Inform ATTACK - Manual
Version 2

Victor Gijsbers (victor@lilith.cc)

April 6, 2011

Contents

Chapter 1

Introduction

1.1 IF and combat

Welcome to the *Inform Advanced Turn-based TActical Combat Kit*, abbreviated as *Inform ATTACK*.[1] ATTACK is a piece of software written in the Inform 7 interactive fiction programming language, to be used by authors who wish to incorporate tactical turn-based combat in their interactive fiction. This is not a stand-alone game. If you are not the author or prospective author of a piece of interactive fiction, this document will be of little use to you.

For a genre that can trace it roots back to *Dungeons & Dragons*,[2] interactive fiction has made curiously little use of the gameplay options provided by tactical combat – options that have been developed and perfected in different ways in many genres of computer games, board games and pen & paper roleplaying games. The heritage of the early interactive fiction is of course evident in non-text games like *Nethack* and in many text-based MUDs; but these are not exactly what we would nowadays call interactive fiction (in the narrower sense). I think that the prevailing attitude among IF authors has been nicely summed up in a recent article by Jimmy Maher:

> Virtually all attempts at creating a satisfying tactical combat engine for IF, for instance, have met with failure, for without a mechanism for knowing where combatants stand in relation to

[1] One of my friends immediately complained that the acronym was not recursive. So, as a special treat for those of you who not only know what HURD stands for but also chuckle about it: you should read "ATTACK" as an acronym for "ADVANCED Turn-based TActical Combat Kit, and "ADVANCED" as an acronym for "ATTACK: D&d Variant for Adaptive Narratives with Computer-Enhanced Dungeon-master".

[2] See the early history of interactive fiction in Jimmy Maher's *Let's tell a story together*: `http://maher.filfre.net/if-book/if-2.htm`.

> one another the engine can allow for little tactical skill from the player. Such systems thus end up feeling like little more than a series of arbitrary die rolls. It has in fact become something of a truism among many players that simulated combat in IF simply cannot work, and that its presence is a justifiable reason not to even attempt a given game.[3]

There are several reasons why there has been little enthusiasm for tactical combat among modern IF authors. As far as I can judge, the most important reasons are these:

- Because of the actual games that have used randomised combat, people have come to loath it as the intrusion of pure chance into what is otherwise a skill-based game. That is, "randomised combat" has become synonymous with "combat where the outcome is purely random" – the typical combat game being one where you type "attack" twenty times in a row, hoping to survive. Such combat is, of course, precisly *not* tactical; but what is in fact a design flaw in such games has come to be perceived as a necessary aspect of randomised combat as such.

- In addition, it has often been remarked that the UNDO and SAVE commands will doom all tactical combat: for either the player can just undo every bad roll of the dice, or the author must deprive the player of two conveniences which have become essential to an enjoyable IF experience. But the idea that the player would resent having no or only restricted access to UNDO and SAVE seems to me unfounded. Indeed, what makes *Nethack* such a great game is *precisely* this restriction. It is not clear that a suitable compromise between tension and convenience cannot be found.

- Finally, interactive fiction authors strive for good prose; and the automatically generated prose of a combat system is thought to be incompatible with this aim. This is an excellent argument. It raises the question whether IF, as a medium, is suited to tactical combat. After all, by choosing IF instead of a graphical medium, one loses certain possibilities – many ways of using geography, for instance. What do we get in return? Two things: prose, and the quasi-unlimited input that a creative player can think up. But if our tactical combat system accepts only a strictly limited range of actions, and if it only spews out

[3] Jimmy Maher, "Lines and Rectangles: Navigating within a Textual Virtual World", `http://www.sparkynet.com/spag/backissues/spag58.html#navigating`. (As the conclusion of the passage makes clear, "virtually all" should be read as "all".)

sentences like "You hit the goblin for 4 damage.", it is definitely not playing to the strenghts of the medium. Although the resulting game might still be fun, one has to wonder why it was written as a piece of interactive fiction.

In my estimation all these problems can be solved. I hope that Inform ATTACK can be a significant part of the solution, and will make it possible to combine interactive fiction and tactical combat in a fruitful way.

1.2 Design principles

Inform ATTACK has been designed on three principles: combat should be tactical; the system should be easily extensible and customisable; and it should be possible to create interesting prose based on what happens during combat.

1. **Combat should be tactical.** Tactical combat is combat where the player can make different decisions, where these decisions significantly affect the flow and outcome of the battle, where smart thinking and courage allow the player to make better decisions, but where it is either hard or impossible to find the best strategy. If any of these conditions is not met, combat will be boring.

   The Inform ATTACK basic combat system already comes with the tactical options, and interplay between these options, to fulfill these conditions. The decisions when to concentrate, when to defend and when to attack are very significant, highly dependent on the circumstances of the moment, but not easy to make even for someone who knows the system well. The initiative system adds another layer of complexity that the wise will be able to use to their advantage.

   A game with more than a handful of fights will certainly want to extend the basic system and add new tactical options, new circumstances, new enemy behaviour, or new items. ATTACK's system is rich enough to make it easy to add such options and ensure that they form a set of live tactical options, rather than a hierarchy of worse and better possibilities. As an example: any system where weapons have only one property, a chance to hit for instance, will have a simple hierarchy of better weapons (with greater chance to hit) and worse weapons (with lower chance to hit). ATTACK gives weapons many standard properties, and in addition makes it very easy to add new rules governing the behaviour of any single weapon or class of weapons.

2. **The system should be easily extendible and customisable.** ATTACK has been designed from the ground up with extensibility and customisablity in mind. Almost the entire combat system is built up out of rulebooks, which are then filled with named rules. It is easy to add rules, replace rules, and remove rules, without having to edit the code of the basic system itself. This makes it possible to define new characters, actions, weapons, circumstances, and so forth, in a completely local manner: you can have all the code pertaining to any of these things in one place, grouped together, even though these rules are called in very different circumstances.

3. **Interesting prose generation should be possible.** Prose generation is governed by a number of rulebooks, which can (and should) be customised and extended at will, and are limited only by how much time the author is willing to invest in writing custom prose for any particular set of circumstances.

   In addition, ATTACK comes with a 'numbers on' and a 'numbers off' mode, which will turn on and off the display of numerical combat calculations based on the preferences of the player or author. It is possible to allow the player to look under the hood at all the modifiers, or to have the game generate straight prose without any numbers intervening at all.

1.3 Some features

- Get started **in minutes**: setting up a basic combat situation will only take a few lines of code, and Inform ATTACK comes with enough complexity to make such a fight interesting in its own right. (Though you will almost certainly want to heavily customise both the system and the prose.)

- An advanced **initiative system** ensures that there is no static order in which the actors take their turns, which makes the combat feel more dynamic and less predictable. Getting more turns than your enemies, or getting them at the most opportune moments, is an important tactical goal.

- An **act/react cycle**, where the player gets to react against attacks initiated against her by a defensive move or some other non-aggressive action, adds to the excitement. It also allows the player to either play it safe, or take big risks for big rewards.

- Six **built-in actions** – attacking, parrying, dodging, concentrating, readying and reloading – provide a basic set of tactical choices that make for interesting combat even before the author adds new possibilities.
- **Artificial intelligence** allows NPCs to choose their enemies, their weapons and their actions in a smart and not entirely predictable way, based on the current circumstances and whatever preferences the author adds.
- A system of **factions** allows you to easily customise who should attack who, and you can change allegiances on the fly.
- The class of **weapons** comes with many interesting properties that allow you to easily define weapons with very different behaviours – from swords and daggers to crossbows and laser guns.

1.4 How to use this document

If you want to understand ATTACK completely, you should read chapter 2, preferably with ATTACK's source code next to it. But if you want to see of what it is capable and want to get a feel for how to do things, chapter 3 might be a better place to start – it offers examples and recipes for some common (and some uncommon) scenarios.

1.5 License

[4] http://www.ifarchive.org/

2. If you *do* choose to release a derivative work under the GPL, you are allowed (but not required) to add these exact same dispensations to its license.

What does this mean? Well, it basically means that if you release your games in the way 99% of all IF authors do, you can use ATTACK without worrying about it any further. If you have any questions, or if you need to use a different license, email me at `victor@lilith.cc`.[5]

The manual you are reading right now is licensed under the GNU Free Documentation License. See `http://www.gnu.org/copyleft/fdl.html`.

1.6 Thanks

Everyone who made Inform 7 possible deserves my praise, as do all the people who stand ready to answer questions on the newsgroup rec.arts.int-fiction, the intfiction.org forum, and the IFMud. Michael Neil Tenuis sent me a large amount of corrections to the text of this manual.

1.7 Version history

- **Version 1; September 28, 2010**. The original ATTACK release. It was labelled as "0.1" in the manual, and had no version number in the extension.

- **Version 2; April 6, 2011**. Adds minor features and fixes a few bugs.

  - Fixed a bug whereby the going action was counted as acting fast, because it triggered the looking action.
  - The *decide whether hate is present* routine now calls a rulebook called the *hate rules*, in which there is one standard rule called the *standard hate rule*. This makes it possible for the author to customise when ATTACK thinks a combat is being fought.

[5] Why, you may ask, is ATTACK not licensed under the Creative Commons Attribution 3.0 license, like all the extensions on the Inform 7 website? Two reasons. One, this extension was a lot of work, and I'm not comfortable with someone using it who isn't planning on letting us all profit from his work in return. Two, the Creative Commons Attribution 3.0 license *is not a software license*. I'm not a lawyer, but it seems almost certain to me that if you stick it on your source code, it only covers that source code, not games compiled from that source code. I want the legal status of ATTACK to be less dubious than that.

- ATTACK now stores its decision about whose turn it is in the *main actor* variable. Thus, if you want something to happen every turn when the player is the primary actor, you can now write an "Every turn if main actor is the player" rule.
- The *handle the combat round* routine now considers a new rulebook called the *starting the combat round rules* after the main actor has been set, but before a prompt has been printed or an AI rulebook has been run. This allows the author to intervene at an otherwise inaccessible point.
- Fixed several typo's in the manual and the Test Dungeon game.
- Solved a bug with the artifical intelligence of the troll in the Test Dungeon game.

Chapter 2

ATTACK: detailed overview

2.1 Preliminaries

2.1.1 Inclusions and use statements

ATTACK needs one extension: the built-in extension *Plurality* by Emily Short, for printing plurals right.

We also define a very short Inform 6 routine which decides *which number is the current_row*; ATTACK uses this internally, but you don't have to worry about it.

2.1.2 Numbers on and off

There are two different ways in which ATTACK can be used: with all the combat-related mathematics hidden under the hood, or with it all out in the open. The first will be preferred by authors and players who don't want any calculations to interrupt the prose of the game; the second will be preferred by players who wish to understand and use the details of the combat system.

To track which mode is preferred, ATTACK defines a truth state called the *numbers boolean*. This is usually true, but can of course be changed by the author. ATTACK also defines two actions that the player can use to change the numbers boolean: *switching the numbers off*, triggered by a command of "numbers off", will run the *standard switching the numbers off rule* and will make the numbers boolean false; *switching the numbers on*, triggered by a command of "numbers on", will run the *standard switching the numbers on rule* and will make the numbers boolean true.

To get a feeling for the difference between these two modes, simply try out these commands in the ATTACK test game.

2.1.3 Some say-phrases

Next, we define several say-phrases, all of which are needed to change the standard printed name of the player from "yourself" to "you". After all, we don't want to see stuff like: "Yourself hits John."

This may seem to be easy enough, since we just declare a rule for printing the name of yourself, the *standard yourself to you rule*, and make it print "you". But this will have the effect of printing "you" in all lowercase letters even at the beginning of a sentence. So we define a number of further say-phrases that will help us avoid this problem. For example, any prose in your game that starts a sentence with the name of the current *global attacker* should not start with "[Global attacker]", but with "[CAP-attacker]". Similarly, we have "[CAP-defender]" for the *global defender*, "[CAP-actor]" for the *global actor*, "[CAP-noun]" for the *noun* and "[CAP-second noun]" for the *second noun*.

2.1.4 Acting fast

As a final preliminary, we define *acting fast*. Any action that should not cost a combat turn ought to be defined as acting fast, e.g.:

> **Example.** If we want singing to take no time, we just add the following:
>
> ```
> Singing is acting fast.
> ```

The actions that ATTACK standardly designates as fast are: examining something, taking inventory, smelling, smelling something, looking, looking under, listening, listening to something, thinking, and all the standard out of world actions. So the player can take inventory, look and examine as much as he likes without being killed by whatever nasty he is currently facing.

We also define the phrase *take no time*. You can call this whenever the current action should count as acting fast; for instance, if an action of the player is stopped by a check rule.

In addition, we need several rules to make sure that going, which automatically calls the looking action, is not counted as acting fast. These are the *first make sure that going is not acting fast* and the *second make sure that going is not acting fast rule*. These rules use the *just moved* boolean, which you will probably not need to mess with yourself.

We don't want the automatic looking action on the first turn to trigger the acting fast machinery; we accomplish this with the *looking at the beginning of the game is not acting fast rule*.

2.2 The basic statistics of actors

2.2.1 Health

Every person has a number called *health*, which is usually 10. It indicates how far the person is from dying.

A person has a number called *permanent health*, which can be used (for instance) to heal the person back to his original amount of health. At the beginning of the game (when play begins), a rule called the *set permanent health to initial health rule* sets the permanent health of every person equal to the health of that person. If you want to have persons whose permanent health is different from their initial health, you must either change one of these values after this rule has run, or you must unlist this rule.

For your convenience, ATTACK provides three phrases for healing persons:

```
Restore the health of Jack.
```

will set the health of Jack to the permanent health of Jack (even if his current health is higher).

```
Fully heal Jack.
```

will set the health of Jack to the permanent health of Jack, but not if his current health is higher.

```
Heal Jack for 4 health.
```

will raise Jack's health by 4, or up to his permanent health, whichever is lower. In addition, it will store the amount of health given to Jack in the global numerical variable *healed amount*, where you can access and print it, if you wish. (It will be overwritten by the next healing.)

> **Example.** Jack's permanent health is 15, but through some means his health has been raised to 20. Now
>
> ```
> Restore the health of Jack.
> ```
>
> will reset Jack's health to 15, while
>
> ```
> Fully heal Jack.
> ```
>
> will do nothing.
>
> **Example.** Jack's permanent health is 15, but his current health is 11. Now

```
Heal Jack for 7 health.
```

will heal him for 4 health, because it cannot heal him above his permanent health. The number 4 is stored in the global variable *healed amount*.

2.2.2 Killed / alive

If at any time the health of the actor becomes 0 or less, that person will be *killed*; otherwise, he is *alive*.

ATTACK does not handle everything you need to do to integrate dead bodies into your game; for instance, in the presence of a killed goblin "goblin, go north" will lead to the somewhat unlikely "The dead goblin has better things to do.". Writing rules to handle these cases cannot be done in ATTACK, because the prose you'll want will be different for each game.

However we do set up a couple of things that almost every author will want to use. We have two rules, the *improper print dead property rule* and the *proper print dead property rule*, that print "dead" before the name of a killed improper-named person or "'s body/bodies" after the name of a killed proper-named person. Thus, "You see John, Queen and a goblin." becomes "You see John's body, Queen's bodies and a dead goblin.". These rules can be circumvented by setting the *printing dead property* to false. ATTACK defines two say-phrases which set this truth state to false and true respectively: *[no dead property]* and *[dead property]*. The recommended use like this:

```
This is some rule:
   say "You kill [no dead property][the noun][dead property].".
```

and in fact, ATTACK uses this scheme to ensure that the player will not see "You kill the dead goblin." after killing the goblin – which would be something of an anti-climax. We also ensure that the phrases "dead", "killed", "body", "body of", "bodies", "bodies of" and "corpse" refer to killed people, and the phrases "alive", "live" and "living" to alive people.[1]

We replace the *can't take people's possenssion rule* by the *can't take living people's possessions rule* – looting dead bodies is a staple of the genre. In order to make looting more feasible, we also add an *after examining a killed person* rule called the *give list of possession on dead person rule*, which prints "On the body/bodies of [the noun] you also see", and then a list of items carried.

[1]Unfortunately, "x John's body" will not be recognised, since "John's" is not recognised as refering to John. I'd like to solve this with "Understand "[something related by equality]'s" as a person.", but that code leads to a fatal error.

2.2.3 Melee and defence

Every person has two basic combat statistics that will, in the absence of more interesting things, determine the probability of hitting. These statistics are two numbers called *melee* and *defence*. The usual values are melee 0 and defence 7.

To score a hit, the attacker's melee + a random number between 1 and 10 must exceed (be strictly higher than) the defender's defence – though both sides of this equation can be modified in various ways. This means that a standard attacker has a 30% chance of hitting a standard defender. It also means that even a small defence bonus can dramatically decrease the chance of hitting, or even lower it to zero; but the basic system's use of concentration and tension has been designed to fit this.

When a hit is scored, the basic damage done is a random number between 1 and the damage die of the weapon used by the attacker (a number defined in the section on weapons; see 2.7).

> **Example.** To create a person whose combat statistics are slightly better than normal, we might type the following.
>
> ```
> Jack is a person.
> The health of Jack is 15.
> The melee of Jack is 2.
> The defence of Jack is 8.
> ```

2.2.4 Factions

Faction is a kind of value, with the standard values *friendly*, *passive* and *hostile*. Every person has a faction; the player is normally friendly, other persons are normally passive. More factions can be added by the author, and the factions of people can be changed on the fly.

Factions can be related to each other by the *hating* relation. In standard ATTACK, friendly hates hostile, and hostile hates friendly. Whether or not factions hate each other can be changed on the fly as well, using a statement like "now neutral hates hostile".

Person A will only attack person B if the faction of A hates the faction of B. So in standard ATTACK, friendly persons will attack hostile persons, hostile persons will attack friendly persons, and all other combinations won't fight. *Passive* is a special faction: passive people will never get a turn during a fight, not even when their faction is made to hate some other faction. (But if the passive faction is hated by some other faction, that other faction *will* attack the poor defenseless passives!)

Example 3. The only thing we need to add to the previous example to make Jack an opponent to our player character is:

```
Jack is hostile.
```

Finally, this section defines a routine to *decide whether hate is present*. This routine returns yes if the *hate rules* succeed, and no if they do not. Based on this decision, we either do or do not run the ATTACK combat round machinery.

The *hate rules* contain one standard rule, a last hate rule known as the *standard hate rule*. It succeeds if at least one alive not passive person enclosed by the location has a faction that hates the faction of at least one alive person enclosed by the location; otherwise, it decides no. In other words, it checks if the factions of any two potential combatants hate each other. If you want ATTACK to behave differently, you can write your own *hate rule*.

2.3 The combat round

Whose turn is it? Which people get to react to the action just taken? These are the questions answered by the routines in this part of ATTACK.

2.3.1 Combat states

Combat state is a kind of value. The values are: *at-Normal*, *at-Act*, *at-React* and *at-Reacted*. Every person has a combat state, which is usually at-Normal. (The 'at-' stands for the name of this extension and is added to avoid namespace clashes with your own code and other extenions. It's not beautiful, but it's not terrible either.)

Every person also has a person variable attached to him called the *provoker*, and an action-name called the *provocation*.

The person who can act this round will be given the at-Act combat state. After the at-Act(ive) person has chosen his action, everyone who is allowed to react to his action (in the basic rules, this will only be the noun of a possible attack action) will be given the at-React combat state. The provoker of each of these persons will be set to the current at-Act(ive) person, and the provocation to the action name of the action taken. (So, if A attacks B, the combat state of B is set to at-React; the provoker of B is set to A; and the provocation of B is set to the attack action. The provoker and the provocation can be used to decide on an appropriate reaction.)

The game then calls on everyone with the at-React combat state, changing their state to at-Reacted after they have declared an action. At the beginning of the next combat round, all combat states are reset to at-Normal.

Many rules check the combat state of a person. For instance, aggressive actions are only allowed to persons with the at-Act combat state.

2.3.2 Initiative

A person has a number called the *initiative*. The initiative of a person is usually 0.

Every round, the person with the highest initiative is the one who will be able to act. If two or more people have the same initiative, ATTACK will decide randomly which of them can act.

The *initiative rules* rulebook is called at the beginning of each combat round, just before the decision is made who can act. Not all rules concerning initiative should be in this rulebook, just the ones that should be called at the beginning of each combat round. (One example of a rule that should be called at other moments will be given shortly.)

The standard system fills the initiative rulebook with three rules, all of which apply to all alive persons enclosed by the location:

1. A *first initiative rule* called the *no low initiative trap rule* sets any initiative lower than -2 to -2. This ensures that nobody can fall too far down in the initiative order.

2. An *initiative rule* called the *increase initiative every round rule* increases everyone's initiative by 2. ("But that doesn't change the initiative order!", you may object. It does in combination with the fact that after attacking, your initiative is reset to 0.)

3. An *initiative rule* called the *random initiative rule* adds between 0 and 2 initiative to everyone's initiative (randomly calculated for everyone separately), to add some randomness to the combat order.

One additional rule concerned with initiative is defined in this part of ATTACK, namely an *aftereffects rule* (a rulebook discussed in subsection 2.4.3) called the *modify initiative based on combat results rule*. It decreases the initiative of the defender by 2 if he was damaged, and that of the attacker by 2 if he failed to damage the defender. This reflects the fact that a successful attack puts you at an advantage, whereas an unsuccessful attack puts you at a disadvantage.

2.3.3 Setting up the combat order

The combat order is decided in the *set up the combat order* routine by setting up a table called the *Table of Combat Order* which contains all alive not passive persons enclosed by the location (the *Combatant entry*) and their initiative (the *Move Order* entry). This table is first sorted in random order and then sorted in reverse intiative order – with the result that a random person with the highest initiative is in the first row. (Only the first row will ever be used; for the next round the entire table is constructed anew, since everything may have changed.)

The *Table of Combat Order* has 50 rows. If your game will ever have more than 50 alive not passive persons in one location, you need to extend this. (And perhaps rethink your design!)

The *set up the combat order* routine also resets the combat state of every combatant to at-Normal.

2.3.4 Main combat round routines

The main combat round routines call the right AI routines to make NPCs act, and govern when the player gets to act. Although it is somewhat unlikely that you would want to mess with this part of the code (which should only be done if you really know what you're doing), understanding the turn sequence is essential to understanding how ATTACK works.

Every combat round is the equivalent of an Inform 7 turn, even those rounds where the player doesn't get to do anything – in such cases, we merely skip the parsing routines, but everything else (running every turn rules, advancing the time, checking timed events, and so on) takes place as normal.

Free actions – actions that you can do without them taking up your turn – are also the equivalent of an Inform 7 turn, but do skip some of the standard rulebooks. Looking, smelling, listening, thinking, examining and taking inventory are defined as such free actions. These actions have a property called *acting fast*, and if you take them, the rules for this individual action are followed and then all the other rules are skipped and a command prompt is printed again. Every turn rules do not run, the time is not advanced, initiative is not recalculated, and so on. The same thing happens when we tell Inform to "take no time", which is as if we could say "act as if the current action is acting fast" (which we cannot, since it is illegal syntax). See for more information subsection 2.1.4.

We will now look at the turn sequence of Inform ATTACK, paying special attention to those rules that are not in standard Inform 7.

1. The **govern combat first part rule** gets called first. If we are acting fast, it does nothing. Otherwise, it does two things. First, it resets some variables, changing the command prompt back to ">" and setting the combat state of all present combatants back to at-Normal. Second, it checks whether hate is present (i.e., whether a fight is going on, see subsection 2.2.4), and if so it runs the two routines *set up the combat order* and *handle the combat round*. It also sets the *main actor* variable (either directly or through the *handle the combat round* routine): this now contains the name of the person whose turn it is to act. This can be useful for writing "every turn" rules.

2. The second rule is the **dont parse command when not the players turn rule**. This ensures that if the player's combat state is at-Normal (i.e., neither acting nor reacting), there will be no command prompt. If we are acting fast, the rule will instead do nothing.[2]

3. After that, we have the standard Inform 7 **parse command rule**, followed by the **generate action rule**. This handles whatever the player wants to do.

4. Then, we get ATTACK's **govern combat second part rule**. If we are acting fast, this rule will do nothing. Otherwise, it will change the player's combat state from at-React to at-Reacted (if applicable); then it will run the *have a reaction* routine; and finally, it will *run the combat*.

5. At this point, we get to the **acting fast rule**, which stops the turn sequence if we are acting fast. Everything after this rule will not be reached if we are acting fast, and everything before this rule except for the parse/generate-action rules turns itself off if we are acting fast. Thus, if we are acting fast, we will skip everything until we get to the next parse command rule.

6. From now on, we have all the usual Inform 7 rules: the **every turn stage rule** is called and will run through the every turn rulebook; timed events take place, the time is advanced, **and so on**.

We see that four specially defined clauses or routines are called: *set up the combat order* was already talked about in subsection 2.3.3; we will presently

[2]Normally, this shouldn't make a difference. We should never be in a situation where acting fast changes the combat state of the player – but if an author accidentally makes this happen, the current clause will stop the interpreter from getting into an infinite loop.

discuss the other three, *handle the combat round*, *have a reaction*, and *run the combat*.

2.3.5 Handling the combat round

The routine called *to handle the combat round* is the routine that makes the attacker act. It chooses the first row in the Table of Combat Order, which has previously been set up by the *set up the combat order* routine. It then puts the name found in the row into the global variable called *global attacker*, and sets the initiative of the global attacker to 0 and his combat state to at-Act.

Then, the *starting the combat round rules* are considered. This rulebook is empty in standard ATTACK, but can be used to intervene at this point of the combat sequence.

If the global attacker is the player, it changes the command prompt to "Act>" and stops – from this point, the parse command rule will take over.

If the global attacker is someone else, his combat AI rulebook is called. This rulebook will decide what the global attacker does, and will carry out the action, and/or store it for being carried out later by the *run the combat routine*. (Attacks, for instance, are carried out in two parts, with the defender having a chance to react in between.) The AI is discussed in section 2.6.

If the action of a non-player global attacker has led to the player having the combat state at-React (for instance, the global attacker has attacked the player), the command prompt is changed to "React>" – from this point, the parse command rule will take over.

2.3.6 Having a reaction

The *have a reaction* routine runs through the Table of Combat Order and makes everyone whose combat state is at-React do something, after which it changes their combat state to at-Reacted. In order to make a person do something, it puts his name into the global variable called *global attacker*, and then calls his combat AI rulebook.

The AI routines will be able to see whether they have to devise an action or a reaction by checking the combat state of the global attacker – if this is at-Act, they must come up with an action, if it is at-React, they must come up with a reaction. It should not be possible to have any other state at this point.

2.3.7 Running the combat

Some actions must be carried out in two different phases. For instance, if A attacks B, we have the following steps:

1. A declares his attack.
2. B gets to react.
3. A carries out his attack.

ATTACK handles this in the following way:

1. The *fight consequences variable* is false. We try the action "A attacking B", the behaviour of which depends on the fight consequences variable. We store the action "A attacking B" in the *Table of Stored Combat Actions*.
2. B's combat AI rulebook is followed.
3. The *fight consequences variable* is now set to true. We try all the actions in the *Table of Stored Combat Actions* (after sorting it in *Combat Speed* order, although this has no effect in standard ATTACK), which in this case will include the action of "A attacking B". This action will now have a different effect, because the fight consequences variable has a different value. Finally, we reset the *fight consequences variable* to false.

It is the third part of this scheme, where we sort the Table of Stored Combat Actions, then run through it, try all the actions put into it, and finally blank it out, that is handled by the *run the combat* routine.

In standard ATTACK, only the attack action follows this scheme, but authors will want to use it for any custom actions that allow a reaction. See also the section on attacking, 2.5.2.

2.4 Striking a blow

The *striking a blow* routine is where the real action takes place: dice are rolled, damage is calculated, life and death are decided. It is a very complicated routine that calls on many rulebooks, seventeen of which are interesting for authors. They are the entrypoints where you can intervene in the calculations and in how the results are reported. Most of your customisation of ATTACK will consists of rules that are to be placed in one of these seventeen rulebooks, so it pays to understand them.

2.4.1 A few preliminaries

At this point, the source defines a number of global variables as well as the weapon kind and the readied property.

The *global attacker* and the *global defender* are persons that vary. Basically, any routine in the game that gets called can assume that the global attacker is the person attacking and the global defender is the person defending.

A *weapon* is a kind of thing. Weapons can be *readied* or *not readied*, and are usually not readied. Readying a weapon is an action. The readied weapon is the weapon one uses in attacks. For more on weapons, see section 2.7.

Two global variables that are weapons that vary are also defined: the *global attacker weapon* is the weapon with which the global attacker attacks, while the *global defender weapon* is the weapon with which the global defender defends.

Several numbers that vary that are used in combat calculations are defined here:

- The *to-hit roll* is the die roll made to see whether the attacker hits the defender.
- The *to-hit modifier* is the bonus that is to be added to the to-hit roll.
- The *damage* is the amount of damage the attack will do.
- The *damage modifier* is the bonus that is to be added to the damage.
- The *final damage* is the amount of damage that the defender actually sustains in the end – i. e., after everything which can add or subtract damage has been taken into account. This is an important variable, because (among other things) the printed text and whether or not you lose concentration depend on whether this number is greater than 0 or not.

2.4.2 The big routine

During *run the combat* (see 2.3.7), attack actions will end up calling the *make A strike a blow against B* routine, where A is the attacker and B is the defender. This routine is complicated, so I will first give an overview of its logic, and then we will discuss the important parts one by one. Those important parts are the seventeen entry points – places (rulebooks) in the

striking a blow routine where the author of a game may want to intervene and change things.

- First, the *global attacker* and the *global defender* are set to the two persons identified in the call to the routine.
- Next, the pressing relation is made true between the attacker and the defender. (We will come back to this in section 2.6.4; the basic idea is to remember who someone has been attacking and to give that person a greater chance to choose the same target in the next round. This makes for consistency in larger fights.)
- We then set the *global attacker weapon* and the *global defender weapon* to random readied weapons enclosed by the global attacker and global defender respectively. (No person should be able to have more than one readied weapon, but if through some coding mistake they do, the game just chooses one to use randomly.)
- We now consider the *reset combat variables* rules, which reset all the values and modifiers stored in the last round to their default values.
- **Entry point 1: does attacking begin?** We now abide by the *whether attacking begins rules*. If these rules decide no, the entire routine is stopped right here, and no blow is struck.
- **Entry point 2: preliminary results of attacking.** Anything that happens just because an attack is going to take place happens here, in the *preliminary results of attacking rules*.
- **Entry point 3: basic attack roll.** We roll the die to see whether the attacker will hit in the *basic attack roll rules*.
- **Entry point 4: apply the attack modifiers.** We calculate and apply bonuses and maluses on the attack roll in the *attack modifiers rules*.
- **Entry point 5: calculate results of the attack roll.** The *calculate results of the attack roll rules* calculate the final numerical result of the attack roll.
- **Entry point 6: show results of the attack roll.** The *show results of the attack roll rules* give the player feedback on the results of the attack roll, if desired.

- **Entry point 7: did the attack hit?** The *whether the attack hit rules* decide if the attack hit or not; if they decide no, entry points 7 to 12 are skipped.

- **Entry point 8: immediate results of hitting.** The *immediate results of hitting rules* can do anything that ought to happen before damage is applied or even calculated.

- **Entry point 9: rolling the die for damage.** In the *basic damage roll rules*, the die is rolled to decide how much damage the hit will do.

- **Entry point 10: modifying the damage.** We now, in the *damage modifiers rules*, add or subtract whatever bonuses or maluses apply to the damage.

- **Entry point 11: calculating the final damage.** At this stage, the *calculate the final damage rules* decide what the final damage is and write it to the *final damage* global variable.

- **Entry point 12: showing the damage.** The *show the final damage rules* can print something to the screen at this point (and will do so in standard ATTACK if the numbers boolean is true).

- **Entry point 13: pre-prose-generation effects.** Anything that needs to be done or determined before we can start printing the results of the blow to the screen is done here.

- **Entry point 14: reporting the results of the blow.** The most important prose generating rulebook, the *flavour text rulebook*, is now followed.

- **Entry point 15: aftereffects.** Unless the player is killed, we now follow the *aftereffect rules*, which govern special results of the attack. (We don't need to do this if the player is dead, and don't want to print anything after the death message.)

- **Entry point 16: remove temporary circumstances.** Now the *take away until attack circumstances rules* reset any bonuses or circumstances that applied only until an attack was made (or defended against). For instance, the attacker's concentration is reset.

- **Entry point 17: final report.** At the final stage, the *final blow report rules* can add any additional prose they wish.

Why do we have so many entry points, when the same results could be gotten with a single rulebook? The point is of course to make it possible to easily position rules at the right points in the process, when the right calculations have been made and the right prose has been printed to the screen. We will discuss examples of rules that go or might go into each of the seventeen rulebooks in the next section.

2.4.3 The seventeen entry points in detail

Entry point 1: does attacking begin?

The *whether attacking begins rules* can preempt an attack from happening before any calculations are made and anything is printed to the screen. You would want to use this to stop a chained player from attacking anyone, for instance, especially if he can become chained during the reaction to his attack. (This can be preferable to having a *check attacking* rule, because there might be other actions that call the striking a blow routine.) If the rules decide no, the attack is stopped.

Standard ATTACK leaves the *whether attacking begins rulebook* empty.

Entry point 2: preliminary results of attacking

The *preliminary results of attacking rules* handle anything that happens as soon as an attack is actually taking place and is certain not to be preempted. This is primarily useful for taking away bonuses that should no longer apply even in the current round; e.g., the monk's divine powers that he has gotten in return for his vow of pacifism.

Standard ATTACK leaves the *preliminary results of attacking rulebook* empty.

Entry point 3: basic attack roll

The *basic attack roll rules* generate a new value for the *to-hit roll*. You will want to intervene here if you want to change ATTACK's standard mechanism (which generates a random number between 1 and 10), for instance, by generating a number between 1 and 20, or adding up the results of three six-sided dice for a more Gaussian distribution. You will also want to intervene here if you wish to print non-standard text here.

Standard ATTACK has one rule in the *basic attack roll rulebook*:

1. The *standard attack roll rule* generates a random number between 1 and 10. It also prints "Rolling [the number rolled]" if the numbers

boolean is true, i.e., if either player or author has stated that they wish to see the numerical calculations.

Entry point 4: apply the attack modifiers

One of the rulebooks the author will wish to use most often are the *attack modifiers rules*. Here we apply bonuses and maluses to the attack roll, usually by adding or subtracting a certain amount from the *to-hit modifier* (a number that varies). Any rule in this rulebook should also print out the applied modifier, with an explanation, if the *numbers boolean* is true.

Standard ATTACK comes with six rules in the *attack modifiers rulebook*:

1. The *melee attack bonus rule* adds the melee of the global attacker to the to-hit modifier.

2. The *concentration attack modifier rule* adds a concentration bonus to the to-hit modifier; see section 2.5.3 for more details.

3. The *parry defence bonus rule* subtracts a certain number from the to-hit modifier if the defender has chosen to parry the attack; see section 2.5.4 for more details.

4. The *dodge defence bonus rule* subtracts a certain number from the to-hit modifier if the defender has chosen to dodge the attack; see section 2.5.5 for more details.

5. The *attack bonus from weapon rule* adds or subtracts the *weapon attack bonus* of the global attacker's weapon.

6. The *standard tension attack modifier rule* adds one half (rounded down) of the current tension; see section 2.8.2 for more details.

Entry point 5: calculate results of the attack roll

This rulebook calculates the final result of the attack roll. It is somewhat unlikely that you will want to change or add to the single rule that ATTACK contains in the *calculate results of the attack roll rulebook*:

1. The *standard calculate results of the attack roll rule* simply adds the *to-hit modifier* to the *to-hit roll*.

Entry point 6: show results of the attack roll

This is a prose generation stage that shows the final result of the attack roll. Standard ATTACK contains one rule in the *show results of the attack roll rulebook*:

1. The *standard show results of the attack roll rule* prints an "=" and the result of the attack roll (if the numbers boolean is true).

Entry point 7: did the attack hit?

The *whether the attack hit rules* decide whether the attack did or did not hit. This is where you want to intervene for any last-minute possibilities of hitting or failing to hit. Effects that grant a certain probability of escaping a blow that would have otherwise connected (such as *Dungeons & Dragons 3's* concealment or *Diablo 2's* chance of blocking) could be implemented here.

If the attack does hit, the rules should succeed; if it does not, the rules should fail. Entry points 7 to 12 are skipped if the rules fail.

Standard ATTACK has one rule in the *whether the attack hit rulebook*:

1. The *standard whether the attack hit rule* succeeds if and only if the to-hit roll is greater than the defence of the global defender. It also finishes the text output of the attack roll (if the numbers boolean is true).

Entry point 8: immediate results of hitting

The *immediate results of hitting rules* should contain anything that happens when an attack hits, but before damage is calculated. If hitting a magma elemental melts your weapons so that they lose their edge and now have reduced damage, starting with the current damage roll, this is the place to make it happen.

Standard ATTACK leaves the *immediate results of hitting rulebook* empty.

Entry point 9: rolling the die for damage

The *basic damage roll rules* roll the die to see how much damage will be applied. If you want to do anything fancier than the standard roll from 1 to the attacker's weapon's damage die, this is the place to be – for instance, rolling two six-sided dice, making damage non-random, rolling twice and applying the higher damage, and so on.

Standard ATTACK has one rule in the *basic damage roll rulebook*:

1. The *standard damage roll rule* chooses a random number between 1 and the damage die of the global attacker weapon. It also starts printing the results of the damage calculations (if the numbers boolean is true).

Entry point 10: modifying the damage

This is another rulebook which authors will want to customise heavily: the *damage modifiers rules* handle all the bonuses and maluses that have to be applied to the damage roll. Weapons, skills, circumstances that allow you to do more damage; armour or spells that absorb damage; this is where you put them.

Standard ATTACK has two rules in the *basic damage modifiers rulebook*:

1. The *concentration damage modifier rule* adds a concentration bonus to the damage modifier; see section 2.5.3 for more details.

2. The *standard tension damage modifier rule* adds one third of the current tension (rounded down) to the damage modifier; see section 2.8.2 for more details.

Entry point 11: calculating the final damage

The *calculate the final damage rules* decide on what the final damage will be, after modifiers have been applied to the damage roll. This is the place to intervene if you want to have effects that multiply the final damage by a certain number (a curse that doubles all damage dealt to the player, for instance), or that are dependent on the exact damage dealt (somewhat unlikely, but think of a monster that is immune to any amount of damage that is not a prime number).

Standard ATTACK has one rule in the *calculate the final damage rulebook*:

1. The *standard calculate the final damage rule* sets the final damage to be the sum of the damage and the damage modifier; unless this sum is less than 0, in which case the final damage is set to 0. (There can be no negative damage in standard ATTACK.)

Entry point 12: showing the damage

If desired, the *show the final damage rules* will print the numerical result of the damage roll. Anything that should be printed to the screen when we are not looking purely at the numbers should be put in the *print flavour text rules*, entry point 14.

There are two rules in the *show the final damage rulebook* in standard ATTACK:

1. The *standard show the final damage rule* prints "= n damage" (if the numbers boolean is true).

2. The *standard report result of blow in numbers mode* rule finishes the numerical prose generation by appending a phrase like "allowing you to espace unscathed", "wounding [the opponent] to [n] health", or "killing you" – based on whether the attack hit, did fatal damage, and was initiated against the player or not.

Entry point 13: pre-prose-generation effects

In the *aftereffects before flavour text rules*, we do whatever needs to be done before we can print the main prose. Only rules that will actually change the prose generated should be put here; all the others should be deferred to entry point 15, the *aftereffects rules*.

For example: if you have a weapon that has an $n\%$ probability of destroying undead completely in a huge flash of light, where n is the damage dealt to the undead by the blow, this is the place to put a rule concerning this: the final damage has already been calculated, and whether or not the undead is destroyed will obviously have to be reflected in the prose generated.

Standard ATTACK has just one rule here, which is rather essential (and allows all subsequent rules to refer to the new health of the defender):

1. The *subtract damage from health rule* does exactly what it says: it subtracts the final damage from the health of the global defender. (We do not need to make the global defender "killed" if his health drops below 1, because being killed is *defined* as having 0 or less health.)

Entry point 14: reporting the results of the blow

The *print flavour text rules* are very important: they generate the prose at the end of the blow. Nevertheless, you probably don't want to write any print flavour text rules. ATTACK uses a print flavour text rule called the *flavour text structure rule* to call four further rulebooks, and these are the ones you should add your own custom rules to. The structure is the following. First, we abide by the *intervening flavour text rules*. If these decide no, the rest of the rules is not run. If they decide yes (or nothing), one of the three other rulebooks is run. If the global defender is alive, we run the *flavour rulebook*. If the global defender is dead and the global defender is the player, we run

the *fatal player flavour rulebook*; if he is not the player, we run the *fatal flavour rulebook*.

There is a fifth rulebook called the *attack move flavour rules*. This rulebook gets called whenever an actor initiates an attack – it is what you see before you have to react.

Four of these five rulebooks have one standard rule in them, as a backup. In general, you will always want to intervene and make the game print something more interesting than the bland description in the standard rule. Since what you want to print will depend heavily on your particular game, ATTACK cannot provide detailed rules for you.

The four pre-defined backup rules are:

- A *last flavour rule* called the *basic flavour rule*, which prints either "*A* misses *B*." or "*A* hits *B*.", as appropriate.
- A *last fatal player flavour rule* called the *basic fatal player flavour rule*, which prints "You are killed by *B*.".
- A *last fatal flavour rule* called the *basic fatal flavour rule*, which prints "*A* kills *B*.".
- And a *last attack move flavour rule* called the *basic attack move flavour rule*, which prints "*A* lunges towards *B*.".

Entry point 15: aftereffects

The *aftereffects rules* handle everything that results from the blow and is not the simple giving of damage. Does the defender get poisoned, frozen, burned? Does he lose his concentration? Will the attacker gain health using his vampiric weapon? Does the defender drop his weapon and beg for mercy? Does he die and explode, dealing damage to all other persons? Does the attacker's weapon or the defender's shield break?

Standard ATTACK comes with four rules in the *aftereffects rulebook*:

1. The *modify initiative based on combat results rule* decreases the initiative of the defender by 2 if he was damaged, and that of the attacker by 2 if he failed to damage the defender.
2. The *lose concentration when hit rule* takes away the defender's concentration if he was dealt any damage.
3. The *decrease ammo rule* decreases the amount of ammunition in the attacker's weapon by 1 (if it is a weapon that uses ammunition). See section 2.8.1 for more information on weapons with ammunition.

4. The *standard reduce tension after hit rule* decreases the tension when damage has been dealt; see section 2.8.2 for more information.

Entry point 16: remove temporary circumstances

Here, in the *take away until attack circumstances rules*, we reset any circumstances that lasted only until the attack was finished. Most special attacks or defences will be implemented using a variable that is set to true when the action is declared, and set to false by this rulebook. For instance, if you parry, the player gets the *at parry* property. During the attack, rules can check whether the defender is at parry. At the end of the attack, in the current rulebook, the player gets the *not at parry* property.

Standard ATTACK has three rules in the *take away until attack circumstances rulebook*:

1. The *lose concentration after attacking rule* removes the concentration of the attacker.

2. The *no longer at parry after the attack rule* removes the *at parry* property from the defender (which he will have if he parried).

3. The *no longer at dodge after the attack rule* removes the *at dodge* property from the defender (which he will have if he dodged).

Entry point 17: final report

The final entry point is the *final blow report rules*. Here you can do any reporting that had to be saved till last, so you can be certain that it will be printed after everything else.

Standard ATTACK has one rule in the *final blow report rulebook*:

1. The *end reporting blow with paragraph break rule* comes *last* in the final blow report rulebook; it prints a paragraph break. (It is assumed that no other rules print paragraph breaks, since they do not know whether the next rule wants to run on or wants to start reporting on a new paragraph.)

2.4.4 Which entry points should I use?

All of that may be a little overwhelming if you are using ATTACK for the first time! Each of the entry points can be useful, and if you wish to use ATTACK extensively you will want to understand them all. But you can do most things with just a handful of the entry points, namely the following:

- The *attack modifiers rulebook* (entry point 4) is used for anything that gives a bonus or malus to the attack roll.
- The *damage modifiers rulebook* (entry point 10) is used for anything that gives a bonus or malus to the damage done.
- The *flavour text rulebook* (entry point 14) should not be added to directly, but the rulebooks it calls on manage all the normal prose generation and should be customised heavily.
- The *aftereffects rulebook* (entry point 15) can be used to implement most 'special effects': results of attacking that are not simply damage dealt.
- The *take away until attack circumstances rulebook* (entry point 16) is used for resetting all the variables your custom actions have set in order to indicate to the rest of the game that they are being performed.

And if you think this is still overwhelming, check out the examples in the next chapter – by looking at how some things are done, you will quickly learn to do them yourself.

2.5 Standard combat actions

ATTACK defines several new actions, and changes the behaviour of several others. The five changed and new actions discussed in this section are *waiting*, *attacking*, *concentrating*, *parrying* and *dodging*. The weapon-related actions *readying* and *reloading* are discussed in sections 2.7 and 2.8.1 respectively; the meta-actions *switching the numbers on* and *switching the numbers off* have been discussed in section 2.1.2.

2.5.1 Waiting

Waiting (commands *wait* or *z*) still has the actor do nothing, but ATTACK defines one additional rule. If and only if the actor's combat state is *at-Act* (which means the actor is having his turn in a combat situation), waiting automatically sets the actor's initiative to be 2 less than that of the person with the highest initiative (excluding the actor himself, who necessarily has the highest initiative of all).

The effect of this rule, the *waiting lets someone else go first rule*, is that waiting acts more or less as letting one person go first – not as letting everyone in the combat go first. This gives the person who waits more control about

when he acts, and getting such control is the only possible motive for waiting (since otherwise, one would always be better off concentrating). Not complete control, by the way: initiative is partly decided by randomness, and more than one person might have the highest initiative score, so you can never be certain how many people get to make a move before it is your turn again.

2.5.2 Attacking

Attacking (commands *attack* or *a*) is obviously an important action in ATTACK. We first unlist the standard Inform 7 *block attacking rule*, since this time, violence *is* the answer. We add "a" as a synonym for "attack", since the player will presumably type this often. And we tell the parser that it is unlikely that the player wishes to attack killed persons, but that it is very likely that he wishes to attack persons whose faction he hates. (This means that when there is only one enemy present, the player can simply type "a".)

We then define six *check attacking rules*, which stop the attack from happening if the noun is not a person (the *only attack persons rule*); if the noun is not alive (*the only attack the living rule*); if the noun is the player (the *do not kill yourself rule*); if the noun is of the player's faction (the *do not attack friendly people rule*); if the noun is of a faction the player doesn't hate (the *do not attack neutral people rule*); or if the combat state of the player is *at-React*, since one can only attack when it is one's turn (the *cannot attack as reaction rule*). All of them ensure that the player's action takes no time if it fails to pass the check.

Once all these checks have been passed, we know that the target is an alive person of a faction the player hates, and that the player is allowed to attack. Carrying out the attack is handled by two *carry out an actor attacking rules* (the "an actor" there makes it apply to NPCs as well as to the PC):

1. The *standard attacking first phase rule* is run when the *fight consequences variable* is false, that is, before people have a chance to react. It sets the *global attacker* to the actor and chooses a random readied weapon to be the *global attacker weapon*; it consults the *attack move flavour rulebook* (which can use the values set in the previous stage); it adds the action of the attacker attacking the defender to the *Table of Stored Combat Actions*, with a combat speed of 10; it changes the combat state of the noun to *at-React*, so that he will have a chance to react; it changes the provoker of the noun to the actor; and it changes the provocation of the noun to the attack action.

2. The *standard attacking second phase rule* is ran when the *fight consequences variable* is true, which will be the case when the *run the combat* routine is moving through the *Table of Stored Combat Actions*. The rule is very simple: if both actor and noun are still alive, it makes the actor *strike a blow against* the noun, thus calling the routine we have discussed extensively in section 2.4.2 and 2.4.3.

2.5.3 Concentrating

Concentrating is a new action which the player can perform by typing either "concentrate" or "c". Every person has a number called the *concentration*, which is usually 0 and can be increased up until 3. Having concentration gives the actor a bonus to the attack roll and a bonus to damage:

Concentration	*Attack bonus*	*Damage bonus*
0	0	0
1	+2	0
2	+4	+2
3	+8	+4

There are no *check an actor concentrating* rules, since you can always concentrate, even as a reaction. There are three *carry out an actor concentrating rules*:

1. A *first* rule called the *standard concentrating improves initiative rule*, which adds the current concentration (that is, the old concentration) to the actor's initiative. This means that if your current concentration is 2, and you concentrate once more, you get a +2 initiative bonus for the next round.

2. A rule called the *standard carry out concentrating rule*, which increases concentration by 1, unless this would take it above 3.

3. A *last* rule called the *standard concentrating prose rule*, which prints a message that the actor is concentrating, and how concentrated (mildly, quite or maximally) that person now is.

We then have two rules that implement the attack and damage bonuses of concentration. The first is an *attack modifiers rule* called the *concentration attack modifier rule*; the second is a *damage modifiers rule* called the *concentration damage modifier rule*.

Next come two rules that take concentration away. An *aftereffects rule* called the *lose concentration when hit rule* takes away the defender's concentration when the damage dealt to him was greater than 0; this rule calls the *let X lose concentration* routine. A *take away until attack circumstances rule* called the *lose concentration after attacking rule* takes away the attacker's concentration once he has attacked (by simply setting it to 0).

ATTACK then defines the *let X lose concentration* routine – which can be called by any of the author's own rules, if so desired. This routine stops immediately if the concentration of X is already 0. If it is not, it sets the concentration of X to zero, writes X to the *concentration loser* global variable, and considers the *lose concentration prose rules*.

There is one standard *last lose concentration prose rule* called the *standard lose concentration prose rule*, which prints "You lose your concentration!" or "[The concentration loser] loses concentration!" You can substitute your own prose by writing a *lose concentration prose rule* and have it stop the rulebook if it considers itself applicable.

Finally, there are a *chance to win rule* called the *CTW concentration bonus rule*, which is needed to have the AI evaluate attacking (more information can be found in section 2.6); and a *carry out going rule* called the *lose concentration on going rule*, which makes an actor lose all concentration when he moves – this ensures that you cannot maximally concentrate in an empty room and then take that concentration with you to your next fight.

2.5.4 Parrying

Parrying is a new action which the player can perform by typing either "parry" or "p". ATTACK gives every person an either/or property: a person can either be *at parry* or *not at parry*.

The *check parrying rule* known as the *cannot parry when not reacting rule* checks whether the attack state of the player is *React*; if it is not, it prints the "You parry, but there is no attack." message and ensures that no time is taken by the (failed) action.

We have two *carry out an actor parrying rules*, and a *last report an actor parrying rule*:

1. The *parrying changes initiative rule* gives the person who parries one bonus initiative – it's an active reaction that gets you close to your enemies and into the thick of combat.

2. The *standard carry out parrying rule* makes the actor be *at parry* – which will trigger an *attack modifiers rule* later.

3. The *standard parry prose rule* prints a single line saying that the defender strikes up a defensive pose.

Now, the *parry defence bonus rule* is an *attack modifiers rule* that calculates and applies the actual attack modifier of the parrying action. Parrying is a little bit complicated, since its effectiveness depends both on the weapon the attacker is using and the weapon the defender is using. You can't parry a shotgun, and you can't parry with a longbow – to name two obvious examples. So how it works is this: every weapon has two numbers, the *active parry max* and the *passive parry max* (defined in the part of ATTACK that defines weapons, discussed in section 2.7). The *active parry max* defines how well you can parry with the weapon, the *passive parry max* defines how well you can parry against the weapon. The attack penalty of the attacker is either the passive parry max of his weapon, or the active parry max of the defender's weapon, whichever is *lower*.

Example: the active parry max of a sword is 4, while its passive parry max is 3. The active parry max of sawed-off shotgun is 2, while its passive parry max is 0. In the first round, I parry your sword attack with my shotgun: the attack penalty of your attack is equal to the lower number of 3 and 2, which is 2. In the second round, you parry my shotgun shot with your sword. My attack penalty is the lower number of 4 and 0, which is 0 – your sword does nothing to deter me.

The *parry defence bonus rule*, then, makes this calculation, applies the penalty, and prints the result (in numbers mode).

A *take away until attack bonuses rule* called the *no longer at parry after the attack rule* takes away the *at parry* status of the defender once the attack is finished.

Finally, there is a *best defender's action rule* called the *CTW parry bonus rule*, which ensures that the AI can accurately identify the effects of parrying on its chances of landing a hit.

2.5.5 Dodging

Dodging is a new action that the player can perform by typing "dodge" or "do". (Problems with the English verb "to do" seem unlikely to me, since it is rarely used in IF commands.) ATTACK gives every person an either/or property: a person can either be *at dodge* or *not at dodge*.

The *check dodging rule* known as the *cannot dodge when not reacting rule* checks whether the attack state of the player is *React*; if it is not, it prints the "You dodge, but there is no attack." message and ensures that no time is taken by the (failed) action.

We have one *carry out an actor dodging rule* and one *last report an actor dodging rule*:

1. The *standard carry out dodging rule* make the actor be *at dodge* – which will trigger an *attack modifiers rule* later.

2. The *standard dodge prose rule* prints a single line saying that the defender gets ready for quick evasive maneuvers.

An *attack modifiers rule* called the *dodge defence bonus rule* implements the defence bonus given by the dodging action. This bonus is equal to the *dodgability* of the attacker's weapon – a property that is defined in section 2.7. So against a weapon with a dodgability of 3, you would get a +3 bonus on your defence; against a weapon with a dodgability of 0, you would get no bonus. This rule also prints the necessary explanation of the bonus for numbers mode.

A *take away until attack bonuses rule* called the *no longer at dodge after the attack rule* takes away the *at dodge* status of the defender once the attack is finished.

Finally, there is a *best defender's action rule* called the *CTW dodge bonus rule*, which ensures that the AI can accurately identify the effects of dodging on its chances of landing a hit.

2.6 Artificial Intelligence

Obviously, ATTACK needs Artificial Intelligence. We *could* just have the NPCs attack every turn, or we *could* have them make a completely random choice – but that wouldn't lead to interesting tactical combat, and would severely limit the things an author can customise. So instead we have a framework of rules that allows the NPCs to make an informed decision about what they wish to do to whom using what weapon.

2.6.1 Preliminaries

We set up three tables: the *Table of AI Combat Person Options*, the *Table of AI Combat Options* and the *Table of AI Combat Weapon Options*, which will be used to store persons, actions and weapons respectively. For each of them, we define a routine that blanks out all the rows – the routines are called by "blank out the [name of table]".

In addition, we define four global variables that AI rules can use to communicate with each other: the *stored_row* (a number that we'll use to indicate

a row of a table), the *stored_person* (a person), the *stored_weapon* (a weapon) and the *stored_action* (a stored action).

2.6.2 Standard attacker AI

We then associate a rulebook called the *combat AI rulebook* with every person. The *combat AI rulebook* of a person is usually the *standard_attacker rulebook*. If you want to override all the AI settings of a person, you should put your own rulebook in its place.

The *standard_attacker rulebook* does four things. First, it calls the *have the AI select a target* routine, which selects an enemy to target – if no enemy is found, the *found-a-target boolean* will be false and the following three steps are skipped. Second, it calls the *have the AI select a weapon* routine. Third, it calls the *have the AI select an action* routine. And finally, it tries the selected action, which has been saved to the variable *stored_action*.

Before we describe these routines in more detail, I should say a few words about the order in which the AI routine makes its choices.

2.6.3 Three steps: person, weapon, action

The perfect AI would consider all possible combinations of a person, a weapon and an action, and decide what the best of them is. But this easily leads to a combinatorial explosion, so ATTACK opts for a simpler scheme. The AI first chooses which person to attack; it then chooses which weapon to use; and it finally chooses which action to perform.

But, you may say, this is needlessly wasteful: if the AI doesn't attack, it doesn't need to spend time deciding on a person and a weapon! This is true, but then again, whether to attack or do something else will depend on the person and weapon chosen. For instance, the AI will choose to concentrate rather than attack if it considers its chance to hit too low; but its chance to hit obviously depends on who it is going to attack.

So although the person-weapon-action sequence is more computation intensive than the action-person-weapon sequence, it also allows for more rational decision making, which is why I have chosen it for ATTACK.

2.6.4 First step: choosing a person

We first define the pressing relation which relates various people to various people and is tested with "if A presses B". The *make A strike a blow against B* routine – the big routine discussed in section 2.4 – makes A press B by

calling the *have A start pressing B* routine. This latter routine makes sure that A presses B and does not press anyone else.

The pressing relation is used by the AI routines when choosing a target. Specifically, all NPCs will prefer to attack persons they press, which generally means the person they have been attacking. They will also prefer (but to a lesser extent) people who press them, which generally means people who have recently attacked them. This gives a certain consistency to combats with multiple people on each side. (It also allows the author to fine-tune the selection of targets, because one can set the pressing relation by hand.)

Finding a person to attack is handled by the *have the AI select a target* routine. This routine fills the *Table of AI Combat Person Options* with all alive persons enclosed by the location whose faction is hated by the faction of the attacker; each person also gets an initial "0" in his *Weight entry*. The routine sets the *found-a-target boolean* to true if it finds at least one potential target. (If there are no potential targets, the rest of the AI rules are skipped.)

For each possible target, we then call the *standard AI target select rules*. The rules in this rulebook can use the *stored_person* variable to check who the potential target is, and the *stored_row* variable to check to which row in the *Table of AI Combat Person Options* they have to write their result. They calculate whether it is a good idea to attack this person or not, and change the *Weight entry* in this table accordingly.

Once the table is complete, it is sorted in random order, and then in reverse Weight order. A random person with the highest Weight entry will then be in row 1, and this person will be chosen as the target – for this purpose, he is stored in the *global defender* variable.

ATTACK comes with no fewer than ten *standard AI target select rules*:

1. The *do not prefer passive targets rule* subtracts 5 from the weight of all persons whose faction is passive. These people pose no threat, so they should not be attacked first.

2. The *prefer targets you press rule* adds 3 to the weight of all persons pressed by the attacker. These will generally (though not necessarily) be people who have just been attacked by the attacker, and it is natural that one does not change one's target too often.

3. The *prefer those who press you rule* adds 1 to the weight of all persons who press the attacker. These will generally (though not necessarily) be people who have just been attacking you, and it is natural to retaliate.

4. The *prefer the player rule* adds 1 to the weight of the player. Just a little incentive to make things more interesting, people! You wouldn't want your allies to take all the hits, would you?

5. The *prefer the severely wounded rule* adds 2 to the weight of anyone whose health is less than 50% of his permanent health; and another 4 to the weight of anyone whose health is less than 25% of his permanent health. Like vultures the NPCs will fall on the wounded and dying. Why? Because it is just good tactics in any game where the amount of damage dealt by a person is not proportional to his health. All players of RPGs and RTSes know that you must concentrate your fire and take out your enemies one by one.

6. The *prefer concentrated people rule* adds 1 to the weight of anyone who has a concentration of 1. This rises to 2 at a concentration of 2, and to 5 at a concentration of 3. This, again, is simply good tactics: such a high concentration *must* be broken before it can be used.

7. The *prefer those with good weapons rule* adds half the damage die of the defender's weapon to his weight. So if you use a weapon that has a damage die of 8 (dealing between 1 and 8 damage), your weight increases by 4. Again, this is good tactics: people who can deal a lot of damage are dangerous and should be killed soon.

8. The *do not prefer good parriers rule* decreases the weight of a person whose readied weapon offers a particularly good defense against the readied weapon of the attacker. (Optimally, we would run this rule after choosing a weapon for the attacker; but, as explained above, we will have to make do with the current approximation.) The weight is decreased by the positive difference (if any) between the parry bonus and the dodge bonus the defender would have against the attacker.

9. The *do not prefer high defence unless concentrated rule* ensures that if an NPC has no concentration, he will go for targets with a low defence; but if he has managed to get a high concentration, he will use this bonus against persons who would otherwise be hard to hit. The rule first calculates the difference between the defender's defence and the attacker's melee. (If it turns out to be negative, it is set to 0, but this is not likely.) The number thus gotten is *subtracted* from the weight if the attacker's concentration is 0; it is *added* to the weight if the attacker's concentration is 2; and it is *added twice* if the attacker's concentration is 3. Nothing is done if the attacker's concentration is 1.

10. The *randomise the target result rule* adds a random number between 0 and 4 to the weight, just to make things interesting and less predictable.

2.6.5 Second step: choosing a weapon

Finding a weapon to attack with is handled by the *have the AI select a weapon* routine. First, we check whether there is by any chance only one weapon enclosed by the attacker; if so, we automatically choose it. If not, the routine fills the *Table of AI Combat Weapon Options* with all weapons enclosed by the attacker; each weapon also gets an initial "0" in its *Weight entry*.

For each possible weapon, we then call the *standard AI weapon select rules*. The rules in this rulebook can use the *stored_weapon* variable to check what the potential weapon is, and the *stored_row* variable to check to which row in the *Table of AI Combat Weapon Options* they have to write their result. They calculate whether it is a good idea to attack with this weapon or not, and change the *Weight entry* in this table accordingly.

Once the table is complete, it is sorted in random order, and then in reverse Weight order. A random weapon with the highest Weight entry will then be in row 1, and this weapon will be chosen – for this purpose, it is stored in the *stored_weapon* variable.

ATTACK comes with no fewer than ten *standard AI weapon select rules*, but two of those will be discussed in the subsection on reloadable weapons, 2.8.1.

1. The *prefer lots of damage rule* simply adds the damage die of the weapon to its weight. More damage is better.

2. The *prefer low dodgability and passive parry rule* subtracts either the weapon's dodgability or the weapon's passive parry max from the weight. It is bad for you if your weapon is easy to dodge or parry – but if it is easy to dodge, it doesn't really matter whether it's easy to parry, and the other way around.

3. The *prefer good active parry rule* adds half the weapon's active parry max to the weight. It's nice if a weapon can be easily parried with, but we divide it by 2 because we always have dodging as an alternative.

4. The *prefer good attack bonus rule* adds 1.5 times the weapon attack bonus of the weapon to the weight (rounded up). Even a +1 attack bonus is a good modifier.

5. The *prefer readied weapon rule* adds 3 to the weight if the weapon is readied and the combat state of the actor is at-Act; and adds 1 if the weapon is readied and the combat state of the actor is at-React.

Readying another weapon costs an entire turn, which is costly, but less costly if you can do it while reacting to an attack.

6. The *prefer readied weapon if attacker almost dead rule* adds 2 to the weight if the weapon is readied and the attacker's health is less than 25% of his permanent health. If you're almost dead, it's not a great idea to spend a turn switching to another weapon. (Switching to a better weapon is an investment which needs time to pay itself back.)

7. The *prefer readied weapon if defender almost dead rule* adds 2 to the weight if the weapon is readied and the defender's health is less than 25% of his permanent health. If your opponent is almost dead, it's more efficient to go for a quick kill than to improve your long term chances.

8. The *randomise the weapon result rule* adds a random number between 0 and 2 to the weight. Since 2 is less than the +3 bonus given to a readied weapon, it is certain that if an actor carries two identical weapons, he will never spend a turn switching from one to the other.

2.6.6 Third step: choosing an action

The third routine that is called by the *standard_attacker rulebook* is *have the AI select an action*. At this point *global defender* contains the target and *stored_weapon* contains the chosen weapon. Now we set up a table of possible actions, the *Table of AI Combat Options*. After blanking this table out, we call the *standard AI action select rules*. For every possible action that the AI can perform, there is a *first standard AI action select rule* which adds this action to the table; and then the other rules in the rulebook will change its Weight entry, after which the table is randomised and sorted. A random action with the highest weight is chosen and stored as the *stored_action*.

One additional *first standard AI action select rule* is the *calculate the chance to win rule*. We will consider it in more detail in the next subsection.

Standard ATTACK comes with twenty *standard AI action select rules*:

1. The *calculate the chance to win rule* will be discussed in subsection 2.6.7.

2. The *consider reloading rule* is discussed in subsection 2.8.1; it sets up the reload action.

3. The *consider reloading rule* adds the readying action to the table, with a base weight of 0.

4. The *consider parrying rule* adds the parrying action to the table. It gives it a base weight of 5 if the combat state of the actor is at-React, and a -1000 base weight otherwise. (We should only parry when reacting.)

5. The *consider dodging rule* adds the dodging action to the table. It gives it a base weight of 5 if the combat state of the actor is at-React, and a -1000 base weight otherwise. (We should only dodge when reacting.)

6. The *consider concentrating rule* adds the concentrating action to the table, and gives is a base weight of 3.

7. The *consider attacking rule* adds the action of attacking the global defender to the table. It gives it a base weight of -1000 if the combat state of the actor is at-React, and a base weight of 5 otherwise. (We should only attack when not reacting.)

8. The *standard attack select rule* adds -100 to the weight of the attacking action when the normalised chance-to-win is 0. It then adds normalised chance-to win minus 5 to the weight. Thus, if the normalised chance-to-win is 0 (0% chance of hitting), attacking gets a -105 penalty; if the normalised chance-to-win is 2 (20% chance of hitting), attacking gets a -3 penalty; if the normalised chance-to-win is 8 (80% chance of hitting), attacking gets a +3 bonus. And so on.

9. The *standard concentration select rule* adds 5 minus the chance-to-win to the weight of the concentration action. Thus, if the chance-to-win is 7 (70% chance of hitting), the concentration actions gets a -2 penalty; but if the chance-to-win is -2 (0% chance of hitting, and in need of at least a +3 attack bonus), concentration gets a +7 bonus. Additionally, if the concentration of the actor is already at the maximum of 3, the weight is decreased by 100.

10. The *concentration influences attacking rule* adds the concentration of the global defender to the weight of the attack action. If the concentration of the global defender is 3, an additional +2 bonus is added. If your enemy is highly concentrated, you must break his concentration before he has a chance to use his bonuses! (ATTACK currently does not add a bonus for the concentration of the attacker, assuming that the added chance to hit takes care of this. But this is certainly something one might wish to fine-tune.)

11. The *standard parry and dodge against attack select rule* only runs if the combat state of the actor is at-React and the provocation is the attacking action. It checks whether dodging or attacking is a good idea against an attack. If parrying will lead to a bonus as good as or better than that of dodging, the parry action's weight is increased by the defence bonus gained, while the dodge action's weight is decreased by 100 (minus the dodge bonus) – unless the defence bonus gained by parrying is 0, in which case the parrying action also gets a -100 penalty. (In such a case, it is better to concentrate.) If parrying will lead to a worse bonus than dodging, the dodge action's weight is increased by the defence bonus gained by dodging, while the parry action's weight is decreased by 100 (minus the parry bonus). The reason to slightly prefer parrying is that it gives a +1 initiative bonus.

12. The *standard ready select rule* gives a -100 penalty to the readying action if the weapon is already readied. If the weapon is not yet readied, it increases the weight of the readying action by 5, and decreases the weight of the attacking and concentrating actions by 100. If we are going to change to a better weapon, we should do so *before* attacking (to maximise the number of attacks done with the better weapon), and certainly before we start concentrating. (It is always better to ready a weapon first and concentrate later, rather than the other way around – concentration can be broken, readying cannot.)

13. The *do not reload weapons that do not use ammo rule* is discussed in subsection 2.8.1.

14. The *do not reload weapons that are not empty rule* is discussed in subsection 2.8.1.

15. The *do not reload weapons that cannot be reloaded rule* is discussed in subsection 2.8.1.

16. The *do not attack with unloaded weapon rule* is discussed in subsection 2.8.1.

17. The *do not concentrate with unloaded weapon rule* is discussed in subsection 2.8.1.

18. The *reload when you have an unloaded weapon rule* is discussed in subsection 2.8.1.

19. The *tension influences attacking rule* is discussed in subsection 2.8.2.

20. The *randomise the action result rule* adds a random number between 0 and 5 to each action in the table, once again to make things more interesting and less predictable.

2.6.7 Chance to win rules

The final part of the AI are the *chance to win rules*. A *first standard AI action select rule* called the *calculate the chance to win rule* is run before any of the normal *standard AI action select rules* start doing their calculations. It does only one thing: consider the *chance to win rules*.

The effect of the *chance to win rules* is to set two numbers that vary, *chance-to-win* and *normalised chance-to-win*. The probability of the attacker hitting the defender with a normal attack is the *normalised chance-to-win* divided by 10 (or, if you prefer, 10% times the *chance-to-win*). So a *normalised chance-to-win* of 7 indicates a 70% chance of hitting. Since all probabilities lie between 0% and 100%, *normalised chance-to-win* lies between 0 and 10. The *chance-to-win* is the calculated probability before it had been limited to the 0 to 10 range. This is useful if you want to know whether the attacker should attempt to make a special attack with a +3 attack bonus; if the *chance-to-win* is 0, the answer might be yes; if it is -4, the answer will certainly be no.

Everything that influences the probability of hitting should have an associated chance to win rule, otherwise the AI will not be able to make rational decisions using full knowledge of the situation.

The standard *chance to win rules* set the chance to win to 10 (*CTW default rule*); add the melee of the global attacker (*CTW melee bonus rule*); subtract the defence of the global defender (*CTW defence bonus rule*); subtract the best possible defensive action of the global defender (*consider best defender's action rule*); add the attack bonus of the weapon (*CTW attack bonus from weapon rule*); and add the attack bonus from tension (*CTW tension bonus rule*). Finally, the *normalised CTW rule* sets the *normalised chance-to-win* to a number between 0 and 10.

The special case here is the *best defender's action rule*, which calls the rulebook *best defender's action rules*. The idea of this rulebook is to call all defensive actions, calculate the possible defense bonus of each, and apply only the best of them. The defender can either dodge or parry, not both; we need to assume that he chooses the best of these actions, and apply the bonus associated with it.

In order to do this, standard ATTACK has two *best defender's action rules*: the *CTW parry bonus rule* and the *CTW dodge bonus rule*. These rules calculate the possible parry and dodge bonus respectively, and then write it

to the number that varies *best defence* if it is higher than the previous best defense. All new defensive actions should contain a similar best defender's action rule!

2.7 Weapons

This part of the source defines the weapon kind and its associated properties; gives all persons a natural weapon; and defines the readying action.

2.7.1 The weapon kind

A weapon is a kind of thing. (In fact, this has been declared higher up in the source, in order to prevent errors.) All weapons have five numbers associated with them:

- The *damage die* defines the amount of damage the weapon can do. In standard ATTACK, the damage is a random number between 1 and the damage die of the weapon. The damage die of a weapon is usually 6.

- The *dodgability* of a weapon defines what defence bonus a defender gets when dodging against the weapon. The dodgability of a weapon is usually 2.

- The *passive parry max* of a weapon defines the maximum parry bonus when parrying *against* this weapon. The passive parry max of a weapon is usually 2.

- The *active parry max* of a weapon defines the maximum parry bonus when parrying *with* this weapon. The active parry max of a weapon is usually 2.

- The *weapon attack bonus* of a weapon defines the bonus to attack rolls given by the weapon. It is usually 0.

The *attack bonus from weapon rule* is an *attack modifiers* rule that applies (and reports, if the numbers boolean is true) the weapon attack bonus to the attack roll. The *CTW attack bonus from weapon rule* is a *chance to win rule* that applies the weapon attack bonus to the chance to win calculations.

2.7.2 Readying

Readying is an action applying to one touchable thing. The player triggers it with the "ready [weapon]" syntax. The action applies to one touchable thing.

The readying action is governed by four standard rules:

1. A *first carry out an actor readying* rule called the *implicit taking when readying rule*. This rule attempts to take the noun if the player doesn't already have it. (ATTACK thus assumes that actors are allowed to take weapons as a free action. You can of course change this.)

2. A *carry out an actor readying* rule called the *carry out readying when enclosing rule*. This rule checks whether the actor encloses the noun, and if so, it sets the noun to readied.

3. A *last carry out an actor readying* rule called the *unready all other weapons rule*. If the noun has been readied, this rule cycles through all other objects enclosed by the actor and unreadies them.

4. A *report an actor readying* rule called the *standard report readying rule*. This rule prints "*A* readies the *B*" if the readying was successful. If the noun is not readied but is enclosed by the actor (this cannot happen in standard ATTACK) it prints: "*A* fools around with the *B*, but fails to ready it.". If the noun is neither readied nor enclosed by the actor, it prints: "*A* attempts to ready *B*, but cannot get it."

There are four additional rules that govern readying. The *readied inventory listing rule* prints " (readied)" after a readied object in the player's inventory. The three rules known as the *unready on dropping rule*, the *unready on putting on rule* and the *unready on inserting rule* unready a weapon when it is dropped, put on something, or inserted into something. (ATTACK makes no assumptions about what happens when a weapon is given to someone – which is anyway impossible if the author does not create rules for it.)

2.7.3 Natural weapons

In order to ensure that every person always has a weapon – which makes a lot of other code easier to write – we endow everyone with a "natural weapon".

A natural weapon is a kind of weapon. It is part of every person. This means that it cannot be dropped, and will not show up in room descriptions or inventory listings. However, it *can* be examined, e.g., when the player

types "examine my natural weapon". The standard response is given by the *standard description of a natural weapon rule*, which says: "Clenched fists, kicking feet – that kind of stuff.".

A natural weapon has the following standard properties: damage die of 3, dodgability of 2, passive parry max of 2, active parry max of 0. (It's not really useful if you manage to parry the opponent's axe with your hand, you see.)

Finally, there is the *ready natural weapons if no other weapon readied rule*, which runs when play begins and every turn thereafter. It cycles through the alive persons enclosed by the location, and if that person encloses no readied weapon, it readies the natural weapon of the person. Thus we can be assured that every person will always have at least one readied weapon. (He should also have at most one readied weapon, but that is handled by the rules above.)

2.8 Plug-ins

Plug-ins are parts of code that you can add to your project if you need them. I hope that at some future point, we will have a large collection of ATTACK plug-ins. Right now there are two standard plug-ins that come with ATTACK: the reloadable weapons plug-in and the tension plug-in. These plug-ins are standard because it is likely that you will want to use them. However, the rest of ATTACK does not depend on them and you can comment them out if you wish.

2.8.1 Reloadable weapons

The reloadable weapons plug-in defines new properties for weapons so that we can have weapons with ammunition that need to be reloaded once they are out of ammo. It is also possible to create weapons that run out of charges but cannot be reloaded – a canister of pepper spray, for instance, or a magic wand with a limited number of spells.

We first give all weapons four new properties, all of which are numbers that vary:

- The *maximum shots* is the amount of attacks that can be performed with the fully loaded weapon before it runs out of ammo. For a six-shooter, this would be 6; for a crossbow, it would be 1. The usual value is 0. Any weapon whose maximum shots is 0 will be treated as a weapon that does not use ammo.

- The *current shots* is the amount of attacks that can still be performed with the weapon before it runs out of ammo. If it drops to 0, the weapon is out of ammo. The usual value is 0.

- The *maximum load time* is the amount of reloading actions if takes to reload the weapon. If the maximum load time is 1, it takes only 1 action to reload the weapon; but if it is 3, it takes 3 actions. As a special value, a weapon with maximum load time -1 cannot be reloaded – once it runs out of ammo, it becomes useless. The usual value is 0.

- The *current load time* is the amount of reloading actions that still has to be taken. This is only useful for weapons with a maximum load time higher than 1: we decrease the current load time by 1 for each reloading action, and once the current load time equals 0, the weapon is reloaded. The usual value is 0.

We also define that a weapon is *unloaded* if its current shots is 0, but its maximum shots is not 0.

Furthermore, we define four text properties for every weapon: the *shots text*, which is usually "shots"; the *reload text*, which is usually "reload"; the *reload stem text*, which is usually "reload"; and the *out of ammo text*, which is usually "You pull the trigger, but nothing happens – you're out of ammo!". These texts are useful because not all weapons using ammunition can be described with the same words. A crossbow has "arrows", not "shots"; a laser gun has "charges", not "shots", and is "recharged" rather than "reloaded".

The perhaps somewhat cryptic *reload stem text* should contain that part of the reloading verb after which we can put "-ed" to get the past tense or "-ing" to get the gerund. Thus, for the laser gun just mentioned, the *reload text* should be "recharge", and the *reload stem text* should be "recharg".

Now we define three rules that show and govern the use of ammunition:

- The *show ammo in inventory rule* prints text showing the amount of ammo in a weapon when inventory is taken; this text either has the form " (x of y shots left)" or " (no ammo; r rounds to reload)" or " (no shots left; cannot be reloaded)". In these phrases, "shots" and "reload" are exhanged for the shots text and the reload (stem) text of the weapon respectively. (And of course, nothing is printed if the maximum shots of the weapon is 0.)

- An *aftereffects rule* called the *decrease ammo rule* decreases the current shots of the weapon used by the global attacker by 1. (Unless the maximum shots of the weapon is 0.)

- A *check attacking rule* called the *cannot attack when out of ammo rule* prints "You pull the trigger, but nothing happens – you're out of ammo!" when the weapon is, in fact, out of ammo. (If you have set the weapon's out of ammo text to something different, that gets printed instead.) Unlike most check rule failures, this one does not count as acting fast – aiming and pulling a trigger takes a round, and makes for a cool if somewhat humiliating waste of the player's round.

We then define the reloading action, which applies to one carried weapon and is triggered by the "reload *A*" syntax. We tell the parser that it is very likely that the player wishes to reload an unloaded readied weapon enclosed by him; and likely that he wishes to reload an unloaded weapon enclosed by him. Thus, you can generally just type "reload".

The reloading action is governed by seven rules: three check rules, three carry out rules, and a report rule.

- A *check reloading rule* called the *cannot reload weapons that use no ammo rule* checks whether the maximum shots of the weapon is different from 0. If not, it takes no time and tells the player that the weapon does not use ammunition.

- A *check reloading rule* called the *cannot reload unreloadable weapons rule* checks whether the maximum load time of the noun is -1. If it is, it is one of those special weapons that cannot be reloaded. The game takes no time and tells the player that the weapon cannot be reloaded.

- A *check reloading rule* called the *cannot reload fully loaded weapons rule* checks whether the current shots is equal to the maximum shots. If it is, no time is taken and the player is told that the weapon is already fully loaded.

- A *carry out reloading rule* called the *ready upon reloading rule* readies the weapon that is about to be reloaded. Thus, in standard ATTACK, reloading gives you a free ready action. If you don't want this, just tell Inform that this rule is not listed in any rulebook.

- A *carry out reloading rule* called the *zero current ammo on reloading rule* sets the current shots to 0 when a reloading action is taken. For weapons that reload in a single round, this makes no difference since the next rule will set the current shots to the maximum shots. But for weapons that take longer to reload, the default behaviour is that you cannot shoot with them until reloading is complete. (This evidently

makes sense for automatic weapons; you might want to change the behaviour depending on the type of weapon you are implementing.)

- A *carry out reloading rule* called the *standard carry out reloading rule* decreases the current load time of the noun by 1. If this makes the current load time less than 1, it resets the current load time to the maximum load time and the current shots to the maximum shots – the weapon is now fully loaded.

- A *report reloading rule* called the *standard report reloading rule* tells us that the actor has reloaded his weapon. If the reload time is more than 1, we are instead told that the actor has started (first round), continued (subsequent) rounds, or finished (last round) reloading.

Finally, we need to define the AI behaviour for reloadable weapons. This influences both the choice of a weapon, and the choice of an action. We deal with choosing a weapon first, by defining the following two *standard AI weapon select rules*:

- The *do not choose an empty weapon that cannot be reloaded rule* gives a -1000 penalty to any weapon that uses ammo, is empty, but cannot be reloaded. Such a weapon has become useless and should not be chosen.

- The *do not prefer weapons that need to be reloaded rule* only runs for weapons that use ammo, and are either out of shots or not readied – these weapons, after all, are the ones that we have to invest time in before we can use them. We then calculate how much time we will be wasting when we use this weapon, and apply an appropriate penalty. (The mathematics are somewhat complicated, but the source code contains examples.)

Now we need rules that will allow the AI to choose when to reload a weapon. Standard ATTACK contains seven of these rules in the *standard AI action select rulebook*:

- The *consider reloading rule* runs *first*, and adds the action of the global attacker reloading the stored_weapon to the *Table of AI Combat Options*.

- The *do not reload weapons that do not use ammo rule* gives a -1000 penalty to the reloading of weapons that do not use ammunition (whose maximum shots is 0).

- The *do not reload weapons that are not empty rule* gives a -100 penalty to the reloading of weapons that are not empty (whose current shots is not 0).

- The *do not reload weapons that cannot be reloaded rule* gives a -1000 penalty to the reloading of weapons that cannot be reloaded (whose maximum load time is -1).

- The *do not attack with unloaded weapon rule* gives a -1000 penalty to attacking with a weapon that must be loaded (maximum shots is not 0) but is not loaded (current shots is 0).

- The *do not concentrate with unloaded weapon rule* gives a -100 penalty to concentrating if the stored_weapon must be reloaded (same circumstances as previous rule).

- The *reload when you have an unloaded weapon rule* gives a +5 bonus to reloading the stored_weapon if it must be reloaded (same circumstances as previous two rules).

2.8.2 Tension

The tension plug-in provides a system that will keep combat fast and interesting even when defence ratings are high compared to attack ratings, and that creates a natural ebb and flow of tension during the combat. Almost any game will benefit from it, which is why it is a standard plug-in. Tension is a kind of 'drama manager' for combat: it makes sure that long periods in which no apparent progress is made – that is, in which no damage is done – are not experienced as boring, but rather as increasing the tension. The way it works is that every turn when no hit is scored, the tension (a number that varies) is increased by one. High tension gives everyone bonuses on the attack roll, thus increasing the likelihood that something will happen, and on the damage roll, thus increasing the stakes.

Tension also works as a check and balance on the combat system: if you have made it too hard for people to hit each other, tension will greatly alleviate this problem.

Tension is implemented using six rules:

- An *every turn rule* called the *standard increase or reset the tension rule* increases the tension by 1 (to a maximum of 20) if hate is present, and sets the tension to 0 otherwise.

- An *attack modifiers rule* called the *standard tension attack modifier rule* adds one half (rounded down) of the current tension to the attack bonus.

- A *damage modifiers rule* called the *standard tension damage modifier rule* adds one third of the tension to the damage modifier.

- An *aftereffects rule* called the *standard reduce tension after hit rule* reduces the tension when the final damage of an attack was greater than 0. The tension is not necessarily reduced to 0; rather, the new tension is 80% of (the old tension - 4), with a minimum of 0. Check the source code for a table that spells it out in detail.

- A *chance to win rule* called the *CTW tension bonus rule* increases the chance-to-win by the tension divided by 2 (thus giving the AI accurate information about how tension modifies the attack bonus).

- A *standard AI action select rule* called the *tension influences attacking rule* increases the weight of the attacking action by the tension divided by 4. (The previous rule also increases the weight of attacking; the current rule adds some extra weight because tension also increases damage.)

Chapter 3

Examples and recipes

3.1 Worked example: a simple dungeon

3.1.1 First blood

Once we have added ATTACK to our project, we can start a fight very easily. The following code suffices:

```
The cave is a room. The player is in the cave.
The health of the player is 20.

A mace is a kind of weapon. The player carries a mace.

The goblin is in the cave. The goblin is hostile.
The goblin carries a mace. The melee of the goblin is -1.
```

The player should be able to win this fight easily (especially if he does not forget to ready his mace): a person usually has a health of 10 and a melee of 0, so we have given the player a health bonus of 10 and the goblin an attack penalty of -1. We have left all the rest of the statistics, including all the statistics of the weapon, at their default values.

3.1.2 Custom weapons

Instead of writing "the player carries a mace", we might want to give the player a custom weapon. And, since this is the start of a typical dungeon crawl, we want to give him a crappy weapon.

```
The unwieldy mace is a mace. The unwieldy mace is readied.
```

```
The weapon attack bonus of the unwieldy mace is -1.
The player carries the unwieldy mace.
```

Since the default weapon attack bonus of a weapon is 0, this mace is slightly but significantly worse. However, we did make the mace 'readied', which means that the player doesn't have to spend a turn readying the weapon.

We have also implemented our first reward: the player can loot the goblin's corpse for the mace. Of course, this will lead to disambiguation problems between the mace and the unwieldy mace, so let's rename the goblin's mace:

```
The goblin carries a mace called the goblin's mace.
```

3.1.3 Simple prose

What fun is fighting a goblin if you just get the standard prose generation of ATTACK? We'd better customise it a bit, although we're going to keep everything quite simple for now. Let's start with a custom message for the goblin's death:

```
A fatal flavour rule (this is the fatal goblin rule):
   if the global defender is the goblin:
      say "[if the global attacker weapon is a mace]With a
        sweeping blow, [possessive of the global attacker] mace
        smashes into the goblin's head[otherwise][The global
        attacker] land[s] a furious blow on the goblin's head
        [end if], ending its beastly existence. Wow! A kill!
        You feel powerful.[run paragraph on]";
      rule succeeds.
```

I would implore you to always write prose generation rules without preconceptions about which event might trigger them. In the game as we have written it now, only the player can kill the goblin; so we might be tempted to drop all that talk about "the global attacker". But we should *never* give in to such temptations, for that way lie horrendous bugs. (Just imagine what would happen if you later decide to give the player a summoning spell.)

Now, a rule for when the player is killed by the goblin:

```
A fatal player flavour rule (this is the fatal player goblin
 rule):
   if the global attacker is the goblin:
      say "Impossible! You stare in disbelief as the goblin
```

```
        [if the global attacker weapon is a mace]'s mace moves
        towards your face, unstoppable and unstopped[otherwise]
        hits you square between the eyes[end if], ending all
        your dreams of fame and treasure.[paragraph break]
        Killed by the first goblin in the first room of the
        dungeon. If only you had listened to your mother.";
      rule succeeds.
```

Next, a rule for when the goblin attacks the player. (This is printed before the player gets to react.) Since this message will be seen multiple times, we'll put some randomness in it.

```
An attack move flavour rule (this is the goblin attacks rule):
   if the global attacker is the goblin:
      say "[one of]The[or]Smiling wickedly, the[or]With a
        scream, the[or]Just when you notice your shoe laces
        are untied, the[as decreasingly likely outcomes]
        goblin [one of]jumps forward to attack[or]attacks[or]
        advances towards[at random] [the global defender].";
      rule succeeds.
```

Now we move on to the normal hits and misses. I'll write one rule that fires when the goblin is the defender and the player is the attacker; and another rule which fires when the goblin is the attacker (independent of who is the defender). This is a good scheme to follow – it means that all prose will be specific to the current monster, while not forcing you to write rules for every possible combination of attackers and defenders.

```
A flavour rule (this is the goblin defender flavour rule):
   if the global defender is the goblin and the global attacker
     is the player:
      if the final damage is greater than 0:
         say "You score a [if the final damage is greater than
           3]brilliant [end if]hit against the goblin[if the
           final damage is greater than 2], who screams in pain
           [end if]![run paragraph on]";
         rule succeeds;
      otherwise:
         say "[if the goblin is at dodge]The goblin dances out
            of the way of your clumsy blow[otherwise if the
            goblin is at parry]The goblin stops your attack
```

```
            with his weapon[otherwise]You attack, but... a
            miss! How unseemly[end if].[run paragraph on]";
         rule succeeds.
```

Notice how we can change the text displayed based on the amount of damage done, and also how we can change it based on whether the goblin did or did not parry/dodge.

```
A flavour rule (this is the goblin attacker flavour rule):
   if the global attacker is the goblin:
      if the final damage is greater than 0:
         say "The horrible little creature manages to hit you
           [one of]in the stomach, sending you reeling[or]on
           your left leg[or]on your behind when you try a fancy
           Star Wars move[at random].[run paragraph on]";
         rule succeeds;
      otherwise:
         say "[if the player is at dodge]You easily avoid the
           goblin's flimsy attacks[otherwise if the player is
           at parry]Clang! You sweep away your enemy's attack
           and move in for the kill[otherwise]'You'll never get
           me, hellspawn!', you cry as the yellow demon
           unsuccessfully attempts to end your life[end if].
           [run paragraph on]";
         rule succeeds.
```

This is already starting to look good – assuming we want to write a somewhat silly dungeon crawl, of course, but all I'm interested in right now is to show you how this stuff works. A few more prose rules should be added to get an even better effect. First, especially since we are changing our miss texts based on whether the goblin parried or dodged, we may not want to announce these reactions when they are decided on. (Just run the game, and you'll see what I mean.) So let's add the following:

```
Report an actor parrying (this is the do not report goblin
  parry rule):
   if the actor is the goblin:
      rule succeeds.

Report an actor dodging (this is the do not report goblin
  dodge rule):
   if the actor is the goblin:
      rule succeeds.
```

Finally, we may want to spice up the text shown when the goblin concentrates.

```
Report an actor concentrating (this is the goblin concentrating
  prose rule):
   if the actor is the goblin:
      now the global actor is the actor;
      if the concentration of the actor is 1, say "The goblin
        starts chanting in a low voice.";
      if the concentration of the actor is 2, say "The goblin's
        chant rises in volume[one of]--he sounds like a shaman
        you once saw on Discovery[or][stopping].";
      if the concentration of the actor is 3, say "The goblin's
        entrancing song rises to a feverish pitch; his eyes
        shine with demonic purpose.";
      rule succeeds.
```

And to have an appropriate message when we break the goblin's concentration by hitting him:

```
Lose concentration prose rule (this is the goblin
  lose concentration prose rule):
   if the concentration loser is the goblin:
      say " The goblin's [bold type]trance[roman type] is
        broken![run paragraph on]";
      rule succeeds.
```

Notice that everywhere in this section, we used rules – which can in principle be made to do *anything*. Absolutely nothing is stopping you from having the villain say something about the current weapon of the player when the villain's health has dropped below half its normal value, for instance.

To round out our prose improvement process, let us turn off the numerical results:

```
When play begins:
   now the numbers boolean is false.
```

3.1.4 Killing the player

One weird thing about our adventure is that once the player is killed, the game doesn't stop. ATTACK doesn't automatically end the game once the player's health drops below 0, because different authors might want very

different things to happen. If we just want to stop the game, we can do the following:

```
Every turn (this is the player death rule):
   if the player is killed:
      end the game saying "You were killed by [the global
        attacker].".
```

3.1.5 Movement and retreat

Of course, a one-room game is a little limited. So let's have:

```
Small cavern is north of cave.
```

The player will now be able to go to the small cavern whenever he likes – even when the goblin is still alive. This may not be desirable, so we'll add:

```
Before going (this is the cannot go when in a fight rule):
   if hate is present:
      take no time;
      say "You cannot abandon a fight so easily!" instead.
```

But perhaps that is a little harsh – the player will never be able to leave a fight, so if he ever blunders into something he can't handle, he'll be doomed. Let's try our hand at making a new action: retreating. When the player retreats, he is moved to his previous location... but only after everyone in the location who presses him or is pressed by him has made a free attack against him. (We could also have everyone who hates the player make a free attack against them, with far deadlier results. In the rules I give below, you can still retreat safely if your enemies have not yet attacked you and you have not yet attacked them.)

To make this easier to test, we'll also add a new location to our example game.

```
Grassy meadow is a room. "The cave gapes to the north."
The player is in grassy meadow.
Cave is north of grassy meadow.

Retreating is an action applying to nothing.
Understand "retreat" and "flee" as retreating.

A person is either retreater or not retreater.
A person is usually not retreater.
```

I'll explain those last two lines in a minute. For now, we need to store the location to which the player will retreat when he types "retreat".

```
The retreat location is a room that varies.
The retreat location is Grassy meadow.

First carry out going (this is the set retreat location rule):
   now the retreat location is the location.
```

which insures that the retreat location is set at the start of the game, and sets it anew after every going action. Note that retreating itself will not set a new retreat location – which is as it should be, since you can hardly retreat back to a fight after retreating from it. Note also that if we do any fancy stuff with movement in our game, we might need to think about how it interacts with retreating.

We now need to add some check rules to retreating:

```
Check retreating (this is the cannot retreat when in the
  retreat location rule):
   if the retreat location is the location:
      take no time;
      say "You cannot retreat now." instead.

Check retreating (this is the cannot retreat when there are no
  enemies rule):
   if not hate is present:
      say "There's nothing here to retreat from." instead.

Check retreating (this is the cannot retreat as reaction rule):
   if the combat state of the player is not at-Act:
      take no time;
      say "You cannot retreat as a reaction--try to survive and
        run away on your own turn." instead.
```

which ensure that you cannot retreat to the location you are already in; that you cannot retreat when there are no enemies to retreat from; and that you cannot retreat when you are reacting to someone else's action – you have to wait for your own turn. And last but rather essentially, here is the rule that carries out retreating:

```
Carry out retreating (this is the standard carry out retreat
  rule):
```

```
   say "Deciding that discretion is the better part of valour,
     you bravely run away.";
   now the player is retreater;
   repeat with X running through alive persons in the location:
      if the player is alive:
         if X presses the player or the player presses X:
            now X does not press the player;
            now the player does not press X;
            if the faction of X hates the faction of the player:
               make X strike a blow against the player;
   if the player is alive, move the player to the retreat
     location;
   now the player is not retreater.
```

We remove all pressing relations (which is good, since otherwise they would be remembered for the next fight, which is illogical) and make every person who hates the player and either presses him or is pressed by him strike a blow. If they fail to kill him, the player is transported to the retreat location, where Inform's standard rules kick in to print a room description.

So far so good, but what about that "retreater" / "not retreater" business? Well, having a variable we can check to see whether the player is retreating allows us to have special rules for blows struck when retreating. Think of a slow ooze monster that has a -4 penalty when striking a retreating character, of a many-tentacled horror that strikes with a +10 bonus, or simply of a nice magic item:

```
The boots of expeditious retreat are in grassy meadow.
They are wearable and plural-named.
The indefinite article is "the".

An attack modifiers rule (this is the boots of expeditious
  retreat grant better retreat rule):
   if the global defender is the player and the player is
     retreater:
      if the player wears the boots of expeditious retreat:
         if the numbers boolean is true, say " - 2 (boots of
           expeditious retreat)[run paragraph on]";
         decrease the to-hit modifier by 2.
```

These boots will certainly come in handy, since they give the player a +2 defence bonus when he is retreating.

3.1.6 Simple AI tweaks

So, the player can retreat, kill the goblin, get a better weapon – let's add some enemies to the small cavern on which he can *use that better weapon.* And what better enemies than frog warriors?

```
A frog warrior is a kind of person.
A frog warrior is usually hostile.
The health of a frog warrior is usually 4.
The melee of a frog warrior is usually -2.
The defence of a frog warrior is usually 7.

A small frog spear is a kind of weapon.
Every frog warrior carries a small frog spear.
The damage die of a small frog spear is usually 2.
```

These creatures really *are* puny – with only 4 (instead of 10) health, a large attack penalty, and an incredibly bad weapon. So we had better put a lot of them in the cavern:

```
A frog warrior called the frog commander is in the Small
  cavern.
A frog warrior called the limping frog warrior is in the Small
  cavern.
A frog warrior called the stout frog warrior is in the Small
  cavern.
A frog warrior called the old frog warrior is in the Small
  cavern.
A frog warrior called the bright green frog warrior is in the
  Small cavern.
```

The prose generated by five of these creatures will be of a horrendous repetitiveness, so if this were a real game, we would absolutely *have* to customise it heavily – preferably, with different prose for the five individuals. (Nobody said writing interactive fiction was easy, did they?) But we've already seen how to do that, so I won't be spending time on that now.

Now the weapon I've given these frog warriors is so bad that they may well start preferring their natural weapon, which is not what I want – and which leads to incomprehensible prose as well. So let's add an AI rule that will ensure that frog warriors use their spears:

```
Standard AI weapon select rule (this is the frogs use spears
```

```
  rule):
   choose row stored_row in Table of AI Combat Weapon
     Options;
   if the Weapon Option entry is a small frog spear:
      if the global attacker is a frog warrior:
         increase the Weight entry by 100.
```

And now for the really interesting stuff. Let's make those frog warriors dangerous and special by giving them a great love of concentrating. Individually, they may be weak, but imagine what they can do when all five of them are spending their turns concentrating, and the player cannot possibly hope to break the concentration of each of them before they can attack him with very nasty bonuses.

```
Standard AI action select rule (this is the frogs love
  concentrating rule):
   choose row with an Option of the action of the global
     attacker concentrating in the Table of AI Combat Options;
   if the global attacker is a frog warrior:
      increase the Weight entry by 5.
```

And again, if this were a real game, we would have to spend time on the prose generated by all those concentration actions.

3.1.7 Special ability: shout

In fact, we may have made those frog warriors a bit *too* good. How can the player possibly hope to defeat them? Well, that's going to be difficult, unless we give him some help. So what about this – a special action, *shouting*, that can be performed both as an action and as a reaction, and that has a probability of breaking the concentration of anyone who hears it?

```
Shouting is an action applying to nothing.
Understand "shout" as shouting.

The shout-victims is a list of persons that varies.

Carry out shouting (this is the standard shouting rule):
   now the concentration of the player is 0;
   truncate shout-victims to 0 entries;
   repeat with X running through alive persons enclosed by the
     location:
```

```
    if the concentration of X is not 0:
       if a random chance of 1 in 2 succeeds:
          now the concentration of X is 0;
          add X to shout-victims;
say "'[one of]BWAAH[or]BLEH[or]GRAAH[or]WRUGH[at random]!',
  you shout as loudly as you can. [if the number of entries
  in shout-victims is 0]Unfortunately, nobody is very
  impressed[otherwise if the number of entries in
  shout-victims is 1]This startles [shout-victims with
  definite articles] so much that [it-they] lose[s]
  concentration[otherwise]This startles [shout-victims with
  definite articles] so much that they lose concentration
  [end if].".
```

This should certainly help, but it's still not easy. Furthermore, if the player was wounded by the goblin, it will be much harder indeed. And what use is retreating if you cannot heal yourself once you've left combat? So without further ado, we come to...

3.1.8 Healing the player

When the player has health and can be damaged, you'll want to add a mechanism to your game that allows him to replenish his health. Some well-known ways to do this are:

1. Health slowly regenerates.
2. Health can be replenished with one-use items (health potions, medikits – and I am suddenly reminded of *Redneck Rampage*'s innovative use of whiskey).
3. The player has special abilities that allow him to heal himself and others (magic spells are always popular).
4. The player can heal himself by getting to a certain place.
5. Health is fully regained after every fight.
6. The player gets health from hitting or killing enemies.

Although they are perhaps not the most common, I would recommend thinking about the last two options for your game. Both of them decrease the time spent on healing, and thus increase the time spent on stuff that is probably

more fun. Another option, well-suited to interactive fiction, is to integrate healing with your story. However, for the current game we will implement option 2 and option 4.

Let's start with option 4. We want to give the player some way of fully healing himself when he gets back to the grassy meadow – this will mean that if you manage to retreat, you can always get back to strength. But this rewards the kind of hit-and-run tactic where you dash into a room, deal a small amount of damage to your enemy, dash back out, heal, and do it all again until he dies; which is not fun. So we'll make it so that *everyone* who is still alive gets healed. (This means that hit-and-run can still work, but only against multiple enemies and if you manage to kill at least one of them.)

```
The shrine is in Grassy meadow. It is fixed in place.

Instead of examining the shrine:
   say "If you pray in this location, the world's wounds will
     be healed.".

Praying is an action applying to nothing.
Understand "pray" as praying.

Carry out praying (this is the pray in grassy meadow rule):
   if the location is the Grassy Meadow:
      say "God's blessing descends onto the world, healing
        friend and foe alike.";
      repeat with X running through alive persons:
         fully heal X;
      rule succeeds.

Last carry out praying (this is the pray somewhere else rule):
   say "Words without thoughts never to heaven go.".
```

Notice that we have set the praying action up in such a way that we can easily add other locations where praying does something interesting. Now we will add some health potions:

```
A health potion is a kind of thing.

Instead of drinking a health potion:
   heal the player for 6 health;
   say "You drink the health potion, [if the healed amount is
```

```
    0]but you were already at full health[otherwise if the
    health of the player is less than the permanent health of
    the player]partially restoring your health[otherwise]
    restoring your health[end if].";
  if the combat state of the player is at-React:
     now the player is adrinking.

The player carries 3 health potions.
```

The last two lines of that instead-rule make it possible to ensure that although (a) drinking a potion can be done as a reaction, nevertheless (b) this will (realistically) give a penalty to your defence against the attack you are reacting to. Here's the code to implement that:

```
A person can be either adrinking or not adrinking.
A person is usually not adrinking.

An attack modifiers rule (this is the drinking is not a good
  defence rule):
   if the global defender is adrinking:
      if the numbers boolean is true, say " + 2 (defender
        drinking a potion)[run paragraph on]";
      increase the to-hit modifier by 2.

A take away until attack circumstances rule (this is the
  defender no longer adrinking rule):
   now the global defender is not adrinking.
```

With these additions, the fight with the frogs has become much more managable. (I just managed to win it. Shouting is great.) But one thing is missing – a way for the player to see how healthy he or she is. There are again many ways to do this, but perhaps one of the best is to show it in the status bar. Here is an implementation:

```
Include Basic Screen Effects by Emily Short.

Table of Fancy Status
left     central     right
"[bold type][location of the player][roman type]"     ""
     "Score: [score]/100"
"You [harm status], and armed with [a random readied weapon
```

```
  enclosed by the player]."     ""     ""

Rule for constructing the status line:
   fill status bar with Table of Fancy Status;
   rule succeeds.

To say harm status:
   if the numbers boolean is false:
      say "are ";
      let n be 10 times the health of the player;
      now n is n divided by the permanent health of the player;
      if n is:
         -- 10: say "unharmed";
         -- 9: say "scratched";
         -- 8: say "lightly wounded";
         -- 7: say "wounded";
         -- 6: say "wounded";
         -- 5: say "severely wounded";
         -- 4: say "severely wounded";
         -- 3: say "[bold type]bleeding copiously[roman type]";
         -- 2: say "[bold type]losing limbs[roman type]";
         -- otherwise: say "[bold type]dying[roman type]";
   otherwise:
      say "are at [health of the player] of [permanent health
        of the player] health".
```

which will appropriately print either prose or numbers to the status bar depending on whether numbers have been turned on or off. You can obviously do many other things with the status bar as well, and a game heavy on combat might want to put a lot of information in it. (But if you put vital information in the status bar, it is generally a good idea to make this information available in some other way as well. Some software – including that used by blind readers and people running the game in a MUD environment – may not show the status bar.)

3.1.9 A blade in the dark

So, once we have defeated the frogs, we should be rewarded with some nice items. Let's have a treasure chest, and let's make it so that it cannot be opened while the frogs are alive:

```
The treasure chest is in small cavern.
```

```
It is fixed in place, container, closed and openable.

Check opening the treasure chest (this is the cannot open
  chest in combat rule):
   if hate is present:
      take no time;
      say "Not in the presence of enemies!" instead.
```

The first item we'll put in the chest is another weapon: a dagger. This is a good opportunity to define another kind of weapon, and show how we can make weapon choice more interesting.

```
A dagger is a kind of weapon.

The damage die of a dagger is usually 2.
The weapon attack bonus of a dagger is usually -2.
The dodgability of a dagger is usually 3.
The passive parry max of a dagger is usually 1.
The active parry max of a dagger is usually 1.

The gilded dagger is a dagger.
The gilded dagger is in the treasure chest.
```

Of course, the dagger kind we just created is simply *terrible*. Why would anyone want to use it? Well, we're going to add the following two rules:

```
An attack modifiers rule (this is the dagger extra tension
  attack bonus rule):
   if the global attacker weapon is a dagger:
      let n be 0;
      now n is the tension divided by 2;
      if n is not 0:
         if the numbers boolean is true, say " + ", n, "
           (dagger benefits from tension)[run paragraph on]";
         increase the to-hit modifier by n.

A damage modifiers rule (this is the dagger extra tension
  damage bonus rule):
   if the global attacker weapon is a dagger:
      let n be 0;
      now n is the tension divided by 3;
```

```
      if n is not 0:
         if the numbers boolean is true, say " + ", n, "
           (dagger benefits from tension)[run paragraph on]";
         increase the damage modifier by n.
```

which double the total bonus tension gives to the attack roll and damage roll made with a dagger. Since tension bonuses can get very high if no hit is scored for a long time (up to a maximum of a +10 attack and +6 damage modifier), the benefit given by these two rules might easily outweigh the weaknesses of the dagger kind. More interestingly, if you use a dagger, you must use different tactics – more defensive, eluding your opponent until the tension has built up to such heights that you can deliver a single extremely effective attack. Which is what fighting with a dagger against people with maces and swords must be like – mostly trying not to get hit, until you get the right opportunity and can stick your blade into the enemy's weak spot.

If we are going to use tension to such an extent, we might as well show it in the status bar. We replace the previous table with:

```
Table of Fancy Status
left     central     right
"[bold type][location of the player][roman type]"
     "Tension: [tension]"     "Score: [score]/100"
"You [harm status], and armed with [a random readied weapon
  enclosed by the player]."     ""     ""
```

and then define:

```
To say tension:
   if the numbers boolean is true:
         say tension;
   otherwise:
      if tension is greater than 15:
         say "very high";
      if tension is greater than 10 and tension is less than
        16:
         say "high";
      if tension is greater than 5 and tension is less than 11:
         say "moderate";
      if tension is less than 6:
         say "low".
```

Well, let's go all the way towards the classic rogue character, shall we? We will give the player a new item that will give him some chance of entering new locations unseen; he will then be able to concentrate and let the tension rise, and suddenly strike from the dark. With a certain probability of being seen every turn, of course – just to keep things interesting.

```
The cloak of shadows is in the treasure chest. It is wearable.

A person is either shadowed or not shadowed.
A person is usually not shadowed.

First standard_attacker rule (this is the do nothing when all
  enemies shadowed rule):
   let p be false;
   let q be false;
   repeat with X running through all alive persons enclosed by
     the location:
      if the faction of the global attacker hates the faction
        of X:
         now p is true;
         if X is not shadowed:
            now q is true;
   if p is true and q is false:
      say "[CAP-attacker] [one of]remains unaware of your
        presence[or]looks around suspiciously[or]sighs in
        boredom[as decreasingly likely outcomes].";
      rule succeeds.
```

In this example, p becomes true if there is at least one person the global attacker hates, and q becomes true if at least one of those persons is not shadowed. So p is true and q is false if the global attacker is in the presence of enemies, all of whom are hidden.

Now we need to make sure that the player actually gets shadowed when wearing the cloak, and stops being shadowed when he attacks.

```
Every turn when the player wears the cloak of shadows (this is
  the blend into shadows rule):
   if not hate is present:
      if the player is not shadowed:
         now the player is shadowed;
```

```
        say "You blend into the shadows.";

An aftereffects rule (this is the attacking breaks shadowed
  rule):
   if the global attacker is the player:
      now the player is not shadowed.
```

And now, we add the possibility of detection. We will use a rulebook to allow for greater customisation – monsters with exceptional perception, items improving the player's sneaky skills, different levels of light for different rooms or different times of day, and so on.

```
The detection probability is a number that varies.
The detection rules are a rulebook.

Every turn when the player is shadowed (this is the possible
  detection rule):
   if hate is present:
      consider the detection rules;
      let n be a random number between 1 and 100;
      if the detection probability is greater than n:
         now the player is not shadowed;
            say "You have been detected!";
      now the detection probability is 0.

A detection rule (this is the standard probability of
  detection rule):
   increase the detection probability by 6.

A detection rule (this is the tension increases probability of
  detection rule):
   increase the detection probability by the tension divided
     by 2.
```

These rules give the player a 5% chance of being seen, and the chance slowly gets higher as the tension grows. The player has approximately 50% chance of remaining undetected until the tension rises to 9.

By the way, since being able to concentrate while remaining hidden is a far too powerful ability, we add:

```
Check concentrating (this is the cannot concentrate
```

```
  while shadowed rule):
   if the player is shadowed:
      take no time;
      say "You need all your concentration just to stay
        hidden." instead.
```

3.1.10 Monster with special abilities

All we need is a suitable victim for our new-found powers. What about a large troll, and one that can heal itself to boot?

```
Troll lair is north of small cavern.

A large troll is in troll lair. The large troll is a person.
The large troll is hostile.

The health of the large troll is 30.
The defence of the large troll is 8.

The large troll carries a huge club.
The huge club is a weapon.
The damage die of the huge club is 10.
The dodgability of the huge club is 5.
The passive parry max of the huge club is 0.
The active parry max of the huge club is 2.
```

I'm imagining this troll as a slow and clumsy beast, but one whose attack packs some truly serious punch when he manages to hit. Right now the club does between 1 and 10 damage; let's raise that to between 4 and 13. And let us ensure that the player cannot use this weapon.

```
Damage modifiers rule (this is the more damage from club rule):
   if the global attacker weapon is the huge club:
      increase the damage modifier by 3;
      if the numbers boolean is true:
         say " + 3 (huge club)[run paragraph on]".

Check readying the huge club:
   say "It's far too massive for you to use in combat."
     instead.
```

That the troll is clumsy can already be seen from the large dodgability of the huge club, but this alone just makes him a *bad* fighter. What we need is increased danger, increased unpredictability. What we need is a random attack bonus.

```
An attack modifiers rule (this is the troll attack bonus rule):
   if the global attacker is the large troll:
      let n be a random number between -4 and 10;
      increase the to-hit modifier by n;
      if the numbers boolean is true and n is greater than 0:
         say " + ", n, " (random troll bonus)[run paragraph
           on]";
      if the numbers boolean is true and n is less than 0:
         say " - ", 0 minus n, " (random troll bonus)[run
           paragraph on]".
```

Furthermore, we need to signal the AI that because of the random troll bonus, the troll can hit even when the odds seem unsurmountable. We'll do two things. First, we add a random modifier to the troll's estimation of his own ability to hit – this random number can be very different from the actual attack modifier, but that's just so the troll doesn't have perfect knowledge of the future. Second, we slightly increase the troll's preference for attacking. Both of these rules have the effect that the troll attacks more and concentrates less (which is what I want).

```
Chance to win rule (this is the troll CTW rule):
   if the global attacker is the large troll:
      increase the chance-to-win by a random number between -4
        and 10.

A standard AI action select rule (this is the troll attack
  select rule):
   if the global attacker is the large troll:
      choose row with an Option of the action of the global
        attacker attacking the global defender in the Table of AI
        Combat Options;
      increase the Weight entry by 2.
```

Now, for some added fun, we are going to add a special action for the troll. Trolls famously regenerate (at least they famously do so in *Dungeons & Dragons* and *Heroes of Might & Magic*), so let us imagine a troll that can

heal its own wounds by covering them with its own magical slime. (If we made a larger game, this would be the point where the player could gather troll slime, one of seventeen ingredients needed to save the dying king.)

```
Slime-healing is an action applying to nothing.

Carry out the large troll slime-healing:
   say "The troll drools large amounts of slime over its arms,
     chest and legs. Wherever the slime touches its wounds, they
     close and apparently heal.";
   heal the large troll for 5 health.
```

This should be obvious enough: we give a description of the action, and have it heal the troll for a certain amount of damage. But now we need the troll to consider this action. In order to have it do so, we need a *first standard AI action select rule*:

```
First standard AI action select rule (this is the consider
  slime-healing rule):
   choose a blank Row in the Table of AI Combat Options;
   change the Option entry to the action of the global attacker
     slime-healing;
   change the Weight entry to -1000.
```

Every new action that the AI can consider should have one such rule. The Weight entry given here is exceedingly low to ensure that unless something intervenes, no NPC will choose this action. (We generally give -100 to an action that would be pointless for an NPC to perform, and -1000 to an action that would be impossible for an NPC to perform. Normal NPC cannot use slime-healing, so we choose -1000.) In the case of the troll, we need something to intervene:

```
Standard AI action select rule (this is the only the troll
  considers slime-healing rule):
   choose row with an Option of the action of the global
     attacker slime-healing in the Table of AI Combat Options;
   if the global attacker is the large troll:
      change the Weight entry to 15;
      decrease the Weight entry by the health of the large
        troll.

Standard AI action select rule (this is the troll heals only
```

```
  20 percent of the time rule):
   if the global attacker is the large troll:
      choose row with an Option of the action of the global
        attacker slime-healing in the Table of AI Combat Options;
      if a random chance of 4 in 5 succeeds, decrease the weight
        entry by 100;

Standard AI action select rule (this is the troll doesnt heal
  when player almost dead rule):
   if the global attacker is the large troll:
      choose row with an Option of the action of the global
        attacker slime-healing in the Table of AI Combat Options;
      if the health of the global defender is less than 8:
         decrease the weight entry by 8 minus the health of the
           global defender.
```

For the most part, these AI rules should be obvious enough. We want the troll to use its healing ability when it is wounded, so we give the healing a weight of (15 - health of the troll), which starts out at -15 and then slowly rises to 14 as the troll's health goes down. We also decrease the weight of the action if the global defender (in our game always the player, since he is the only opponent of the troll) is very low on health – in that case, the troll ought to prefer simply attempting a fatal blow.

Perhaps less obvious is the line that decreases the probability of the troll healing itself by 100 in 80% of the cases. The reason to add this is that we want to limit the troll's ability. If it could heal itself every turn, the player would never be able to kill it. There are other ways of limiting the use of an ability – a mana cost, a cooldown period – but if an ability can only be used by NPCs, making them not choose it with a certain probability is the easiest way, implementation-wise, to limit an ability.

3.1.11 Victory!

When the troll is dead, the adventure is over. We need a way to indicate victory.

```
Every turn when in the grassy meadow (this is the victory
  rule):
   if there are no hostile alive persons:
      end the game saying "You emerge into the sunlight,
        victorious!".
```

We've done it – created a small adventure that can in fact be won, but only with effort and tactics. (I just managed to beat it, but I only had 1 health left.) Of course, it still needs a lot of polish – especially in the prose department. The prose for all the monsters needs to be customised, we need room descriptions and descriptions of the items to actually tell the player what they are. We also very probably want to indicate how much damage the monsters have sustained – either by printing something about this in the status bar, or by indicating it in the prose after a hit has been scored, or in the descriptions of the monsters printed after an examine command. Also, we are printing the player's score to the status bar, but we're not awarding him any points!

But none of that is any different from what you would normally have to do when writing a piece of interactive fiction. The only ATTACK-specific things left to do are creating some test commands, and changing Save and Undo.

3.1.12 Test commands

If you're testing a game that uses ATTACK, you will want to do two things in a special 'not for release' section. First, you will turn off randomness, so that your replays will actually replay. Here's how to do that:

```
Section - Testing - Not for release

When play begins, seed the random-number generator with 1081.
```

or any other number, of course. Then, we add some cheating commands to get through the fights we do not want to test:

```
Plunking is an action applying to one thing.
Understand "plunk [something]" as plunking.

Carry out plunking:
   say "You plunk [the noun].";
   now the health of the noun is -5.

Plonking is an action applying to nothing.
Understand "plonk" as plonking.

Carry out plonking:
   say "These fools are no match for you, my lord!";
```

```
repeat with X running through all alive persons enclosed by
  the location:
   if the faction of the player hates the faction of X:
      now the health of X is -5.
```

Plunking kills a single person, plonking kills every enemy in the room you're in. You'll get a programming error when you try to plunk a non-person, but hey, this stuff is not for release anyway.

3.1.13 Save and Undo

What to do with save and undo? You can do nothing special, you can disable one or both of them, or you can somehow limit them. Perhaps saving and undoing are only allowed outside fights, or perhaps saving can only be done in certain places, or at the cost of certain items. These are important decisions, and you should consider your game design as a whole.

If you use ATTACK only to spice up a few scenes in an otherwise non-combat, non-tactical game, you're probably best off leaving Save and Undo intact. The farther you move to a hard-core tactical game, the nearer you can go to the *Nethack/Diablo* paradigm, where undo does not exist, and saving immediately quits the game. (You cannot stop the player from making backup copies of his save files, of course, but anyone low enough to stoop to *save scumming* isn't worthy of your game anyway!)

One neat trick is provided by the *Dice-lock* extension written by S John Ross. You can have that extension *lock the dice* before every action, with the result that any player hoping to change the random factors in their rolls by undoing and trying the action again will be foiled. (But unless you disable undo completely, or disable it during combat, the player will still be able to undo their actions and try other actions instead.)

Another possibility is using my *Permadeath* extension, which implements rogue-like saving: you can have only one save per game; saving immediately quits the game; and dying makes your save unrecoverable. This extension can be found on the Inform 7 website.

For the present game, I suggest we disallow undoing when a fight is going on; and allow saving only in the grassy meadow. The latter is easy

```
Check saving the game (this is the can only save in meadow
  rule):
   if the location is not the grassy meadow:
      say "You can only save outside, where God can see you."
        instead.
```

although it does involve a dubious theology. Messing with undo, however, is *not* easy, because *there is no undoing action*. That's right. Undo has been hard-coded into Inform 6, probably for historical reasons. Luckily, we can use the *Conditional Undo* extension by Jesse McGrew.

```
Include Conditional Undo by Jesse McGrew.

Rule for deciding whether to allow undo (this is the no undoing
  during a fight rule):
   if hate is present:
      take no time;
      say "You cannot undo during a fight.";
      deny undo.
```

Which concludes this example game. By the way, feel free to use any code from this example game (which I hereby release into the public domain[1]), and, for that matter, any ideas in this manual that strike your fancy. Originality does not exist. We're all footnotes to Plato. Or the Jahwist. Or Gygax.

3.2 Further recipes

Here I hope to add further recipes in future releases. Some staples of the RPG-genre might be useful – limiting the use of special abilities with something like mana, summoning allies to your side, necromancy, ranged weapons (which could be useful in rooms with "sub-locations"), and so on.

[1] I'm talking about the example game here, not the ATTACK extension itself.