selectively add more expressive components. We have carried out several additional case studies with this tool, investigating Koenig's potential for inventing new operational logics, and we will report on these results elsewhere. We also note that the language contributions of this paper from the game theory community are not limited to the logic of action and observation.

Given this, we are also eager the expand the range of analytical tools for Capricorn. Specifically, we have in progress several extensions to the system, including support for more complex predicates, and tools for visualizing and manipulating logically constrained structures. These tools will be used in future work to support the development of new games.

Capricorn could lead to novel approaches to expressive range of logic programs as well.

Linear logic has been used independently to study games, especially in the context of strategic domains (e.g., Benthem 2002; Benthem, Pothos, and Elmer 2005; Benthem, Carmona, and Chaudhuri 2010). For instance, Ding et al. used linear logic to model the notion of strategy in a game theory problem to validate the space of possible memory outcomes. Related frameworks, such as Point nets (Kumar and Rajan 2009) and the Mathematics of Games (Bentzen et al. 2011), have been used to similar extent. These investigations have shown that linear logic provides a useful framework for analyzing what a predicate like $\text{at}(x, t)$ must be instantiated at every character and location to make use of it in the model, and also for analyzing what constraints must be satisfied by strategies. Handling predicate rules requires an extension to the logic of action and observation, and this is something we suspect will be difficult to prove of soundness in the language, or applicable like exhaustion analysis are we able to do in Capricorn.

Several other systems have been proposed to address the problem of game analysis. One of the most prominent is the coding of game mechanics for analysis. Our work has very similar goals to those in this line of the literature (the paper presented to VCG 2012 is a good example of this work, see Benthem et al. 2013). But Capricorn is not limited to describing games based on a single type of game mechanics. Instead, Benthem, Carmona, and Mauw (2010), BOOG (Ewerhart 2005) and Ewerhart et al. (2010) are proposals for domain-specific languages for specifying game mechanics. These languages provide a logic-based foundation for the language design. Despite this, Capricorn is not limited to analyzing games based on a single type of game mechanics. Instead, it is designed to support them through such means as abstraction specifications and piecewise character grammars (South, Nelson, and Mauw 2009).

Acknowledgments

This work was partially funded by grant NFRP-08-007-1-100, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

Amirij, M., and M. Ragni. 2009. Matching games with Petty rules. *Braiding New Generations*. Conference on Games, Play, Fun, and Interactions. Springer, Berlin, Germany, 1–10.

Bauer, A., G. Corradi, M., and Chircagian, R. 2013. Linear logic for linear memory. *EURO-CLP*. Artificial Intelligence and Logic Programming. Springer, Berlin, Germany.

Champagne, R., Poggenpohl, A., and Fratioscini, P. 2005. Software engineering based on formal methods: a adaptive case studies. *IEEE TSE*, 31(1), 1–12.

Dong, E. D., Hoffmann, S., Champagne, B., and Spivack, G. 2011. How neural-fuzzy decision logic in the game planning process can support the player in a strategy board game. *AI&Society*, 25(1), 1–12.

Dosman, J. 2012. *Simulating mechanics to study memory games in games*. In *Artificial Intelligence in the Game Design Process*, 1–10.

Dosen, M., Levine, J., Larson, C., Schmid, T., Thompson, T., and Tschirhart, J. 2013. Towards a video game paradigm: two approaches to the design of *Computational Thinking*. *Games*, 4(5), 1–10.

Gardner, H. 1985. *Linear logic: Theoretical Computer Science* 30(1):1–10.

Hague, R., Howard, F., and Probin, G. 1993. A framework for building graphical games. *ACM SIGART Newsletter*, 1(1), 1–10.

Hartmann, K., Drak, A., Das, S., and Radl, M. G. 2011. Toward supporting novices with procedurally generated game worlds. *Proceedings of the Conference on Computer Games (CG)*. 2011 IEEE Conference on, 297–304. IEEE.

Hinde, J. L., and Miller, D. 2011. Logic programming as a language for game design. *Journal of Logic and Computation*, 21(2), 327–363.

Jarvenpaa, K., Kurki-Suonio, R., Heikkilä, M., and Syrja, K. 1996. *Computer game design: from system dynamics to Software Engineering*. 1996. Proceedings. 12th International Conference on Computer Games.

Krebs, Z. 1997. *Tetris: a play*.

Lavelle, S. 2013. *Pentangle puzzle engine*. <http://www.pentanglepuzzles.com/>.

MacLennan, B. B. 2011. The design of Kali: a very visual programming language for children on the XBox 360. *AI&Society*, 25(1), 11–20.

Marcus, C., Ferreira, F., and Bauer, A.-G. 2013. Generating story worlds in linear logic programs. In *Artificial Intelligence and Logic Programming*. Springer, Berlin, Germany.

Marcus, M., and Ferreira, A. 2013. *Playful Aspects in building a fully interactive dungeon*. In *Game Development and Design*, 1–10.

Marcus, M., and Waidipha Pras. 2008. *Defining operational logic*. Digital Game Research Association (DGR).

McLean, I., Thomas, M., Samuel, B., Turner, B., Marcus, M., and Waidipha Pras. 2010. *Competence in Use*: A fully model-based solution to software-on-paper perplexity. In *Proceedings of the Intelligent Software Technologies (IST) Workshop*, 55–56. ACM.

Mohlin, B., and Marcus, B. 2006. *Solid: a tool for automated generation of 3D environments*. In *Proceedings of the Interactive Digital Storytelling and Entertainment*. Springer, 139–150.

Monk, J. 2004. *The Joltone Designer's Manual*. Platonic Software, Inc.

Graves, J. 2006. *Rage Against the Machine*. *Journal of Video Game Theory and Scholarship* 7(1):72–82.

Hoban, J. C., Grove, A., and Marcus, M. 2013. *Mobile computation: cities for games*. In *ASIDE*.

Graves, J. C., Hoban, J. C., and Marcus, M. 2008. Applying planning to memory games: Narrative State Space. *ACM SIGART Newsletter*, 1(2), 1–10.

Schulte-Nolte, A., and Schummans, C. 2008. *Calib* – a logic framework for generating game environments (systems). In *Proceedings of the 2008 Conference on Automated Reasoning (CAFR'08)*, 520–535. Springer.

Smith, G., and Whistler, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Automated Content Generation in Games*, 6. ACM.

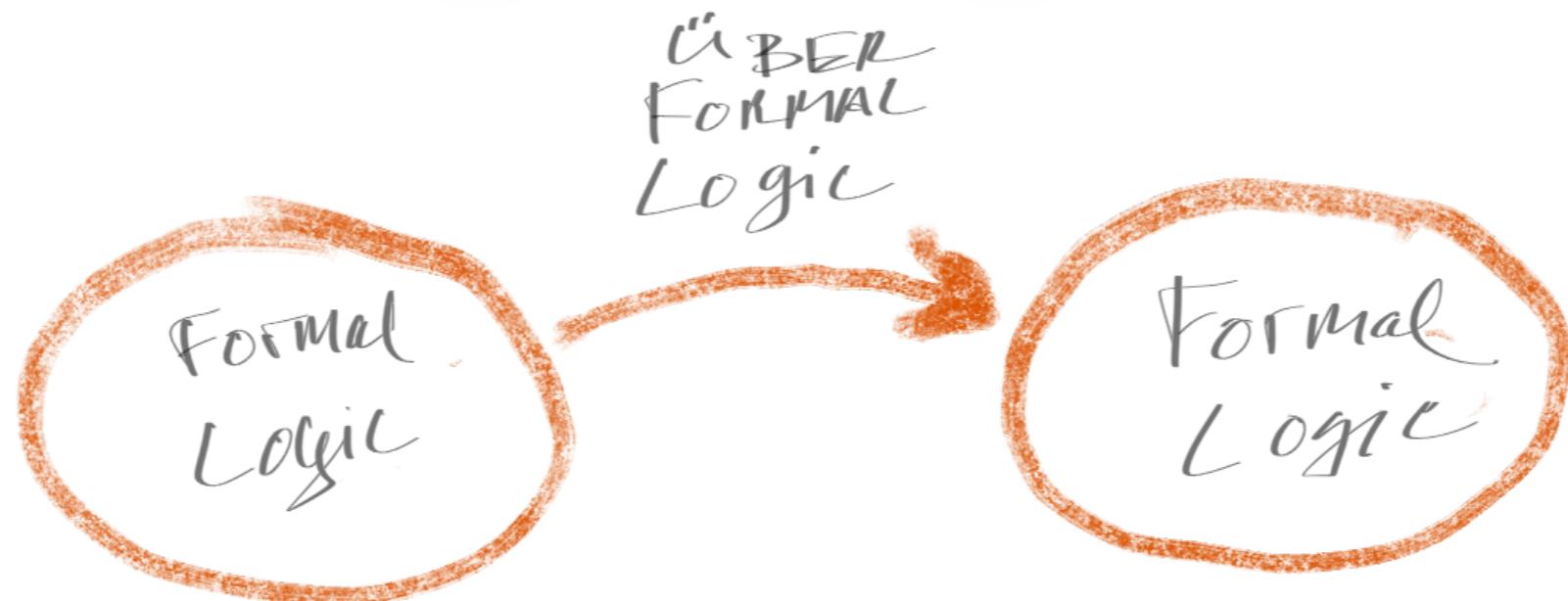
Smith, A. M., Nelson, M. I., and Marcus, M. 2009. *Artificial Intelligence for Game Design*. In *AGDF*.

Smith, A. M. 2012. *Modulating Egocentric Game Design*. Ph.D. Dissertation, University of Southern California, Los Angeles, CA.

Smith, and Bush, M. 2011. *Automated game design via mechanic generation*. In *Proceedings of the 2010 AAAI Conference on Artificial Intelligence*.

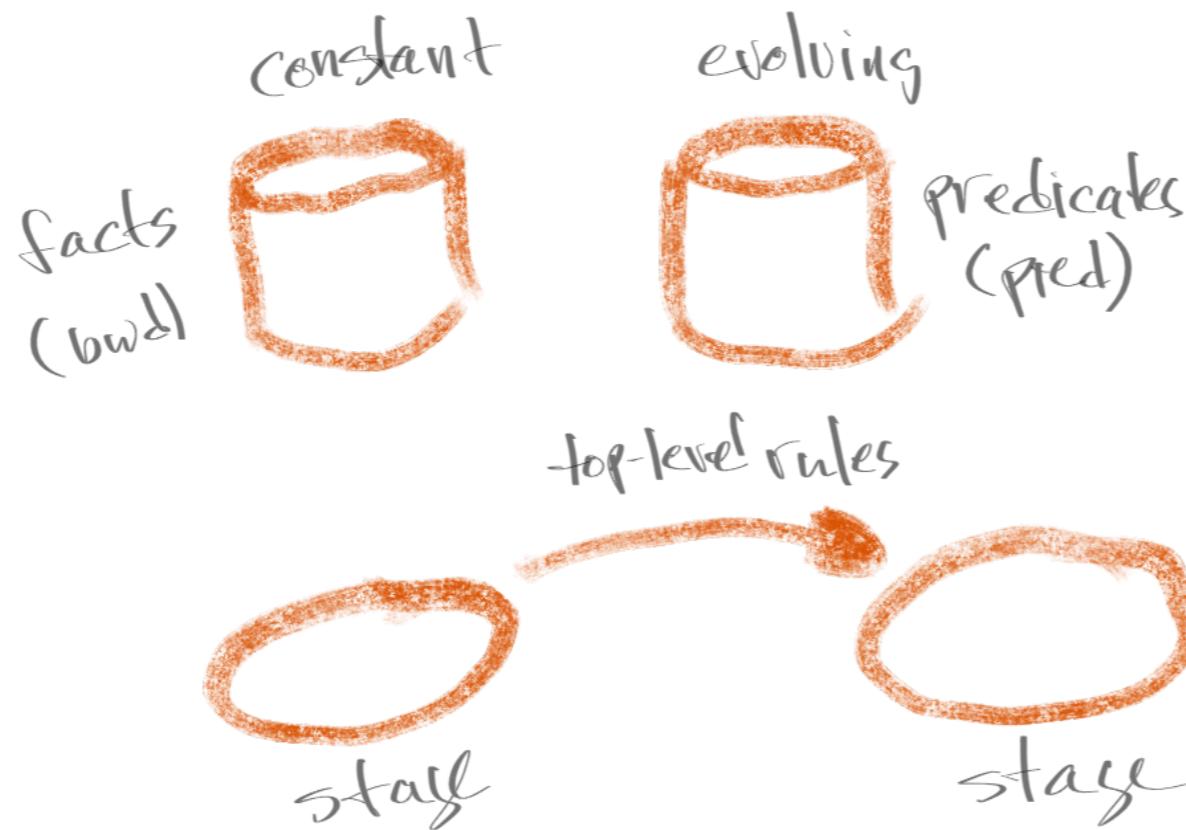
I view the code from the perspective of thinking of Ceptre as a language for writing practical gaming software, instead of as a way to formally analyze the required operations.

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:

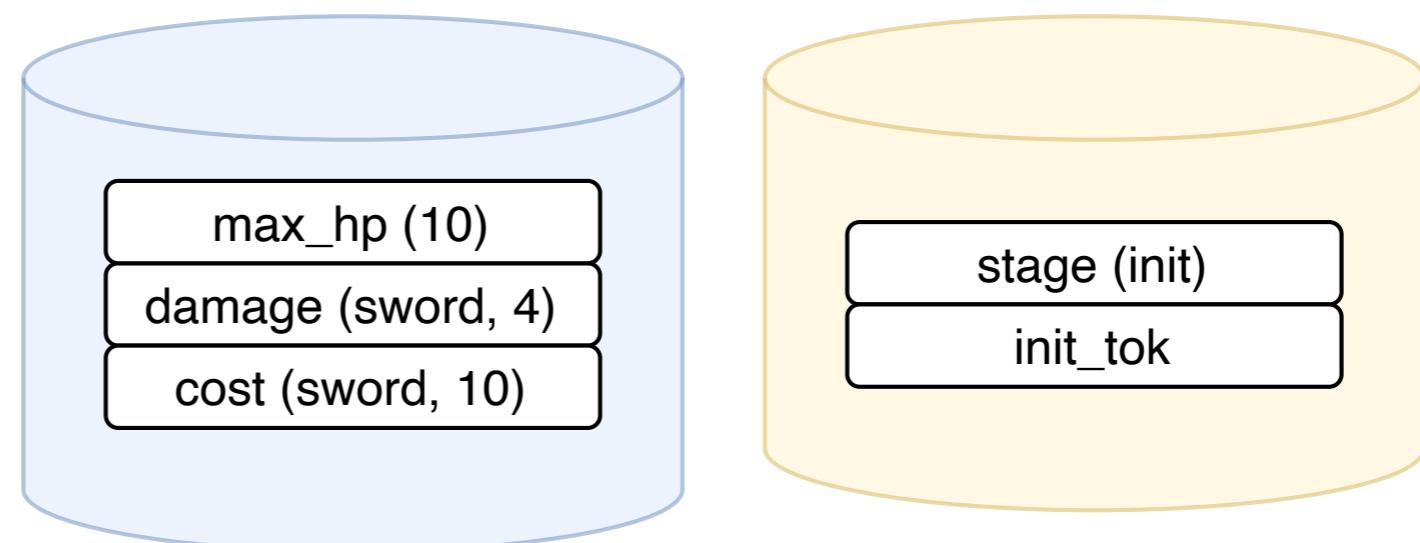
```
context init_ctxt = {init_tok}.

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 * 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}.

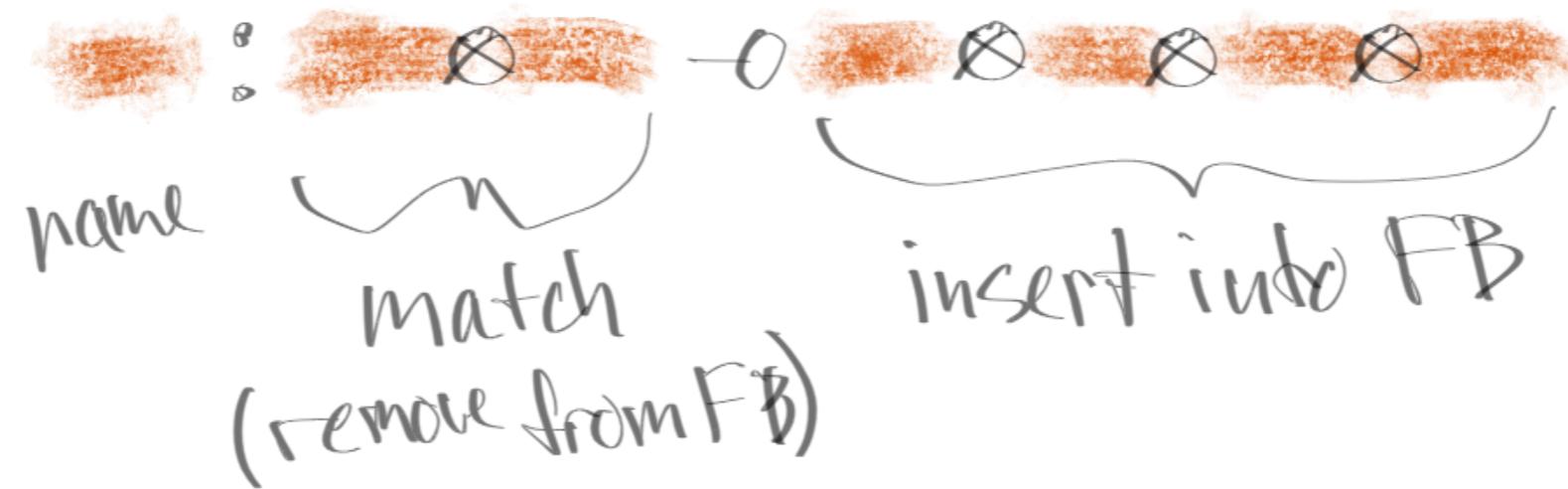
stage init = {
    i : init_tok * max_hp N
    -o health N * treasure z * ndays z * weapon_damage 4.
}
```

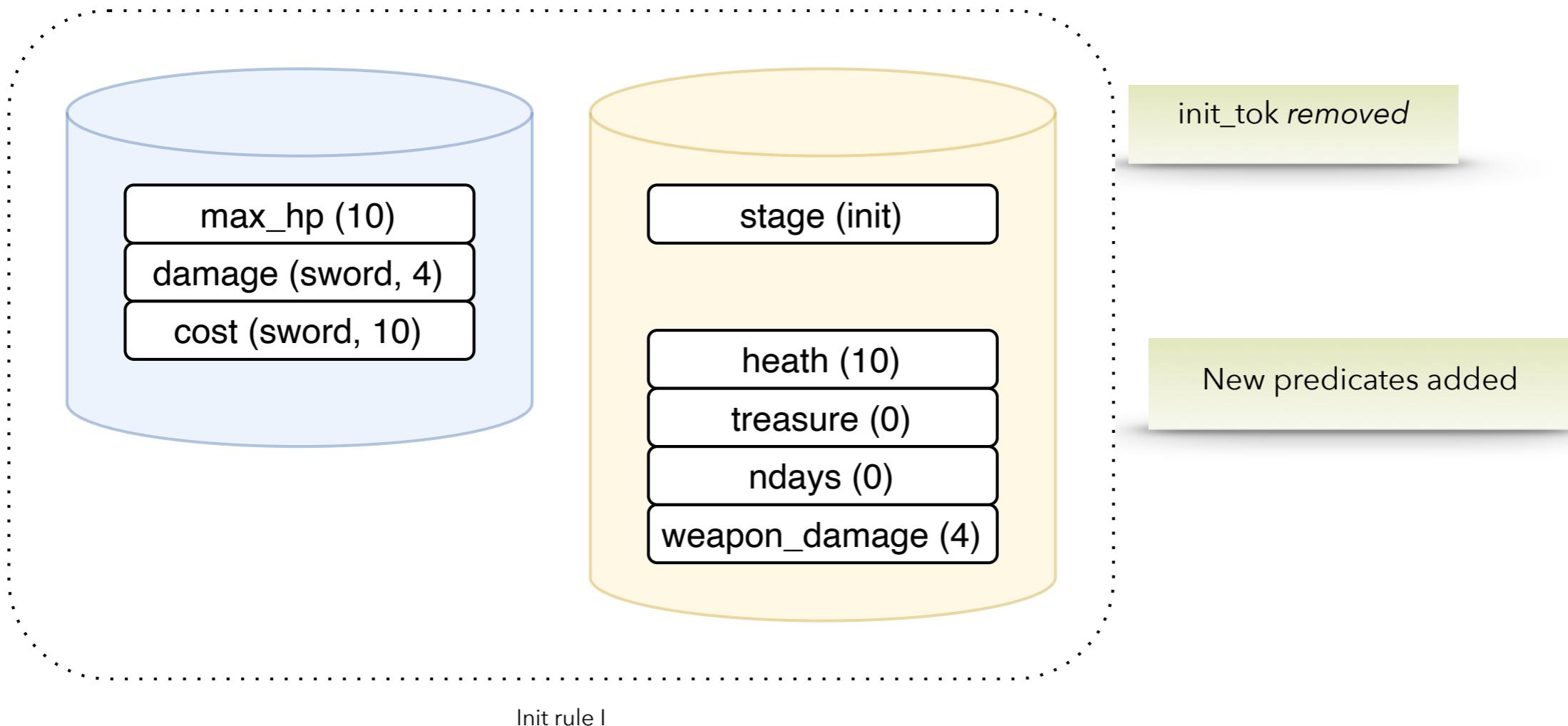
We define the rest of the game using a “screen” idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

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

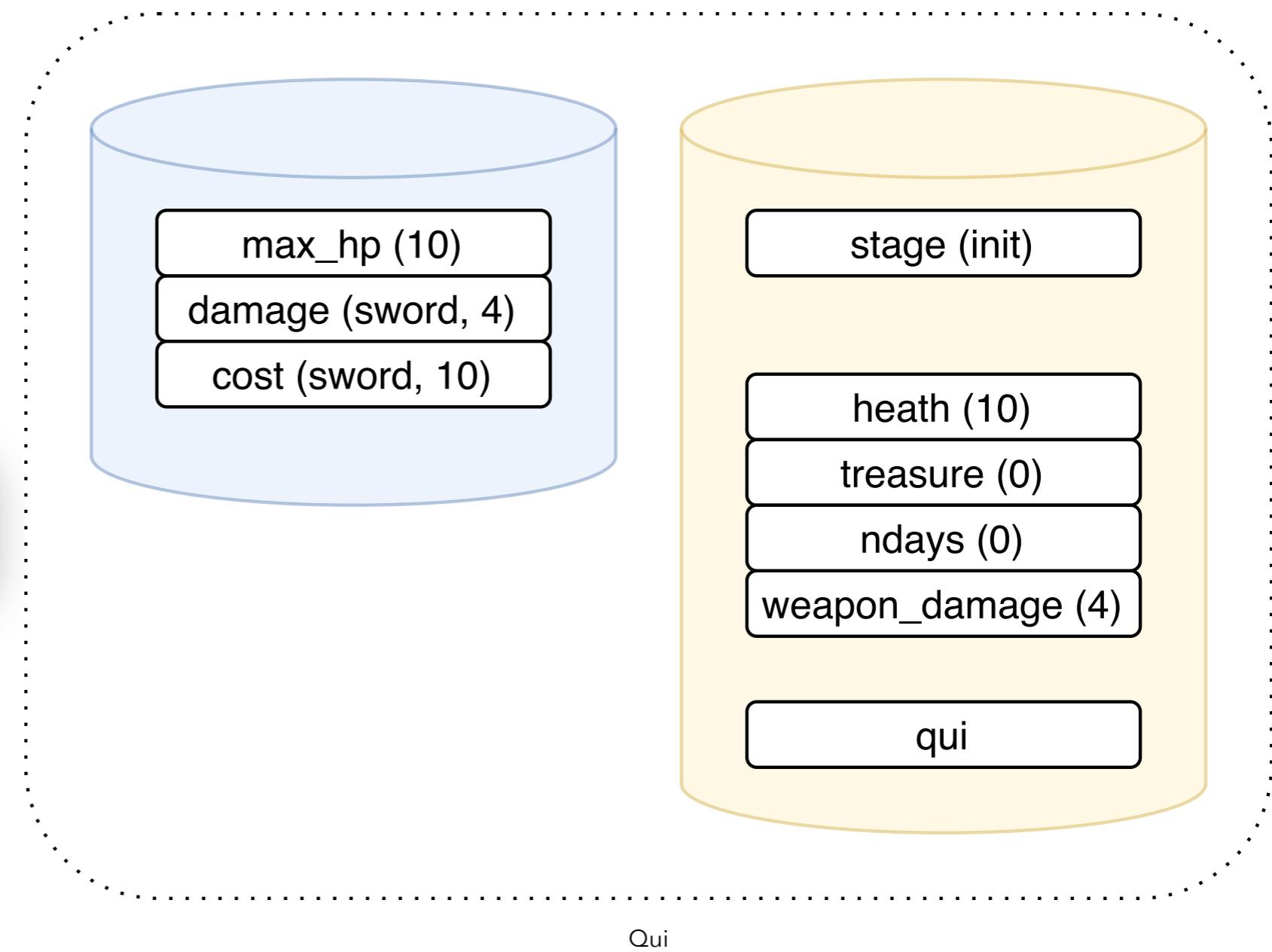
stage main = {
    do/rest : main_screen -o rest_screen.
    do/adventure : main_screen -o adventure_screen.
    do/shop : main_screen -o shop_screen.
```

Ceptre code stage init

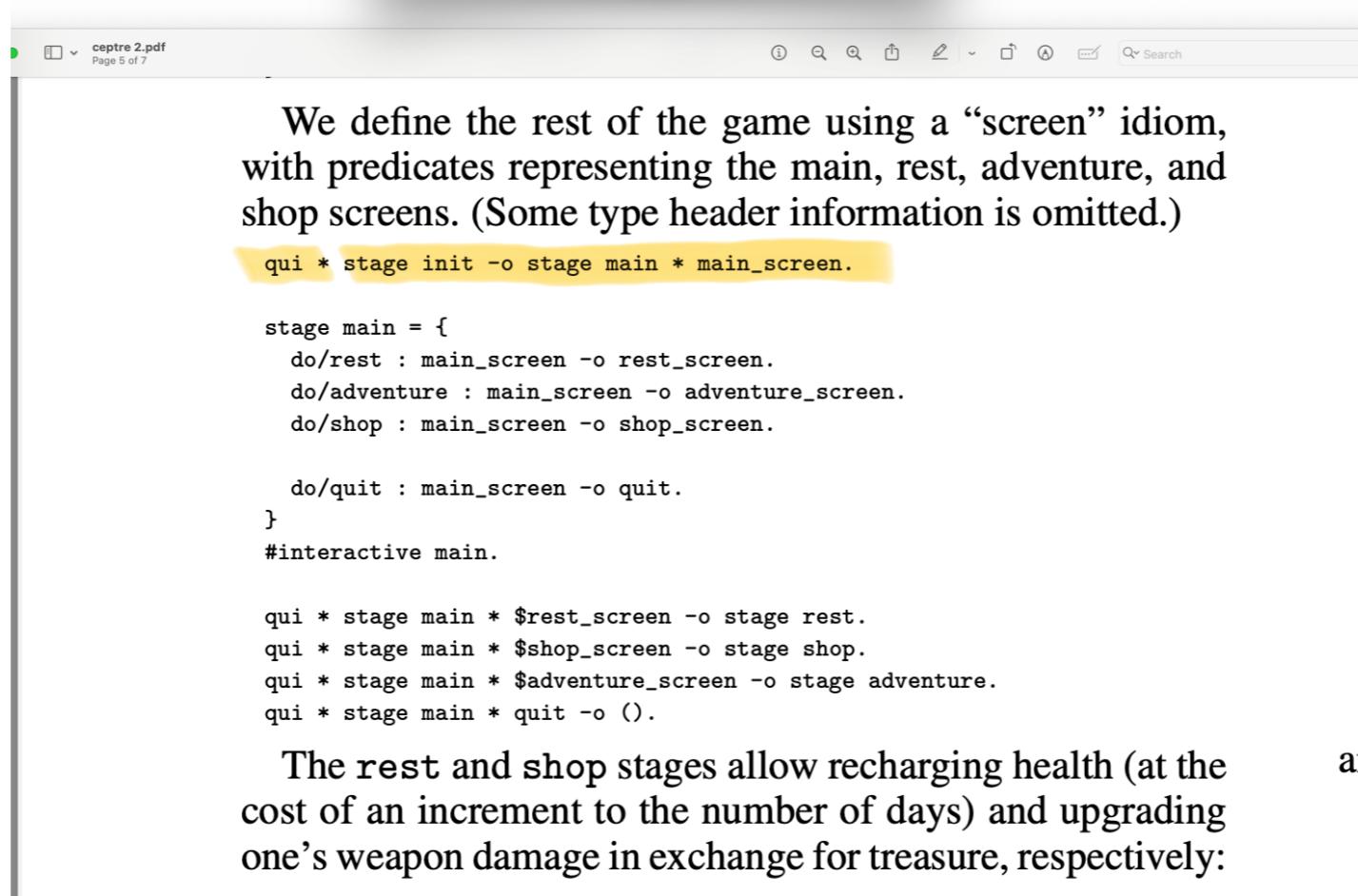




When no more rules can be enabled
(matched), insert *qui* into the factbase.



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

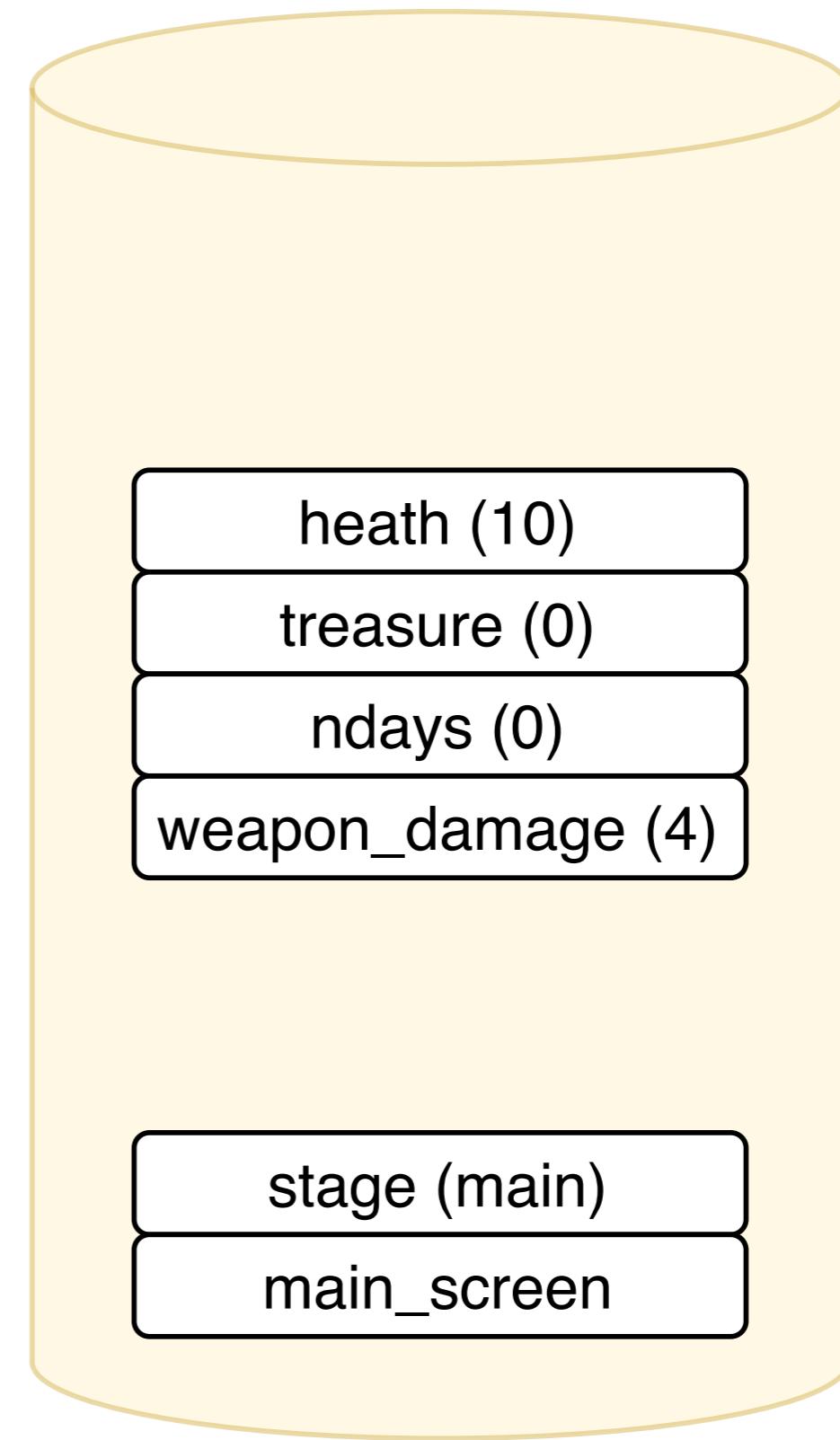
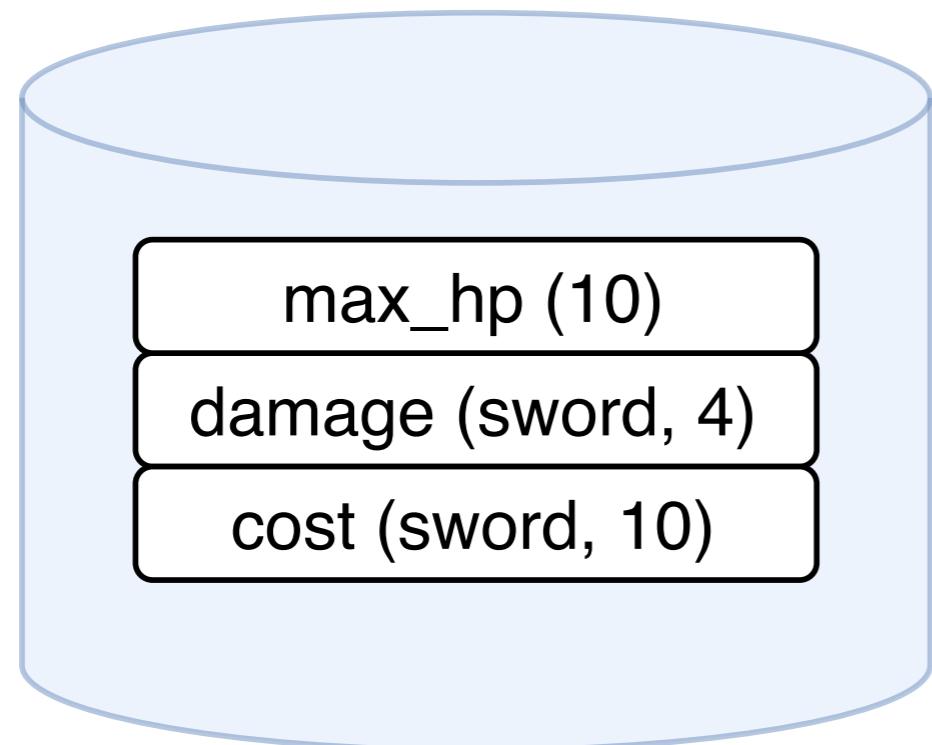
 Remove *qui* from FB

 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..

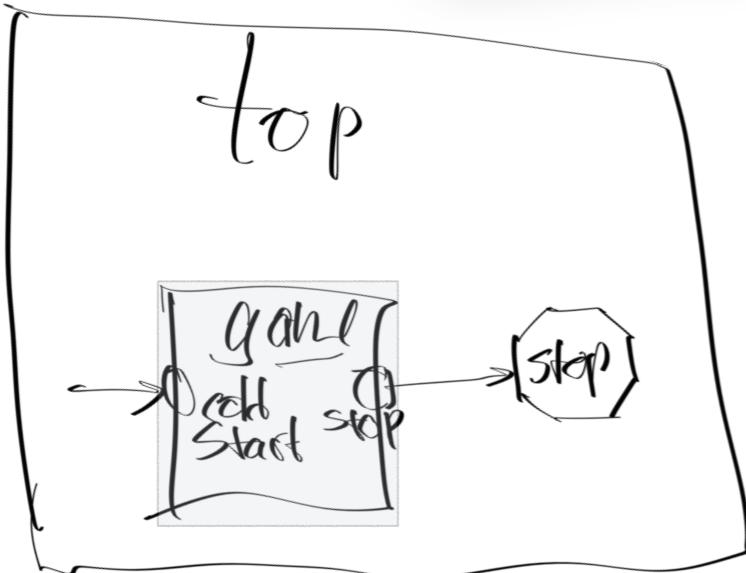
The following pages are freehand sketches that give my best understanding of the design of the game. I choose to explain the design using a set of layers.

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).

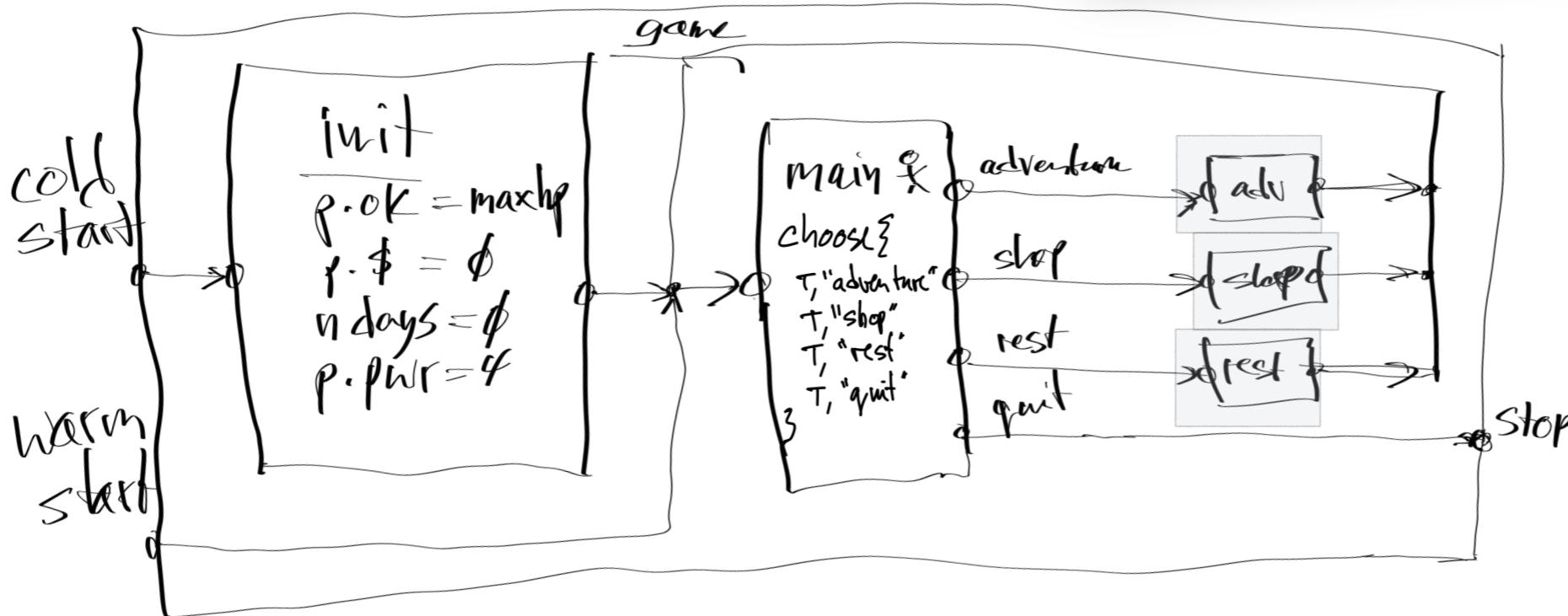
Layers Init and Main



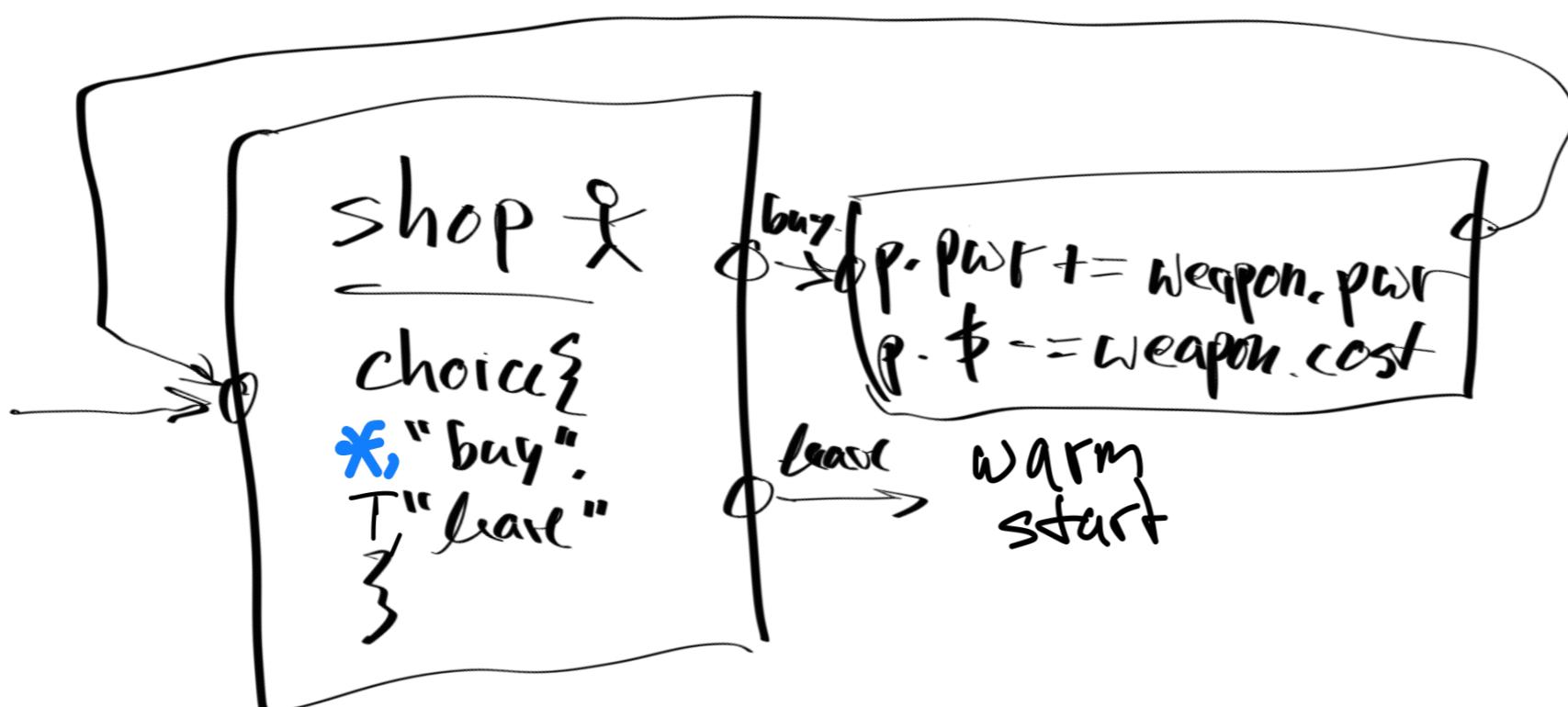
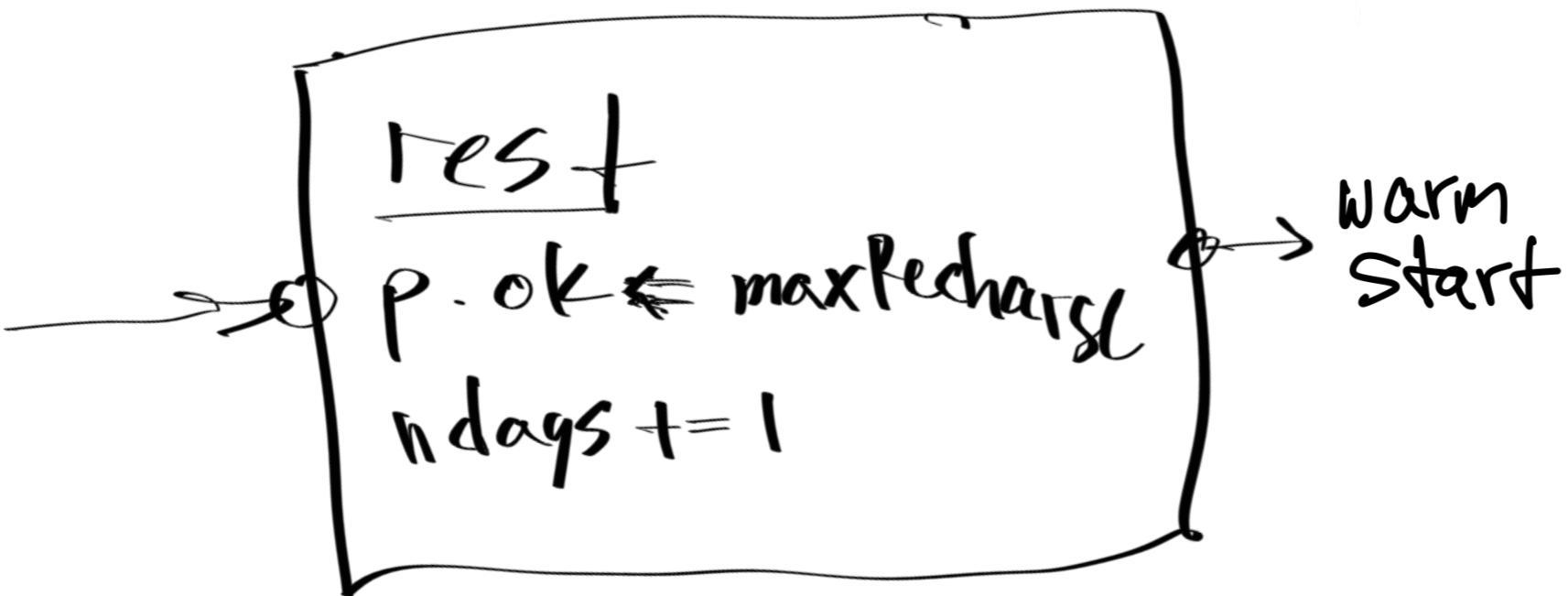
The game begins with a cold start - some variables are initialized, then the main loop is entered.

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

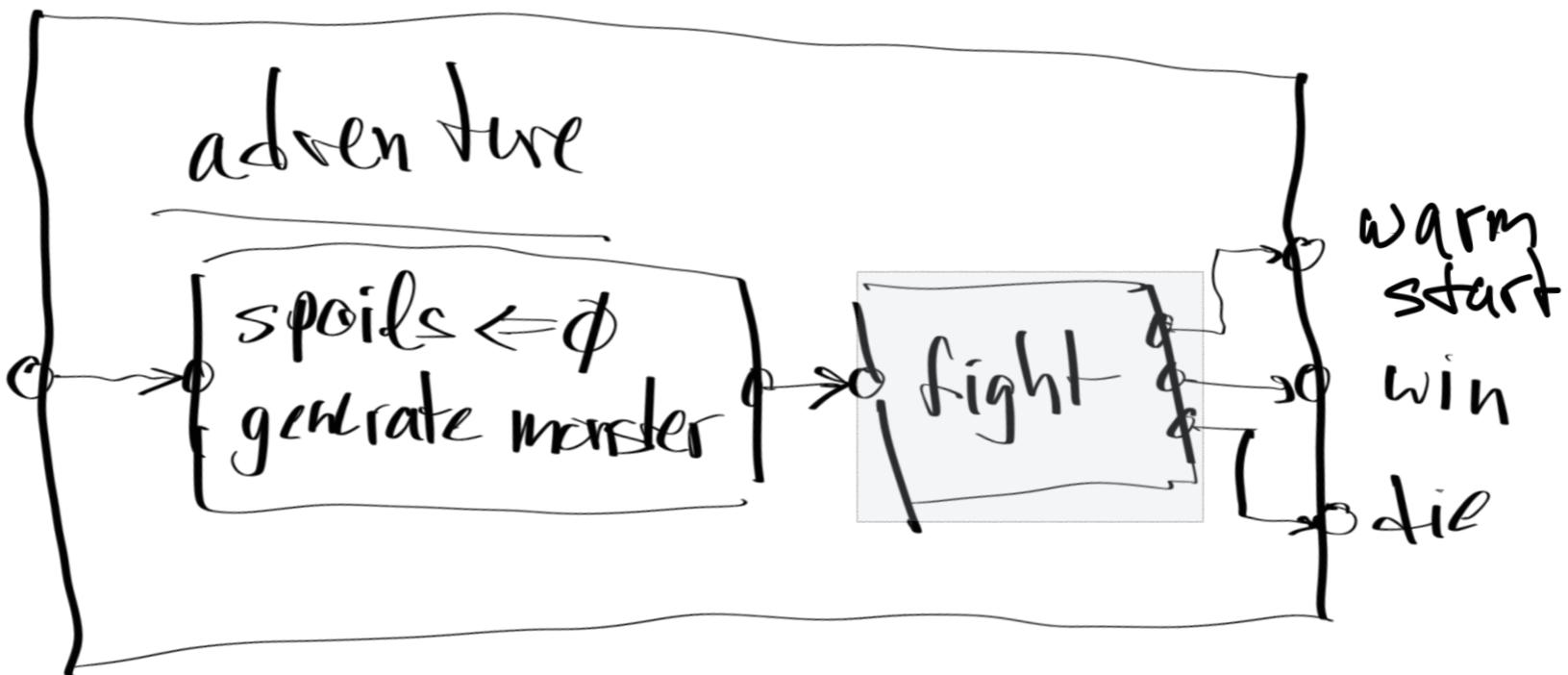


If the player chooses "rest", the player's health is restored to a maximum value, but, 1 day is lost.

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

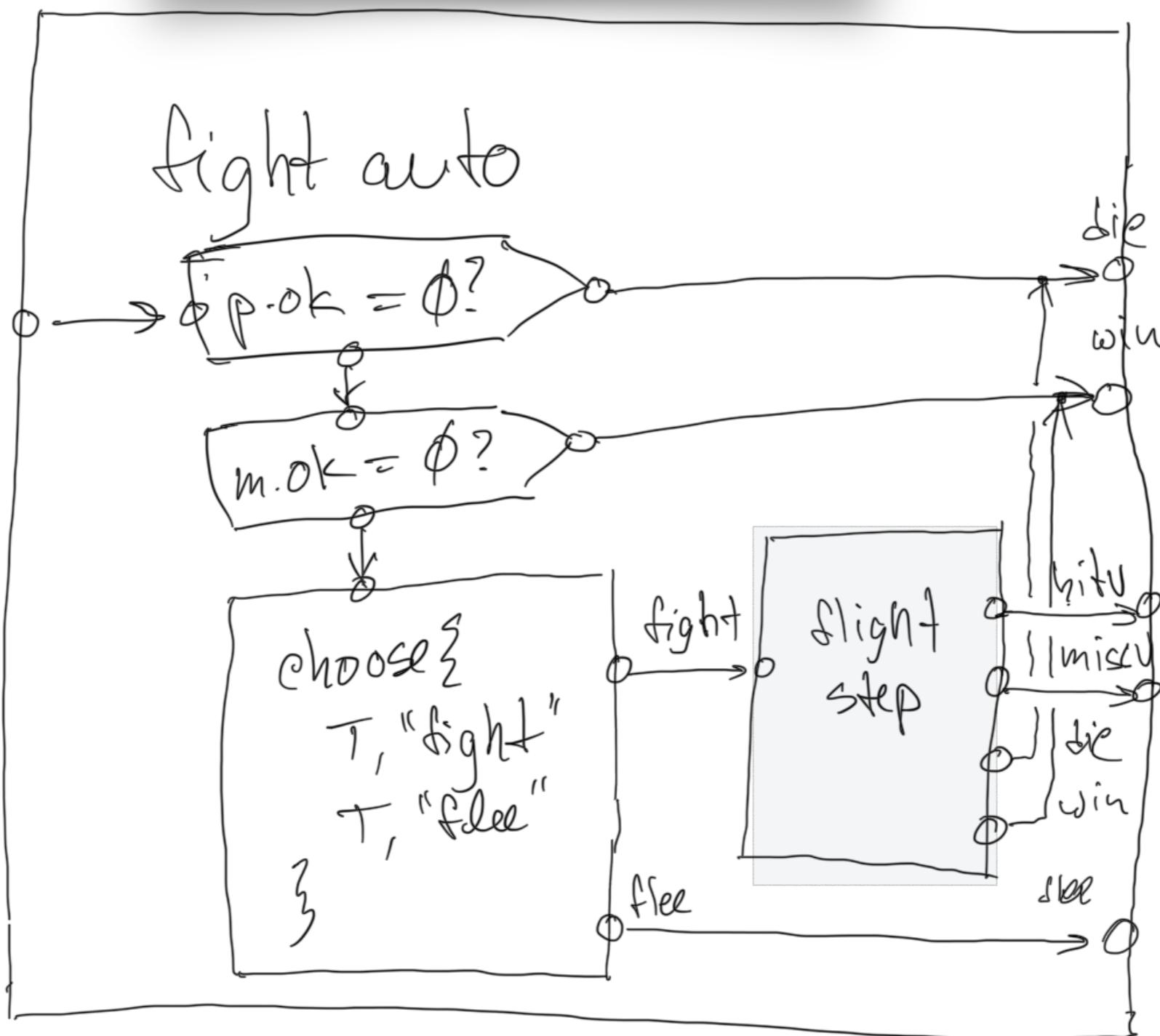
- The player wins the game (all monsters have been vanquished)
- The player loses the game ('dies')
- The player flees from the current battle and foregoes all winnings in a given round of fighting, the game continues with a warm start.

Layer Fight



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

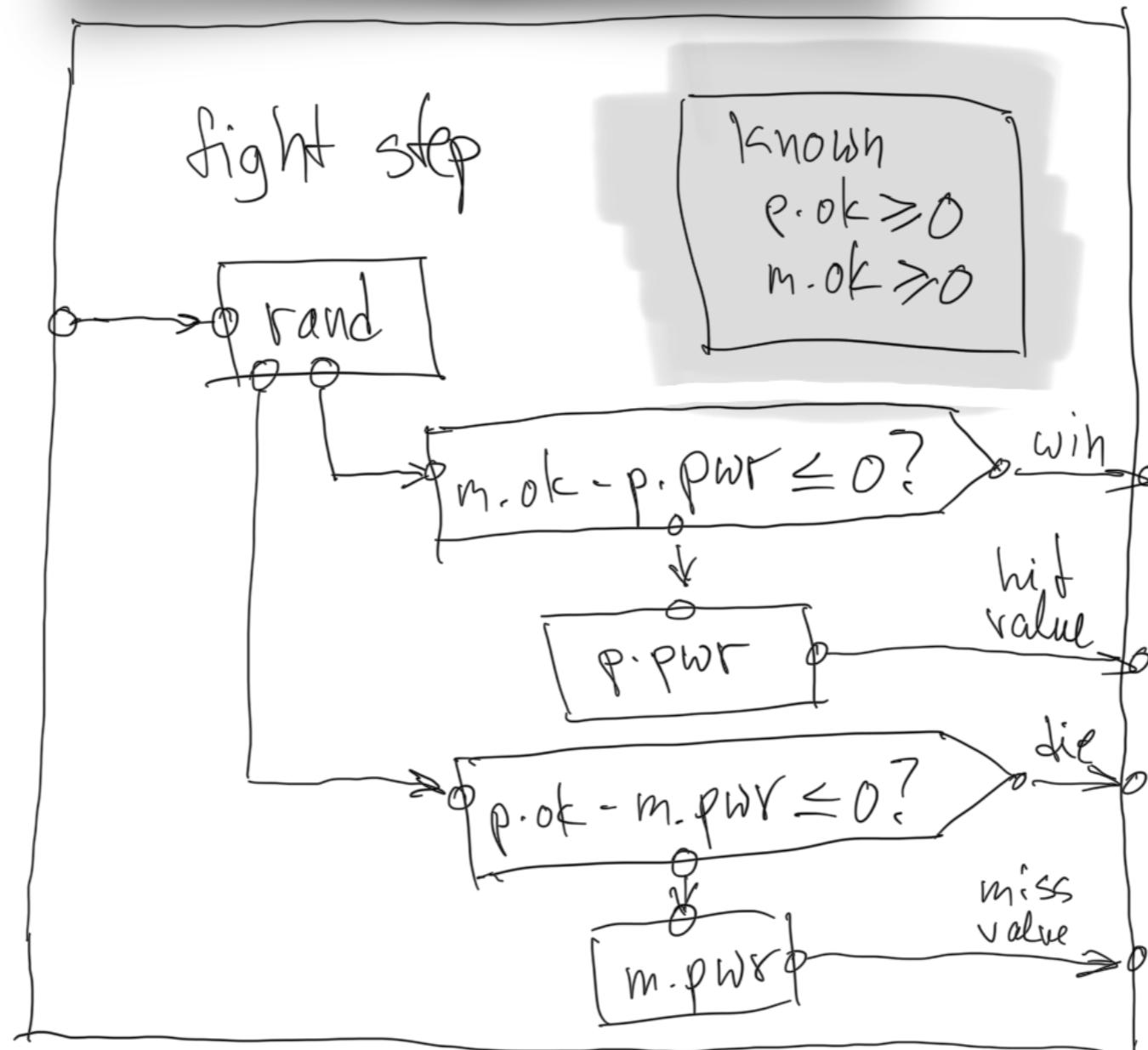
Layer Fight Auto



Fight auto runs the fight loop.

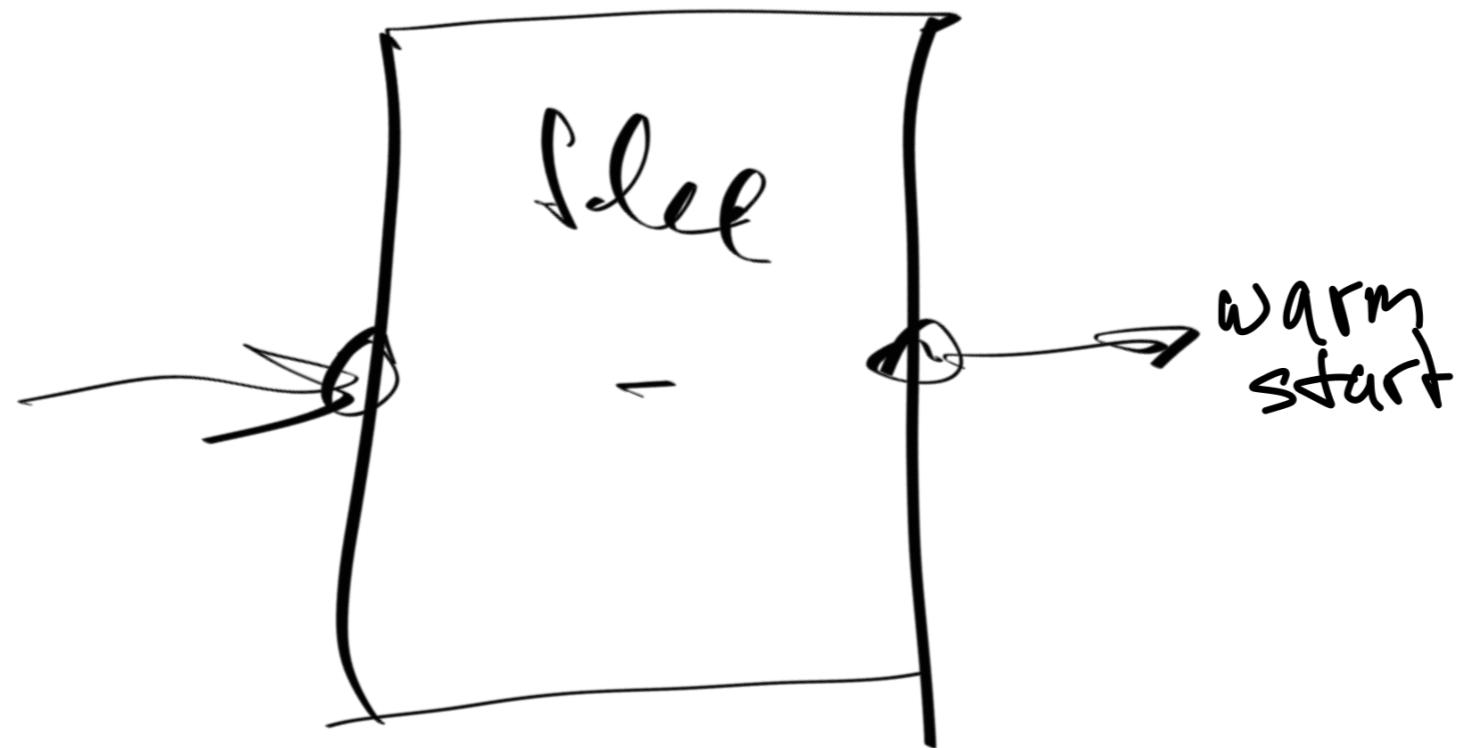
Fight auto returns 1 of 5 possibilities: a hit value, a miss value, a die event, a win event, a flee event.

Layer Fight Step



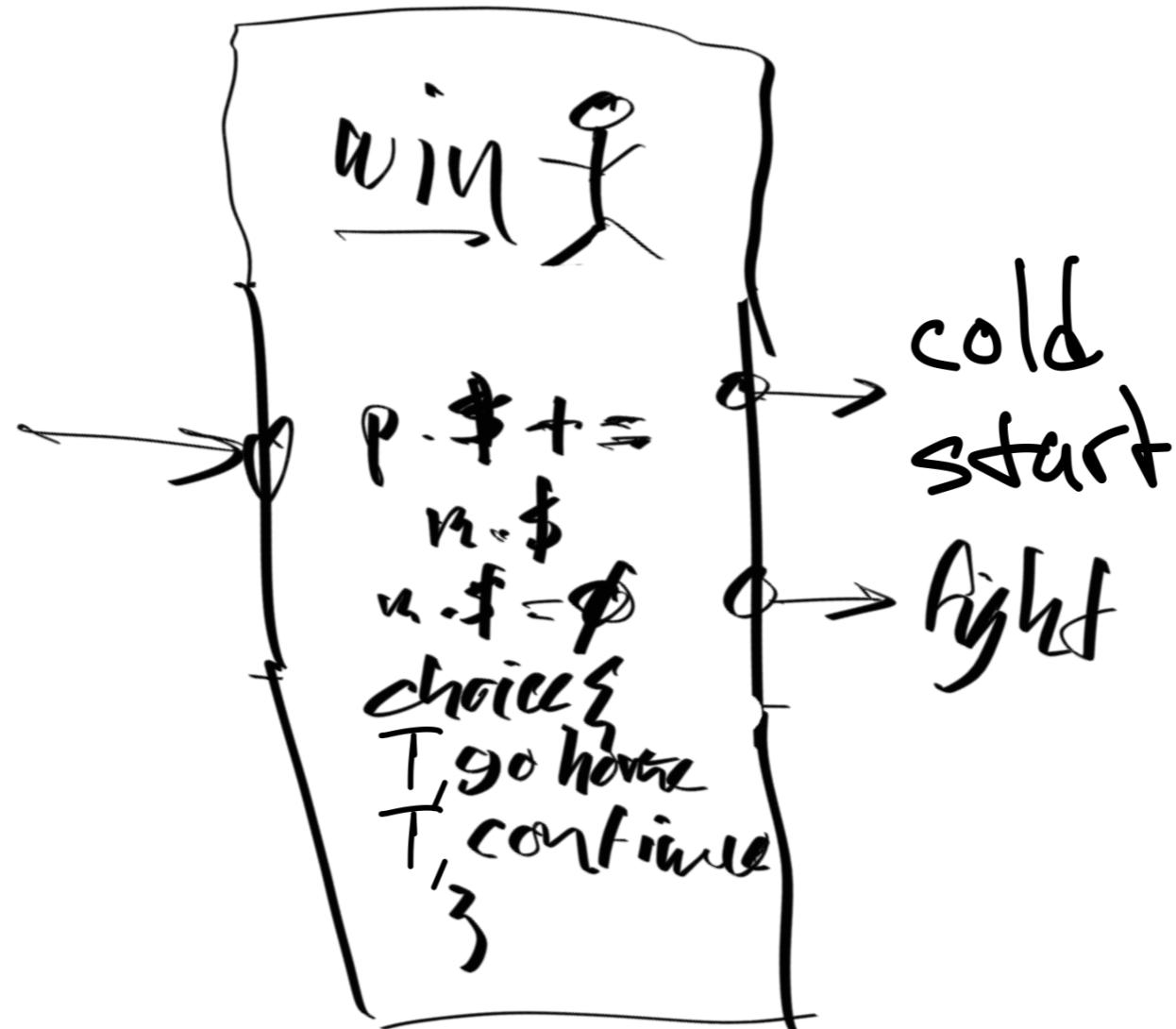
Fight step picks a hit or a miss at random, then produces a "win" or "hit value" event, or, produces a "die" or "miss value" event.

Layer Flee



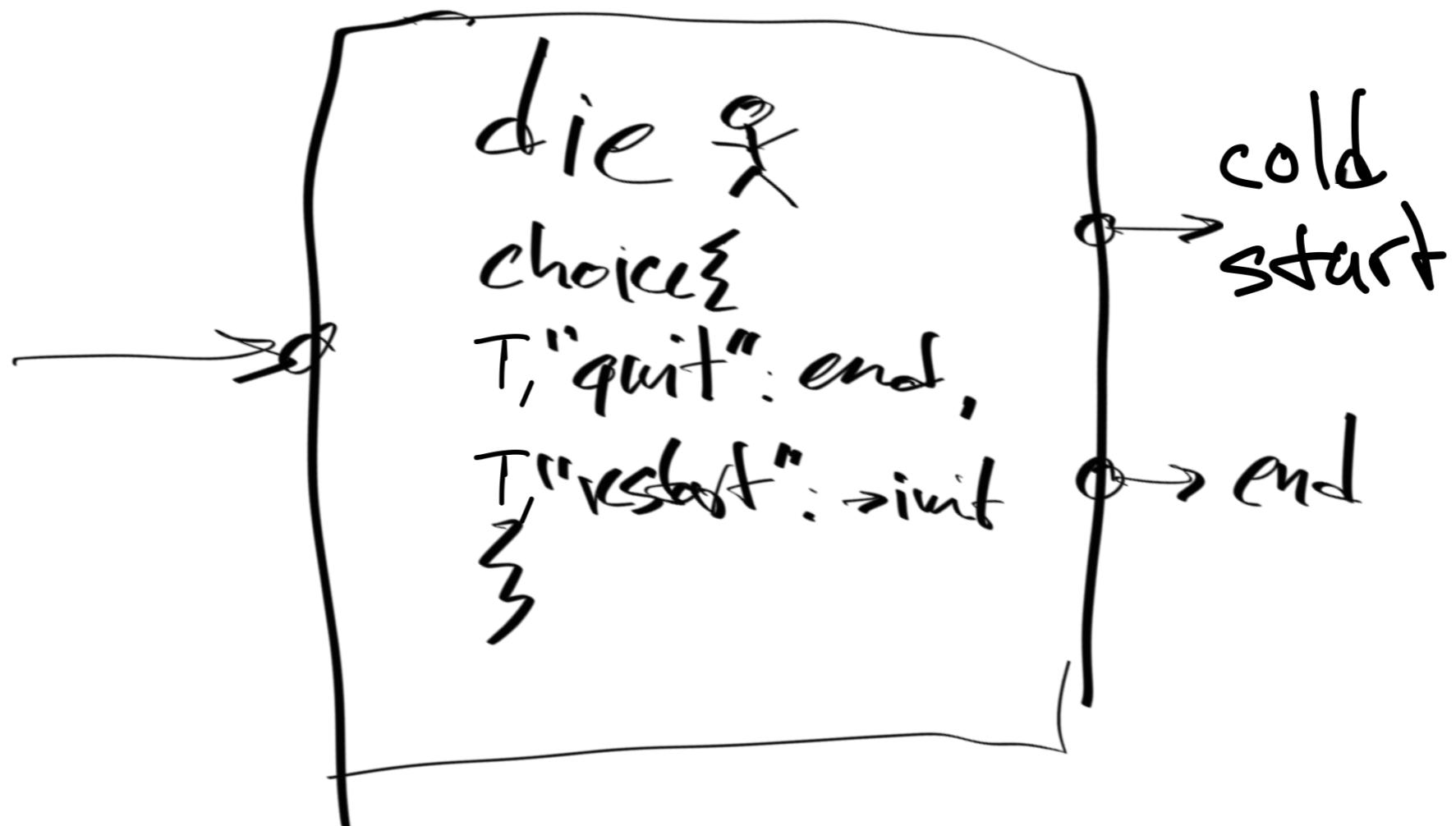
The player foregoes all \$'s dropped by the monster and goes back to the main loop.

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

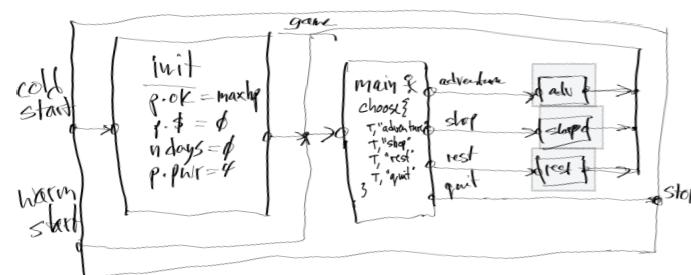
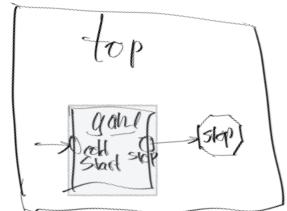


The player lost the game (player's health has dropped to 0). The player can play again or quit the game.

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.

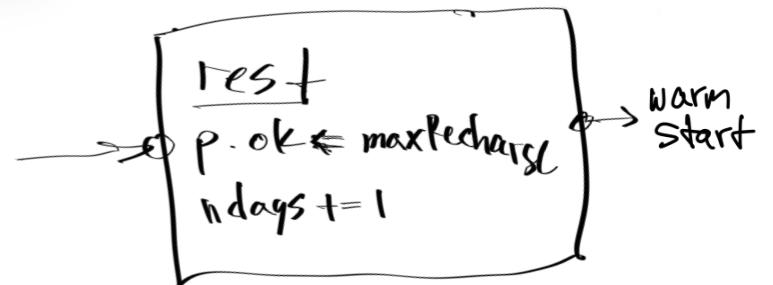
The player is asked to pick one of the rules.

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

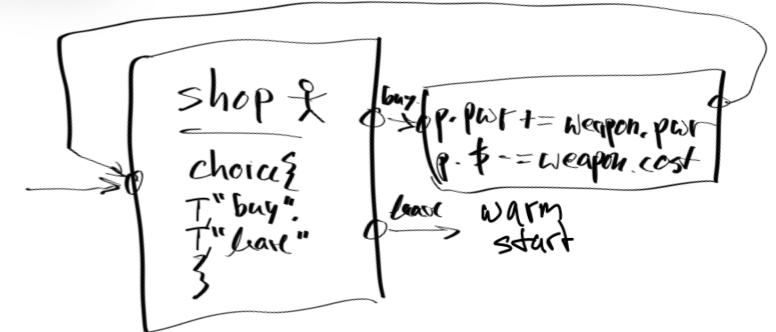
N := max (Max, HP + Recharge)
FB += health (N)
FB += ndays (NDAYS + 1)

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.
  
```

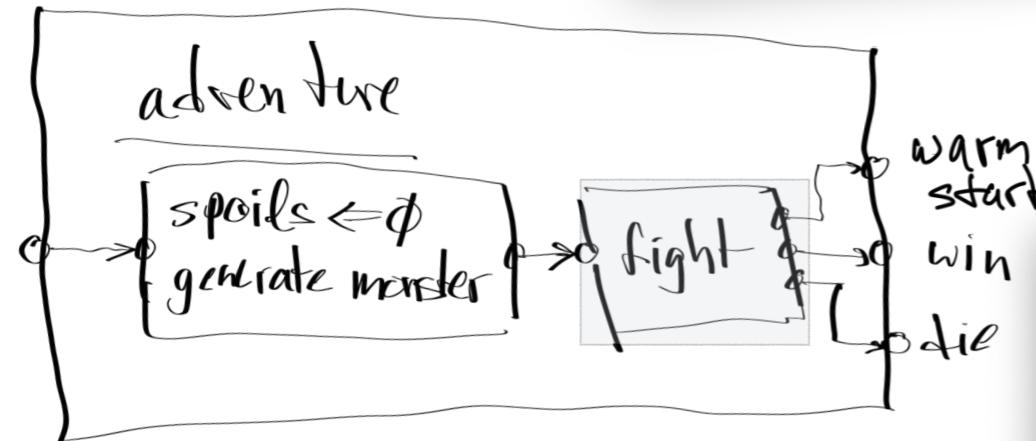
```

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.
```

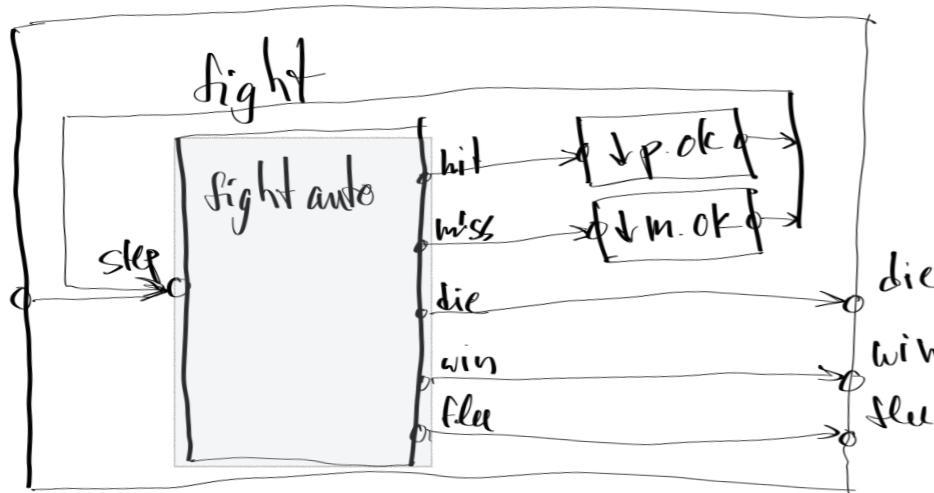
```
stage fight_init = {
    init : fight_screen -o gen_monster * fight_in_progress.
    gen_a_monster : gen_monster * monster_size Size
        -o monster Size * monster_hp Size.
}
```

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.
```

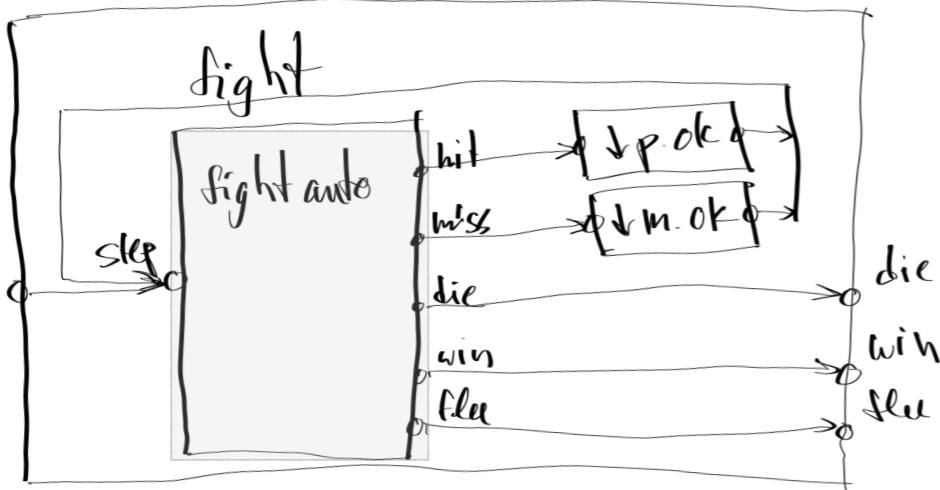
Reverse-engineering this code is overly difficult.

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).

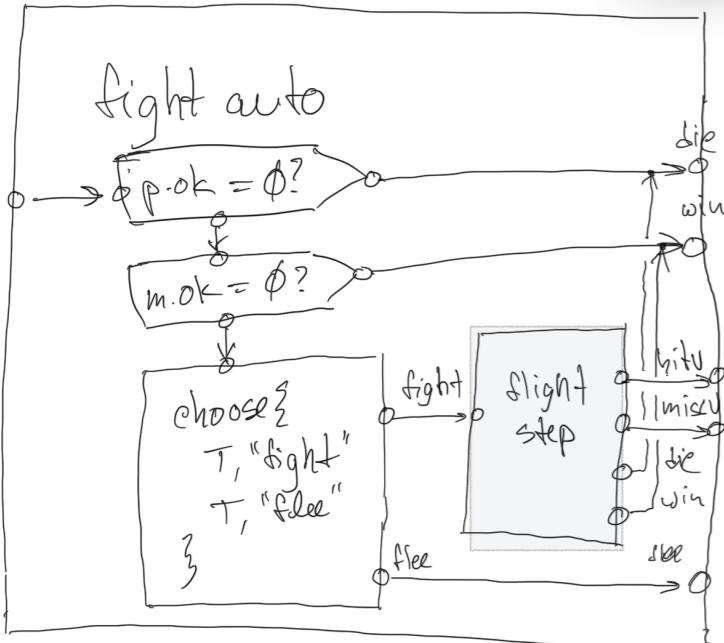
N.B. When 1 output fires, the other 4 outputs produce *nothing*. Not *nil*, not *false* - nothing, no event whatsoever.

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

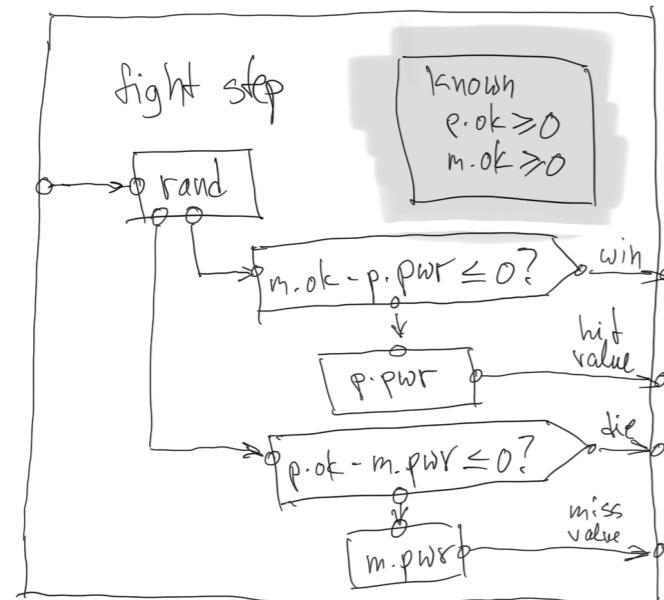


```
stage fight_auto = {
    fight/hit
        : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
            * subtract MHP D (some MHP') -o monster_hp MHP'.
    win
        : fight_in_progress * monster_hp MHP * $weapon_damage D
            * subtract MHP D none -o win_screen.
    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.  

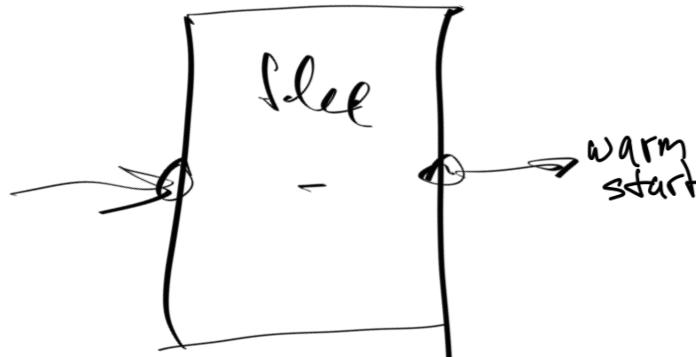
qui * stage fight * $flee_screen -o stage flee.

```

"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.

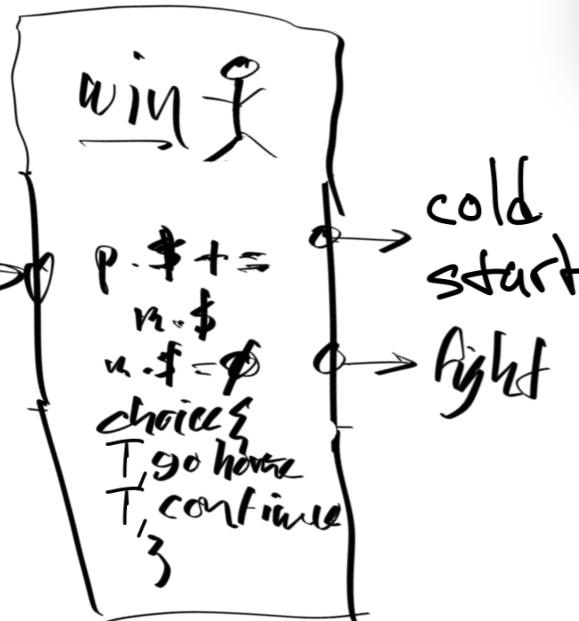
Code Layer Flee



```
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

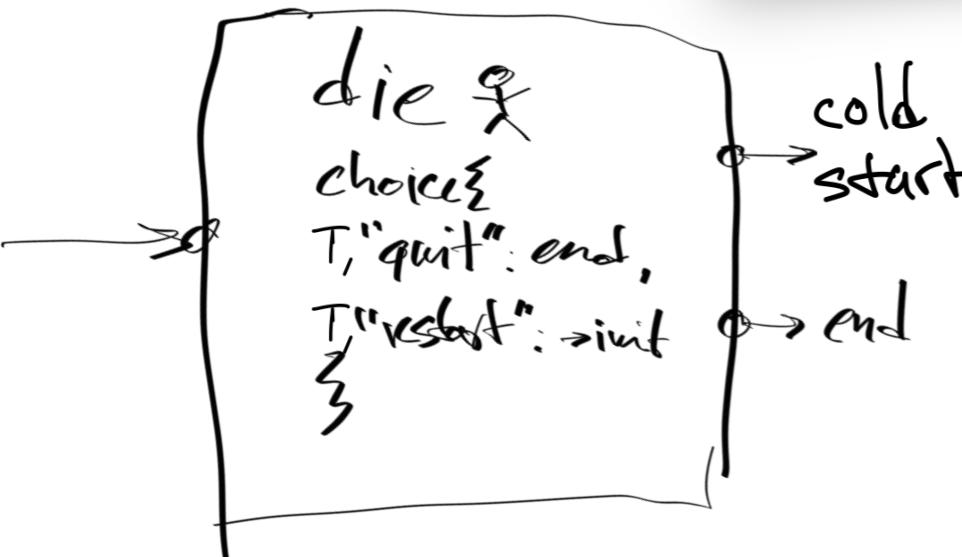


```
go_home_or_continue : pred.  
stage win = {  
    win : win_screen * monster Size * drop_amount Size Drop  
        -o drop Drop.  
    collect_spoils : drop X * spoils Y * plus X Y Z  
        -o spoils Z * go_home_or_continue.  
    go_home : go_home_or_continue  
        * spoils X * treasure 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



```
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/>