A Guide to TAG Design

by

Few Jar Smith

Table of Contents

1 – Introduction

This guide assumes that the reader has played a TAG such as “The Quest of the Ruby Red Dragon” and is wanting to learn how to write their own game - see **Appendix A – TAG Webware**.

There is a learning curve here – so the reader needs to be determined. Creating a TAG is **writing intensive** – so a real keyboard is very useful.

Hint: print out this guide so that it can be reviewed while designing a game. Here is the copyright notice allowing one to do so:

NOTE: Document cross references are in colored text such as **5 - Good Design** - and depending on the pdf viewer in use can be clicked (or control-clicked) to jump to that reference.

NOTE: This guide often refers to various pieces of text that appear on the computer screen. Using quote marks every time would become very confusing. Consequently most pieces of quoted text are shown in **bold** face. And computer scripts are often rendered in Courier font and inside a gray box like thus:

```
this is a piece of game scripting,
literally what would show
on the computer.
```

When a person plays a language based adventure game, the computer displays a series of situations that the game player must imagine themselves to be in. This imaginary self is called the *hero*, and is distinguished from the actual human sitting in front of the computer (who is called the *player*). The player types commands into the computer telling the hero what to do. Usually the hero will discover a number of tools which the hero can acquire to help win the game. These tools are also called “movable objects” whereas stuff that always appears in a particular scene is called “furniture”.

What the imaginary hero is supposed to see is written on the computer screen. The hero is capable of doing whatever the game designer writes into the game. The human person who designs a game for someone else to play is called the *author*.

At the beginning, the *player* and *author* are the same person – who takes turns designing parts of the game and then playing the game to see that those parts work as desired – then going back to fix and create more parts. Later the author will want a separate person to be the player – since this will expose problems the author missed. When doing this it is important NOT to INTERRUPT or HELP the player – just quietly take notes.

The commands "understood" by the hero are a kind of simplified English such as:

```
get sword
kill dragon
push button
go north
climb up stairs
put medicine in cauldron
shoot werewolf with silver bullet
```

To create a consistent illusion, the adventure game keeps track of a variety of information. In particular it needs to keep track of:

which scene the hero is in,
which scene each tool is in,
which tools the hero has hold of.

In addition a game might keep track of:

which creatures are still alive in which scenes,
which doors have been unlocked,
how much money the hero has,
how much longer the hero's torch will give light,
etc. etc.

A TAG game has a variety of mechanisms for doing this bookkeeping.

To keep track of game information it is necessary to make up a separate name for each of the various objects, scenes, and creatures in the game.

A name must be chosen for each scene the hero might be in and for each object that can be moved from place to place.

Thus, an object that can be moved from scene to scene by the hero would have its own name. Objects that only appear in a particular scene need not be named and are called "furniture". There also needs to be a name for each creature that exists independently of a scene. However a "room monster" that only appears in one scene can be considered to be part of the "furniture" of that scene and so need not have its own name.

Each name chosen by the game author must be different and must satisfy the following conditions:

(1) A name must begin with a lower case letter.
(2) A name cannot have spaces, or any kind of punctuation in it – except for the apostrophe (') and the underline (_).
(3) A name cannot have any capital letters in it.
(4) A name *can* have numerals (i.e. digits) in it.

Some examples of legitimate names are:

```
john            silver_dragon            r2d2
i'mallright      xb3_'ntv
```

and some examples of invalid names are:

```
John     red wagon     r2-d2     im!allright
k2f#37z
```

However, the author should use names that will be meaningful to the player. In particular the names of scenes should be meaningful since they are displayed in a modified form during the play of the game.

A novice author will have some difficulty because of their natural tendency to capitalize the first letter of a name and the first word of a sentence. **Beware!!** Capital letters have an unusual role in TAG design and their traditional English use will have mysterious effects.

Each capital letter **A** through **Z** is used to stand for ***something else***. This can be a window on the computer screen or a data table of management information or a set of "rules" that understand player commands or just a small piece of information such as a name or a number.

When a capital letter is used to keep track of a name or number, it is called a ***setting*** – some other people might call it a *variable*. What a setting keeps track of is called its ***value*** and can be changed in the game from time to time. (This has very little to do with the x, y, and z of Algebra class).

2 – A Small Sample

2.1 Start a Game

To provide a background for future discussions, the reader should become an author and actually create a simple game.

This game will be called **simple**. It will consist of just three scenes named as follows:

```
living_room      dark_forest      tree_top
```

Notice the use of an underline instead of a blank in the names.

What the hero (i.e. the player) will see in each of these scenes is described below:

`living_room`

> You are in your living room. A beautiful oriental rug covers most of the floor. Through a large picture window, you can see the residential area outside. A hallway leads from the living room to your front door. Another door opens into your family room. There are several lovely large framed paintings, one of a dark forest, one of sailing ships, and one of a circus.

`dark_forest`

> You are in a dark and gloomy forest, surrounded by huge oak trees. Moss is growing on the north side of each tree.

`tree_top`

> You are at the top of the oak tree. There is a giant bird's nest. In the bird's nest is an egg of gold.

The game is going to allow the hero to magically travel to the forest by touching the painting of the forest. Once in the forest, any attempt to go north, south, east, or west will leave the hero in the same scene – however the hero will not know this. That will give the illusion that the forest is very large. Finally, the hero will be allowed to climb a tree to get to the scene with the golden egg. When the hero picks up the egg, the game will be over.

To begin with this simple game will not deal with tools, only with the movement of the hero from place to place.

To create this game, locate and select the **designTag.html** web page (see **Appendix A – TAG Webware**). Once that application starts it will show two areas. On the left will be a game editing area colored aqua and on the right will be a game playing area currently holding a copyright notice.

The application menu bar is colored violet near the top of the screen. Each button (except **BEGIN** and **PLAY TAG**) will produce a drop-down menu of selections.

It may not be necessary, but click on **DESIGN** and select **New Game**.

Clicking on **ROUTINES** will show a list of named items called *routines* – such as **introduction**, **setup**, **arrival**. These items are automatically put into each new game – they provide management for the game and will later contain a *routine* to manage each of the scenes added to the game. The author can change any of these routines. One of the routines – **introduction** – manages the very first scene the player encounters. It is a good place to put game author copyright information, if any.

The author must create a routine to manage each of the scenes of the game. What is written inside a routine is called its *script* – in the sense of the script of a play telling the actors what to say and do.

For now (after clicking **ROUTINES**), select the **introduction** routine. A small editing page should appear below in the game editing area. It may be arranged in lines differently than shown.

```
you are playing a text adventure game.
here and there you will find things
useful for
your quest.
some commands are\:\n \b8 look at, go,
list my stuff, quit.
\n\n press return \a
G room1
```

The peculiar notations **\: \n \b8 \a G room1** will be explained later. Also note that the sentences of the description are not capitalized.

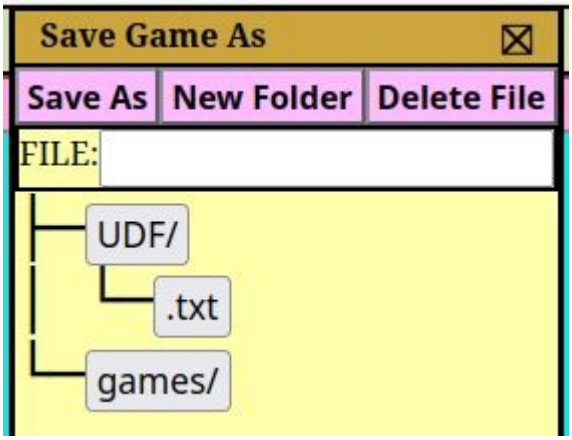
Change the last line so that it now reads:

```
G living_room
```

which means that after reading the introduction, the player is to be sent to the **living_room** scene.

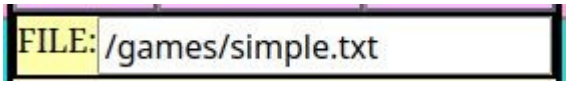
2.2 Save the Game

Now is a good time to save the game. Click on **DESIGN** and select **Save Game As** and a small image of the **Virtual File System** (see **Appendix B – Virtual File System**) will appear:



showing a button for the **/games/** folder:

Click that button and finish typing in the name of the game:



where the **.txt** will be automatically supplied, should the reader forget it.

Then click **Save As** and that action should be acknowledged in a status message at the top left side of the web page.

After having saved the game once and thus chosen its file name (here it was "/games/simple.txt"), the author can more easily save further changes by clicking **DESIGN** and choosing **Save Game**. The application will remember the game file's name. On computers with a keyboard, typing a Ctrl-S will often save the game as well.

2.3 Initial Test

There is not much in the game yet, but it *can* be tested. Click on **BEGIN** and the game playing area will display the introduction just edited – except without the peculiar notations.

Now act as the game player and press the return/enter key as requested.

Notice that the right hand game playing area now has two display areas titled **Introduction** and **Commands**. These are called *game windows*. There is an error reported in the **Commands** game window:

```
game halted: No such destination as
living_room from introduction
```

which is correct because we have told the game to send the hero to a scene named **living_room** but have not yet created such a routine.

2.4 Create Living Room

Click **ROUTINES** and select **[NEW ROUTINE]**. The author will be asked to provide a name for the new routine – it may suggest one. In any case use the name:

living_room

and click **MAKE**.

There will now be an editing box for that routine. It will appear in the aqua colored panel of the application on the left. The editing boxes for the routines being edited will each have their own title bar. An editing box can be moved around by dragging its title bar. On a laptop or desktop computer, the editing box can be resized by dragging its lower right corner.

Or one can use the  symbol in the editing box title to enlarge the box and the 

resulting symbol in the editing box title to resize it.

To dismiss an editing box, click on the  symbol in its title bar. That will NOT

destroy the routine – it can be redisplayed by selecting it from the **ROUTINES** menu.

Type the following script into the editing box for the living_room:

```
you are in your living room.  a beautiful
oriental rug covers most of the floor.
through a large picture window, you can
see the residential area outside.   a
hallway leads from the living room to
your front door.  another door opens into
your family room.   there are several
lovely large framed paintings, one of a
dark forest, one of sailing ships, and
one of a circus.
```

being careful NOT to use any capitals.

The arrangement of the script into lines is not important since the TAG game window automatically re-formats it to fit.

Now save the game again and click **BEGIN** to test it.

Become the player and press the return key as requested. The game will send the hero to the **living_room** and display what they see in that scene. A second game window titled **Commands** will appear for the player to send commands to the hero.

The upper game window will now be titled **Living Room** since the game program automatically modifies the routine's name when displayed as a scene title. It will capitalize each word of the title, replace underlines (_) with spaces, and omit any numerals. Hence the author can name several similar but different scenes with names like:

room1 room2 room97

without the hero/player being clued that they are different. This trick is used in the "downs" scenes in the ruby game.

2.5 Hero Movement

Back to game editing.

To allow the hero to obey player commands, the commands must be written into the script of the scene where they apply. Below are three such command "rules" that should be added to the bottom of the edit box for the **living_room** routine

```
[go hallway = there is no point in doing that.]

[go family room = it's too noisy in there]

[touch forest painting = you are getting dizzy!
    and seem to be falling into the picture
    ... :p10 G dark_forest]
```

All of these "rules" have the format:

[*command* **=** *consequence* **]**

and the last rule uses these *actions*:

:p10 pause for 10 tenths of a second

`G dark_forest` move the hero to the **dark_forest**

All the things done in the *consequence* of a rule are called *actions*. Running the actions of a rule is called *firing* the rule. The script of a routine is also just a series of actions. Most of these actions just display text in a game window. Other actions are written using capitals or punctuation and have game management effects.

The setting **G** is used to tell the game where the hero should be sent to. It's value should be the name of the routine describing that scene.

The game management routine named **play** monitors the value of the **G** setting and whenever it detects a change, it moves the hero to the scene **G** now refers to. The play routine will be described in another document. However the author can view the routine by selecting it from the **ROUTINES** menu.

The first two rules above simply display text in the **Commands** game window. They give the player excuses why the hero will not follow such perfectly reasonable commands. These are called "put-off" rules and should be used sparingly. Often the author will later replace them with more responsive commands.

Another example of a put-off would be the rule:

```
[ look under rug = it is very dirty under
there. ]
```

Notice that the words in the consequence part of a rule are usually displayed in the **Commands** window and the words describing a scene are displayed in the scene window. Again, that is managed by the **play** routine.

Here are more rules to add to the living_room script:

```
[ touch painting = touch which painting\? ]
[ touch (circus/ships) painting = nothing
happens. ]
```

The first of these rules is a "hint" to the player that suggests a more complete command.

Notice the use of the action **\?** to actually display a question mark – necessary because a **?** signals a management action known as a *query*.

The second rule introduces a new feature

```
(circus/ships)
```

which allows the rule to match either of two different commands:

```
touch circus paintings
```

or

```
touch ships paintings
```

.

The ***pattern*** portion of a rule in front of the equal sign **=** is more than just the sequence of words desired in the command and its elements can match a variety of different command words or sequences of command words. See **9 - Command Patterns** for more detail.

The player should now **BEGIN** the game again and experiment with all the commands except the “forest painting” command

Finally, the player should try the command:

```
touch forest painting
```

which will cause an error reported in the status line:

This caused by an error deliberately introduced in a rule given previously.

Look at the text of the **living_room** routine and find the notation:

```
dizzy! and
```

which was meant to add an exclamation mark after the word **dizzy**.

Instead of being a character to be written to the game window, the **!** is a management operation asking for a routine named **and**. But there is no such routine (none is wanted) and that caused the error message.

Replace the notation **!** by **\!** and that problem will be solved. Remember that almost all punctuation has a special management function. To avoid that function, prefix the punctuation with a back slash **** .

However, after testing the game again, the command to touch the forest painting will now produce a different error message:

```
game halted: No such destination as
dark_forest from living_room
```

The reader has already learned about this kind of error and how to fix it - namely, create a **dark_forest** routine:

```
you are in a dark and gloomy forest,
surrounded by
huge oak trees. moss is growing on the
north side of each tree.
```

and while at it, create a **tree_top** routine:

```
you are at the top of the oak tree.
there is a giant bird's nest.
in the bird's nest is an egg of gold.
```

When one routine edit box covers up another, it can be dragged around by its title bar or dismissed by clicking the ☒ symbol in the title bar. Dismissing an edit box does NOT destroy the routine and it can be retrieved again using the **ROUTINES** menu. On a computer with a mouse, the lower right corner of an edit box can also be dragged to change its size.
When the game runs the actions of a rule's consequence, the rule is said to be *fired*.
Now add the following rules to the **dark_forest** routine:

```
[ go (north/south/east/west) =
        okay S0 ]
[ climb tree = G tree_top ]
```

which allow the hero to appear to move around in the forest or to climb a tree. The **S0** action doesn't actually move the hero – they stay in the **dark_forest**.

Start the game again and play until the hero gets to the tree top.
And finally add this rule to the tree-top scene:

```
[ get egg = the egg is made of pure gold.
  this will provide you with enough money
  to live in comfort for the rest of your life.
  the game is over. :q ]
```

The action **:q** stops the game – no more commands can be given.

It is a pretty simple game, but can be played to its conclusion.

Don't forget to save it.

2.6 Restart Game

Before reading the next Section, it would be well to acquire another authoring skill.

The author will have noticed that changing a routine in the game designer side does **not** immediately fix problems in the game player side. There are several reasons for that but the most important one is that:

The player side does <u>not</u> actually play the game the author is writing.

Instead it plays a modified version of the game called a **.tag** file. This is encoded in binary to prevent other players from easily seeing the secrets of the game. Any updates to the design have to be re-encoded into a new **.tag** file.

The **BEGIN** button not only starts the play of the game, but it also encodes the entire game design file into its **.tag** form.

Consequently getting design changes to work requires use of the **BEGIN** button and thus the player will start the game *all over again* after each update of the design (ouch !!!)

Unfortunately the SAVE and RESUME buttons will not help because a saved game position only applies to the exact version of the game used to produce it.

But it **is** possible to fast forward to a scene after **BEGIN** -ing the game again. Simply bring up the edit box for that scene (using **ROUTINES**) and click its **GO TO** button. Typically that edit box is already present because the author just made changes to that routine.

BEGIN the game and fast forward to the tree top and get the egg.

3 - Distributing a Game

A TAG game can exist in three forms – all of which are text files and can be attached to an email, put on a thumb drive, printed out, etc. They all can be viewed, but only the **.txt** design file will make much sense.

The original game design text (like what was created in Section 2) is called a *design* file and should have a file name extension of **.txt** – since it is a readable text file.

The author is unlikely to want to share all their clever tricks and secrets – much less a road map to winning the game. So the game can be encoded and saved as a **.tag** file in which all the game text and logic has been hidden using a binary coding. This is a good form for distributing the game to other game players.

Finally, the game can be converted to a ready-to-go web page **.html** file. If this is hosted on a web site, it can be played simply by clicking on its name. If it is downloaded to a computer, the internet browser on that computer can locate it as a local file and execute it by selecting it.

In every case, begin by starting the **designTag.html** application and clicking **DESIGN**. Then select **Load Game** and choose the game to be distributed.

To produce a web page for the game, click **DESIGN** and select **HMTL to UDF**. The result will be placed into the **UDF.**

The UDF is explained in appendix **Appendix C – The UDF**. From that point on, the file is part of the user's file system and can be treated however the user wants.

To produce a **.tag** file, click **DESIGN** and select **Tag to UDF**. The result will again be placed into the UDF.

When the file in the UDF is distributed, the recipient should save it in their file system. Most browsers allow the user to find and open a web page (i.e. an **.html** file) in their file system. Or if the file appears on the user's desktop, selecting it will usually open it with the default web browser.

If the distribution was a **.tag** file, a good place to save it is the **UDF** on the user's computer. Then the recipient can play it by starting the **designTag.html** application (or

the **playTag.html** application) on *their* computer, clicking on **PLAY TAG**, selecting **/UDF/.tag** , and choosing the distributed file from the resulting downloaded file window.

The **.html** version of a game includes a cut-back version of the game play logic from the playTag application as well as an encoded version of the game design. So that is larger than just the **.tag** file version of the game.

The application **playTag.html** only contains the game playing logic from the designTag application. A user having **playTag.html** need only save the **.tag** versions if they keep a number of games to play.

4 - Movable Objects

4.1 Tools

Movable objects (aka tools) are things that the hero can find in one scene or another, pick up, carry around, use, and put down. Because such an object has an existence independent of any particular scene, it needs to have is own named routine to explain it.

For example, consider the "simple" game designed in Section 2. Suppose that a "rope" is to be added to the game. The purpose of the rope is to enable to hero to climb a tree. The rope will be placed in the family room to begin with. Without the rope, the hero should be unable to climb the tree.

Load the **simple.txt** game and select the **living_room** routine. It currently should read:

```
you are in your living room.  a beautiful oriental
rug covers most of the floor.  through a large
picture window, you can see the residential area
outside.  a hallway leads from the living room to
your front door.  another door opens into your
family room.  There are several lovely large framed
paintings, one of a dark forest, one of sailing
ships, and one of a circus.

[go hallway = there is no point in doing that.]
[go family room = it's too noisy in there]
[touch forest painting = you are getting dizzy!
    and seem to be falling into the picture
    ... :p10 G dark_forest]
[ touch painting = touch which painting\? ]
[ touch (circus/ships) painting =
    nothing happens. ]
```

Change the **go family room** rule (highlit in pale green) to be:

```
[ go family room = G family_room]
```

and create a new scene named **family_room** that reads:

```
you are in your family room.  the kids have left it in
an awful mess.  there is junk strewn all over the
place. the only door goes back into the living_room.
[go living room=G living_room]
```

At this point the novice game author is tempted to make a mistake. Since the rope is originally found in the family_room, it seems natural to place the description of the rope directly into the **family_room** routine shown above.

DO NOT DO THAT

4.2 Creating Tools

There will need to be new routine named ***rope**** which will describe the rope and explain what the rope can do. This kind of routine is called a "tool" and its name should have an asterisk at the end.

If the author forgets to put an ***** at the end of a tool's routine name, the lack can be repaired in its editing box by clicking **MORE** and selecting **Be tool**.

Create the routine **rope*** and edit it to be:

```
-a +strong+nylon rope =
[ climb tree = G tree_top ]
```

Instead of writing the "rope" into the family_room scene, edit the (already existing) **setup** routine so that it has this line at the bottom:

```
@ rope family_room
```

which tells the game management routines to show the rope when the hero goes into the **family_room**. The above action adds information to something called the @ table.

The action of the **–** and **+** signs will be explained later in **9.1 Pattern Elements**

The player should now try **BEGIN**-ing the modified game. Once the player gets the hero to the family room, the player should try each of these commands:

look around

list stuff

get nylon rope

look around

list my stuff

drop rope

look around

list stuff

get strong nylon rope

list my stuff

to see what they do. Notice how the game management is creating the illusion of there being a rope in the family room – without ever changing the description of that scene in its game window.

4.3 View @ Table

Click **DESIGN** and select **See Tables** and the editing side of designTag will now show the actual content of the @ table. The player can watch its content change as the player exercises the **drop** and **get** commands. This table will not be visible to another player who only has the **.tag** form of the game.

Since a rope is now available for climbing trees, the rule

```
[ climb tree = G tree_top ]
```

should be removed from the **dark_forest** scene. Do that.

The player should now restart the game with **BEGIN**, go to the family room, get the rope, and go to the living room.

Please Note: From now on, directing the player to follow a game playing command really means to have the player give the hero that command.

Just to try it out, drop the rope (in the living_room) and touch the forest painting, arrive at the dark_forest and try to climb a tree. The hero should now be unable to do that ! ! !

Now restart the game, go through all of the steps, only keep the rope and then touch the forest paining. The hero should now be able to climb the tree.

4.4 Alternative Action

Unfortunately, all this editing has allowed a strange thing to happen.

Restart the game, go get the rope, but stay in the family room. Then try the command:

climb tree

Oops!!! Because the **climb tree** rule is now part of the rope, and the rope is with the hero, its rule is active and can be used to climb a tree the hero cannot even see.

To prevent the rope's unwanted activation, change the rope routine's rule:

```
[climb tree = G tree_top ]
```

to now read:

```
[climb tree = ?S=dark_forest  G tree_top /
    what tree\? ]
```

which introduces a new kind of action called a *query*:

?S=dark_forest

and a new idea called *alternative action*.

A game can query data and act differently depending on the response. A part of a script that depends on a response is called a *alternative*. Typically there is one alternative when the response is true and another alternative when it is false. This is written in a game using a question mark **?** and a slash **/** like this:

? *query what-to-do-if-response-is-true* **/**
what-to-do-if -response-is-false

In the above "climb tree" rule, the consequence of a matching command was:

```
?S=dark_forest  G tree_top /
    what tree\?
```

The **?S=dark_forest** part is the query and asks whether the hero is in the dark_forest or not. The **G tree_top** part is the action chosen when the hero is in the dark_forest and the **what tree\?** part is the message displayed otherwise. The **S** setting always keeps track of what scene the hero is in. Writing each alternative on a separate line is an authoring choice and makes no difference to the play of the game.

Try this out.

Save the game.

4.5 Challenge

There is another problem. After the hero gets the egg and wins the game, the egg still appears in the nest. Since the player has won, perhaps this is not worth worrying about.

If the author decides to fix this, they should first save the game under another name and only make changes to this new version of the game. The unchanged version of **simple.txt** game will be used in later sections of this guide.

To save a game under a new name, click **DESIGN** and select **Save Game As**. Click the button for the /**games/** folder. Then complete the new name in the **FILE** box and click **Save As**.

The problem occurred because the egg is really a movable object, but the game designed it as "furniture".

So, the author needs to create a tool named **egg*** perhaps with a description such as:

```
-a +shiny+golden egg =
```

and needs to move the "get egg" rule out of the **tree_top** routine and put that rule in the **egg*** routine. The normal browser cut and paste operations can be used.

Since the egg is no longer furniture, remove its description from the **tree_top** description. and add a line

```
@ egg tree_top
```

to the **set_up** routine. Now the game management routines will automatically handle showing the egg and getting the egg.

However, it does not make sense to "get" the egg, when the hero already has it, so the rule that was just moved from the **tree_top** into the **egg*** routine needs to be altered somewhat. Change its command pattern from:

```
get egg =
```

to something like:

```
look +at egg =
```

The game could now be extended to require the hero to somehow get back to the living_room and maybe take the egg to a bank to cash it in, etc. etc.

Getting the hero back to the **living_room** could be as simple as adding a **`rub egg`** rule with a **`Gliving_room`** action or allowing the hero to wander the forest and find a

magical door in one of the tree trunks or even something more elaborate wherein the real goal of the adventure is not getting the egg at all, but rather somehow getting it back home.

By the way, a **rub egg** rule needs to have some kind of indirect hint in the description of the egg or perhaps a **look at egg** rule in the egg tool itself:

```
[look +at egg = it is really smooth to the touch.]
```

DO NOT simply have the game tell the hero to rub the egg although a good TAG player does know they need to "look at" stuff.

Maxim:

> The player commands the hero,
> The game should NOT.

5 - Good Design

Writing a good game is different from writing a good novel or short story, although they both require a skill with words. However, a novelist is not only master of the "world" where the novel takes place but is also master of the protagonist(s) – the hero(s) do exactly what they are written to do. On the other hand, a TAG author is master of the game "world" and can limit what the hero can do **but** the ***player*** controls the hero, not the author. This is hard to see at first because the author takes turns being the player.

The following is an example of the dialog for a "game" where the author is acting as a novelist instead of a game designer:

> **You are playing a text adventure game.**
> **Press the return key for each command.**
> **----**
> **You are in your living room. There are pictures on the wall one of which shows a circus, one a dark forest and one some sailing ships.**
> **Press return**
> **----**
> **You go in the family_room you see a rope on the floor and pick it up.**
> **Press return**
> **----**
> **You go into the living_room**
> **You are in your living room. There are pictures on the wall one of which shows a circus, one a dark forest and one some sailing ships.**
> **Press return**
> **----**
> **You touch the forest painting**
> **After fainting, you find yourself in a dark forest.**
> **Press return**
> **----**
> **You climb a tree.**
> **You are in a nest with a golden egg.**
> **Press return**
> **You pick up the egg and become very wealthy and happy. The game is over**

All the player can do is press the return key when told to.

Select https://fewjarsmith.com/Tag/badgame.html to run this game, keeping in mind that it is an example of what NOT to do.

The **badgame** breaks some rules of good game design:

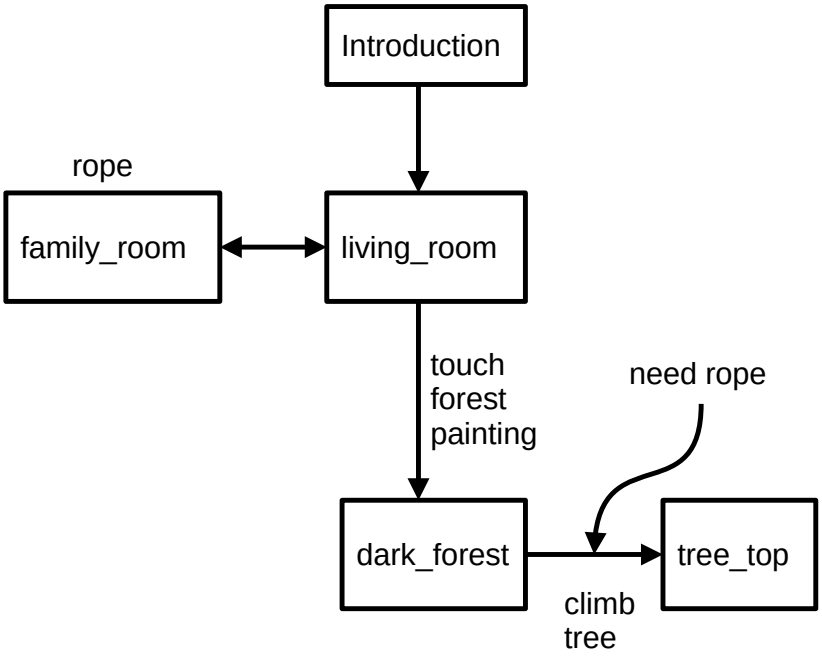
1. DON'T Boss the Hero.
2. DO Let the Hero make mistakes.
3. DOORS should open BOTH ways – in other words:
   If the Hero can ENTER somewhere, then the Hero can LEAVE.
4. If the Hero can GET something, then the Hero can DROP it.
5. DON'T describe Hero activity in the Scene window.

For example, the **simple.txt** game designed in Section 2 breaks rule 3 twice. Once the hero touches the forest painting, there is no way back to the living room. And once the hero climbs a tree, there is no way to climb down. Rule 4 is pretty much guaranteed by the basic game management routines and the @ table. Rule 2 is somewhat obeyed since the hero can touch the "wrong" paintings, but overuse of "put-off" rules should be avoided.

These design flaws are to be expected since **simple.txt** was an overly simplified training example. The reader could try to expand the game to improve it following the above rules. Of course, all rules are made to be broken – but always with good reason.

In any case, the reader should design a more elaborate game of their own before trying to read the following more technical sections of this guide, although the reader could first do a quick scan through Section 6 without getting too bogged down in the detailed logic of flags, settings, and groupings.

A useful tool for planning a game is a conceptual map of its scenes. Use a labeled box for each scene and draw arrows showing how the hero can move from scene to scene. Most arrows should point two ways (Rule 3). Draw this in pencil so it can be expanded and revised a lot. Below is a map for the simple game developed in Section 2. The map makes it clear that rule 3 is broken a lot since most of the arrows are one way.


Introduction
rope
family_room
living_room
touch
forest
painting
need rope
dark_forest
tree_top
climb
tree

6 - Action Logic

6.1 Definitions

When the designTag or playTag applications interact with the game player they follow the scripts coded in the **.tag** file. Each routine has its own script. When the application follows the script for a routine, it said to be *running* that routine. Sometimes the words *performing* or *executing* will be used instead of *running*. Each script is composed of a series of *actions*. Some actions can be arranged in groups using various kinds of parentheses and the slash mark(**/**).

The actions in a script can display information in a game window, change the values of *settings* and *flags*, *accept* input from the player, *query* the values of settings and flags, interact with data *tables*, and direct the application to choose between *alternatives*. Certain actions can visit another routine and come back to finish running the first routine. This can result in visits to routines that visit other routines which visit other routines, etc. – eventually returning to the original routine. Much of the organization of a game can be managed by such visiting – for example, the **play** routine *visits* each of the scene routines when the hero enters them.

The notation for visiting a routine is just an exclamation mark (**!**) in front of its name and can be read as “execute this”.

The basic unit of information in a game (called a *value*) is a number or a piece of text. There is no limit on the size of the text, but it is usually fairly small. Text values which are like a routine name need not be quoted. Other text values should be placed inside a pair of quotation marks **"**. A value can be kept in a setting, in a data table, or in a flag. Tables will be explained in Section **7 - Data Tables**. Settings have already been introduced in Section **2 – A Small Sample**.

Recall that a *setting* is named by a capital letter (**A**-**Z**) and is shared by all of the routines of the game. A *flag* holds one piece of data and has a lower case name (like a routine name) and belongs to a particular routine known as its *owner*. The flag name can refer to different flags depending on the owner and each can hold different data. Some of the standard basic management routines use a flag.

By default a flag’s owner is the name of the routine where the flag is written. To specify some other owner, use the notation:

```
flag ^ owner
```

NOTE: each time **BEGIN** is clicked, **all** the flags of **all** the routines are turned off.

A series of actions from a routine can be arranged into a *group*. This is done by placing a pair of parentheses around those actions. There are four kinds of groups. The ordinary **(** and **)** will form a *plain* group or an *indexed* group. The braces **{** and **}** will form a *repeating* group. And the brackets **[** and **]** help form a *rule*.

A group can be divided up into two or more *alternatives* using slash marks (**/**).

These groups must be properly nested inside one another. A smaller group inside a larger group is called a *child* and the larger group is called its *parent*. The child must fit completely into one of the alternatives of the parent. Thus a slash mark appearing in a child divides up the actions of that child but has no such effect on the parent.

6.2 Plain Groups

The purpose of dividing a group into alternatives is to allow the response of a query to direct the subsequent execution of a script.

Remember that a plain group begins with an open parenthesis **(** and ends with a closing parenthesis **)** and is divided into alternatives by slashes **/** .

In general, a plain group is run as follows -

1. It starts with the first alternative, running its actions, queries, and children in order left to right.

2. If it encounters a slash mark, it skips all the remaining alternatives of the group and so finishes the group.

3. If it encounters the closing parenthesis, it has also finished the group.

4. If it encounters a query responding true , it continues running the same alternative.

5. If it encounters a query responding false, it skips the rest of that alternative and continues with the next alternative (if any). If doing this skipped the last alternative, then step 3 applies.

6.3 Experiment

It is possible to experiment with many game features without actually having to write an entire game. The author can click the **DESIGN** menu and select the **Empty Game** option. An empty editing box will appear for the **begin_game** routine – which is the one routine every game must have. In general whenever the author clicks the **BEGIN** button, it is the **begin_game** routine which starts the game running.

To try this out, select the **Empty Game** option and put some message (all lower case letters and no punctuation) into the **begin_game** routine and click **BEGIN**.

In general, to experiment with some game feature, just write the feature into the **begin_game** editing box and click **BEGIN**. A list of possible script actions can be found in **Appendix E – Action Reference.**

Here is a small script to try:

```
+m \v$m
-m \v$m
$m help \v$m
```

If this was done correctly, the game window will now show:

```
1 0 help
```

As shown above, one way to use game data is to display it in a game window using the **\v** action. The data being displayed can be a flag, a setting, or some kind of calculation.

A *comment* is information that can be added to the end of any script line by using a double semicolon **;;** followed by the comment. Comments are never seen by the player, they are a design aid for the author - in this case an explanation of what the script line does. When experimenting with the following script, the comments should be left out.

Below is a script that demonstrates how to display a flag, a setting, and a calculation:

```
;; set up some game data
+b ;; turn on the b flag
H 31 ;; set H to be 31

;; now show that data to the player
the b flag is \v$b \n
the "H" setting is \vH \n
twice 6 is \v(2*6) \n
```

In a real game, do not use anywhere near this many comments since that will obscure the meaning of the script rather than clarify it.

Try out all these examples as before, using the **Empty Game** option.

The last three lines of the above example are not very typical. Usually the player has to deduce what is going on by playing the game. So game data is often used to provide hints. The scripting to do this is called a *group* (see **6.1 Definitions**) and is usually enclosed in a pair of parentheses. The alternatives in the group are separated by the **/** symbol. For example:

```
you are feeling
( ?H<10  poorly / ?H<40 okay / pretty good)
today.
```

which means that If H is less than 10, display **poorly** but if H is less than 40, display **okay**, otherwise display **pretty good**.

In all three alternatives, that display is prefixed with **you are feeling** and finished with **today** and a period. As can be seen, a question mark can be thought of as an “if” and a slash mark can be thought of as an “else” or “but” or “otherwise”.

However, in a real game, flags and settings are set up in one routine and used in some other routine. As another experiment, move the “you are feeling … \n” script into a new routine named **show_health** and change the **begin_game** to be:

```
H31 !show_health
```

Always test run any suggestions made by this guide!

The above experiment only exercised the middle (**?H<40**) alternative. To test all three, change **begin_game** to be:

```
H8 !show_health
H31 !show_health
H43 !show_health
```

Or use the following **begin_game** script to allow the player to choose the values of **H** to be tested:

```
{ what health \a H:a1 !show_health }
```

and since that is a repeating group, the game will ask for the health indefinitely.

To stop an unruly repeating group, change the **begin_game** script and press BEGIN again.

6.4 Nested Groups

Nested groups are typically used when there are separate values being queried. Suppose that the hero’s health is kept in setting **H** and that there is a **sword*** tool. Create this **fight** routine (in an empty game):

```
a huge ogre jumps through the door.
( ?@sword me
   (
      ?H>20 you kill the ogre. /
      the ogre kills you. H0
   ) / you can't defend yourself. H0
)
(?H<1 you are dead. /
    the ogre vanishes leaving a
    pile of coins. M+20
)
```

To exercise it use this **begin_game**:

```
M0 H30 @sword 0
!fight
```

which makes the hero healthy enough but without a sword.

Here is a more elaborate **begin_game** that allows the player to set the health and sword possession before exercising the **fight** routine and would repeatedly test that routine:

```
M0 { \n health and sword\? \a
   H:a1
   (?#:a2 @sword me / @sword 0)
   !fight
   \n "H=" \vH "M=" \vM
}
```

The **\a** action waits for the player to enter a command. Once that is done, the action **H:a1** changes the **H** setting to be the first number the player types and the query **?#:a2** checks whether the second number typed is nonzero. This setup is only for *testing* the fight routine. In a real game, the **H** and **sword** would depend on other parts of the game and the **fight** scene would only be visited once instead of being in a repeating group. The reason for the **M0** is subtle - try leaving it out and killing several ogres. A good rule is to always initialize a numeric setting with an actual number (like 0) before using it.

As practice understanding plain groups and their alternatives, try to predict what each of the scripts below will display depending on the queries involved. Then actually run the script to check the prediction. Use the techniques developed above with **begin_game** setting the flag(s) and a routine named **testme** to hold the script being tested (also see **Answers-1**).

This depends on flag **b^testme**:

```
there is a
( ?b very large / tiny little )
bug on the table.
```

This one depends on flags **c^testme**, **b^testme**, and **f^testme** and provides six different outputs:

```
inside the
( ?c coffin lies / ?b box sits / knapsack is )
a ( ?f frog / lizard)
```

This one has a child group nested inside a parent group:

```
( ?f the fighting ( ?m man / woman ) hit /
   some wimp ran from
) the bear.
```

6.5 Repeating Group

Remember that a repeating group begins with an open brace **{** and ends with a closing brace **}** and can be divided into alternatives by slashes **/** .

In general, a repeating group is run as follows

1. It starts with the first alternative, running its actions, queries, and children in order left to right.

2. If it encounters a slash mark, it **returns to the beginning** of the first alternative

3. If it encounters the closing brace **}**, it **returns to the beginning** of the first alternative.

4. If it encounters a query responding true , it continues running the same alternative.

5. If it encounters a query responding false, it skips to the beginning of the next alternative – however if there is no next alternative then it skips the rest of this last alternative and thus finishes the group – this is the only way out of the group.

Thus the only way to stop a repeating group is to have a query in the last alternative which returns false.

Here is a simple example to try (using **Empty Game**):

```
A1
{ "A is" \vA \n A+1 ?A<9 }
all done
```

This should count from 1 to 8 with each count on a separate line. Its essential parts are:

```
A1
{ . . . . A+1 ?A<9 }
```

As practice write a script that counts from 10 to 90 by fives (see **Answers-2**).

Like a plain group, a repeating group can have several queries – see this somewhat tricky example:

```
M30
{ ?M<40 \vM is small M+5 / \vM is large
M+20  ?M<100 } done
```

Try to work out how it counts using **M** before actually testing it. **Answers-2** has a detailed presentation of its execution.

6.6 Indexed Group

There is one more kind of group called an *indexed group*. It has the form:

```
| formula ( alternative-1 / alternative-2 / etc. etc )
```

and appears much like a plain group. However the game finds the value of the formula and uses it to choose one of the alternatives to be run. A value of 1 chooses alternative-1, a value of 2 chooses alternative-2, etc. The other values are ignored. As an example, suppose setting **H** records how fierce a particular dragon is. Suppose 1 means not very fierce all the way up to 4 meaning extremely fierce. Then the indexed group:

```
| H(irked/rather upset/
    fairly angry/really really mad )
```

would describe the dragon after the hero poked it.

7 - Data Tables

A *data table* is a rectangular table whose rows represent the known instances of a relationship and whose columns contain the terms of those relationships. Each table manages a different relationship.

An example is the following table showing children and their mothers and fathers:

mary	alice	john
alice	jean	peter
john	betty	robert
jean	liza	albert

This table would record that mary was the child of alice and john, alice was the child of jean and peter, john was the child of betty and robert, and jean was the child of liza and albert.

The following script would create such a table (called ***C**) and fill in the indicated instance rows:

```
*C<3 1> ;; table dimensions: 3 columns, 1 key column
*C(mary alice john)
*C(alice jean peter)
*C(john betty robert)
*C(jean liza albert)
```

The **<3 1>** notation indicates that the newly created table should have 3 columns. The **1** indicates that the first term must be unique - no two rows can have the same first term.

Try this out (using **Empty Game** as usual) and select **See Tables** from the **DESIGN** menu. An editing like box will appear showing the various tables. Fortunately someone only playing the game's **.tag** form cannot see the tables and uncover secrets for winning the game.

7.1 Table Queries

Once established, a table can be used (usually in another routine) to query facts about the relationship using either a *table query* or a *table runner*. A *table query* has the form:

```
?table ( selector ) collectors .
```

where *table* is either **@** or one of the ***A** through ***Z** table names; where the *selector* is a parenthesized list of values; and where *collectors* is a list of capital letters naming settings. The selector can be empty and the collectors can be empty.

The query searches the named table to see if the prefix (i.e. front) of any of its rows match the selector. If it finds a matching row, then the query is true and the row terms following that prefix are set into the collector settings. If none can be found, then the query is false and no change is made to the collector settings.

For example, using the ***C** table shown above, the table query:

```
?*C(alice)XY.
```

is looking for a row in the ***C** table which starts with **alice**. It finds **(alice jean peter)** and so the query is true and **X**'s value is set to **jean** and **Y**'s value is set to **peter**.

As another example the query:

```
?*C(john betty)Z.
```

is looking for a row which starts with **john** followed by **betty**. It finds **(john betty robert)** and so the query is true and **Z**'s value is set to **robert**.

As a third example, the query:

```
?*C(mary betty)P.
```

is looking for a row which starts with **mary** followed by **betty**. There is no such row. The only row starting with **mary** is the row **(mary alice john)** and its second column is **alice** not **betty**. Thus the query is false and no change is made to the setting **P**

As an example of an empty selector, the query:

```
?*C( )BV.
```

is looking for any row at all (since **( )** is a prefix of all rows). It finds the first row of the table **(mary alice john)** and sets **B** to **mary** and **V** to **alice**. The size of a selector plus the number of collectors need not total the width of the table and so **john** is ignored.

As an example of empty collectors, the query:

```
?*C(john betty).
```

finds the row **(john betty robert)**, so the query is true, but no settings are changed.

The period ending the collectors list can usually be omitted when the next action does NOT begin with a capital letter. When unsure of that, leave the period in the table query.

If no row matches the selector, then the query is false and no changes are made to the collectors.

If the selector is empty **()**, and the table is not empty, then the query is true and the first row's columns are placed into the collectors.

It is advised that collector letters be **X** or **Y** or **Z** and that those settings not be used to store any game-wide information. Keep in mind that the collector settings will get changed by any table processing.

Here are some more ideas for data tables - the terms of the relationship are in bold face and the table dimensions are indicated at the front:

<3 1> **sword** cost **50** has **medium** strength
<3 2> **sword** against **ogre** has **50**% winning odds
<2 0> **red_key** opens **brass_door**
<4 2> **bobs_shop** has **swords** costing **50** gold, inventory **2**
<2 1> **flash_light** lasts **4** hours
<3 2> from **woods1** go **north** to **woods2**
<3 2> going from **valley** to **hilltop** takes **4** hours
<2 1> eating **apple** increases health by **2** points

7.2 Table Runners

A powerful use of a data table is in conjunction with a special kind of repeating group - known as a *table runner*. It has the general format:

```
~ collectors table (selector) { actions }
```

where *collectors* is a list of capital letters naming settings; where *table* is either **@** or one of the ***A** through ***Z** table names; and where the *selector* is a parenthesized list of values. The selector can be empty and the collectors can be empty.

The **~** symbol could be read as "for every".

Note that for a table runner as opposed to a table query, the collectors come first and capture a prefix of a matched row whose right hand end (suffix) matches the selector.

The table runner will be repeated just once for each matching row of the data table. A matching row is one that has the selector as a suffix - namely whose right hand end matches the selector.

For each matching row of the data table, the left hand column values in the row are placed into the collector settings *before* the actions of the group are run.

However, the usual rules for a repeating group also apply and so a false query in the last group alternative will terminate the group's repetitions prematurely.

A few examples may help in understanding this.

One simple use of a table runner is to display the table in a game window. Suppose the the ***M** table records the relation "**object** colored **color** costs **price**". Here is what such a table might look like:'

sword	silver	35
axe	bronze	21
sword	bronze	15
ring	gold	47
axe	black	24

The following script would display all the rows of the ***M** table. Notice the switch in order for the object and its color - the display order need not be the column order.

```
~XYZ*M() { a \vY colored \vX costs \vZ coins \n }
```

As another example. suppose that the game uses a table ***W** to keep track of how many silver coins each movable object is "worth". Its rows might be something like:

sword	**25**
gold_ring	**13**
secret_map	**52**

Here is a piece of script to calculate the total amount of treasure **T** the hero is "worth":

```
T0
~XY*C() { (?@Xme T+Y) }
;; hero's worth is in T
```

Some notes about the example:

1. The routine name **me** is used to refer to the hero.
2. Trying it will require creating an **@** table.
3. Settings **X**, **Y**, **Z** should be reserved for temporary calculations - particularly for holding values collected out of a table row.
4. Using **( )** as the suffix in the table runner means that it runs every row of the table since **( )** matches the end of every row.
5. The **?@Xme** query needs to be in its own plain **( )** group because its failure would otherwise prematurely terminate the table runner's repetitions.

Here is an alternate script to do the same thing using a table runner for **@** and a table query for ***C** :

```
T0
~X@me { (?*C(X)Y. T+Y) }
;; hero's worth in T
```

Some comments about this script.

The table query **?*C** could be false and **@** might have more rows - so the query needs to be kept in a plain **( )** group so its failure doesn't terminate the table runner.

7.3 Table Deletion

An action of the form:

```
–  table (  selector )
```

will delete all of the matching rows of the table - where the selector is matched against row prefixes. If none of the row prefixes match the selector, then none are removed.

As an example, suppose the the ***M** table records the relation "**object** colored **color** has magic **power** value" and has these rows:

sword	silver	medium
club	brown	low
ring	silver	high
sword	brass	low
ring	brass	none
plate	silver	medium

The action **–*M(ring)** would delete the 3rd and 5th row.
The action **–*M(ring brass)** would only delete the5th row.
The action **–*M()** would delete all of the rows.
The action **–*M(ring silver medium)** would not delete any rows.

A script to delete all of the **silver** objects would need a table runner:

```
~XY*M() { ( ?Y=silver –*M(X Y) ) }
```

Deleting rows from a table within a table runner for that same table, can have strange effects - although the above script will work.

8 - Simple Game Plus

As a more complicated illustration of flags, queries, and groups, reconsider the **dark_forest** scene in the **simple.txt** game developed in Sections 2 and 4. Load that game using the **DESIGN Load Game** option.

Suppose the game author decides that the hero needs to go north before being able to find a tree that can be climbed - even with the rope. This can be most easily done by rewriting **dark_forest** as:

```
you are in a dark and gloomy forest, surrounded by
huge oak trees
moss is growing on the north side of each tree.
[go north=Gdark_forest2]
[go (south/east/west) = S0]
[climb tree = they are too huge.]
```

Remove the scripting from the **rope*** routine leaving only its descriptive pattern:

```
-a +strong+nylon rope =
```

Create a new scene named **dark_forest2**:

```
you are in a dark forest of large oak trees.
[climb tree = ?@rope me G tree_top / it's too hard. ]
[go north = G swamp]      ;; rule-1
[go :e = G dark_forest]   ;; rule-2
```

Never mind the **swamp** for now. Notice that the climbing rule has been moved from the **rope*** script into the **dark-forest2** scene but only works if the hero has the rope.

There is a trick used above regarding the rules commented as rule-1 and rule-2. Each command the player gives can fire only one rule - the first one whose pattern matches that command. Since a **go north** command is matched by rule-1, that is the one (and only one) rule which will fire for that command - sending the hero to the swamp (which

is not yet written). However a **go east** command would not match the rule-1 and so it matches rule-2 and sends the hero back to the **dark_forest** scene.

As a game gets more complex, it becomes harder and harder to check it is working as desired. The author certainly doesn't want to play from the beginning all the way to the scene being worked on - every single time they change the design. But as explained elsewhere, the design changes do not take effect until the **BEGIN** button is pressed and so the author (playing the game) is sent back to "square one". There is a simple and powerful solution to this problem called the "cheat room". This is a scene designed for the author to use that is not available to other players. The author puts in it any useful actions they need and then uses the **GO TO** button of the **cheat** editing box to get to that scene and use those actions. Create a new routine named **cheat** with this script:

```
[rope = @rope me ]
[drop rope = @rope 0 ]
[forest = Gdark_forest]
[forest 2 = Gdark_forest2]
```

This script has no scene description since a player (other than the author) will never see it. However, the title **Cheat** will be visible once the author does the **GO TO** for that script.

The commands are abbreviated forms of what a player would have to say. The actual cheat rule commands don't really matter because only the author will see them. The only tricky feature is the "go to dark_forest2" cheat because if the author types:

forest2

the command is interpreted as two separate words **forest** and **2**.

As the game gets more complex, the cheat room can also get more complex.

Use the cheat room to check that the hero can no longer climb trees in the **dark_forest** scene and needs the rope to climb a tree in the **dark_forest2** scene.

The hero is going to need a bow for some more changes made below. Create a **bow*** routine:

```
-a bow +and +arrow =
```

and ignore the logistics problem of a bow without arrows for now. The **bow*** could be put in the family_room via a:

@ bow family_room

action in the **set_up** routine. However, the author should later feel free to devise a more difficult way to get the bow. Perhaps the **family_room** should have a locked cabinet - after all a bow is dangerous. Then the player will have to solve the "find a key" problem.

A general comment: when designing a game, the author can first create a "bare bones" version of the game and add lots of interesting features later on. However, at some point the author needs to "bite the bullet" and release the game for play - after which game tinkering should be discouraged. To avoid the temptation to "fix" the game, switch to a new release of the game. For example, this expanded **simple.txt** game should be saved as **simpleV2.txt**

Now change the **dark_forest2** scene as shown below. However, do ***not*** write the line numbers into the routine. They are shown in the script below to help explain how it works.

```
1:  you are in a dark forest of large oak trees.
2:  ( ?-k \C
3:       a huge grizzly bear suddenly attacks you.
4:       [ shoot grizzly +bear with bow=
5:            ?@bow me you shot him in the heart.
6:            he continues charging at you :p15
7:            but drops dead at your feet +k S0 /
8:            you don't have a bow.
9:            there isn't time to do anything else
10:           and the bear chases you right
11:           into a swamp G swamp ]
12:      [ :e = there isn't time to do that
13:           and the bear chases you right
14:           into a swamp G swamp ]
15:   /
16:      a huge grizzly bear lies dead
17:      on the ground.
18:   )
19:   [climb tree = ?@rope me G tree_top /
20:      it's too hard. ]
21:   [go north = G swamp]
22:   [go :e = G dark_forest]
```

Lines (2-18) above are the changes to the previous version of **dark_forest2**. They make up a plain group of actions (see **6.2 Plain Groups**). They also illustrate that the rules for a scene can be mixed in with the description of the scene - since those too are just actions.

The query for that group is **?-k** which tests a flag **k** indicating the grizzly has been killed. The **-** in the query means that the query is true when the flag is off. Since all flags are turned off by the BEGIN button, this query will be true the first time the **dark_forest2** scene is entered and the player will see that a grizzly is attacking.

This grizzly is a "room monster" and so there is no routine for it.

Not only will the player see the grizzly, but the two rules (4-11) and (12-14) will become available for matching player commands. The **:e** pattern in rule (12-14) means that the remaining game rules are ignored, since the **:e** pattern will match any command. In a basic game, the rules given in a scene are tried first. So any rules for tools present and any generic rules will be ignored. The player either shoots the bow or gets chased into the swamp. And yes - this is an example of breaking the "DONT BOSS THE HERO" rule.

Lines (5-11) are the consequence of the "shoot bear" rule and form a plain child group with two alternatives chosen depending on whether the hero has the bow or not. The first alternative of this group is lines (5-7). Line (7) is interesting because it manages the presence of the grizzly using the actions:

```
+k S0
```

where the **+k** turns on the k flag indicating the grizzly is killed and the **S0** causes the hero to "re-enter" the scene and this time the game skips lines (3-15) and executes lines(16-17) instead, showing the player that the grizzly is dead. The **S0** action is typically used when some part of a scene has changed, but the hero has not actually left and re-entered the scene. The player does not know that **S0** has been activated - they only see the content of the **S** game window change.

The last step is to create a new scene **swamp** which serves as a dead end trap for the hero. The only way out is to give up and restart the game:

```
you are knee deep in the smelly loathsome mud
of a dismal swamp.
you have lost all sense of direction.
[ go :e = it seems the same wherever you go. S0 ]
[ drop ~X@me = it falls into the mud and
    is swallowed up @X0 ]
```

The **go :e** rule overrides any other **go** rules in the game because the rules from the scene have priority and the **:e** matches any destination.

The **drop** rule uses a table query (see **7.1 Table Queries**) to override the generic drop rule and move the dropped tool (named in **X**) to nowhere (represented as **0**). This rule is added because a trick some game players use to detect that they are stuck in the same scene is to drop a tool, go somewhere and notice the tool is still visible.

Use an expanded version of the cheat room:

```
[rope = @rope me]
[drop rope = @rope 0]
[bow = @bow me]
[drop bow = @bow 0]
[forest = Gdark_forest]
[forest 2 = Gdark_forest2]
[clear k = -k^dark_forest2 ]
```

to check that all of this stuff is working, namely:

1. A grizzly attacks the hero in the dark_forest2
2. If not killed with a bow, the grizzly chases the hero into the swamp
3. If killed, the dead grizzly shows up in the scene
4. Once in the swamp, the hero is lost
5. Only if the hero kills the grizzly with the bow, can he climb a tree and win.

9 - Command Patterns

The game interacts with the player by requesting and accepting commands in a game window. Each command is broken down into a series of *words.* A *word* is either a number (all digits) or is spelled only using letters and apostrophes(**'**). All other text in a command is ignored. In addition the words are converted to lower case.

For example if the command was:

Go, now, John, quick-ly i'to the 55BREACH !!!

then its 9 words would be:

go now john quick ly i'to the 55 breach

In a basic game, the command is matched against the patterns in a list of rules. The first rule whose pattern matches the command is then *fired*. *Firing* a rule means executing the group of actions in the consequence of the rule. Recall that a rule has the format:

[*pattern* **=** *consequence* **]**

where the *consequence* is essentially a plain group of actions. However, no parentheses are needed for that group because the **[ ]** encloses its parts. In more detail a rule can be:

[*pattern* **=** *first-alternative* **/** *second-alternative* **/** *etc. etc* **]**

although the consequence need not have more than one alternative and in fact need not have any actions at all !

Groups of actions have been discussed in detail in section **6.2 Plain Groups.**

A script for a tool will also have the format;

pattern **=** *consequence*

where the pattern serves to describe the tool as well as matching references to the tool in a command. That pattern will also be called the *description* of the tool.

9.1 Pattern Elements

This section will examine the structure of a *pattern*.

A *pattern* is composed of a series of pattern *elements* - each of which matches zero or more words of the command in order. Suppose the element *alpha* matches **get big** and the element *beta* matches zero words and the element *gamma* matches **sharp pointy sword**, then the pattern

```
alpha beta gamma
```

matches the command: **get big sharp pointy sword.**

Most pattern elements are just the words wanted in a command. For example the pattern:

```
get back money
```

has three elements and matches the command: **get back money** It also matches the command: **Get !! BACK - # $ money ..**

Two other simple but useful pattern elements (first seen in **4.2 Creating Tools**) are:

```
+ word
```

which matches the word if it come next but otherwise matches zero command words and:

```
- word
```

which always matches zero command words (and thus is ignored). The latter kind of pattern is mostly used when describing tools with the **\d** action.

Another simple pattern element is:

```
:e
```

which matches the rest of the command regardless of its length. A silly example is the pattern:

```
-ignore -me :e
```

which will match absolutely any command. The **:e** element is not allowed in a tool description.

9.2 Pattern Group

A pattern element can also be a "group" of pattern alternatives enclosed in parentheses:

```
( pattern / pattern / etc. etc )
```

The enclosed child patterns are tried one at a time - if a match is found, the remaining patterns are ignored and matching continues with the elements after the group.

As an example, the pattern:

```
go (home/back to town) (again/right now)
```

matches these 4 commands:

go home again
go home right now
go back to town again
go back to town right now

but does *not* match any of these commands:

go back to right now
home again
go home
go home back to town again
go home again now

9.3 Word Capture

A setting letter can be a pattern element. It will always match any one word. That word will become the value of the setting. For example the pattern

```
marry B to G in church
```

will match the command:

marry Robert to ALICE in church

and will set **B** to be **robert** and **G** to be **alice**. Remember that the words of a command are converted to lower case before being matched. The same pattern will match the command:

marry phillip to dorothy in church

but this time **B** will be set to **phillip** and **G** will be set to **dorothy**.

The problem with this kind of pattern element is that there is no restriction on the word being captured - any single word is matched and captured.

A word capture is not allowed in a tool description.

9.3 Value Match

The pattern element

```
:v formula
```

finds the value of the formula and matches it to the next word of the command.

When the formula refers to a flag, the notation:

```
:v$ name
```

can be shortened to **$name**. The flag must belong to the current routine script.

A value match is generally not allowed in a tool description but the special case **$name** is allowed.

9.4 Tool Recognition

If ***tool*** is some formula whose value is the name of a tool (aka movable object), then the pattern element **!*tool*** attempts to match the description of that tool against words in the command. If there is no such tool, the matching fails. Such a pattern element is called a *tool recognition*.

Recall that the descriptive pattern for a tool both describes the tool and also matches references to the tool in a command.

For example if **shovel*** has the script:

```
-a +large (shovel/digger) =
```

then the pattern

```
dig hole with !shovel deeper
```

matches the commands:

dig hole with large shovel deeper
dig hole with large digger deeper
dig hole with shovel deeper
dig hole with digger deeper

but does not match the commands:

dig hole with large shovel sooner
dig hole with small shovel deeper
dig jole with large spoon deeper
dig hole with large shovel

NOTE: A tool recognition is **not** allowed inside a tool description. For example the **shed*** script:

```
-a shed with !shovel inside =
```

is **not** allowed. However the **@** table can be used to "place" the **shovel** inside the **shed** and the **shed** should have a rule for "looking" inside it.

```
-a +small shed =
  [look inside shed = you can see ~X@shed { \dX }]
```

There is also the question of whether the **shed** is really a movable object. Is there a forklift somewhere in the game which can be used to move the shed ? ? ?

9.5 Table Search

A *table search* pattern element has the form:

```
~ collector table ( selector )
```

where *collector* is a single capital letter naming a setting; where *table* is either **@** or one of the ***A** through ***Z** table names; and where the *selector* is a parenthesized list of values. The selector could be empty - although that would be strange.

The table search pattern element searches through the indicated table for rows with a suffix matching the selector. If the first column of such a row is a tool, then its description is matched against the next part of the command. If that match is successful, then the name of the tool is saved in the collector setting. If unsuccessful, the next matching row (if any) is tried. If none of the rows have a matching tool, then the table search fails. Notice that this is similar to **7.2 Table Runners**

The **~** symbol could be read as "find a matching".

Suppose, as an example, the table ***H** had these rows:

hammer	**fixes**	**nails**
sword	**cuts**	**ogres**
switch	**brings**	**light**
saw	**cuts**	**wood**

and that **sword*** had the script:

```
-a +sharp bronze sword = ...
```

and the **saw*** had the script:

```
-a +shiny steel saw = ...
```

then the pattern:

```
use ~X*H(cuts wood) on tree
```

would match either of the commands

use shiny steel saw on tree
use steel saw on tree

and in either case would set **X** to **saw**.

A table search is not allowed in a tool description.

9.6 Tool Display

When a tool is used in a display action like:

```
\d tool
```

that action displays the description of the tool in the game window. Recall that the script for a tool has the form:

```
pattern = consequence
```

where the pattern is also the description of the tool. Each element of the tool's descriptive pattern is displayed as follows:

1. A ***name*** is displayed as itself.
2. A ***number*** is displayed as itself.

3. A *group* is displayed by displaying its first (perhaps only) alternative child pattern, the other child patterns are ignored.
4. A ***+name*** is displayed as the ***name***.
5. A ***-name*** is displayed as the ***name*** - this is the only purpose of this kind of element.
6. An element **$flag** displays the value of the flag, its owner is the tool being described.
7. Other kinds of pattern element are illegal in a tool description.

As noted earlier, the pattern element types :

word capture, value match, tool recognition, table search, and *:e*

are not allowed in a tool description (except for **$flag**) .

Some variability in the description of a tool can be made using the **$flag** notation but normally should be avoided.

Instead of modifying the description of a tool it is usually better simply to replace the tool with a similar one in the @ table. The piece of script below uses a table query to replace a tool named **boat** with another one named **boat2**:

```
(?@(boat)X. -@(boat)  @boat2 X)
```

It is enclosed in a group to keep the query from affecting actions outside of the piece.

9.7 Rule Sets

Not all rules are available for firing at a given time. A "go north" rule would not be appropriate for the hero in a "living room". This understanding has been implicit in this guide so far. But there is a specific mechanism for organizing which rules are appropriate - the rule set. A rule set is a list of rules that can be searched for a rule matching the current player command. There are 26 possible rule sets named **A** through **Z**. However a basic game as created by the **New Game** option only uses three: **C** , **B**, and **A**. The basic game routine named **play** searches those sets in that order. It collects rules from the hero's current scene in rule set **C**. It collects rules from the tools the hero carries in **B**. It also collects rules from the tools in the current scene in **B**. Finally rule set **A** will have all the generic rules used in game play such as the "drop" and "get" rules.

Every time the hero enters a "new" scene, the basic game management recreates rule set **C**. That way rules from some other scene don't become inappropriately available to the player. It also recreates rule set **B** since a new scene might have different tools laying around. Even the rules from the tools carried by the hero are reconsidered since they might be conditional on the current scene. As an example consider this tool routine named **book***:

```
-a +leather +bound book =
    (?S windy_cliff
        [open book=the stiff wind blows away
            some of its pages] /
        [open book= it automatically opens
             to a page with the word \qalacazam]
    )
```

This is a perfectly legal script. When a rule is written in a script, it is considered to be an action. It is not the action of firing that rule, it is the action of collecting the rule into a rule set. Rather than label every written rule with the name of the rule set that collects it, the game keeps track of which rule set is the current rule set. When a rule is encountered in a script, it is added to the end of that current rule set.

The book script above will only collect one of the two rules shown, depending on whether the hero is on the **windy_cliff** or not. The two alternative scripts are rule collections.

The same effect can be had with this alternate scripting

```
-a +leather +bound book =
    [open book = ?S windy_cliff
        the stiff wind blows away some
        of its pages] /
        it automatically opens to a page with
        the word \qalacazam]
    ]
```

which only collects one rule with the two alternatives placed inside its consequence.

Whether to put the choice of alternatives inside the consequence of one rule or to have two alternative rules is completely up to the author.

In any case the basic game management re-creates the **B** rule set both for tools in the "new" scene and for tools carried by the author.

Rule set **A** is populated by the **begin_rules** routine which is visited once by the **begin_game** routine. If the author wishes to have additional generic rules, they should be added to the end of the **begin_rules** routine. However, if the entire game is supposed to change in some generic fashion, then just prefix the additional rules with the action

```
:A
```

which changes the current rule set to be set **A**.

Appendix A – TAG Webware

The program to design a TAG game is named **designTag.html** and is a web page located at:

https://fewjarsmith.com/Tag/designTag.html

The program just to play TAG games (but not to create them) is named **playTag.html** and is a web page at:

https://fewjarsmith.com/Tag/playTag.html

The (self-contained) web page to play "The Quest of the Ruby Red Dragon" is at:

https://fewjarsmith.com/Tag/rubyTag.html

The (self-contained) web page to play "The Pacific Atoll" is at:

https://fewjarsmith.com/Tag/atollTag.html

All these programs are self-contained web pages. They can also be downloaded to the user's computer and run from its User Download Folder (see **Appendix C – The UDF**). Each one is stand-alone and does not need access to the others. This document was written for version 0.0.4C.

Each of these programs is free to use, copy, and distribute – only requiring that no alterations be made to them.

Once the user has downloaded a copy of these programs, there is no longer any need to access the web site **fewjarsmith.com** since they do not rely on access to that web site.

These programs **do** require the use of a modern internet browser – such as a recent copy of Chrome, Firefox, Safari, or Opera for a desktop or laptop computer. The operative standard is ECMASCRIPT_2021.

Some tablet and cell-phone browsers may work although neither the built-in nor application virtual keyboard will be very satisfactory for large amounts of text input.

For playing a game, the applications have a virtual keyboard that will provide useful type-ahead selections - see **Error: Reference source not found**.

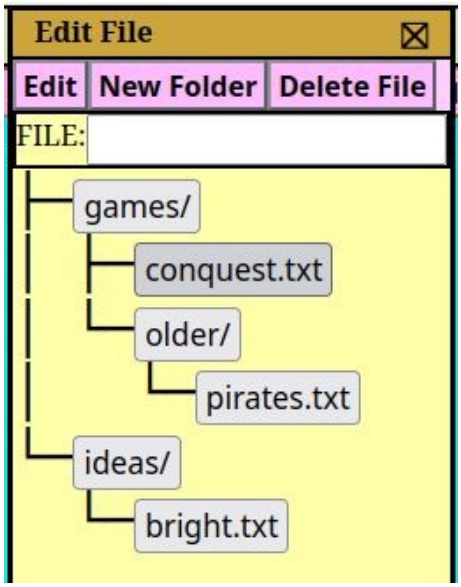
To use the application virtual keyboard, the user needs to start the application web page, click on FILE and select **Settings** - see **Appendix H - Application Preferences**. There are several keyboard layout options available and the font size of the keys can be altered to suit. Be sure to **Save** the desired settings.

This introduction to TAG design is located at: https://fewjarsmith.com/Tag/tagGuide.pdf

Appendix B – Virtual File System

Web page browsers are generally prevented from freely accessing the user's computer files. As a substitute, most browsers allocate some "local storage" for each web site that the user visits. This can persist through several visits to that web site. Usually about 4 megabytes or so of space is available. This local storage is owned by the browser and so can be read and written by web pages.

The TAG applications structure this local storage to appear like a traditional file system with files and folders - called the *virtual file system*. The folder separation character is a forward slash (/) as is standard for web page names. As an example, if the virtual file system tree structure was:



then the full name of the pirates game file would be **/games/older/pirates.txt**

The problem with using local storage is that it is **erased** when the history of its web site is cleared and that it is not shared between different browsers. In addition if duplicate instances of the **designTag.html** web page are being used (perhaps from different web sites), then each has its own virtual file system. In that case it would be necessary to do a **Backup** from one application instance and then a **Recover** from the other using the same backup file from the UDF. The same applies to the virtual file system for one browser vs the virtual file system for another browser.

The virtual file system tree will show a root folder named **/UDF/**. This provides the TAG application some access to the UDF (see **Appendix C – The UDF**) . This **/UDF/** virtual folder is not kept in the browser local storage but is mapped to the real UDF folder in the computer's file system. The notation **/UDF/** will refer to its appearance in the virtual file system. The word **UDF** will refer to the real computer folder. This is admittedly confusing.

The files shown in a virtual file tree under the **/UDF/** name are actually file name extensions. Attempting to load such a "file" will actually redirect the user to a browser window showing files in the real **UDF**. By default, this window will only show files having the selected extension. The user will have to select a file from this window to complete the load operation. Thus loading a real computer file requires two steps: (1) select a file name extension under **/UDF/** and once the browser window appears, (2) actually select a file from the real **UDF**. All this is necessary because web applications do not have direct access to the computer's file system (and they shouldn't).

Saving data into a name prefixed with **/UDF/** will actually save the data into the real **UDF** but possibly with a modified name. A web page is not allowed to overwrite computer files. An attempt to save a file to a name already existing in the real **UDF** will cause the file name to be slightly modified by insertion of some kind of version number - such as (1) or (2). Some browsers will also append the file name extension **.txt** to files they do not recognize (such as **.tag** files).

So although the virtual file system is convenient to use, care must be taken to back it up to the user's computer file system fairly ***OFTEN***. This is easily done in a TAG application by clicking **FILE** and then selecting **Backup**, which will bundle the entire virtual file system into a single file and save it into the UDF (User Download Folder) with a name like:

playTag20260429-2243.fsbk

which shows the year, month, day, hour, and minute the backup was made.

To recover the virtual file system from such a backup, click **FILE** and then select **Recover**. The browser will allow the user to select a file for recovery – usually found in the UDF.

Or the user can elect to keep individual game files directly in the **UDF**.

Appendix C – The UDF

The UDF (User Download Folder) is a generic name for the folder in the user's computer file system where the browser saves downloaded files. Each brand of browser may use a different folder for this purpose and most will allow the user to select what that folder is. Since the user is usually allowed to view other folders in place of the UDF, some browsers consider the UDF to be the last folder so viewed by the user. It is recommended that the user establish a single folder for this purpose and give it an obvious name.

However, to find out where the UDF is located on a particular computer, start **designTag.html**, click **FILE** and then select **Recover**. The browser will automatically open its UDF for selection. Note where it shows the UDF is located and then abort the recovery.

The designTag application specifically uses the UDF for the following menu selections:

Top Bar Menu	Selection	Action
DESIGN	Tag to UDF	Encode game to **.tag** file & save in the UDF. Some browsers will rename it to **.tag.txt**
DESIGN	HTML to UDF	Make game into HTML file & save in the UDF

Both the designTag and playTag applications use the UDF in the following menu selections:

Top Bar Menu	Selection	Action
FILE	Backup	Save entire virtual file system in the UDF
FILE	Recover	Restore entire virtual file system from the UDF

Appendix D – Load a Design

Suppose that some file for a game design has been downloaded into the UDF (see **Appendix C – The UDF**) and it is desired to study it and play it – suppose the file's name is **xyz.txt**

Click the **DESIGN** menu, select the **Load Game** option, and select the "file" entry **/UDF/.txt -** after which the browser will show a window giving all the files in the UDF with the file name extension **.txt** . Select the name **xyz.txt** which should now appear in that window. The status box should now say the game has been loaded. The game can be examined using the **ROUTINES** menu. The game can now be played by clicking **BEGIN**. If the game is saved, a version will go back in the UDF but with a modified file name. If the **Tag to UDF** option of **DESIGN** is used, the game's tag file will be saved in the UDF.

To save a copy of the game into the virtual file system, click DESIGN, select **Save Game As,** and select a file name that is not in the /UDF/ folder.

Appendix E – Action Reference

A script is mostly made up of actions. Many of the actions involve *formulas*. A formula is a notation which has a *value* that is a number or piece of text. Formulas are often *recursive*. Being recursive is a property of most languages. English is highly recursive – consider this sentence for instance:

> The man lived in a house under an oak tree that was planted by a wizard who wore a purple robe emblazoned with pink and red flowers as large as an open palm of an ogre who was only of medium height, etc. etc.

The script actions are described below by **generic** models with red colored letters referring to parts of the action that can vary from instance to instance of the model. Black characters just stand for themselves and do not vary from instance to instance. Italic words are descriptive of the part. The following colored letters are used:

- **A** stands for any capital letter
- **x** stands for any formula
- **r** stands for any *name* (like a routine name)
- **...** stands for any piece of text without question marks in it
- **f** stands for any flag notation – this can be:
  a *name* (usually one lower case letter) or
  a *name* followed by **^** and a formula

A *name* is a piece of text only containing lower case letters, digits, underscores (_) and apostrophes ('). A name must start with a lower case letter.

A *number* is a piece of text only containing decimal digits.

A *row* is a series of zero or more formulas. If necessary, they are separated by blanks and NOT by commas.

The possible actions in a script have been grouped into tables below depending on their first character. The options in a table are not listed in alphabetic order, but rather in order of usage. Many actions contain formulas.

E.1 Formulas

Below are the possible formulas and their values. A value is a number or a piece of text. Remember that **A**, **x**, etc are generic.

A	the value of the setting **A**
name	the *name* itself
number	the *number* itself
$f	the value of the flag **f** remember that a flag is a name **r** or a notation: **r ^ x** where the value of **x** is the name of the owner
"…"	the piece of text …
:ax	a word from the most recent player command where the value of **x** determines its position (1= first, 2=second, …) . If there is no such word, the value is the empty text ""
:n	a special marker text used in data tables - it is called the *null character*.
(x+x)	value of the first **x** plus the second **x**. The **x** is generic, so each **x** can be a different formula.
(x-x)	the value of the first **x** minus the second **x**.
(x*x)	the value of the first **x** times the second **x**.
(x/x)	the value of the first **x** divided by the second **x**, keeping the fractional part (if any)
(x//x)	the value of the first **x** divided by the second **x**, discarding the fractional part (if any)
-x	zero minus the value of the **x**.
?*query*	the response of the query converted to a number: true converts to 1 and false converts to 0. A query is a special kind of action – see farther below.
%x	a random number from 1 up to and including the value of **x**

:m	a number from 1 to 5 produced by matching player input commands to certain rules

A formula (**x** ~~~~ **x**) with more than two terms is allowed. For example:

```
(A+7*B/12)
```

which means:

> Take the value of **A**, add 7 to that, multiply that sum by the value of **B,** divide that product by 12.

Notice this is not the scheme used in Algebra class - arithmetic is performed strictly left to right. But if the author had wanted the value of **A** plus the result of 7 times the value of **B,** that could be requested by using inner parentheses:

```
(A+(7*B))
```

E.2 Basic Actions

Miscellaneous Actions:

Ax	change the setting **A** to have the value of **x**
!x	visit the routine whose name is the value of **x**. If there is no such routine, the play of the game stops with an error message.
+f	turn flag **f** on
–f	turn flag **f** off
$f x	change flag **f** to have the value of **x**
@x x	Use the @ table to record the first **x** as being at the second **x**.
***A**(*row*)*	Add the values of the row to the table named ***A**
.	display a period in the game window, followed by two blanks. Arrange to capitalize the next word.
,	display a comma in the game window

:A	collect rules into the rule set referred to by **A** . That rule set will become the current rule set.
(	begins a *plain group* – see **6.2 Plain Groups**
{	begins a *repeating group* – see **6.5 Repeating Group**
[	begins a rule, the rule is collected into the current rule set.
|x(	begins an indexed group - see **6.6 Indexed Group**
:m	run the consequence of the most recently matched rule.

Examples of **Ax** are:

A7	change setting A to be 7
M128	change setting M to be 128
MW	change setting M to match setting W
Q$f	change setting Q to be the value of flag f
F(24+1*5)	change setting F to be 125 – since 24+1 is 25 and that times 5 is 125
U(B*3)	change setting U to be three times the value of setting B

E.3 Arithmetic Actions

+	add the value of **x** to setting **A**
A-x	subtract the value of **x** from setting **A**
A*x	multiply setting **A** by the value of **x**
A/x	divide setting **A** by the value of **x**, keep fraction(if any)
A//x	divide setting **A** by the value of **x**, discard fraction(if any)
$f+x	add the value of **x** to the value of **f**
$f-x	subtract the value of **x** from the value of **f**
$f*x	multiply the value of **f** by the value of **x**
$f/x	divide the value of **f** by the value of **x**, keep fraction(if any)
$f//x	divide the value of **f** by the value of **x**, discard fraction(if any)

E.4 Output Actions

All action notations that start with a back slash(****) involve a game window. There can be up to 26 different game windows visible to the player. They are named **A** through **Z**. Only one is active at any given time – it is known as **the** game window.

name	display the *name* in the game window.
number	display the *number* in the game window.
\v`x`	display the **v**alue of `x` in the game window.
\n	start a **n**ew line in the game window.
\c	**c**apitalize the next text displayed in the game window.
\j	**j**oin the next displayed text to the previous display without a blank between them.
\a	**a**ccept a command from the player and break it into words at the blanks.
\h	erase the content of the game window and start displaying again at its upper left corner – known as **h**ome.
****`A`	make window `A` be the game window.
\t`x`	change the **t**itle of the game window to be the value of `x`. The title is displayed in modified form with underscores turned into blanks, digits omitted, and individual words capitalized.
\q`x`	display the value of `x` in **q**uotes in the game window.
\b`x`	output `x` many **b**lanks
\d`x`	display the **d**escription of the tool named `x` in the game window.
\e	**e**rase the game window and stop displaying it

There is one more game window action:

\A<x x x x x>

which creates (or recreates) a game window referred to by `A`. There are 5 values inside the **< >** pairing. In order they provide:

window horizontal position (from left edge)
window vertical position (from top edge)

window width
window height
window title (as in a **\tx** action)

The game window dimensions are given in tenths of a percent of the player area so that a width of 500 would (for example) mean half the width of the player area. NOTE that the dimensions are **not** pixel sizes.

E.5 Queries

Recall that a query answers some question about the game data - so its response is either "true" or "false". "Falsy" values are 0 and "". All other values are considered "truthy".

?A	is the value of setting **A** truthy
?A=x	does the value of setting **A** equal the value of **x**
?A<x	is the value of setting **A** less than the value of **x**
?A>x	is the value of setting **A** greater than the value of **x**
?A#x	is the value of setting **A** different from the value of **x**
?#x	is the value of **x** truthy
?–*query*	is the response of the *query* false
?@x x	does the @ table say the first **x** value is located at the second **x** value
?[*pattern* **=]**	does the current player command match the pattern
?f	is the flag **f** truthy
?$f=x	does the value of flag **f** equal the value of **x**
?$f<x	is the value of flag **f** less than the value of **x**
?$f>x	is the value of flag **f** greater than the value of **x**
?$f#x	is the value of flag **f** different from the value of **x**
?%x	will be true randomly a certain percent of the time - that percent is given by the value of **x**
?:rx	is the value of **x** the name of a **r**outine
?:tx	is the value of **x** the name of a **t**ool - ignoring the *
?:m ~~~~	rule **m**atching - not described in this guide
?*table* ~~~~	See **7.1 Table Queries**

E.6 Pattern Elements

The table below summarizes the various kinds of pattern element:

word	matches the *word*
number	matches the *number*
(*alternatives***)**	matches one of the *alternatives*
+ *word*	matches the *word* if it is next in the command - otherwise continues to the next element of the pattern.
– *word*	not used in matching
:e	matches the rest of the command illegal in a tool description
A	matches one command word which is captured into the setting **A**. illegal in a tool description
:vx	matches the **v**alue of the notation **x** illegal in a tool description
$f	matches the value of the flag **f** owned by the routine. the **^** *owner* notation cannot be used here
!x	the value of **x** is assumed to be a tool, this matches whatever the description of that tool matches. illegal in a tool description
~A*A(*row*)	explained in **9.5 Table Search** the first **A** is a setting and the second **A** is a table illegal in a tool description
~A@x	explained in **9.5 Table Search** illegal in a tool description

Appendix F – Problem Answers

Answers-1

The script below from 6.4 depends on the **b** flag.

```
there is a
( ?b very large / tiny little )
bug on the table.
```

b flag	display
0	there is a tiny little bug on the table.
1	there is a very large bug on the table.

The script below depends on flags **c**, **b**, and **f** and provides six different outputs:

```
inside the
( ?c coffin lies / ?b box sits / knapsack is )
a ( ?f frog / lizard )
```

c	b	f	display
1	any	1	inside the coffin lies a frog
1	any	0	inside the coffin lies a lizard
0	1	1	inside the box sits a frog
0	1	0	inside the box sits a lizard
0	0	1	inside the knapsack is a frog
0	0	0	inside the knapsack is a lizard

This script has a child group nested inside a parent group:

```
( ?f the fighting ( ?m man / woman ) hit /
   some wimp ran from
) the bear.
```

f	m	display
1	1	the fighting man hit the bear.
1	0	the fighting woman hit the bear.
0	any	some wimp ran from the bear.

Answers-2

This script from 6.5 counts from 10 to 90 by fives

```
A10
{ "A is" \vA \n A+5 ?A<95 }
all done
```

Below is a blow by blow demonstration of running the following script:

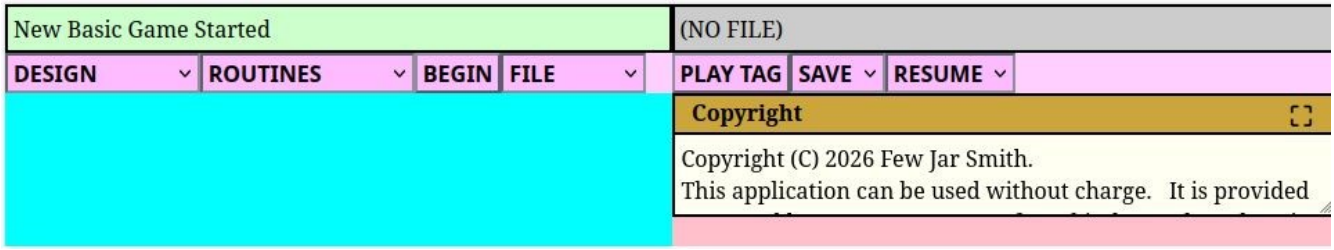
```
M30
{ \vM ?M<40 is small M+5 / is large M+20
  ?M<80 } done
```

step	action	display
M30	set M to 30	
{	begin repeating	
\vM	display M	**30**
?M<40	query is true	
is small	display text	**30 is small**
M+5	add 5 to M, M now 35	
/	skip next alternative	

}	go back	
\vM	display M	**30 is small 35**
?M<40	query is true	
is small	display text	**30 is small 35 is small**
M+5	add t to M, M now 40	
/	skip next alternative	
}	go back	
\vM	display M	**30 is small 35 is small 40**
?M<40	query is false, skip this alternative	
is large	**display text**	**30 is small 35 is small 40 is large**
M+20	add 20 to M, M now 60	
?M<80	query true,continue alternative	
}	go back	
\vM	display M	**30 is small 35 is small 40 is large 60**
?M<40	query is false, skip this alternative	
is large	display text	**30 is small 35 is small 40 is large 60 is large**
M+20	add 20 to M, M now 80	
?M<80	query false, exit repeating group, skipping the }	
done	display text	**30 is small 35 is small 40 is large 60 is large done**

Appendix G - User Interface

This describes the user interface for the **designTag.html** web application



The light green *status* box on the upper left holds status messages such as the completion of an operation or an error message.

The light gray *file* box on the upper right holds the virtual file name of the **.txt** game being designed or the **.tag** game being played. Any mention of the folder **/UDF/** is fictional and indicates the file was loaded from the UDF see **Appendix C – The UDF.**

The left half of the web page window is dominated by an aqua area used to hold routine editing boxes. It can also hold panes showing settings, tables, or rules. The menu bar above that area is used by the game author.

File operations will often display a *file tree* for the virtual file system. It will also show files being in the **/UDF/** folder. That is a fiction. Attempting to access such files will redirect the user to the User Download Folder in the computer's file system. Attempting to save a file whose name begins with **/UDF/** will redirect the content to a file in the User Download Folder. Because the browser does not allow a web page to overwrite files in the computer's file system, it will modify the file name being saved with something akin to a version number. This varies from browser to browser.

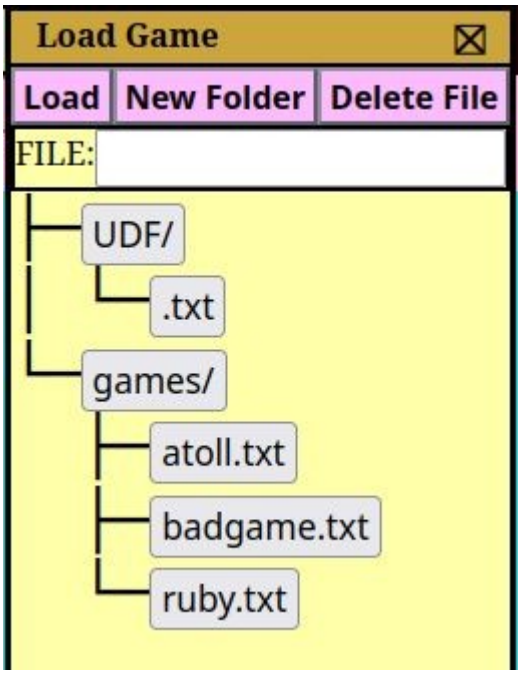
G.1 DESIGN Menu

Clicking **DESIGN** provides the options described below.

However, if the user has just finished running a game using the **Play Tag** menu, then only the **Load Game**, **New Game**, and **Empty Game** options will appear. To return to game designing, select **Empty Game** and the full **DESIGN** menu will then appear.

Load Game

A pane will appear holding the file tree for all the **.txt** files in the virtual file system.



Clicking one of those buttons will transfer its full path name to the FILE: box. Clicking the **Load** button will then load that design file and populate the **ROUTINES** menu with its script names. The status box will display how many routines were loaded and the file box will show the full path name of the file that was loaded. Trying to load the **/UDF/.tx**t entry will redirect to the UDF and allow the user to choose a game from there.

New Game

Any game being designed will be removed from the designer without notice. The user must have previously selected **Save Game** if they wished it saved. A completely new game will be started. The basic game management routines will be loaded. The status box will announce the game creation and the file box will indicate it has no assigned file name yet.

Empty Game

This is similar to **New Game** except that no basic management routines are loaded. The only routine created will be the **begin_game** routine and it will be blank.

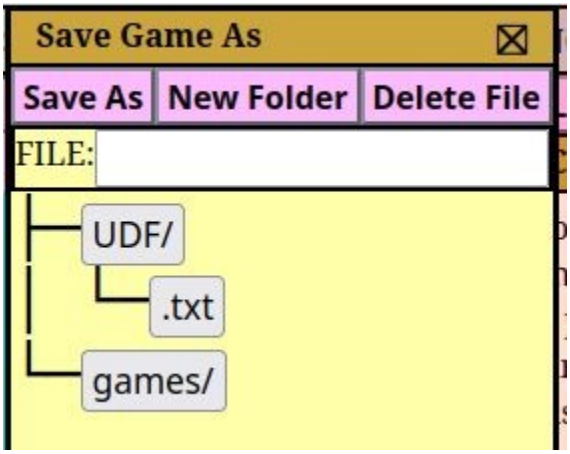
Save Game

If the file box indicates "(No File)", then this operation reverts to the **Save Game As** operation.

Otherwise the game being designed is saved as a .**txt** file in the virtual file system and a message to that effect appears in the status box.

Save Game As

A pane will appear holding the file tree for all the **.txt** files in the virtual file system.



Click the desired folder (usually **games/**) and complete the file name in the **FILE:** box. It will need a file name extension of **.txt** . The status box will confirm the game is saved and the file box will show the file path you chose. Selecting a name in the **/UDF/** folder will save the file in the computer's **UDF** folder - possibly modifying the file name with a version number.

Saving additional changes to the game can be made with the **Save Game** operation without having to reselect the file path.

Save Tag

The **.tag** form of the game will be saved to the virtual file system under almost the same file name but with **.txt** replaced by **.tag** . If no file name has been chosen yet, an error will be reported in the status box.

Tag to UDF

The **.tag** form of the game will be saved to the **UDF** under almost the same file name but with **.txt** replaced by **.tag** . If no file name has been chosen yet, an error will be reported in the status box.

HTML to UDF

The **.html** form of the game will be saved to the **UDF** under almost the same file NEED TO REVISE THIS XXX

name but with **.txt** replaced by **.html** . This form of the game is self contained and ready to play on most internet browsers. The browser will have to be pointed at the file. Many browser's have a **File** menu which can locate any **html** page on their computer. Many operating systems allow the user to set up an icon for an **html** file and arrange that a double-click or right-click opens that file. Many tablet computers use a single tap for the same purpose. Also, if a trusted web page has a link to a game **html** file, then activating that link will run the game.

Edit Title

A title can be attached to a game. It will be saved in the **.txt** file and also in a resulting **.tag** or **.html** file. When the game is run, that title will appear as the web page title - usually at the very top of the web page.

View Settings

A view pane will appear in aqua design area which will keep track of current values of the game settings. It does not allow them to be edited.

View Tables

A view pane will appear in aqua design area which will keep track of currently defined data tables. It does not allow them to be edited.

View Rule Sets

A view pane will appear in aqua design area which will keep track of current rule sets. It only shows one rule set at a time as selected by its capital letter at the top of the pane. It does not allow them to be edited.

Game Stats

A view pane will appear with game statistics:

1. **Word Count** is the total number of words used in the game design.
2. **Vocabulary** is the total number of <u>different</u> words used in the game design - this includes routine names. It cannot exceed about 65000. For comparison, the ruby game has a vocabulary of 352.

3. **Block Width** is a technical feature of group sizes. It cannot exceed about 65000. For comparison, the ruby game has a Block Width of 141.
4. **Code Count** gives the script size of the **.tag** file omitting its vocabulary. For comparison, the ruby game has a Code Count of 3711. The upper limit for Code Count is about 4 billion.

Prime Vocab

If there is a virtual keyboard, this option loads its type-ahead feature with the words from the routines currently in the game. It can be used repeatedly during the design of the game. The **Auto Prime** preference can cause this option to be exercised for each new game loaded by a **Load Game** operation.

If the type-ahead becomes overloaded with misspellings, the author should save the game, reload it, and then activate the **Prime Vocab** again.

Append

Some game files can just be collections of useful routines - essentially functioning as a library. This option allows the author to select such a library and add its routines to the game currently being developed. None of these routines will replace existing routines.

Update

Some game files can just be collections of useful routines - essentially functioning as a library. This option allows the author to select such a library and add its routines to the game currently being developed. Each of these routines would replace an existing routine with the same name.

G.2 FILE Menu

The virtual file system is not intended as a fully functional computer file system and so not all the usual operations are provided.

Clicking **FILE** provides the options described below.

Edit File

This provides a pane showing the *file tree* of the virtual file system. If seeing that tree was the only purpose, then dismiss the pane with the ⊠ button.

Clicking a button naming a file will only copy its path name into the **FILE:** box, it will not edit it or delete it or anything. Use the menu items of the file tree pane to perform those operations.

The **Edit** item will produce a text editor for the file. All virtual file system files are assumed to be text files.

The file tree will also show a folder named **/UDF/** . This is a fiction. Attempting to edit a file in that "folder" will redirect the user to the User Download Folder to choose a file from there. The editor can edit such files. Attempting to save one will instead save the file into the **UDF** - although perhaps with a version number inserted into its name. This allows the author to use the **UDF** instead of the virtual file system.

New File

This immediately provides a text editor for a new text file. Use its **Save As** menu to actually save the text. It will then provide a file tree pane. Place the new file name in the **FILE:** box and click **Save As**. If the provided file name is illegal, a revision will be suggested and no action will be taken yet. If the name is legal, the text in the editor will be saved in the virtual file system under that name.

The file tree will also show a folder named **/UDF/** . This is a fiction. Attempting to create a file in that "folder" will instead save the file into the **UDF**. This allows the author to use the **UDF** instead of the virtual file system.

There is a way to copy one file to another. Use **Edit File** to locate the file to be copied. Copy it to the clip board. Dismiss the editor. Select **New File** and copy the clipboard into the new resulting editor. Then click **Save As** and enter the new file name into the **FILE:** box and again click **Save As**.

Backup

The entire virtual file system is bundled into a single file and saved in the UDF under a name such as:

playTag20260429-2243.fsbk

which shows the year, month, day, hour, and minute the backup was made.

Making a backup after any large amount of design work is highly recommended. It is also recommended just before the author intends to make major revisions.

Recover

This opens a window showing the **UDF** files with extension **.fsbk** . Selecting one of these will erase the current virtual file system and replace it with the virtual file system saved in the selected **.fsbk** file. It is good policy to perform a **Backup** just before a **Recover** - in case changes have been made that are not captured by a previous **Backup**.

Copy App

The web page application currently running (probably an instance of **designTag.html**) will be downloaded into the **UDF**. Many browsers will run a web page from its URL but not save it into the user's file system. This provides a work-around.

Prefers

See **Appendix H - Application Preferences**

Appendix H - Application Preferences

The user can tailor some features of the TAG web page applications - such as font sizes, work area sizes, the use of a virtual keyboard, and the colors of some of the buttons and panes. A copy of the color choices is kept in a virtual file named **/colors** and a copy of the other preferences is kept in a virtual file named **/preferences**. An **.html** game page does not provide direct access to the virtual file system.

These are JSON text files and can be directly edited by the author. However it is STRONGLY recommended that they only be changed using the **Prefers** option of the **FILE** menu. It is possible to create invalid versions of these files that prevent the TAG applications from running. In that case it will be necessary to run the TAG application with the query **?reset** which will delete the **/preferences** and **/colors** files before running the application.

The first time a TAG application is used on a computer with a touch screen, the application will assume that a virtual keyboard is needed. It will change the preferences to be appropriate for a tablet or cellphone. The user can modify and save the preferences as explained below and thereafter the chosen preferences will be used when the application is started. Also, if the computer has a real keyboard, then both the real keyboard and the virtual keyboard can be used if wanted.

To modify application preferences, choose the **Prefers** option of the **FILE** menu. This provides a preferences window showing the application preferences which can be edited and a menu bar with two buttons: **SAVE** and **COLORS**.

The **SAVE** button applies the changes, commits them to the **/preferences** file, and then exits.

The preferences are explained below. Changes will usually be visible immediately. If the **Preferences** pane is dismissed (without saving those changes), the changes will still remain in effect until the application is restarted (i.e. reloaded).

Work On

There is a button with the name of a game design to be automatically loaded whenever the designTag application starts. Click the button to provide a pane showing the files of the virtual file system. Select the desired file and click the **CHOOSE** button.

To cancel a **Work On** selection, choose an empty path name.

Font Size

This affects the font used in menu bars and game windows and in routine editing boxes, etc.

Keyboard

See **Appendix I - Virtual Keyboard**

Key Size

The font size of the keys in the virtual keyboard. The font size of the type-ahead bar is instead controlled by the **Font Size** preference. Otherwise large key sizes (to help use of the keyboard) will limit the number of type-ahead words visible.

Portion

This provides a slider that controls how the web page is divided between the routine editor on the left and a the game playing area on the right.

Sides

On a computer with a small screen, trying to present both the design area and the play area in the same screen will likely make one or both areas too small. By selecting a **Sides** value of **Single**, the screen will switch to only showing one of the two areas. In that mode of operation a >> button will appear in the design area menu bar and << button will appear in the play area menu bar. These allow the author to switch the display back and forth between showing the two areas. The **BEGIN** button will automatically switch the display to the play area.

Selecting a **Sides** value of **Both** will return to the usual display showing both design area and play area.

Auto Prime

Setting this option "On" will cause every **DESIGN Load Game** operation to prime the virtual keyboard (if any) with the words in the routines of the newly loaded game. This will somewhat ease the process of entering lots of game design text from the virtual keyboard.

Appendix I - Virtual Keyboard

A virtual keyboard is available for computers, tablets, and cell phones which lack a real keyboard. This provides acceptable performance for playing a game, but is less adequate for writing a game since that is very text intensive. Tablets and cell phones also provide a builtin virtual keyboard but that tends to occupy too much screen space and provide features really not needed by an author or player. The TAG virtual keyboard is designed to occupy very little space - only three rows of 10 keys per row. Its font size can be adjusted to be some balance between ease of selection and leaving enough room for the application window panes. Using a tablet stylus may allow having the key size smaller.

The **Keyboard** preference (see **Appendix H - Application Preferences**) allows selection of one of the **Qwerty**, **Workman**, **Colemak**, or **Alpha** keyboard layouts. Choosing **---** as the layout disables the virtual keyboard.

NOTE: Selecting a virtual keyboard may move the web page focus away from the game player input - in which case, the user needs to tap/touch that input box so that it gets the focus again and will receive keyboard input. If touching the virtual keys seems to have no effect, that is the reason.

Above the virtual keyboard is a line showing a selection of type-ahead choices which can be tapped on to finish a word being entered. This makes it easier to play a game since the choices are tailored to the scenes recently viewed by the player. As the player types in the keyboard the type-ahead choices are updated to match the word so far typed. It also adjusts when the input is back spaced. If the type-ahead only suggests longer versions of the desired word, choose one of them and backspace over the unwanted characters.

When editing a game routine, don't backspace over mistakes - just keep on entering text. For some browsers, the misspellings will be selected (underlined) and touching them will provide easy spelling correction.

The standard keyboard function keys are located in unusual places:

> ⏎ is the return/enter key - located at the top in the middle - except the **qwerty** keyboard shows it at the bottom right.

There is no space bar - there is a single space key in the middle of the bottom row. And clicking a type-ahead will automatically space between words.

⌫ is the back space key - located at the bottom on the right.

⇧ is the "shift" key - located at the bottom on the right.

There are **four** shift levels - reached by repeatedly touching the shift key ⇧ . The un-shifted (first) level is the lower case alphabet, blank key, backspace, and enter key. In authoring a typical game **.txt** file this accounts for about 82% of the text.

The next shift level is the action punctuation and accounts for about 11 ½ % of the text.
The next shift level is the capital letters and accounts for only about 1 ½ % of the text.
The last shift level contains the digits, some arithmetic symbols, and the three punctuation marks not used for actions: ` and **&** and **;** .

When a character is selected from the first shift level (the action punctuation) the keyboard automatically returns to the lower case un-shifted level.

The first two shift levels should account for all but about 3% of the game design text input.