Paul Tarvydas

CEPTRE -WALKTHROUGH OF THE DUNGEON CRAWLER EXAMPLE

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

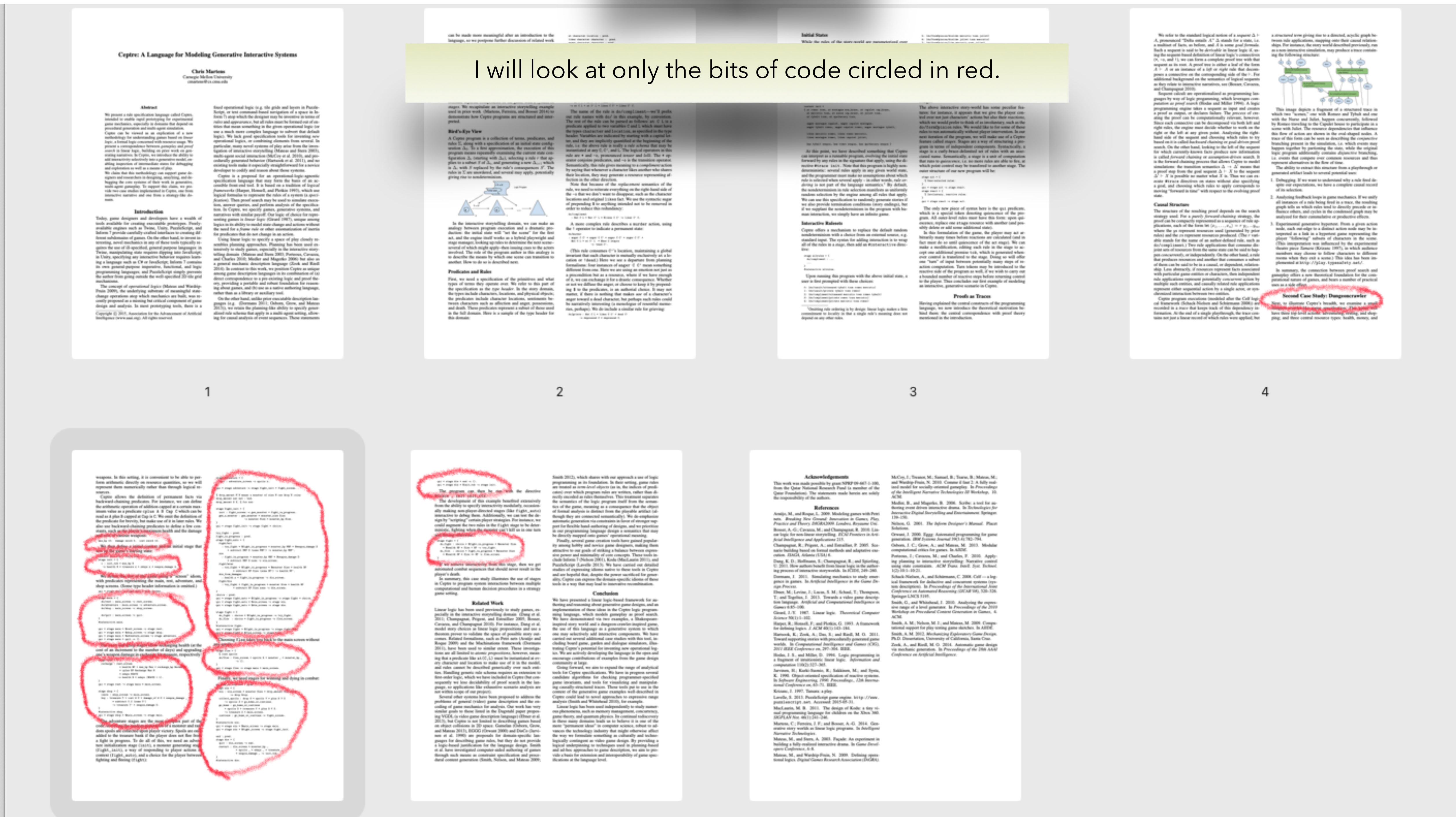
THE PAPER

https://www.cs.cmu.edu/~cmartens/ceptre.pdf

Source code for a more involved version of RPG:

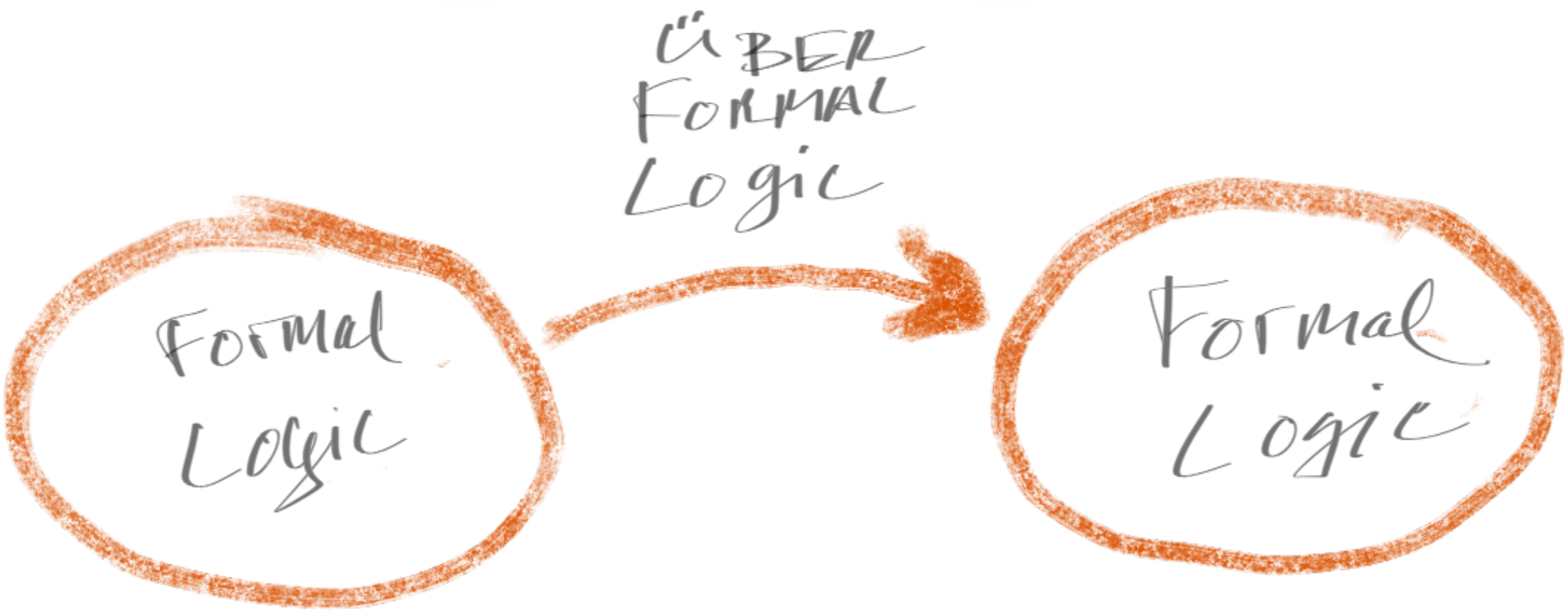
https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg.cep

AGENDA



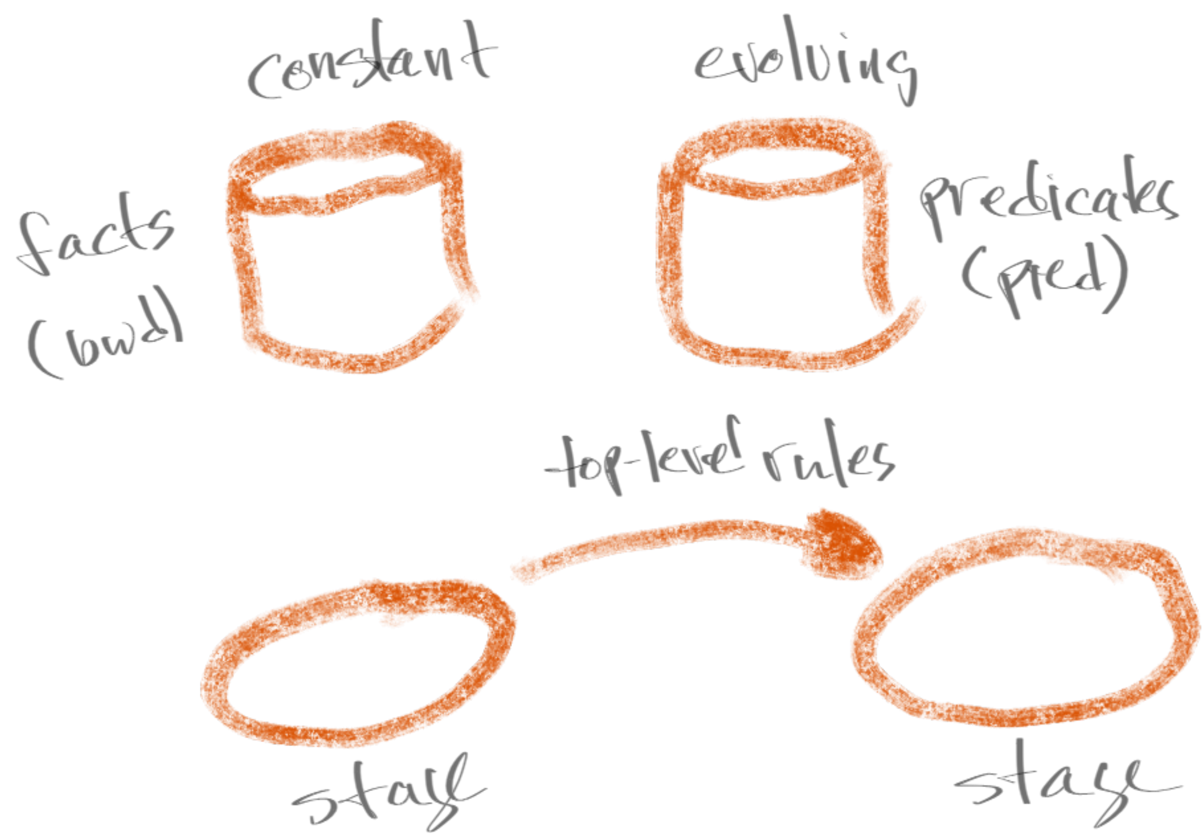
I view the code from the perspective of thinking of Ceptre as a language for writing practical gaming software, instead of as a way to formally analyze the required operations.

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

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:

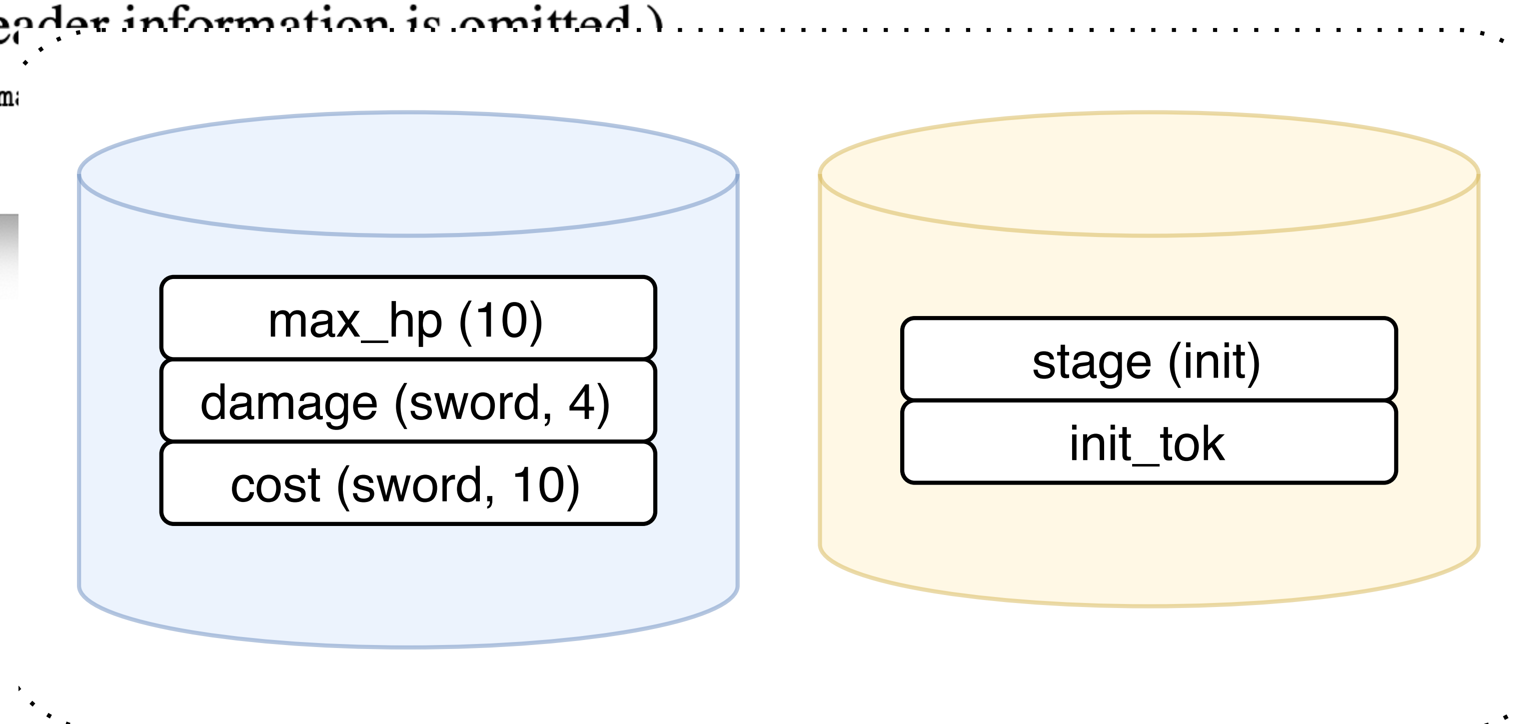
```
max_hp 10.  damage sword 4.  cost sword 10.
```

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

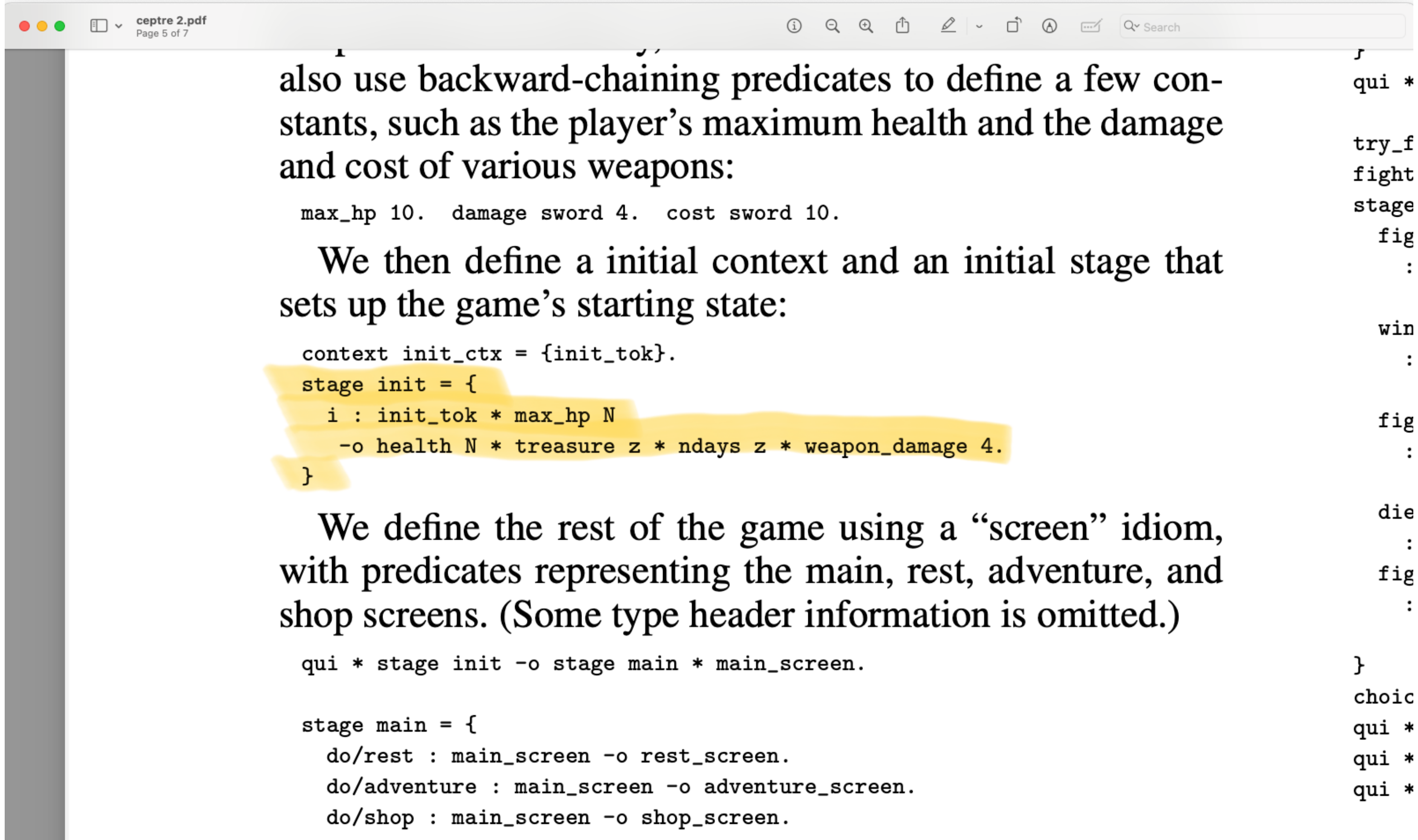
```
context init_ctx = {init_tok}.
stage init = {
  i : init_tok * max_hp N
   -o health N * treasure z * ndays z * weapon_damage 4.
}
```

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



ceptre 2.pdf
Page 5 of 7

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:

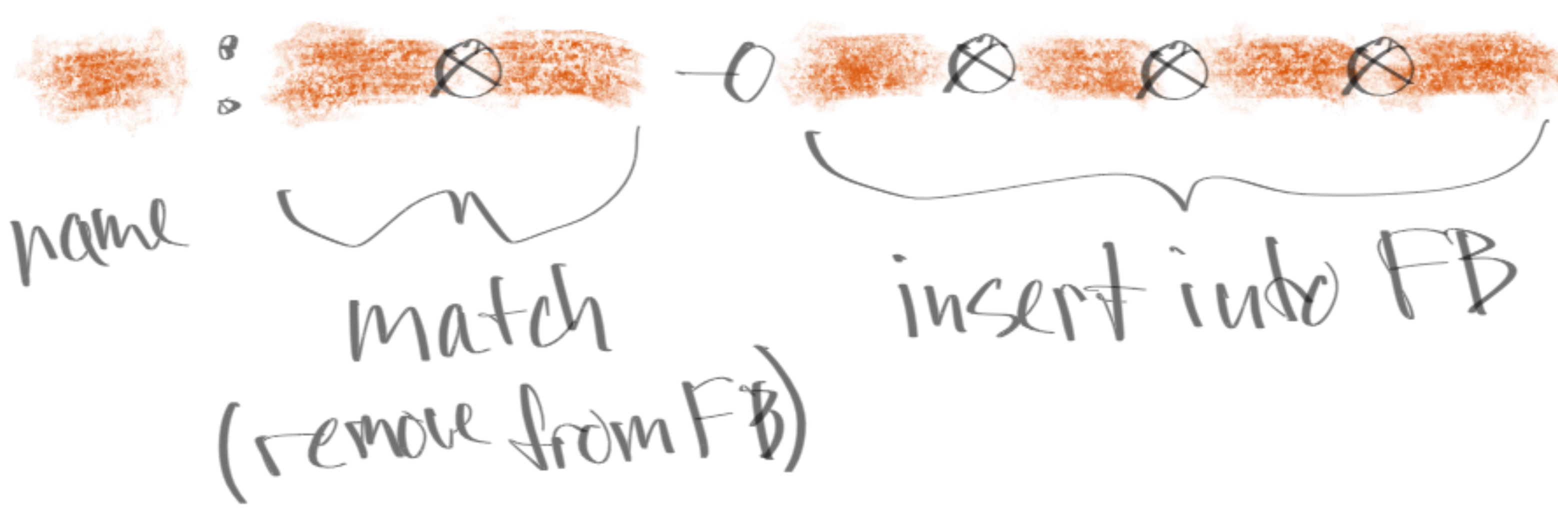
```
context init_ctx = {init_tok}.
stage init = {
  i : init_tok * max_hp N
   -o health N * treasure z * ndays z * weapon_damage 4.
}
```

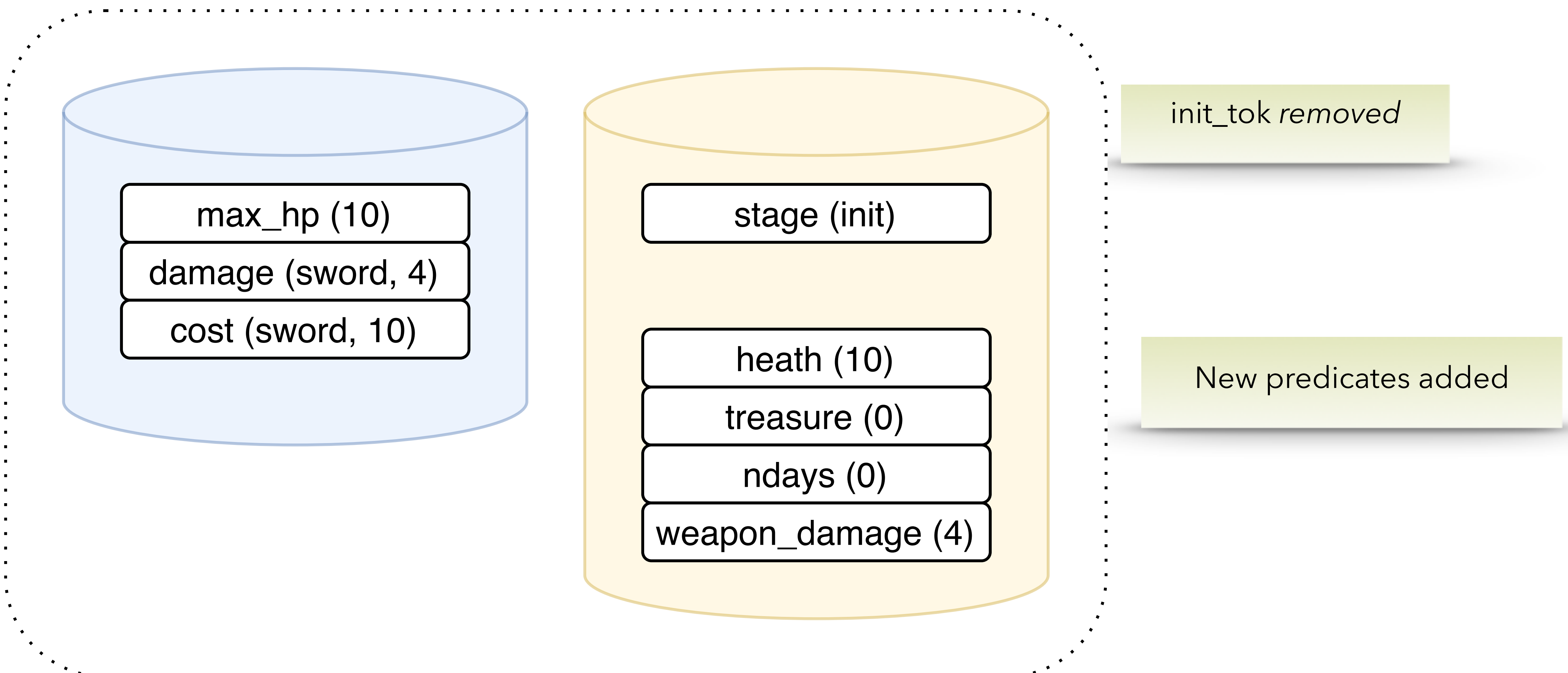
We define the rest of the game using a "screen" idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

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

```
}
qui *

try_f
fight
stage
  fig
    :

  win
    :

  fig
    :

  die
    :
  fig
    :

}
choic
qui *
qui *
qui *
```

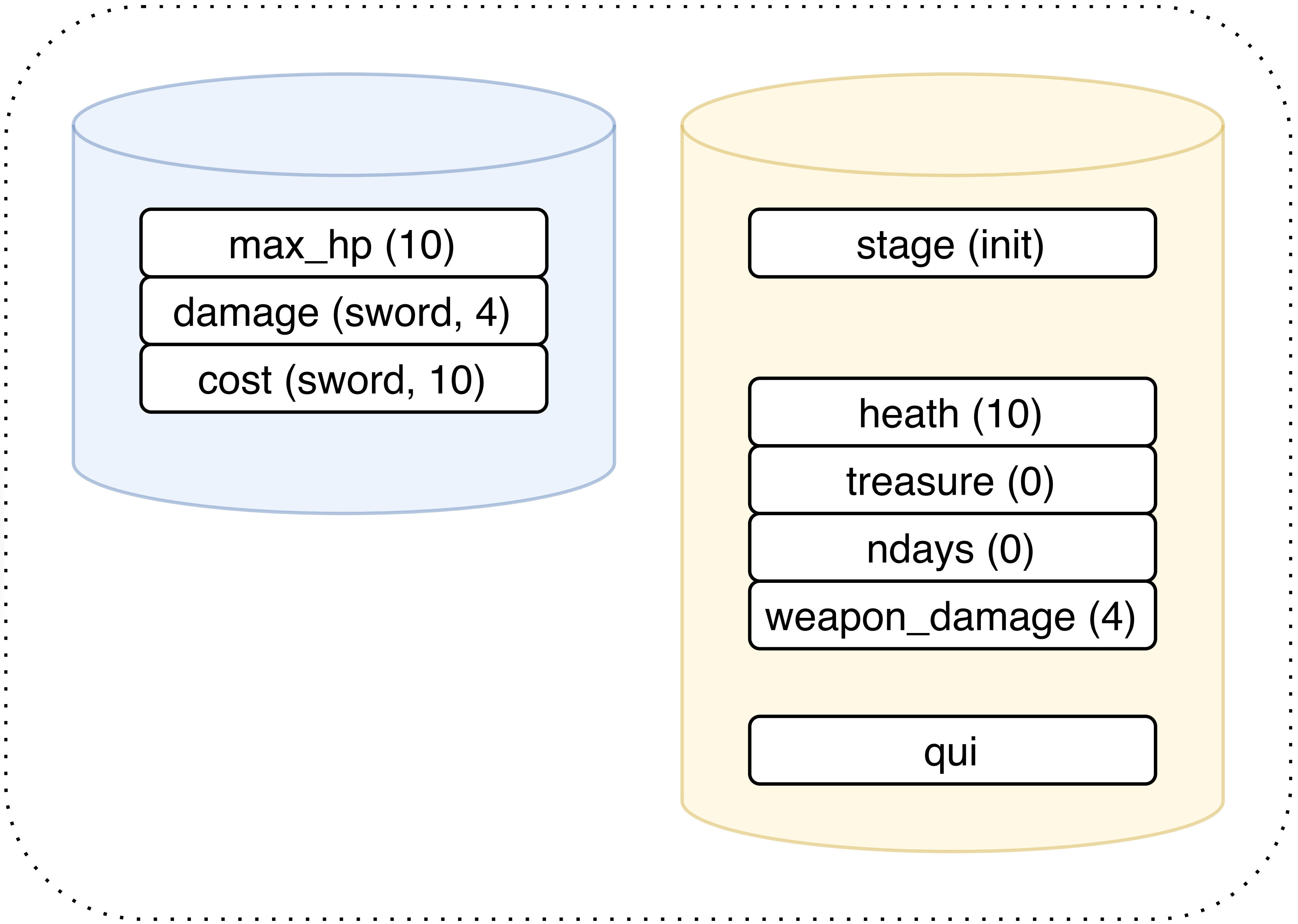
Ceptre code stage init




max_hp (10)
damage (sword, 4)
cost (sword, 10)
stage (init)
heath (10)
treasure (0)
ndays (0)
weapon_damage (4)
init_tok *removed*
New predicates added

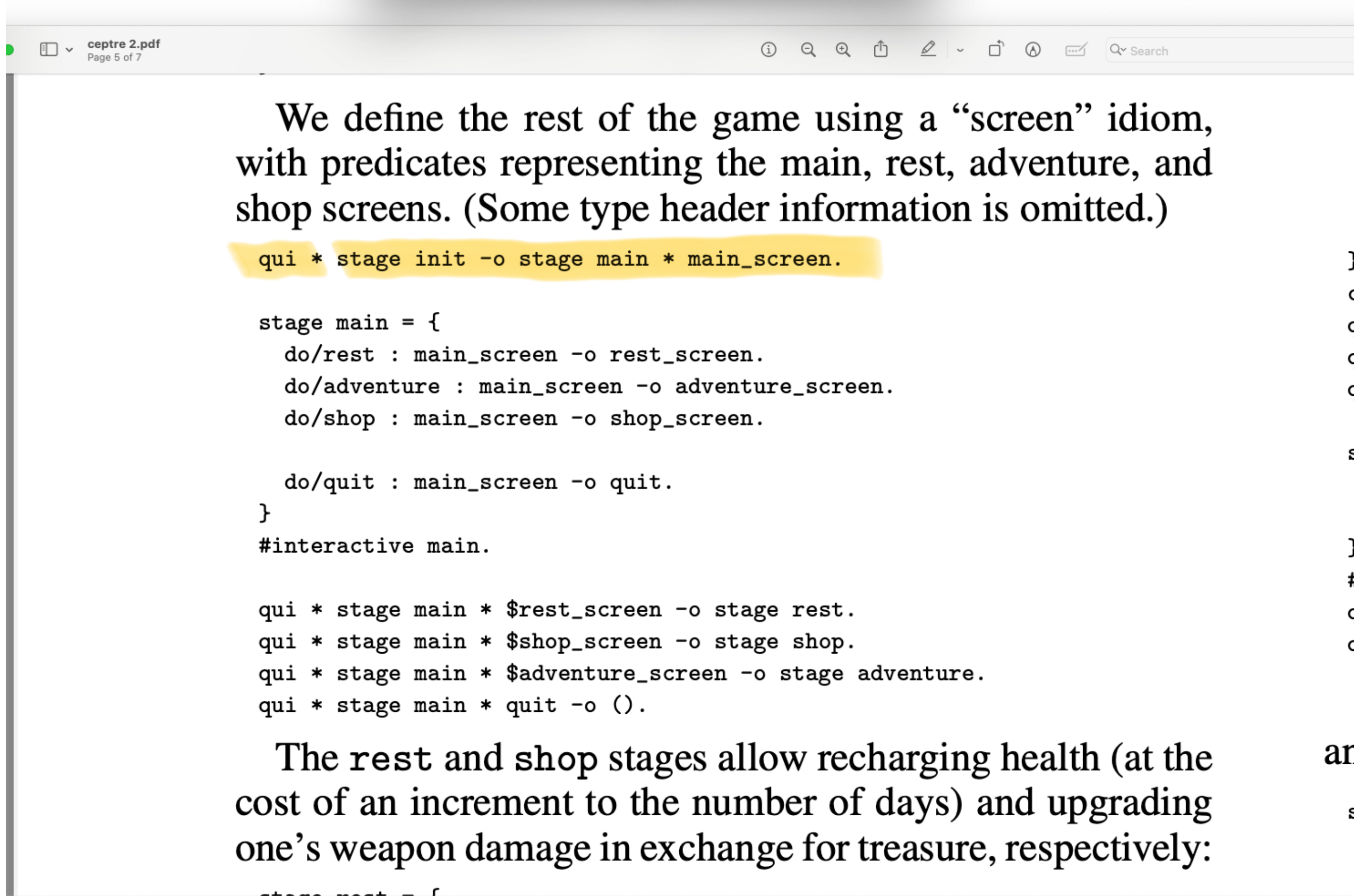
Init rule I

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



Qui

First Transition



ceptre 2.pdf
Page 5 of 7

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.

  do/quit : main_screen -o quit.
}
#interactive main.

qui * stage main * $rest_screen -o stage rest.
qui * stage main * $shop_screen -o stage shop.
qui * stage main * $adventure_screen -o stage adventure.
qui * stage main * quit -o ().
```

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

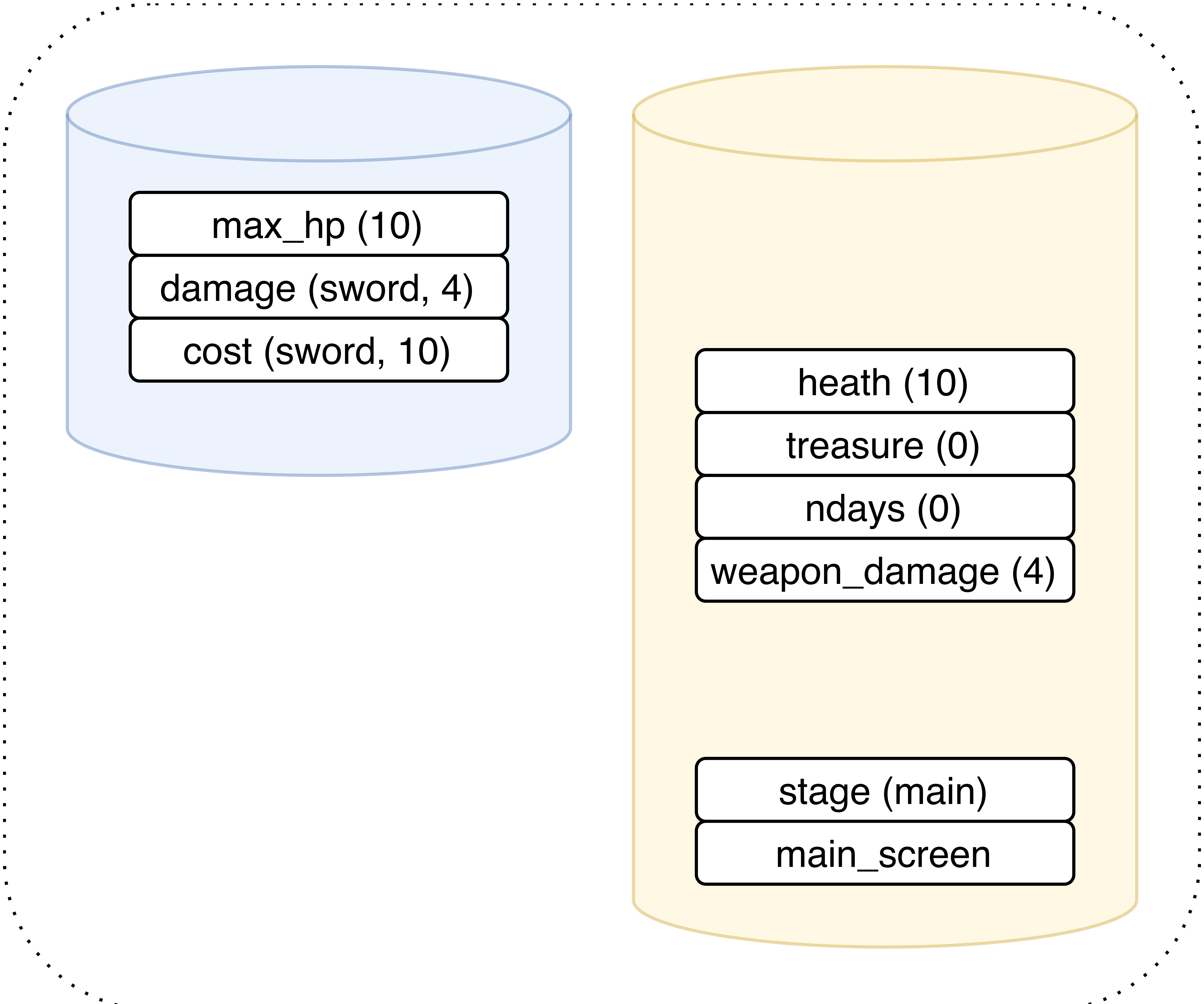
IF state (init) IS *quiescent* then {

Remove *stage (init)* from FB
Remove *qui* from FB

AND

Insert *stage (main) into FB*
Insert *main_screen* () into FB

}


max_hp (10)
damage (sword, 4)
cost (sword, 10)
heath (10)
treasure (0)
ndays (0)
weapon_damage (4)
stage (main)
main_screen

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 a various techniques for designing games..

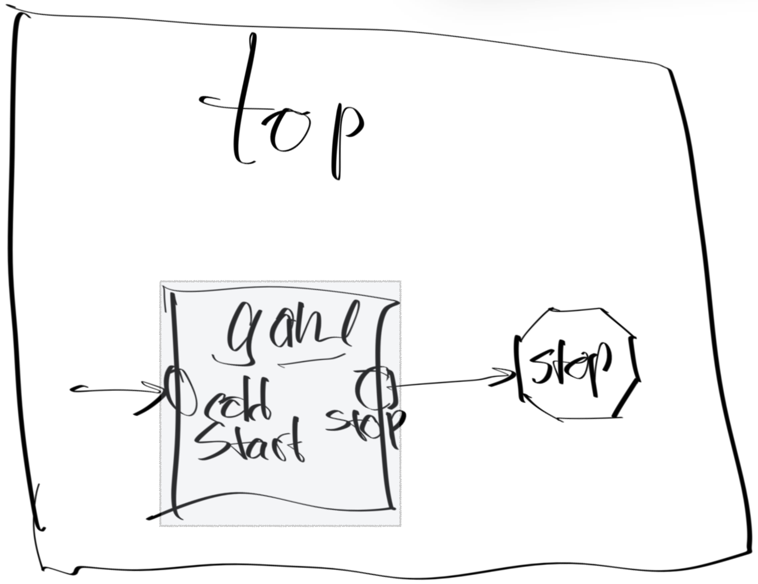
The following pages are freehand sketches that give my best understanding of the design of the game. I choose to explain the design using a set of layers.

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).

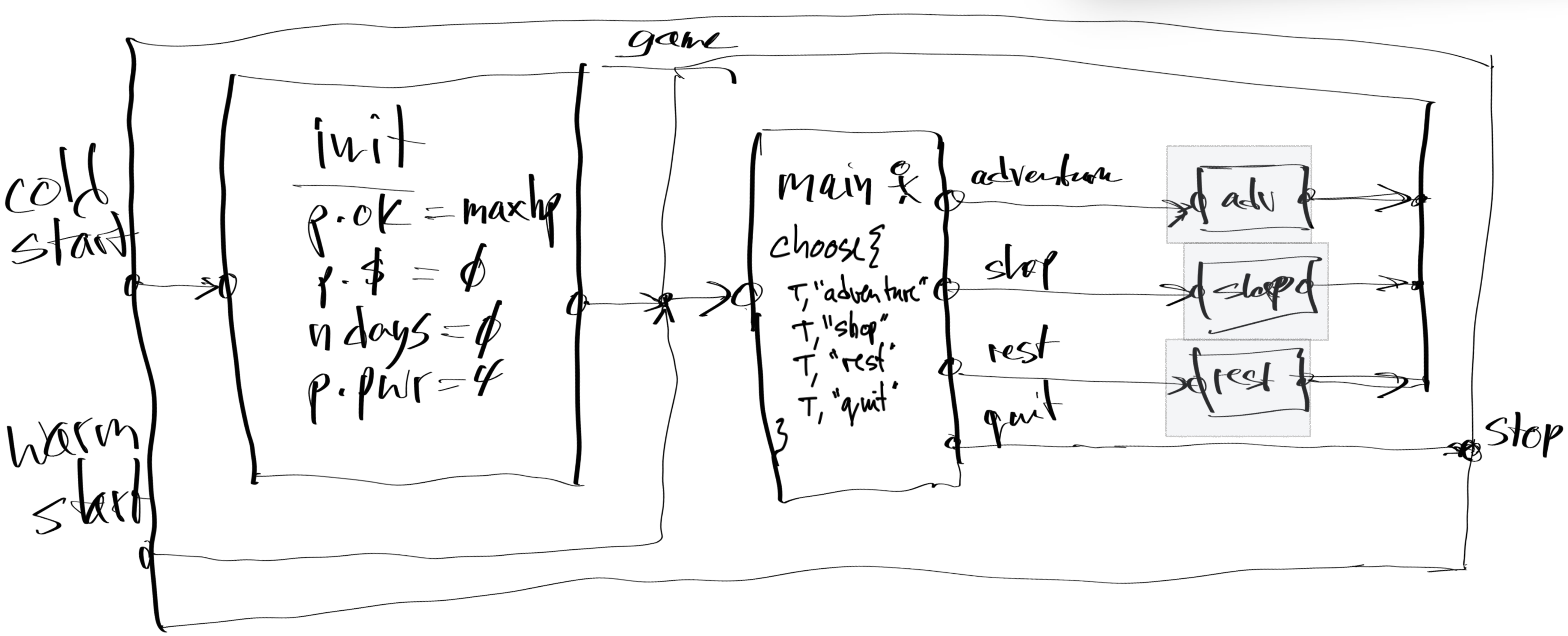
Layers Init and Main



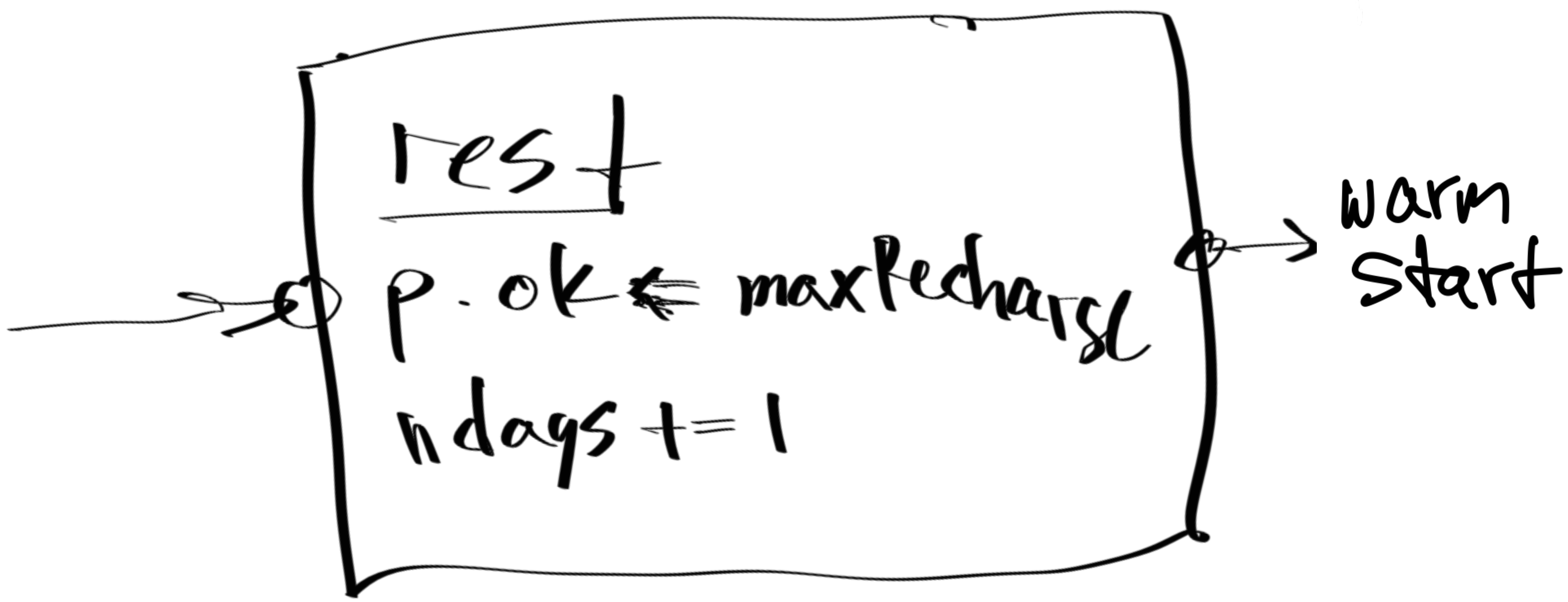
The game begins with a cold start - some variables are initialized, then the main loop is entered.

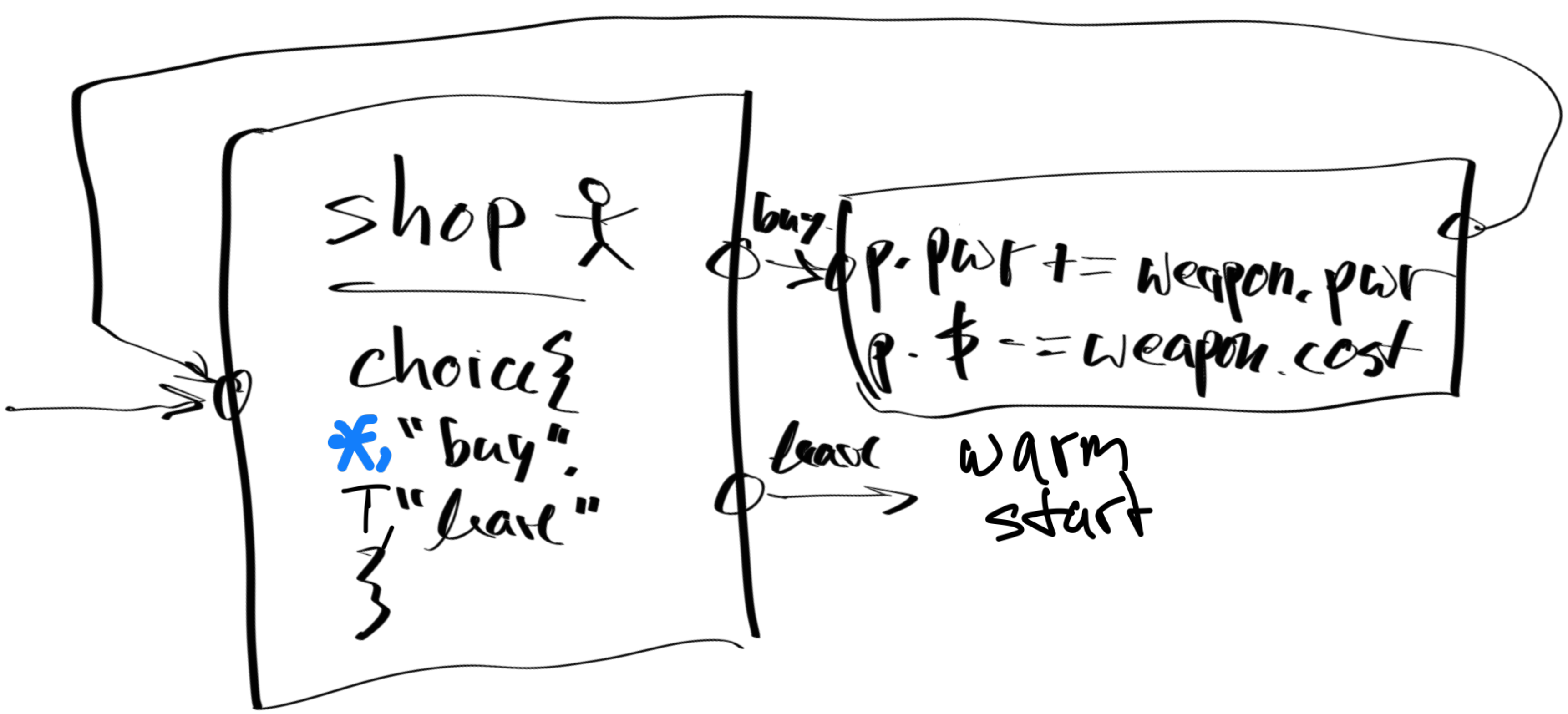
The player must choose from 4 courses of action "adventure", "shop", "rest", "quit".

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



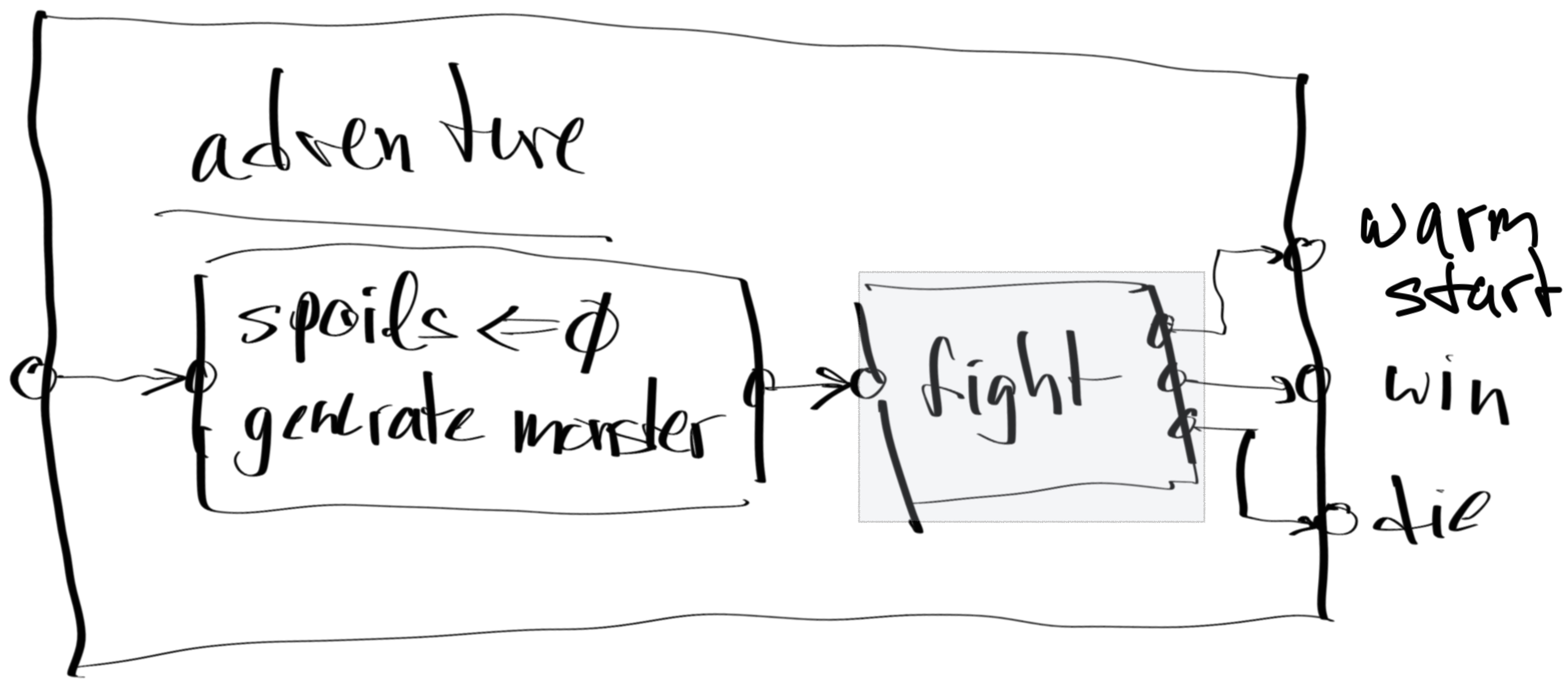


If the player chooses "rest", the player's health is restored to a maximum value, but, 1 day is lost.

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

* p.$ ≥ 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.

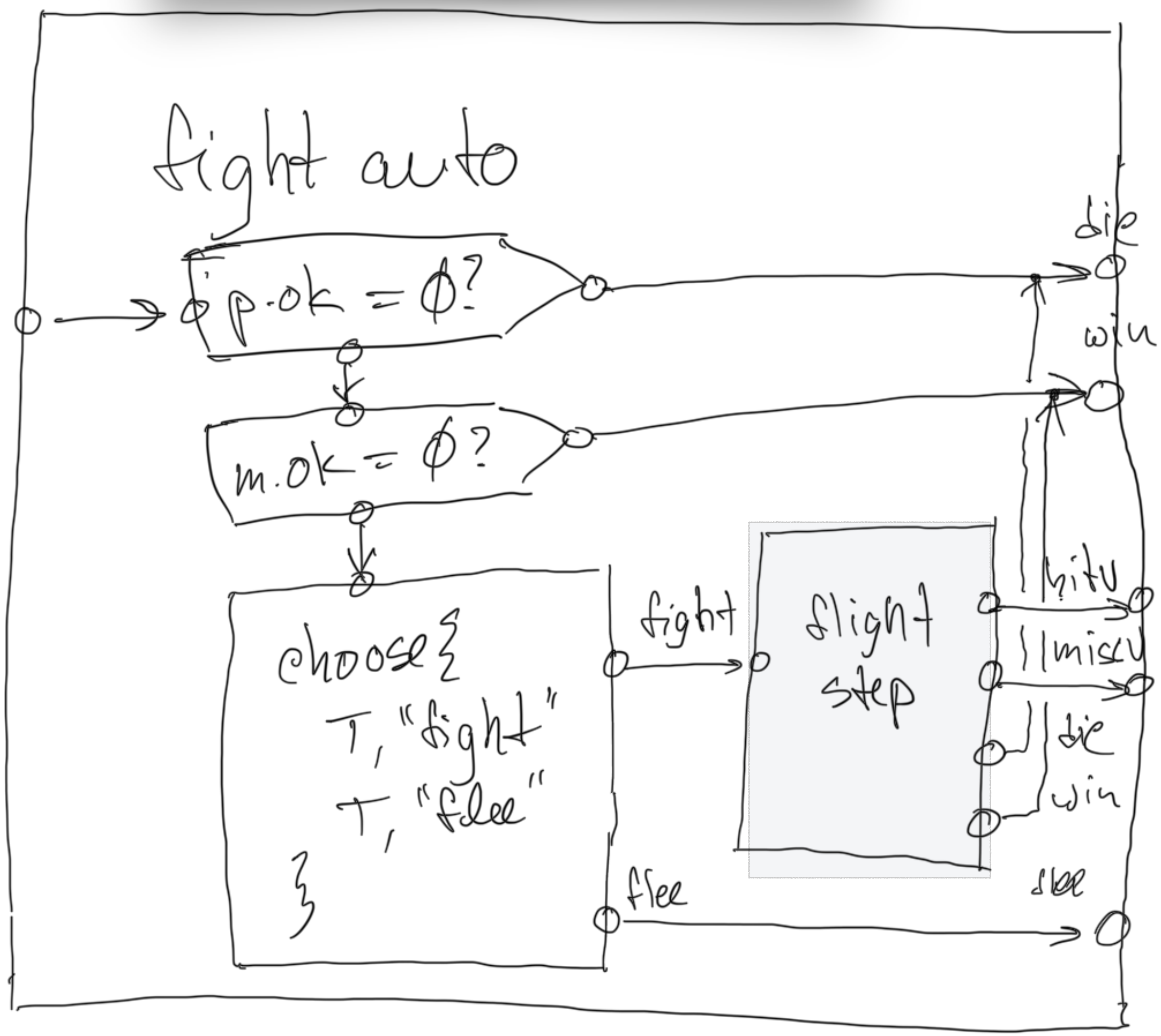
The adventure can end in one of 3 ways

- 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.

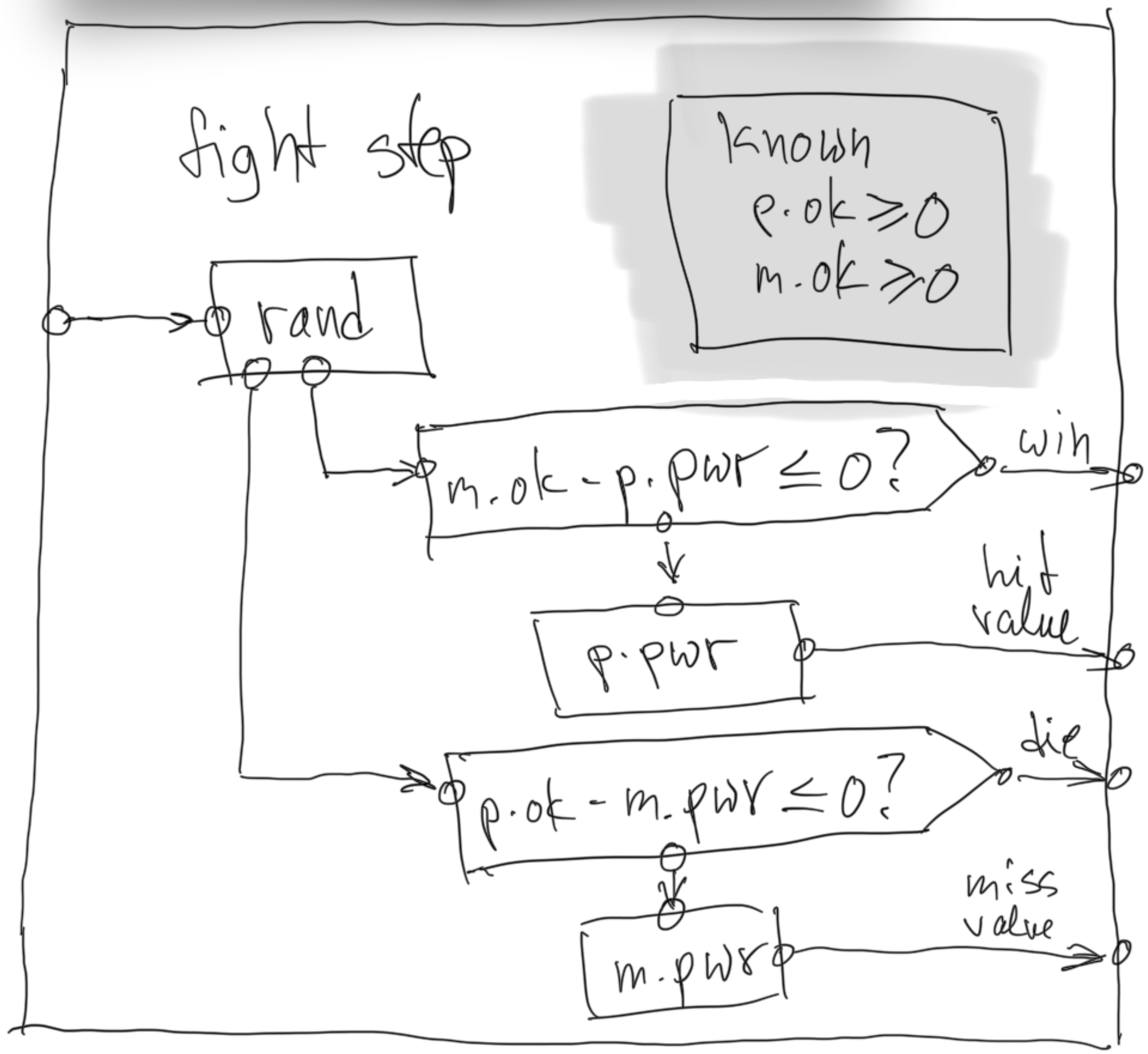
Layer Fight Auto



Fight auto runs the fight loop.

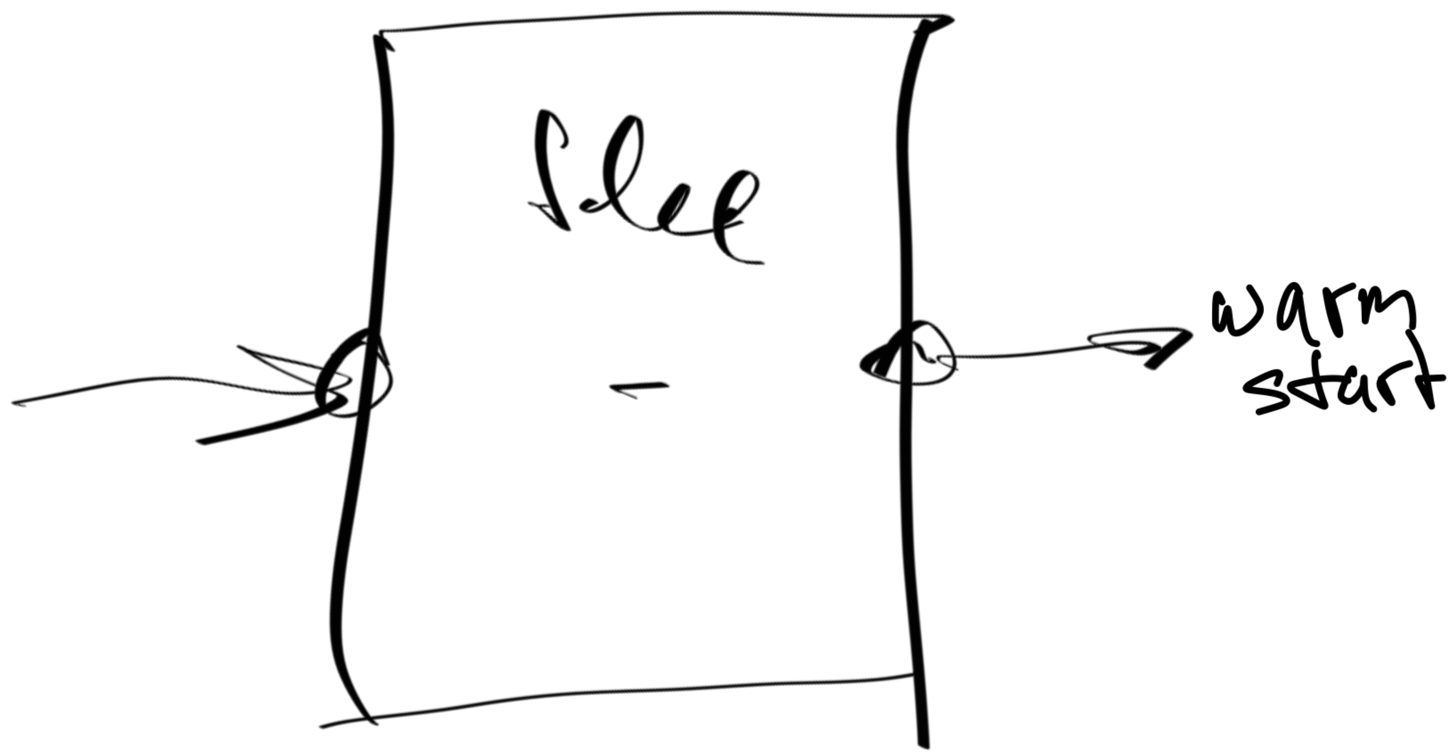
Fight auto returns 1 of 5 possibilities: a hit value, a miss value, a die event, a win event, a flee event.

Layer Fight Step



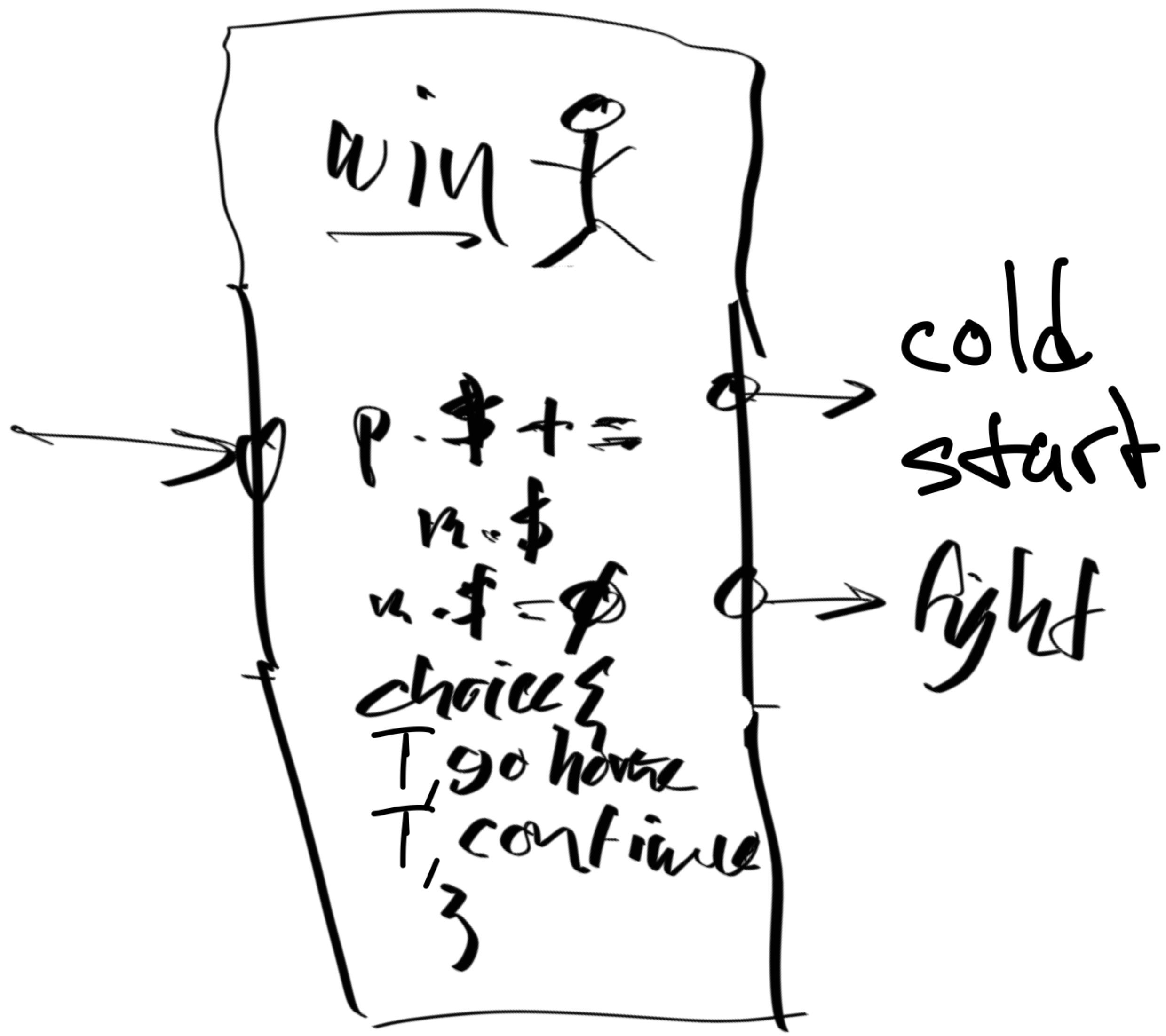
Fight step picks a hit or a miss at random, then produces a “win” or “hit value” event, or, produces a “die” or “miss value” event.

Layer Flee



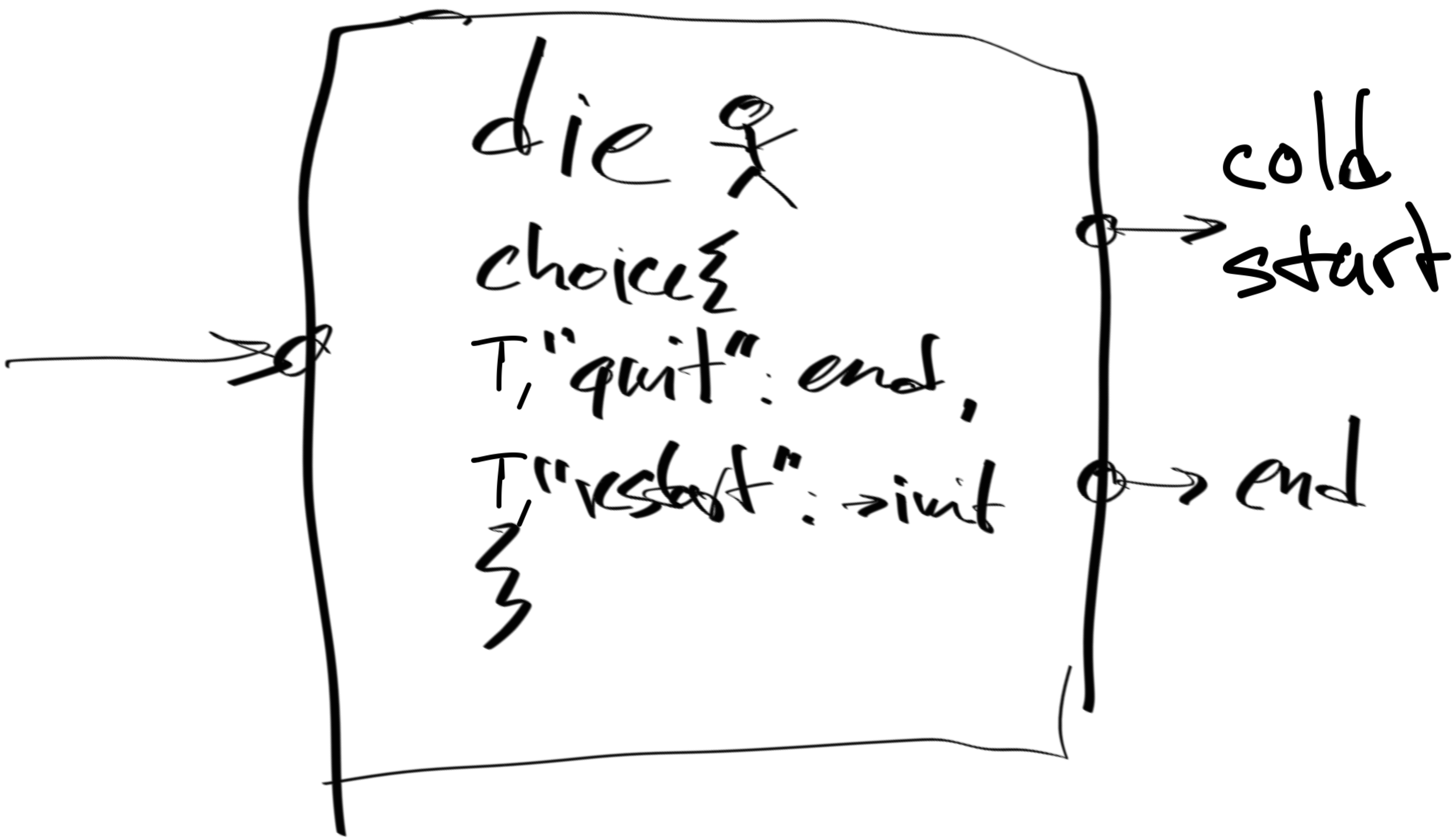
The player foregoes all $'s dropped by the monster and goes back to the main loop.

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

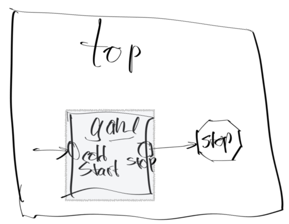


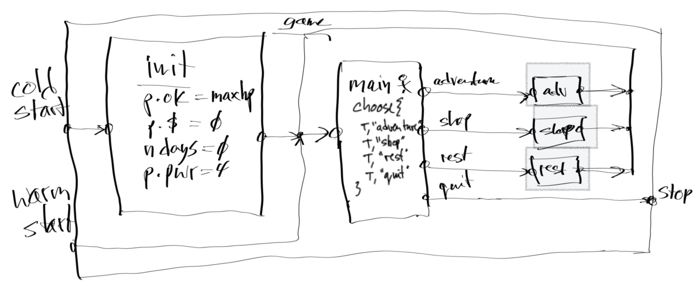
The player lost the game (player's health has dropped to 0). The player can play again or quit the game.

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.
```

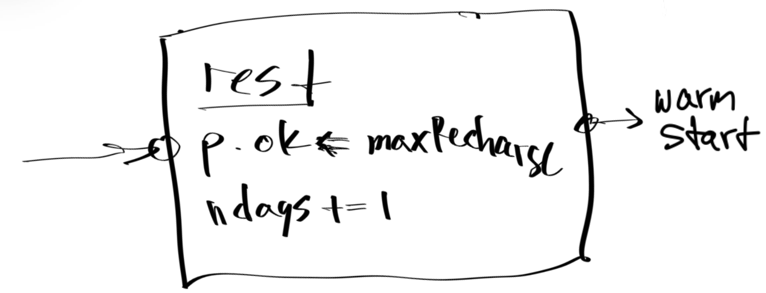
In the main loop of the game, four rules are (always) enabled, since each rule consumes a `main_screen` predicate.

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

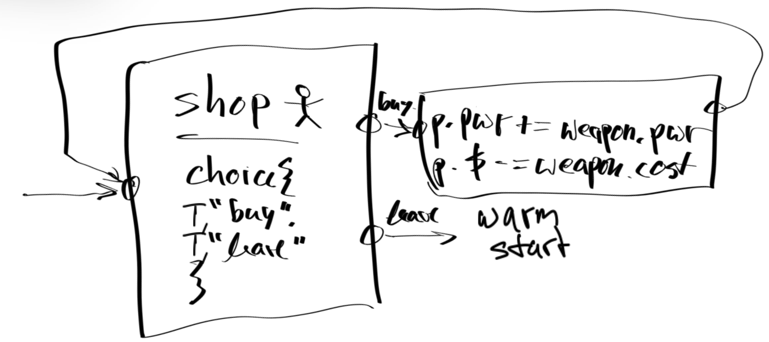
HP
Max
Recharge
N
NDAYS

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 $.

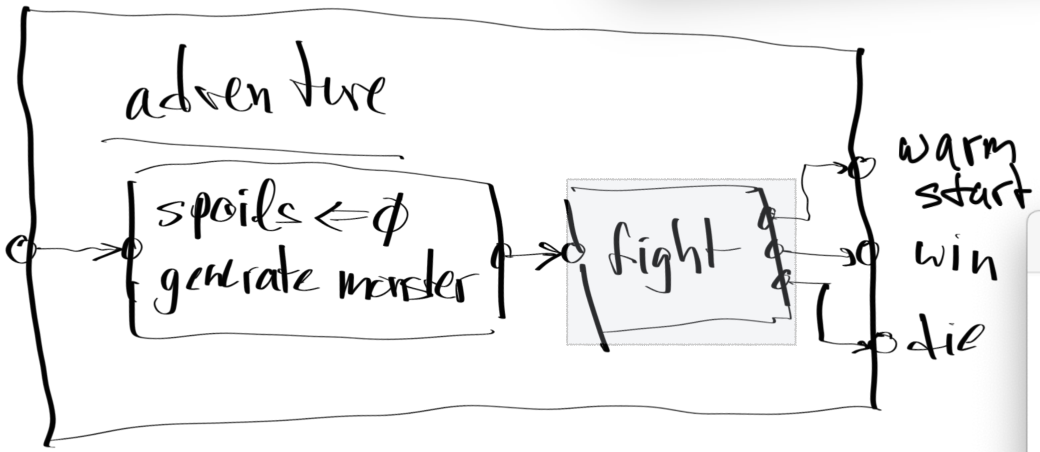
```
stage shop = {
  leave : shop_screen -o main_screen.
  buy : treasure T * cost W C * damage_of W D * weapon_damage _
          * subtract T C (some T')
          -o treasure T' * weapon_damage D.
}
#interactive shop.
qui * stage shop * $main_screen -o stage main.
```

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

let T = p.$ in
let W = any weapon (retry-able)
let C = cost of the given weapon
let D = power of the given weapon
If a weapon can be found that satisfiesT-C >=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`)
}

The rule 'buy' is enabled only if the player has enough $, ie. `(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.
```

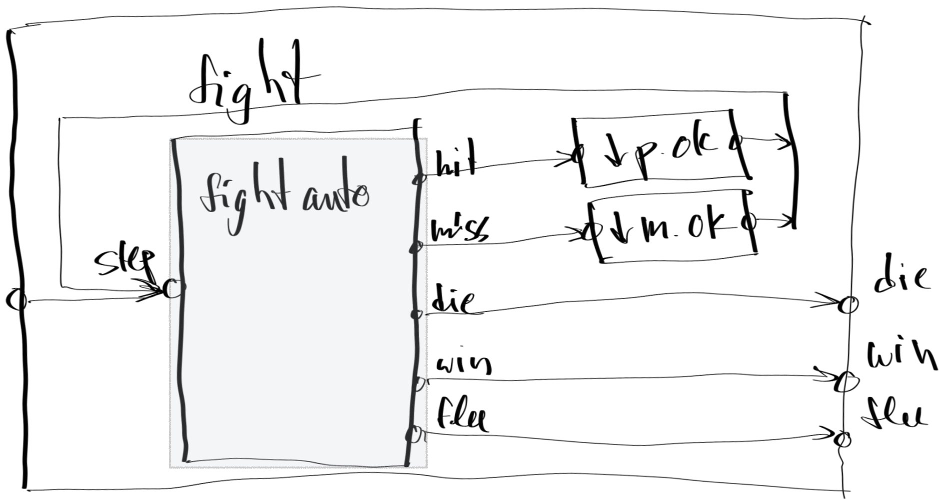
```
stage fight_init = {
  init : fight_screen -o gen_monster * fight_in_progress.
  gen_a_monster : gen_monster * monster_size Size
                    -o monster Size * monster_hp Size.
}
```

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.
```

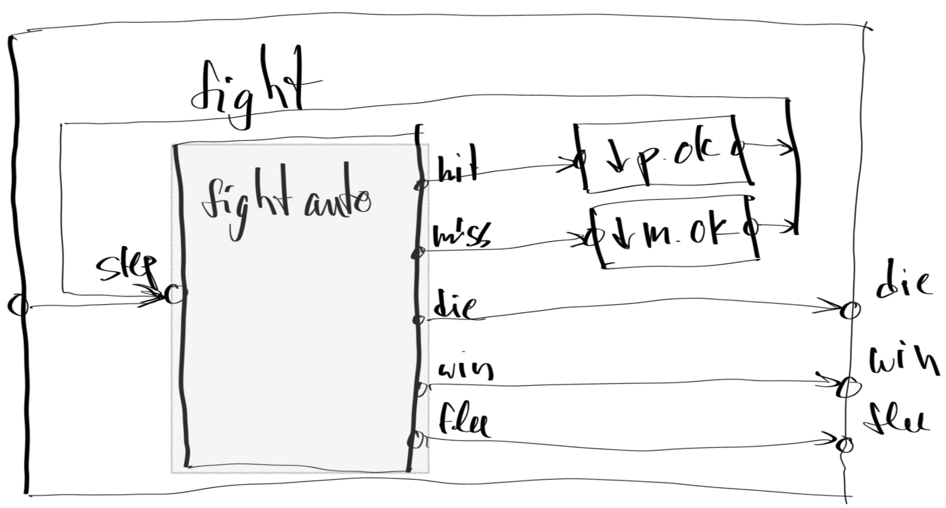
Reverse-engineering this code is overly difficult.

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).

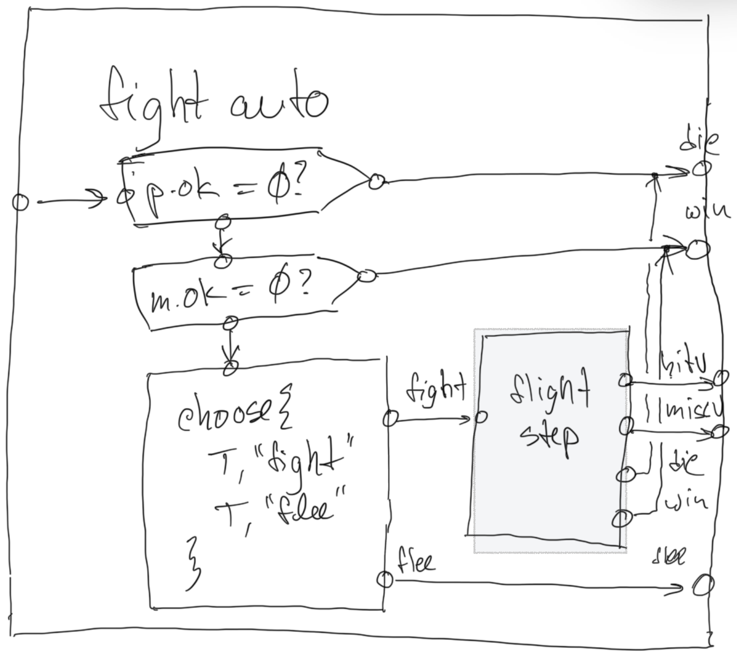
N.B. When 1 output fires, the other 4 outputs produce *nothing*. Not *nil,* not *false* - nothing, no event whatsoever.

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

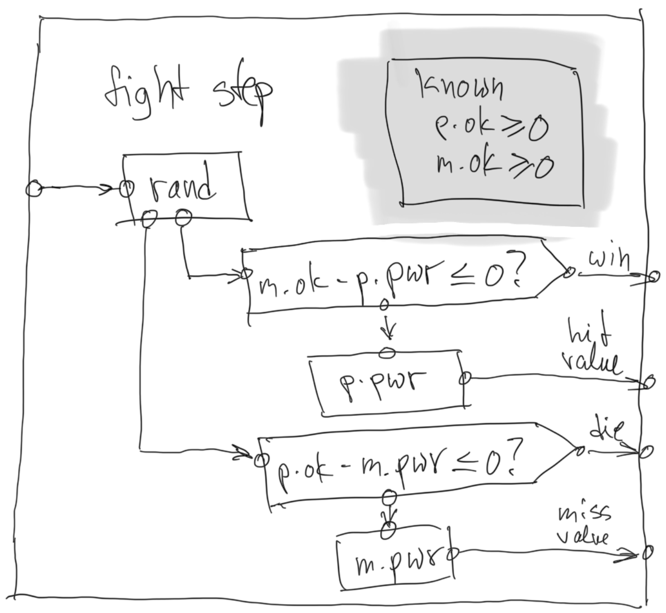



```
stage fight_auto = {
  fight/hit
    : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D (some MHP') -o monster_hp MHP'.
  win
    : fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D none -o win_screen.
  fight/miss
    : try_fight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

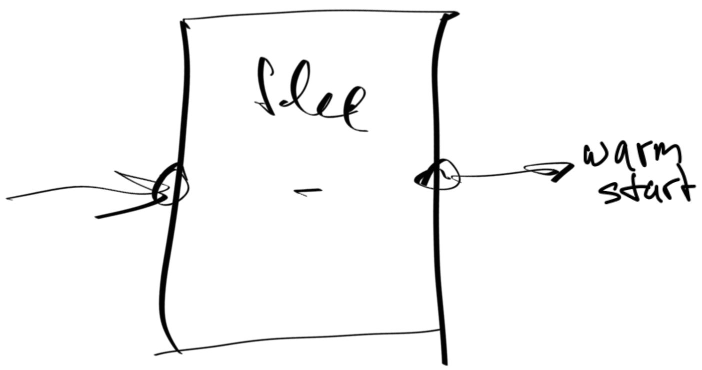


```
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

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.

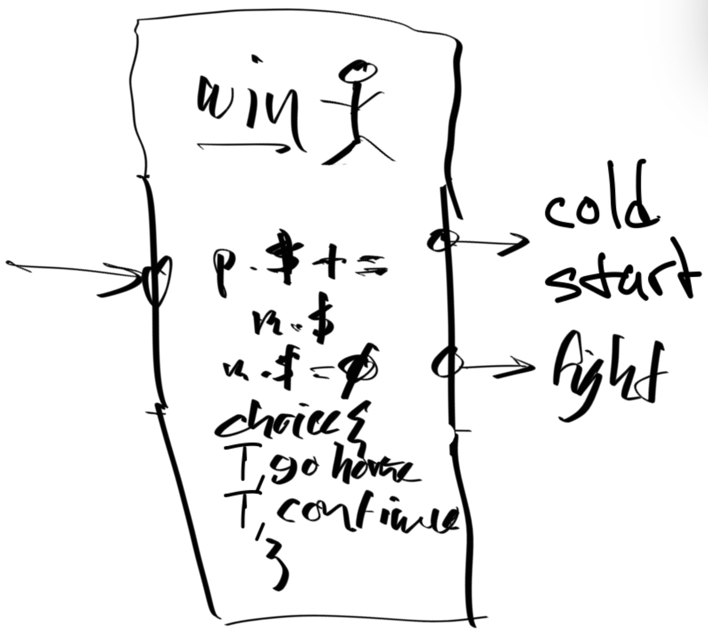
Code Layer Flee




```
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

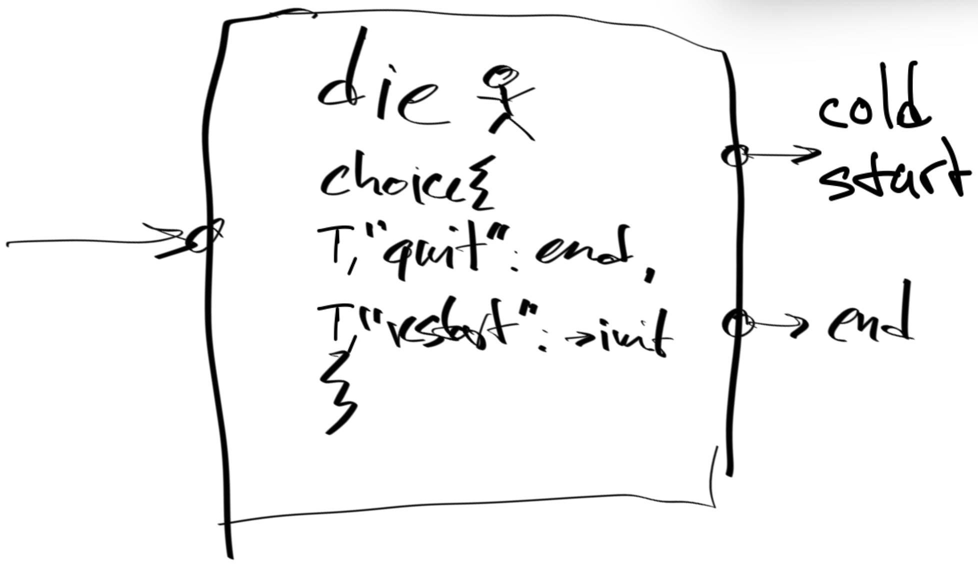



```
go_home_or_continue : pred.
stage win = {
  win : win_screen * monster Size * drop_amount Size Drop
          -o drop Drop.
  collect_spoils : drop X * spoils Y * plus X Y Z
      -o spoils Z * go_home_or_continue.
  go_home : go_home_or_continue
      * spoils X * treasure Y * plus X Y Z
      -o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