Inspiration Pad Pro 3.0

Table of Contents

1 Inspiration Pad Pro

1.1 Introduction

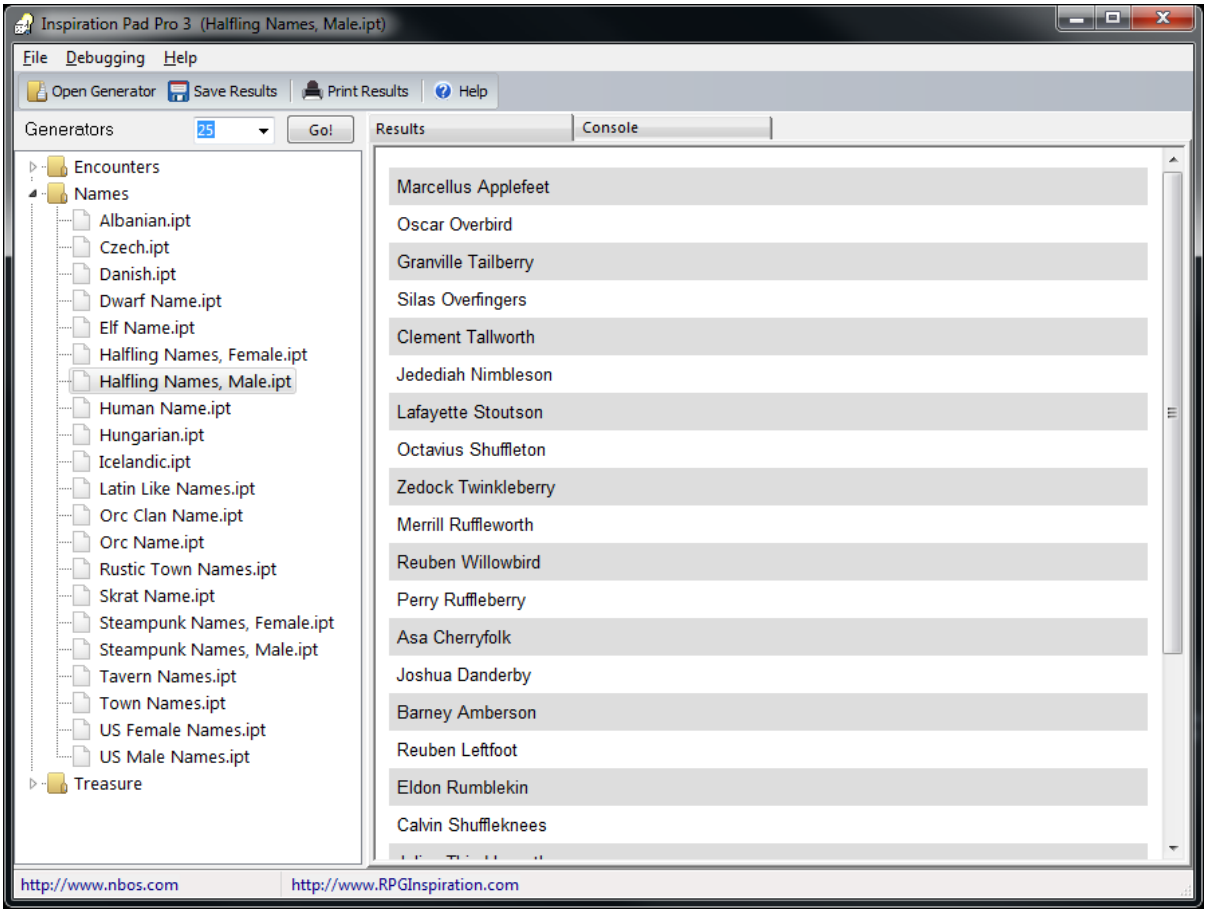
So what is Inspiration Pad Pro, and why will it change my life?

Inspiration Pad Pro is a program you can use to generate all sorts of information for your campaign. You can generate names, treasure, encounters, town information...just about anything.

Inspiration Pad is what's called a 'rules' based generator. It uses a set of user defined rules to control how it generates items. These rules are stored in text files that contain simple collections of items and commands that tell the program how to generate results.

The Inspiration Pad Pro client program

The client program is what you're probably using now. This is the stand-alone, Windows-based program that lets you run generators.



The Inspiration Pad Pro client program

The Inspiration Pad Pro engine

The Inspiration Pad Pro engine is the expression evaluation engine used by several different programs that let them your Inspiration Pad Pro generators. In addition to being used by the Inspiration Pad Pro client program, the Inspiration Pad Pro engine is used to run generators within other programs from NBOS - Fractal Mapper, AstroSynthesis, and the Inspiration Pad Pro command line programs.

1.2 What's New in 3.0?

If you are familiar with previous versions of Inspiration Pad Pro, here are the changes in version 3.0:

User Interface Changes

- New built-in table editor, so that tables can be edited right within the program.
- New debugging tab that outputs more information about how tables are generated and any errors that occur.

Inspiration Pad Language

- An all new Expression engine capable of mathematical, textual, and comparison expressions. Variables, dice rolls, and math expressions all now use the simpler {} syntax.
- There's a new type of table, called Dictionary Tables, that allow for non-numeric (text) keys for items.
- New Filters: +- (PlusMinus), Each, and EachChar

1.3 Rolling on a Generator

To run a generator, use the folder tree on the left side of the main window to navigate the categories. When you select a generator in the tree, it will be run. To run it another time, either click the name of the generator again, or click the Go button

To alter the number of repetitions run against the generator, alter the repetitions box just above the list of generators and then click the Go button.

You can also run a generator by clicking on it's file in Windows Explorer.

1.4 Exporting Generators

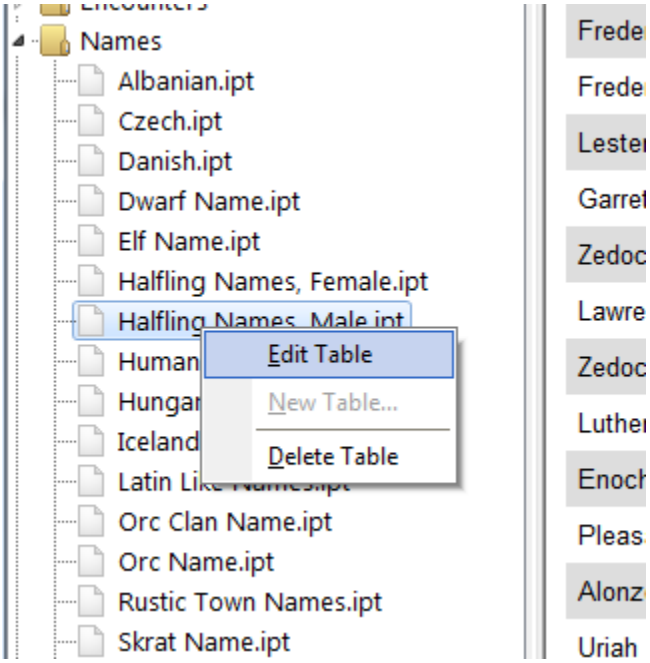
Since Inspiration Pad generator files can rely on the content of other generator files, it can be unwieldly to send a generator and all related files to a friend or to upload them all to a web site. To make it easier, Inspiration Pad lets you combine all the generators into a single self contained .ipt (generator file) file that's easier to send out.

To create a self contained .ipt file, first select the desired file from the table list. Then, select File - Export Generator from the menu. Select a file name for the new generator file (be sure not to overwrite the original!), and press Ok. You can then send that single file to friends, or upload it to your site, without the need to send any secondary files.

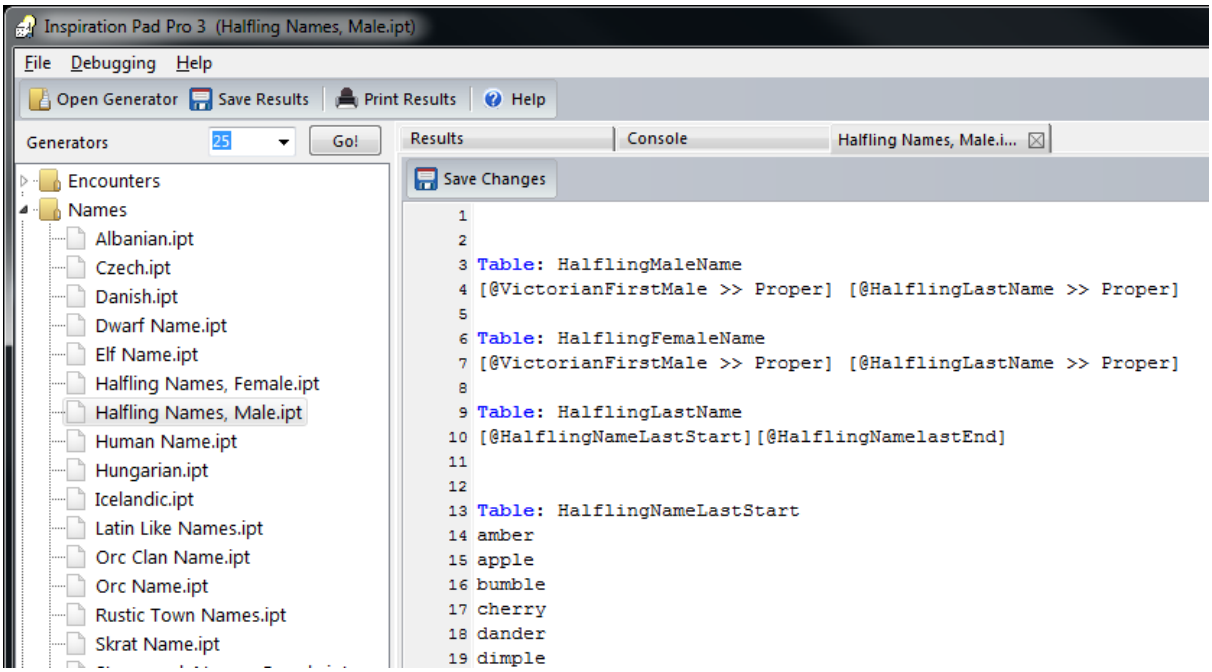
1.5 Editing Generators

Inspiration Pad Pro features a built in table editor, allowing you to make changes to your generators right from within the program.

To edit a generator, right click over its name, and select Edit Table from the pop-up menu.

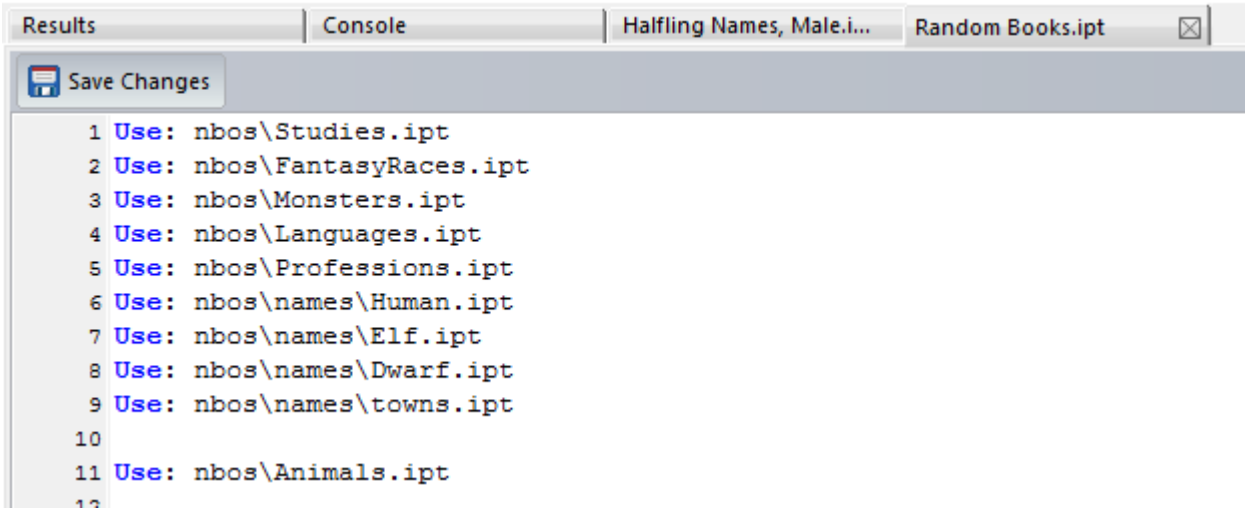


When you do this, your table will be opened in a new editing tab.

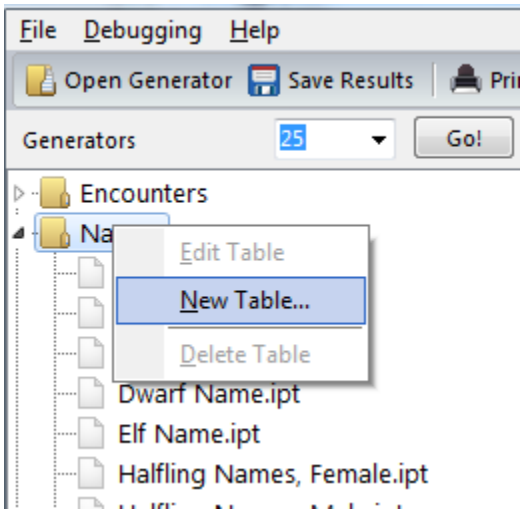


In the editor, you can make your changes. To see them in action, save your changes by clicking the Save Changes button, and then click the Go button.

To edit a table that is include via a Use 41 command, Ctrl-Click (hold down the Ctrl button while clicking) on the line with the Use 41 command, and the library file will be opened for editing.



To create a new generator in one of the categories, right click over the category name and select New Table.



2 Generator Files - a crash course

2.1 Generator Files

The Inspiration Pad uses special data files, called Generator files, when generating random information. Each type of random generator - for example, Elf Names, Dwarf Names, NPCs, etc - has a corresponding generator file. These files are text files that contain the various commands and tags that tell IP how to generate things.

To create a new generator, or to edit an existing one, you would create a new text file or edit one of the existing generator files with a text editor (notepad or other text editor - not a Word Processor). Generator files are stored in the 'Generators' subdirectory under the directory in which the Inspiration Pad is installed. For example, if you installed the Inspiration Pad into 'c:\program files\nbos\InspirationPad', then the generators would be located in the 'c:\program files\nbos\InspirationPad\generators' directory. Within that Generators directory, the individual generator files are then stored in directories based on their category.

The most important part of generator files are the *Tables* that they contain. These are similar to the tables you're familiar with from RPG's. The first table listed in the file is the main table for the generator. Any additional tables encountered are considered sub-tables. By linking multiple tables together,a wide variety of random content can be generated.

To start off with, we'll look at a simple generator file. Take a look at this file:

```
Table: Humanoid
Goblin
Kobold
Orc
Gnoll
```

As you can probably guess, this generator would be used to randomly generate a type of humanoid. When this is run, the program will randomly pick from the list of humanoids. In this case, it's a table that randomly picks between Goblin, Kolbold, Orc, and Gnoll. It's output might end up looking something like this, if run 6 times:

```
Goblin
```

```
Orc
Goblin
Kolbold
Gnoll
Orc
```

The first command in the file is the 'Table:' command. This tells the program that everything following is data for a table called 'Humanoid'. Each line after the 'Table:' command is an item in that table.

Now, say we wanted to make Orcs show up more frequently. We can weight individual items in a table to control their frequency. For example:

```
Table: Humanoid
2:Goblin
4:Kobold
10:Orc
Gnoll
```

In the above, we've added some relative weights of each item by adding a number and a colon (':'). In this example, Goblin is weighted at 2, and Kobold at 4. When the table is run, Kobolds will show up twice as frequently, then, as Goblin. Similarly, Orc, weighted at 10, will show up 5 times as often as Goblin (2), and 2.5 times as often as Kobold. You'll notice that Gnoll does not have a weight assigned. If an item doesn't have a weight assigned, it's weight is assumed to be 1. So in this case, Orc wil show up 10 times more frequently than Gnoll.

Another way to weight tables is to define dice roll and a range of values for each item in the table. This creates a 'look up' table, commonly found in game books. For example:

```
Table: Humanoid
Type: Lookup
Roll: 1d10
1-2:Goblin
3-5:Kobold
6-9:Orc
10:Gnoll
```

You'll see two new commands - Type and Roll. The 'Type: Lookup' command tells IP that the weighted values form a lookup table. The 'Roll' command tells IP to roll 1d10, and look the result up based on the various ranges assigned to each item. For example, if IP rolls a 6 on a d10, the result would be Orc.

Now, what happens if you specify a dice roll combination that might result in a number not assigned to a table item? For example, what if we had assigned '1d12' to the Roll command in the above table? It would then be possible for the dice roll to come up 11 or 12, which have no corresponding value in the table. Normally, when this happens, the result for that roll would be nothing. But you can control this by adding another command to the table, like such:

```
Table: Humanoid
Type: Lookup
Roll: 1d12
Default:Orc
1-2:Goblin
3-5:Kobold
6-9:Orc
10:Gnoll
```

A new 'Default' command has been added to the table. Now, if the dice roll ends up 11 or 12, the

default result would be what's listed after the colon in the Default command. In this case, Orc.

2.2 Dice Rolling

Inspiration Pad Pro lets you embed dice rolls right into your table items. This will let you randomly generate numbers for various purposes - hit points, town populations, and treasure, for example.

To embed a dice roll in your table, wrap the desired dice roll in curly brackets, {}. For example, to roll 3d6, you would put '{3d6}' into one of your table items.

Here's a look at a table with some dice rolling embedded:

```
Table: Humanoid
Type: Lookup
Roll: 1d10
1-2:Goblin, hp {1d6}
3-5:Kobold, hp {1d4}
6-9:Orc, hp {1d8*2}
10:Gnoll, hp {2d6+2}
```

If this table was run 5 times by IP, the results might look like:

```
Orc, hp 7
Orc, hp 6
Gnoll, hp 10
Goblin, hp 3
Kobold, hp 2
```

You can also combine dice expressions together, such as

```
{4d6+8d8}
```

Which in this case would roll 4d6 and add it to 8d8.

Similarly, you can create complex math expressions using dice rolls. For example:

```
{1d6 * 10 + (2d4 * 8d8)}
{1d100 / 1d6}
{max( 1d100, 1d100)}
{round( 1d20 / 6) + 20}
```

See Expressions 20 for more information about complex math expressions and math functions.

2.3 Rolling on Tables

2.3.1 Sub-table Rolls

Simple tables are useful, for sure. But to generate volumes of complex random results, you'll want to take advantage of the Inspiration Pad's ability to call sub-tables.

There are two tags - special commands that you embed into table items - that let you roll on sub-

tables. They are the '[@' and '[#' tags. (There's actually a few more advanced options as well, such as the table pick, but lets keep it simple for now)

In the previous example, a table was created to randomly generate a Humanoid and his/her/its hit points. Say now, we wanted to randomly assign armor and weapons to the creature. What we could do is create 2 sub-tables, one called 'armor' and one called 'weapons', and call them from our main table.

```
Table: Humanoid
Type: Lookup
Roll: 1d10
1-2:Goblin, hp {1d6}, [@Armor], [@Weapons]
3-5:Kobold, hp {1d4}, [@Armor], [@Weapons]
6-9:Orc, hp {1d8+2}, [@Armor], [@Weapons]
10:Gnoll, hp {2d6+2}, [@Armor], [@Weapons]

Table: Armor
None (AC 12)
Hide Armor (AC 13)
Leather Armor (AC 14)
Crude Chain Armor (AC 15)

Table: Weapons
None (claws/bite)
Club (1d6 damage)
Barbed Javelin (1d4+1 damage)
Spear (1d8 damage)
Short Sword (1d6 damage)
Long Sword (1d8 damage)
Great Sword (1d10 damage)
```

In the above example, you'll see two new tables, and some special tags added to the original Humanoid table. The [@Armor] and [@Weapons] tags in each line of the Humanoid table tells IP to roll once on the Armor table, and once on the Weapons table, and to insert the results in that spot. If this table is run 5 times, the results might look something like:

```
Orc, hp 7, Crude Chain Armor (AC 15), Great Sword (1d10 damage)
Orc, hp 5, None (AC 12), Barbed Javelin (1d4+1 damage)
Orc, hp 9, Leather Armor (AC 14), Club (1d6 damage)
Kobold, hp 4, Crude Chain Armor (AC 15), Great Sword (1d10 damage)
Gnoll, hp 11, Crude Chain Armor (AC 15), Spear (1d8 damage)
```

Now, we're getting pretty close to a useful Humanoid generator. But, what if we don't want Kobolds to be wielding Great Swords? What we could do is weight the weapon sub-table, and then call it using the [# command with a specific dice rolling combination:

```
Table: Humanoid
Type: Lookup
Roll: 1d10
1-2:Goblin, hp {1d6}, [@Armor], [#{1d6} Weapons]
3-5:Kobold, hp {1d4}, [@Armor], [#{1d6} Weapons]
6-9:Orc, hp {1d8+2}, [@Armor], [#{1d12} Weapons]
10:Gnoll, hp {2d6+2}, [@Armor], [#{1d6+6} Weapons]

Table: Armor
None (AC 12)
Hide Armor (AC 13)
Leather Armor (AC 14)
Crude Chain Armor (AC 15)
```

```
  Table: Weapons
  Type: Lookup
  1:None (claws/bite)
  2-3:Club (1d6 damage)
  4:Barbed Javelin (1d4+1 damage)
  5-6:Spear (1d8 damage)
  7-9:Short Sword (1d6 damage)
  10-11:Long Sword (1d8 damage)
  12:Great Sword (1d10 damage)
```

The '[#' command tells IP to pick a particular item from a table, or to roll on it using a specific dice combination. In the above example, each type of Humanoid has a dice roll combination assigned to its corresponding table item. You can see that Goblins and Kobolds can only end up with None, Club, Javelin, or Spear. Orcs can end up with any weapon. And Gnolls, because we're rolling with 1d6+6, can only end up with those weapons greater than or equal to 7 on the table - Short Sword, Long Sword, and Great Sword.

The results might looks something like:

```
  Goblin, hp 2, Leather Armor (AC 14), Club (1d6 damage)
  Kobold, hp 4, Leather Armor (AC 14), Club (1d6 damage)
  Orc, hp 5, None (AC 12), Spear (1d8 damage)
  Gnoll, hp 8, None (AC 12), Great Sword (1d10 damage)
  Orc, hp 4, Crude Chain Armor (AC 15), Barbed Javelin (1d4+1 damage)
  Goblin, hp 6, None (AC 12), Club (1d6 damage)
  Orc, hp 8, None (AC 12), Long Sword (1d8 damage)
```

There's no practical limit to how 'deep' you can nest sub-table calls. For example, a sub-table can call a sub-table, in which one of the items calls another sub-table, and on and on and on.

For example, you can modify the Weapons table from above to include another sub-table call - this time to a MagicWeapon table to determine if the weapon carried by our humanoid is magical.

```
Table: Weapons
  Type: Lookup
  1:None (claws/bite)
  2-3:Club (1d6 damage)
  4:Barbed Javelin (1d4+1 damage)
  5-6:[@MagicWeapon] Spear (1d8 damage)
  7-9:[@MagicWeapon] Short Sword (1d6 damage)
  10-11:[@MagicWeapon] Long Sword (1d8 damage)
  12:[@MagicWeapon] Great Sword (1d10 damage)

  Table: MagicWeapon
  20: \z
  2: +1 Magic
  1: +2 Magic
```

The \z in the first item in the MagicWeapon table means nothing - literally. Use this when you want to return no data from a table.

If the table is run now, the output might look something like:

```
  Goblin, hp 2, Crude Chain Armor (AC 15), Club (1d6 damage)
  Kobold, hp 3, Hide Armor (AC 13), Club (1d6 damage)
  Orc, hp 5, Hide Armor (AC 13), +1 Magic Short Sword (1d6 damage)
  Orc, hp 9, Leather Armor (AC 14), 13 Short Sword (1d6 damage)
```

```
   Orc, hp 7, Leather Armor (AC 14), 14 Short Sword (1d6 damage)
```

2.3.2 Table Parameters

Inspiration Pad Pro lets you pass parameters into table calls using the With command. When you pass a parameter into a table, the program automatically creates variables named {$1}, {$2}, etc, that contain the values of the parameters.

For example:

```
table: example
this is [@red_html with red]

table:red_html
<font color=red>{$1}</font>
```

In the example, the table *red_html* is called with the text 'red' as a parameter. The program automatically creates a variable called {$1} that contains the parameter, 'red'. The red_html table takes the parameter, and wraps an html tag around it to make it output in red.

More than one parameter can be passed into a table using the With command. Each parameter should be separated by a comma:

```
table: example
this is [@multicolor with red, green, blue]

table:multicolor
<font color=red>{$1}</font>, <font color=green>{$2}</font>, and <font color=blue>{$
```

The results of this would look (in Inspiration Pad Pro or via a web interface) like:

```
This is red, green, and blue
```

Notice than when more than one parameter is passed into a table using the with statement, a numeric variable is made for each one based on the order in which the parameter is passed. The first parameter is contained in a variable named {$1}, the second in {$2}, the third in {$3}, etc. There is no limit to the number of parameters that can be passed to a table.

Variables, dice expressions, and table rolls can all be passed in to tables as parameters. For example:

```
table: example
this is [@html_red with [@name]]

table:html_red
<font color=red>{$1}</font>

table: name
Tom
Joe
Mike
```

```
Grogar the Conquorer
Steve
```

This example passes a table call to the *name* table as a parameter to the *html_red* table.

2.3.3 Deck Picks

In many cases, you'll want to roll several times on a table without duplicating results. A good example might be generating a list of NPC skills. Inspiration Pad Pro features a special kind of sub-table call, the *deck pick*, to handle such situations.

The deck pick rolls on a table, returns a result, and then removes the result from the table so that it can't be selected again. Think of it as drawing cards from a deck of cards.

To use a deck pick, make your table call with these tags

```
[! <table-name> ]
```

or

```
[!<n> <table-name> ]
```

You can see this differs from the standard sub-table roll in that it uses an exclamation, !, instead of an ampersand. Like a normal table roll, you can either call the table with just its name, or include a number of repetitions, *<n>*, to run against the sub-table.

Example:

```
table: deckpickexample
Skills: [!5 skills >> implode]

table: skills
swordsmanship
riding
basketweaving
dancing
lock picking
swimming
metal working
brewing
climbing
moving silently
```

This example tells the program to do a deck pick against the skills table, picking 5 skills without duplicating them. The ' *>> implode*' command just tells the program to separate the results with commas. The output would look something like:

```
Skills: swordsmanship, swimming, brewing, basketweaving, moving silently

Skills: metal working, swimming, climbing, moving silently, brewing

Skills: moving silently, swimming, swordsmanship, riding, metal working

Skills: climbing, brewing, moving silently, lock picking, basketweaving
```

Notice that each list of skills has no duplicates.

Shuffling

When you use deck picks, the table items are removed from the table when they are selected. What if you want to put them back? Well, you do the same thing you do with a deck of cards... you shuffle them!

At the top of a table (after the table definition, but before any table items), you can tell the program to reset the list of items in specific tables that you plan to make deck pick calls to.

To shuffle a table, put the command:

```
Shuffle: <table-name>
```

at the top of your table. This tells the program that you intend to use a deck pick to call items on <table-name>, and that the program should re-set the items in that table to make them all available prior to making your deck picks.

Example:

```
table: deckpickexample2
[@4 MakeNPC >> implode <br><br>]

table: MakeNPC
shuffle: skills
A [|human|elf|dwarf] with the following skills: [!5 skills >> implode]

table: skills
swordsmanship
riding
basketweaving
dancing
lock picking
swimming
metal working
brewing
climbing
moving silently
```

This tells the program to generate 4 NPC's by rolling on the MakeNPC table. Each call of the MakeNPC table makes 5 deck pick rolls against the skills table. The output might look like:

```
A human with the following skills: basketweaving, climbing, lock picking, swimming,

A human with the following skills: climbing, metal working, lock picking, swimming,

A dwarf with the following skills: brewing, dancing, riding, swimming, swordsmanshi

A elf with the following skills: basketweaving, moving silently, swordsmanship, met
```

Notice that, while some of the NPC's share skills, none of them have the same skill listed twice in their list of skills. This is because a Shuffle command was placed at the top of the MakeNPC table, telling the program to reset the list of skills so that they are all available each time a new NPC is generated.

2.3.4 Inline Table Rolls

For quick random picks, Inspiration Pad Pro features *Inline* tables. These are simple sets of options you can place right inside a table item from which the program randomly selects. Inline tables are very useful when you just want to pick randomly and equally from a small set of options.

Inline table rolls are identified using the following tags:

```
[|<option1>|<option2>|<option3>|....|<optionN>]
```

Each option is separated by a pipe character, |, and you can have any number of options. Options in inline tables cannot be weighted, or have lookup keys assigned to them. When the program picks from the options, it just picks randomly from the list, with each option having an equal chance.

Example:

```
Table: InlineExample
At the bar you see a [|male|female] [|fighter|cleric|mage|thief]
```

Which might output:

```
At the bar you see a female mage
At the bar you see a male thief
At the bar you see a male fighter
At the bar you see a female fighter
At the bar you see a male cleric
At the bar you see a female thief
```

2.3.5 Using external tables

Once you've made a bunch of useful tables, you'll want to use them over and over again. For example, say you make a great name generator for your campaign. You may want to then use it in several different generators - encounter generators, book authors, adventure patrons, etc.

Inspiration Pad Pro lets you create a library of useful tables and set them aside for re-use by other generators.

You can tell the program to use the tables contained in another file by issuing a Use command at the top of your generator, as such:

```
Use: <table-name>
```

For example, say you have a great name generator you want to use stored in the file mynames.ipt. To access the tables contained in mynames.ipt in your script, you'd do something like:

```
Use: mynames.ipt

Table: MakeNPC
NPC Name is [@table_from_mynames.ipt]
```

Where these library files are located is very important. When you Use another generator file to access it's tables, the program needs to be able to find the file so that it can be opened. It looks for the file in the "Common" directory, located under the directory where Inspiration Pad Pro is installed. Typically, this is "C:\Program Files\nbos\InspirationPadPro". Thus, the "Common" directory is usually "C:\Program Files\nbos\InspirationPadPro\Common\". Of course, if you chose to install the program into a different directory, these directories would be different.

It's possible - in fact, it's advisable - to organize your library files into multiple directories. To do this, just create the directory structure under the Common directory, and then reference the path in your Use statement. For example:

```
Use: names\mynames.ipt
```

This would include the file named "C:\Program Files\nbos\InspirationPadPro\Common\names\mynames.ipt" into your generator.

You can include any number of files using the Use command. To include more files, just add a Use command for each file being used. In addition, any of the 'used' files can also Use any number of other files, which in turn can include other files, and so on. In all cases, the program searches for the files to use based on their path relative to the Common directory.

Note: the CGI version of the program, as well as generators run from Fractal Mapper and AstroSynthesis cannot access the Common library, so generators run from them should be exported as a single file[2].

2.3.6 Dictionary Tables

In addition to using numeric based tables (either Lookup or Weighted), Inspiration Pad Pro also supports something called a Dictionary table. This lets you use non-numeric values as keys. Dictionary tables can only be called using the Pick syntax, `[#<item> <table>]`.

For example,

```
table: dictionary_example
set: class=[|fighter|mage|thief|cleric|other]
[#{class} hitdice] for {class}

Table: hitdice
type: dictionary
default: hd6
fighter: hd10
mage: hd4
cleric: hd8
thief: hd6
```

In this example, an inline pick is used to set the variable 'class' to a random class. That value is then used as the key for a pick on the 'hitdice' table. The result is the corresponding hit dice for the class, or 'hd6' if the class is not found.

2.4 Variables

2.4.1 Variables and Constants

Often you'll generate a piece of data - a name, for example - and wish to use that in several different places in your generator. Inspiration Pad Pro lets you do this by allowing you to assign values to variables.

Variables need to be declared before your tables are run, so that the program knows what variables are assigned and what their values are. There are two types of variables available in IP: *Variables* and *Constants*. They are both referenced the same way in tables. They differ only in how they are handled when they are declared. More on that in a bit. To declare a variable, you'll use either the "Set:" (for normal variables) or "Define:" (for constants) command. To reference a variable or constant in your table, wrap it's name with curly braces, like this: {variablename}.

Here's a simple variable example:

```
Table: NPC
Set: name =Kebis
A man named {name}
A woman named {name}
```

(Note: previous versions of Inspiration Pad Pro also supported using this format, {$<var>} for variables. This is still supported for backwards compatibility.)

Here you can see the Set command being used. In this command, we're telling the program that we want to create a variable called 'name'. To this variable, we'll assign the value 'Kebis'. You'll notice that in the table items, special tags are used, {name}. These tags tell the program to insert the value of the variable called 'name' at that point. So, the result of running this table a few times would look something like:

```
A man named Kebis
A man named Kebis
A woman named Kebis
A man named Kebis
A woman named Kebis
A woman named Kebis
```

Now, variables are like any other table item. They can contain dice roll tags, embedded sub-tables, and inline picks. So, to extend the example:

```
use nbos/Human Names.ipt
Table: NPC
Set: Name = [@HumanName]
A man named {name}
A woman named {name}
```

(assume a file called Human Names.ipt exists, and has tables for creating human names)

You can see that a sub-table roll was assigned to a variable. When the table is run several times, a new 'name' is generated each time, with the results looking something like:

```
A man named Varanol
A woman named Sharor
A woman named Wanisrane
A man named Wanorent
A woman named Branesel
A man named Belodal
```

Once you've set a variable, you can use it any number times in a table to retrieve the same value. Here's a further expanded example, where each line uses the same variable twice:

```
use nbos/Human Names.ipt
Table: NPC
Set: Name = [@HumanName]
A man named {name}, who's father was also named {name}
A woman named {name}, who's mother was also named {name}
```

Which might result in:

```
A man named Tororrane, who's father was also named Tororrane
A woman named Kelenfel, who's mother was also named Kelenfel
A woman named Branrane, who's mother was also named Branrane
A woman named Lanidal, who's mother was also named Lanidal
A man named Belend, who's father was also named Belend
```

Notice, that for each roll on the table, you get a different name. But, while processing each individual roll, the value for {name} stays the same for that particular table item.

Also, once a variable is set, you can reference it from other tables:

```
use nbos/Human Names.ipt
Table: NPC
Set: Name = [@HumanName]
A man named {name}, [@Parent]
A woman named {name}, [@Parent]

Table: Parent
who's father was also named {$name}
who's mother was also named {$name}
```

And output might be:

```
A woman named Torrick, who's father was also named Torrick
A woman named Yanfel, who's father was also named Yanfel
A man named Keledac, who's mother was also named Keledac
A woman named Rasalt, who's father was also named Rasalt
A man named Sharart, who's mother was also named Sharart
```

Constants

Earlier we mentioned a type of variable called a *Constant*. A constant is similar to a normal variable in most ways, except one. When a constant is defined using the "Define:" command, if the value assigned to it has sub-table rolls or dice rolls, those sub-table or dice expressions are not evaluated at the time the variable is set, but rather at the time it is used.

Consider this example of a normal variable, and then look below for how a constant differs.

```
Table: Treasure
Set: roll={3d6}
You find {roll} copper pieces, {roll} silver pieces, and {roll} gold pieces!
```

Since the variable {$roll} is evaluated once - when it's 'Set' - the value for {$roll} each time it is used within the same table item is the same. The results might look like:

```
You find 7 copper pieces, 7 silver pieces, and 7 gold pieces!
You find 11 copper pieces, 11 silver pieces, and 11 gold pieces!
You find 11 copper pieces, 11 silver pieces, and 11 gold pieces!
You find 5 copper pieces, 5 silver pieces, and 5 gold pieces!
You find 12 copper pieces, 12 silver pieces, and 12 gold pieces!
```

But, if we change the Set command to a Define command to change {$roll} from a normal variable to a constant like this:

```
Table: Treasure
Define: roll={3d6}
You find {roll} copper pieces, {roll} silver pieces, and {roll} gold pieces!
```

The results might end up like this:

```
You find 15 copper pieces, 7 silver pieces, and 9 gold pieces!
You find 3 copper pieces, 11 silver pieces, and 5 gold pieces!
You find 14 copper pieces, 9 silver pieces, and 10 gold pieces!
You find 6 copper pieces, 8 silver pieces, and 12 gold pieces!
You find 10 copper pieces, 5 silver pieces, and 5 gold pieces!
```

As you can see, the dice expression {3d6} which is assigned to {roll} is evaluated each time {roll} is referenced, rather than just once.

Why use constants? If you have a large table, and a finite set of dice rolls or table calls that you use over and over again, constants are a handy way to be sure that all such references stay the same. It's easier, for example, to change the value assigned to a constant in one place, than it is to search through a large table and change each occurrence of that particular dice expression or sub-table roll.

2.4.2 Inline Variable Assignment

You already know that you can assign a variable using the Set command. There's another way to assign a variable as well, called *inline assignment*. That's a fancy term for assigning the variable right when you first use it.

To assign a variable this way, you'll tell the program to assign an expression to a variable right within a table roll or math expression using an '=' or '==' sign.

The basic usage is with a single equals sign. For example:

```
table: example
[@name=pcname] was the name, the name is [@vartest]

table: pcname
Joe
Mike
Grogar the Usurper
Tom

table: vartest
{name}
```

In this example, the expression [@name=pcname] assigns the results of a roll on the pcname to a

variable named {$name}. In addition, since it's within a line in a table item, the value of {$name} is output. This, for example, might be the result if run 5 times:

```
Joe was the name, the name is Joe
Tom was the name, the name is Tom
Tom was the name, the name is Tom
Joe was the name, the name is Joe
Grogar the Usurper was the name, the name is Grogar the Usurper
```

You can see that the variable {$name} is accessed a second time within the table 'vartest'. Just like any other variable, once it is assigned it as available for use within any sub-table roll.

You can tell the program to *only* perform the variable assignment and not output the variable by using double equal signs, '==', like:

```
table: example
[@name==pcname]the name is [@vartest]

table: pcname
Jill
Cindy
Marsha the Barbarian Queen
Jen

table: vartest
{name}
```

In this case, when the inline assignment is done, the result is not output. But when the variable is referenced again at the end of the table item, it is output. So the results might be something like this if run 4 times:

```
the name is Jill
the name is Cindy
the name is Jen
the name is Marsha the Barbarian Queen
```

In addition to assigning variables in table rolls, you can also assign them in a similar fashion within math expressions.

For example the following evaluates a math expression, assigns the value to a variable called {myvar}, and outputs the result.

```
{myvar=1+2+3}
```

While this evaluates the expression and assigns the value to the variable, but does not output the result:

```
{myvar==1+2+3}
```

2.4.3 Variable Scope (Kinda)

The variables in Inspiration Pad tables are all of a type of what's called a 'global' variable. This means that once a variable or constant is set in one table, it's value is available to be used within any other table.

Whenever a table (or sub-table) is rolled on, any variables that are set or defined in that table have their values set. If you roll on a particular sub-table 5 times, for example, each time that sub-table is called, it's variables are set.

There is a way, though, to get true 'application scope' global variables in Inspiration Pad. These are variables that are set outside of tables. Variables set outside of tables are set only once - when the generator file is loaded and run.

To assign a variable outside a table, put your Set or Define command before any Table commands:

```
#My town generator
Use: nbos/Towns.ipt
Set townname=[@MakeTownName]

Table: GenerateTown
{townname} is a [@TownDescription].
```

(assume that Towns.ipt was a file that had some town tables in it)

Here, the variable {townname} is set outside of any tables. This variable is set once, when the generator file is run, and that's it. It is available from within any table within the file, and from within any table within any files included by using a Use command.

2.4.4 Built-in variables

Inspiration Pad Pro automatically initializes several variables whenever a generator is run. These variables are for the most part designed to assist you in adapting your tables to versions of Inspiration Pad which may not be running in the traditional Inspiration Pad Pro program interface (for example, via the command line version of the program, or as a name generator in AstroSynthesis).

The default set of variables initialized are:

```
{app} - The name of the program ('Inspiration Pad Pro', 'Fractal Mapper', etc) that
        running the generator.
{version} - This is the version of Inspiration Pad evaluation engine.
{cli} - This is set to 'yes' if running the command line version, empty if not.
{os} - Operating system (Windows, Linux, FreeBSD)
{app} - Application running the table (Inspiration Pad, AstroSynthesis, Fractal Map
{builddate} - Date the executable was built (command line program only)
{hostlanguage} - The language as configured in the program's user interface.
                 Does not apply to the command line version of the program.
{fullpath} - Full path and file name of generator being run
{docpath} - The path portion of the generator file's full path.
{self} - File name of generator being run
{date} - The current date using localized format settings
{time} - The current time using localized format settings
{formatting} - Formatting output to be used by filters.  Default is 'html'
{rep} - The current repetition.  This is automatically incremented for each repetit
```

You can see what all the built-in variables are set to by running a generator such as:

```
MaxReps: 1
Table: Vars  \n&
```

```
App - {app} \n&
Version - {version} \n&
Operating System - {os} \n&
Build Date - {builddate} \n&
CLI - {cli} \n&
Date - {date} \n&
Time - {time} \n&
Full Path - {fullpath} \n&
File - {self} \n&
Formatting - {formatting}
```

(the & at the end of each line is a line continuation character, which lets you extend long lines across multiple lines. The \n is a line break)

2.5 Expressions

Let's say you're rolling up treasure, and you want to keep a running gold piece value of the total hoard. Not a problem if you use Inspiration Pad Pro's expressions!

Math expressions can be evaluated by placing them between curly braces as such:

```
{<expression>}
```

Where <*expression*> is the expression to be evaluated.

(note: previous versions of Inspiration Pad Pro used the format `{!<expression>}`, which is still supported by the current version of Inspiration Pad Pro to enable backwards compatibility).

Here's some examples:

```
{1+2+3}
{(4+6) * 2)
```

The valid mathematical operators supported include: +, -, /, *, and ^. The ^ operator is a power operator, such at a^b is the same as 'a raised to the b power'.

Variables, dice expressions, and table rolls can be embedded within mathematical expressions to enhance their utility. For example:

```
{a+1d6+[@sometable]}
```

In the above example, the variable {a} is added to 1d6 and the result of a roll on 'sometable'.

In addition to arithmetic operators, you can also use comparison operators, such as =, <, <=, >, and >=. Typically you'd use these in conjunction with the `if()` 21 function

Example:

```
{if (1d20 > 15, 5, 0)}
```

Which would roll a d20, and if it was greater than 15 returns 5, otherwise 0.

```
{if( 3d6 = 18, 4, 0)}
```

Which would return 4 if a 3d6 roll resulted in 18, otherwise 0.

2.5.1 Text Expressions

But wait, as they say in late night infomercials,... there's more!

Inspiration Pad Pro's expressions can also be used for handling text. Text can be added, compared, and passed to text functions.

Examples:

```
{'hello' + 'world'}
```

Outputs 'helloworld'

```
{if (b > a, 'bigger', 'smaller')}
```

Returns 'bigger', if variable b is greater, alphabetically, than a.

```
{substr( 'Inspiration Pad Pro', 13, 3)}
```

Returns 'Pad'

2.5.2 Built In Functions

In addition to arithmetic operators (+,-,/,*), a number of special functions can be called within expressions. The available functions are:

Math Functions

```
Math Functions

max(n1, n2,...,nN) - the maximum value of any number of passed values.
min(n1, n2,...,nN) - the minimum value of any number of passed values.
sqrt(n) - the square root of n
abs(n) - absolute value of n
round(n) - n rounded to the nearest whole number
floor(n) - n rounded down to the nearest whole number
ceil(n) - n rounded up to the nearest whole number
sign(n) - the sign of n, returns -1 if negative, 1 if positive
```

Example of using functions within math expressions:

```
;this divides the result of a d20 roll by a d6 roll, and returns the result rounded
{round( 1d20 / 1d6)}
```

Text Functions

```
Text Functions

length( s) - returns the length of the string s.
```

```
trim( s) - returns the string s, with any leading or trailing spaces removed.
substr( s, start, length) - returns a portion of the string,
     starting at character start, and extending length characters.
     If length is 0 (or omitted), the result extends to the length
     of the string.
```

Example of using text functions:

```
{a=substr('abcdefghijk', 2, 5}
```

which would set 'a' to 'bcdef'

Conditional Functions

```
Conditional Functions

if( a, b, c) - if 'a' evaluates to true, then returns value 'b',
   otherwise returns value 'c'.
```

Example of using if()

```
{ if( 1d20 >= 15, 'hit', 'miss')}
```

Rolls 1d20, and if it is greater than or equal to 15, returns 'hit', otherwise 'miss'.

2.5.3 Inline Variable Assignment with Expressions

Expressions can be assigned to variables [17], just like table results. For example, the following assigns the result of an expression to {myvar}:

```
{myvar=1+2+3}
```

"Quiet" assignment - just do the calculation, assign the variable, but don't output the results

```
{myvar==1+2+3}
```

2.6 Conditionals

Often when running a generator, you'll want to evaluate an expression only if a certain condition is met. Inspiration Pad Pro provides the [when] and [when not] commands to allow you to do this.

The format of a When command is:

```
[when] <conditional-expression> [do] <expression> [end]
```

Where <*conditional-expression*> is a conditional expression in the form of: 'a > b', 'a = b', or 'a < c'.

Alternatively, if no operator is given (<, >, =) then the conditional is evaluated as true if it contains non-blank space, and false if the conditional is empty. Expressions can contain variables, dice rolls, and other table picks.

For example, say you have a variable in your table called {$race}. If you want to call a table called DwarfProfessions only if {$race} contains 'dwarf', the conditional statement might look something like:

```
[when]{$race} = dwarf[do][@dwarfprofessions][end]
```

Or as another example, if you only want to roll on a table if the variable {$gold} is greater than some number, you might write:

```
[when]{$gold} > 50[do] [@bigtreasure][end]
```

Keep in mind, the expressions between the [when]...[do] do not need to be variables. You can also use table rolls, dice rolls, constants, or any other valid expressions:

```
[when] [@race] = dwarf[do]....[end]
[when] [@encounter1] = [@encounter2] [do]...[end]
[when] {3d6} > {!{$level} + {1d8}}[do]...[/end]
```

When doing a less than or greater than comparison, if both sides of the expression are numeric, a numeric comparison will be performed. Otherwise the comparison is made based on the alphabetical order of the expressions (ie, 'Z' is greater than 'A').

You can also use the [when] command to branch two different ways by adding an [else]. The format when doing this is

```
[when] <conditional-expression> [do] <expression> [else] <expression> [end]
```

For example:

```
[when]{$race} = dwarf[do][@dwarfprofessions][else][@otherprofessions][end]
```

When Not

The second form of the command is the [when not] command. This works exactly as the [when] command, only in the opposite direction. The [do] expression is evaluated if *<conditional-expression>* is false, and the [else] expression is evaluated if *<conditional-expression>* is true.

For example:

```
[when not]10 < 5 [do] orc [else] ogre [end]
```

would output:

```
orc
```

because the expression 10 < 5 is false.

The [when not] command is useful for checking if a variable contains a value.

For example:

```
[when not]{$myvar}[do] [myvar==orc] [end]
```

Which would set the value of {$myvar} to 'orc' if {$myvar} does not have a value.

2.7 Nesting (not bird related)

You know that you can make sub-table calls, do math, and roll dice...but did you know you can use them inside each other?

Placing a table call within a table call, or dice within math expressions, or tables within dice expressions is called 'nesting'. Variables, dice expressions, math expressions, and table picks can all be nested within each other to any depth.

Examples:

Nesting a table call within a dice expression:

```
table: nesting
{1d[@dietype]}

table: dietype
4
6
8
10
12
20
```

Nesting a math expression inside a dice roll (rolls a d6):

```
table: nesting
{1d{!3+3}}
```

Nesting a dice roll inside a table call, to specify the number of repetitions to call:

```
table: nesting
[@{1d6} weapons]
```

Randomly picking the name of a table to roll on by nesting a table call inside a table call:

```
table: nesting
[@[@gettable]]

table:gettable
a
b

table:a
this is table a

table:b
this is table b
```

Nesting a table call inside a variable expression to randomly pick the name of a variable from a table:

```
table: nesting
```

```
set:a=this is a
set:b=this is b
{$[@var]}

table: var
a
b
```

Note: Conditionals *cannot* be nested

2.8 Comments

Generators can get complex, and it helps to be able to annotate parts of a generator to remind yourself what it does. To this end, Inspiration Pad Pro lets you add comments to your generators. Lines blocked out with comments are not evaluated by the program when the generator is run. The comments are strictly for human readers of the generator.

Line Comments

Line Comments let you 'comment out' an entire line in your generator file. Three types of comment tags may be used to comment out a line, the hash, the semi-colon, and the double slash. To comment out a line, place one of these tags at the beginning of the line.

For example:

```
# This is a comment - the hash is a comment character when used at the start of a l

; So is this - a semi-colon is a comment character too

// This is a comment too!
```

Inline Comments

You can also use the double slash to place comments at the end of a line that is to be evaluated. When you do this, anything following the double slash is ignored by the program when evaluating a line.

For example:

```
Table: Example
The name is [@npcnames] //everything from the slashes to the end of the line is a c
```

2.9 Filters For Fun

Filters in Inspiration Pad Pro are a way to process the results of table calls in a number of different ways. For example, filters can sort results, convert text case, and apply font styles. You can see a list of supported filters under the Filter Reference 43.

Filters can be used within any table call item (those that are enclosed with square brackets, [..]). Filters cannot be used within expressions. But, expressions can be placed into a literal text 31 block, and then filtered.

Filters are called by listing them after a >> in a table call. For example, to convert the text of a table roll to upper case, you'd use the Upper filter as such:

```
Table: filterexample
[@monsters >> upper]

table: monsters
orc
goblin
bugbear
kobold
```

This might return:

```
BUGBEAR
```

As you can see, 'bugbear' was rolled on the *monsters* table, and then passed through the *upper* filter, resulting upper case text.

Some filters work against table rolls with multiple results. For example:

```
Table: filterexample
[!4 monsters >> sort >> implode]

table: monsters
orc
goblin
bugbear
kobold
lizardman
elf
ogre
hill giant
```

This picks 4 monsters from the monsters table, and then outputs them in sorted order.

```
bugbear, goblin, lizardman, orc
```

Filters can also be chained together, with the results of one filter then passed to another. In the above example, the list of monsters was sorted, and then passed to the implode 46 filter to sperate the results with a comma.

2.10 Special Characters and Escaping

Inspiration Pad Pro has a number of special characters and character combinations that you may find useful.

& (Line Continuation)

When used *at the end of the line* within a table item, the ampersand acts as a line continuation marker. This means that the next line is not treated as a new entry in a table, but rather as just a continuation of the previous line. You can use this to break up complex table entries to make them more readable.

Example:

```
table: example_table
this is line one &
this also gets included in line one!
```

\a (A or An evaluation)

Inserts either "a" or "an", based on the next occurring letter. If the next letter is a vowel, it returns 'an', otherwise, 'a'. For example, "\a antelope" would return "an antelope", and "\a dragon" would return "a dragon". (There's a few exceptions to this English grammatical rule, and the program will try to guess if it sees one it recognizes)

Example:

```
table: example_table
\a antelope
\a tiger
\a dragon
\a elf
```

Might output:

```
a dragon
a tiger
an antelope
a tiger
an elf
a dragon
```

\n (Line Break)

This inserts a line break at that point *in table results*. Don't confuse this with the line continuation character, which is used for formatting generator files. This controls line breaks in the actual output being generated.

This is frequently quite handy, since table items cannot wrap multiple lines, but you'll often want the output of a table item to span multiple lines. See the example tables for copious use of this tag.

Example:

```
table: example_table
this is line 1\nthis is line 2\nthis is line 3
```

Would output:

```
this is line 1
this is line 2
this is line 3
```

\t (Tab Character)

Inserts a tab character (ascii code 9) at the point. This is useful mostly when you're outputting plain text. If you're outputting HTML (which you are if you're using the normal, non-command line version of the program), this won't do much.

\z (Empty Character)

This means nothing. literally. It tells the program to stick no value at that point. This is useful mainly when you want one table item to return no value. Normally, if a line in a table is completely blank, the program will ignore the line thinking it's just normal spacing between tables. If you place \z as the table item, the program will know that you really intended to return a blank table item.

Example:

```
table: magic_weapon_bonus
5: \z
2: +1 Magic
+2 Magic
```

_ (Space Character)

Inserts a space at the point. For example, [@5 sometable >> Implode ____] would tell the program to seperate the individual sub-table rolls by four spaces. If you had just used spaces, the program would think this is just empty space and would ignore it.

Note that for output that gets generated as HTML, multiple spaces may not always be rendered as anything other than a single space by the program or a web browser. To force rendering of spaces in HTML, use html's own space expression, ' '.

\ (Escape Character)

The backslash is the generic escape character. Use this when you want to output text that otherwise would be evaluated by the program.

For example, if you want to actually output the text "{3d6}", rather than evaluating the dice expression, place an escape character before it, like:

```
table: escape_example
For treasure roll \{3d6}
```

Which would output:

```
For treasure roll {3d6}
```

Instead of evaluating the dice rolls like:

```
For treasure roll 12
For treasure roll 11
For treasure roll 11
For treasure roll 15
```

2.11 Prompting for input

Certain types of generators - treasure generators for example - may require different types of results based on different situations. You usually wouldn't give an orc the same amount of treasure as a giant. And 1st level characters don't usually encounter the same creatures 10th level characters encounter. You could in theory make a separate generator for each situation. Or you can use the Prompt command, which lets you ask the user for parameters to a generator, and then set up your generator to adjust to selected options. When a Prompt command is used, a box is displayed below the list of generators which allows the user to enter or select the options that are requested in the generator file.

When a generator is run, the values selected for the prompts are stored in variables named {$prompt1}, {$prompt2}, etc, sequentially in the order they were listed in the generator file. While you can include Prompt commands in files that are being included with a Use command, its not recommended as doing so may alter the order in which prompts are displayed depending upon the order the files are included.

Take a look at this example:

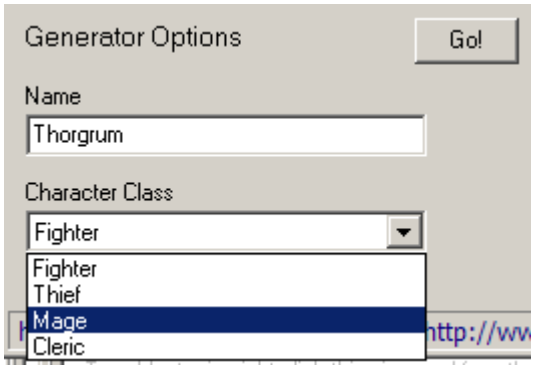
```
Prompt: Name {} Thorgrum
Prompt: Character Class {Fighter|Thief|Mage|Cleric} Fighter

table: prompt_example
You selected - Name of {$prompt1}, class of {$prompt2}
```

Prompts can take two forms - as a free-entry text that allows the user to enter in any text, or as a pick list of pre-defined options.

The first prompt in this example is a simple free-entry text prompt. This creates a simple text box labeled 'Name', which prompts the user to type in a name. The double braces, {}, indicate that there are no 'pick list' items for this prompt. That it is a free-entry prompt in which the user can type whatever they want. After the double braces is where the default prompt value can be specified. This is what the prompt value starts off as until the user changes it.

The second prompt is an example of a pick list. In this example, the label for the prompt is 'Character Class'. In between the braces is a list of options. These will be displayed to the user as a drop-down box from which they can select one of the options. Each option is separated with a pipe character, | (as they are with inline table picks). After the braces is the default value. The default value should match one of the available options.

When this generator is selected, the Generator Options box would look something like:



When generators featuring prompts are first selected, they will run using the default values.

Prompts are not displayed when the program is run via the command line interface or CGI.

3 Reference

3.1 # (comment)

The hash character, #, can be used as a comment character by placing it at the start of a line. Lines that start with # will not be evaluated by the Inspiration Pad Pro engine.

Example:

```
Table: example
#this is a comment and wont be evaluated
this is not a comment, and is treated as an item in the table.
```

3.2 // (comment)

Double slashes indicate a comment. Lines that start with double slashes are not evaluated by the Inspiration Pad Pro engine. In addition, double slashes can be used as inline comments, which cause all remaining text on a line to be ignored.

Example:

```
Table: example
//this is a comment and wont be evaluated
this is not a comment, and is treated as an item in the table.
this is not a comment //everything from the slashes to the end of line is a comment
```

3.3 ; (comment)

The semi-colon character, ;, can be used as a comment character by placing it at the start of a line. Lines that start with ; will not be evaluated by the Inspiration Pad Pro engine.

Example:

```
Table: example
;this is a comment and wont be evaluated
this is not a comment, and is treated as an item in the table.
```

3.4 >> (filters)

Use the >> operator to tell the program to apply one of the supported filters. The filter characters are placed within a table call.

Usage (each line demonstrates filter use with a different type of table call):

```
[@table >> filter]
[#table >> filter]
[!table >> filter]
[literal-text >> filter]
```

Where *table* is a valid table call, *literal-text* is any literal text 31, and *filter* is one of the supported filters 43.

Example:

```
[@MasterElfName >> Proper]
```

Might convert "telaryn telorbrnar" to "Telaryn Telorbrnar"

Example:

```
 [@TownName >> Upper]
```

Might convert "Brekville" to "BREKVILLE"

Multiple filters can be assigned to a table call as such:

```
[@3 Spells >> Proper >> Sort]
```

This would roll 3 times on the Spells table, apply the Proper filter to the results, and the Sort the results so that the spells are returned in alphabetical order.

See the Filter Reference 43 for a list of supported filters.

3.5 [text] (literal text)

Text between brackets indicates literal text that should be output. Normally this isn't necessary as any text within a table item is output as-is. But this option is provided to allow you to send literal text into a filter.

Example:

```
Table: example_table
[all upper case >> upper]
```

would return "ALL UPPER CASE"

3.6 [!table] (deck pick)

The [!table] tag performs a 'card deck' pick from the specified table. A deck pick selects an item from the table, and then removes that item from the table so that the item cannot be selected in subsequent table rolls. The status of removed items can be reset - making them available again - by use of the Shuffle 39 command.

Usage:

```
[!sometable]
[!n sometable]
```

Where *sometable* is the name of the sub-table to roll on, and *n*, if used, is the number of times to roll on that table. Tables always start in a shuffled state each time a generator repetition is run. So if you run a generator 25 times, tables are reset before each repetition.

Table calls may include parameters and filters and inline variable assignments in the form:

```
[!var=n sometable with parameters >> filter]
```

Example:

```
[!myvar=5 skills with {$class}, {$level} >> sort]
```

Which might roll 5 times on the *skills* table (without duplicates), passing variables *{$class}* and *{$level}* as parameters, filtering the result through the *sort* filter, and then assigning the final outcome to a variable named *myvar*.

See Deck Picks 11 for more information.

3.7 [#table] (sub-table pick)

The [#table] tag tells the program to pick the table item at a specified index in a table.

Usage:

```
[#n table]
[#table]
```

Where *table* is the name of the table to roll on, and *n*, if specified, is the item to select in that table. If *n* is not specified, the item that is picked in table is the item that's at the current index of the current table item being processed. So, if the 5th item in a table is selected, and that item has a table pick tag such as [# sometable], the fifth item in sometable would be picked.

Example:

```
Table: GetFifth
Return the fifth item in a subtable - [#5 NextTable]

Table: NextTable
This is the first item
This is the second item
This is the third item
This is the fourth item
This is the fifth item
```

Would result in:

```
Return the fifth item in a subtable - This is the fifth item
```

While:

```
Table: GetEach
Return the first item in a subtable - [#NextTable]
Return the second item in a subtable - [#NextTable]
Return the third item in a subtable - [#NextTable]

Table: NextTable
This is the first item
This is the second item
This is the third item
This is the fourth item
This is the fifth item
```

Might return the following if run 3 times:

```
Return the first item in a subtable - This is the first item
Return the third item in a subtable - This is the third item
Return the second item in a subtable - This is the second item
```

Table calls may include parameters and filters and inline variable assignments in the form:

```
[#var=n sometable with parameters >> filter]
```

Example:

```
[@myvar=12 treasure with {$class}, {$level} >> Proper]
```

Which would pick item number 12 in the *treasure* table, passing variables *{$class}* and *{$level}* as parameters, filtering the result through the Proper filter, and then assigning the final outcome to a variable named *myvar*.

See Sub-table rolls 7 for more information.

3.8 [@table] (sub-table roll)

The [@table] tag tells the program to roll on a specified table, and replace the [@table] tag with the results of that table roll.

Usage:

```
[@sometable]
[@n sometable]
```

Where *sometable* is the name of the sub-table to roll on, and *n*, if used, is the number of times to roll on that table.

Note that dice rolls, math expressions, variables, and other sub-table rolls may be nested 24 inside table calls. For example:

```
   [@{1d6} TreasureTable]
   [@{1d6} {$tablename}]
```

Table calls may include parameters and filters and inline variable assignments in the form:

```
[@var=n sometable with parameters >> filter]
```

Example:

```
[@myvar=5 skills with {$class}, {$level} >> sort]
```

Which might roll 5 times on the *skills* table, passing variables *{$class}* and *{$level}* as parameters, filtering the result through the *sort* filter, and then assigning the final outcome to a variable named *myvar*.

For more information, see Sub-table Rolls 7.

3.9 [|option|option|option] (inline table)

The [|option|...|..] command tells the program to select evenly from one of the state options.

Usage:

```
[|<option1>|<option2>|<option3>|....|<optionN>]
```

Where <*option1*> through <*optionN*> are various options available to pick. Any number of options can be placed in this tag. Each option needs to be seperated with a pipe symbol - the "|". (note that is not a capital letter i, or an exclamation point. It's a pipe, located above the backslash character on most keyboards).

Example:

```
Table: InlineExample
At the bar you see a [|male|female] [|fighter|cleric|mage|thief]
```

Might output:

```
At the bar you see a female mage
At the bar you see a male thief
At the bar you see a male fighter
At the bar you see a female fighter
At the bar you see a male cleric
At the bar you see a female thief
```

See Inline Table 13 rolls for more information.

3.10 \ (escape)

The backslash, \, is the generic escape character. Use this when you want to output text that otherwise would be evaluated by the program.

For example, if you want to actually output the text "{3d6}", rather than evaluating the dice expression, place an escape character before it, like:

```
table: escape_example
For treasure roll \{3d6}
```

Which would output:

```
For treasure roll {3d6}
```

Instead of evaluating the dice rolls like:

```
For treasure roll 12
For treasure roll 11
For treasure roll 11
For treasure roll 15
```

3.11 _ (space)

The backslash and underscore combination, _, inserts a space at that point. For example,

```
[@5 sometable >> Implode \_\_\_\_]
```

would tell the program to separate the individual sub-table rolls by four spaces. If you had just used spaces, the program would think this is just empty space and would ignore it.

Note that for output that gets generated as HTML, multiple spaces may not always be rendered as anything other than a single space by the program or a web browser. To force rendering of spaces in HTML, use html's own space expression, ' '.

3.12 \a (a/an evaluation)

The backslash and a combination, \a, tells the program to insert either "a" or "an", based on the next occurring letter. If the next letter is a vowel, it returns 'an', otherwise, 'a'. For example, "\a antelope" would return "an antelope", and "\a dragon" would return "a dragon". (There's a few exceptions to this English grammatical rule, and the program will try to guess if it sees one it recognizes)

Example:

```
table: example_table
\a antelope
\a tiger
\a dragon
\a elf
```

Might output:

```
a dragon
a tiger
an antelope
a tiger
an elf
a dragon
```

3.13 \n (line break)

The backslash and n combination, \n, inserts a line break at that point in the *in table results*. Don't confuse this with the line continuation character, which is used for formatting generator files. This controls line breaks in the actual output being generated.

This is frequently quite handy, since table items cannot wrap multiple lines, but you'll often want the output of a table item to span multiple lines. See the example tables for copious use of this tag.

Example:

```
table: example_table
this is line 1\nthis is line 2\nthis is line 3
```

Would output:

```
this is line 1
```

```
this is line 2
this is line 3
```

3.14 \t (tab)

The backslash and t combination, \t, inserts a tab character (ASCII code 9) at the point. This is useful mostly when you're outputting plain text. If you're outputting HTML (which you are if you're using the normal, non-command line version of the program), this won't do much.

3.15 \z (nothing)

The backslash and z combination, \z, means nothing. literally. It tells the program to stick no value at that point. This is useful mainly when you want one table item to return no value. Normally, if a line in a table is completely blank, the program will ignore the line thinking it's just normal spacing between tables. If you place \z as the table item, the program will know that you really intended to return a blank table item.

Example:

```
table: magic_weapon_bonus
5: \z
2: +1 Magic
+2 Magic
```

3.16 Default

The Default command defines a value for a table if a lookup roll or deck pick does not find a matching entry in the table.

Usage:

```
Default: fighter
```

3.17 Define

The Define command assigns a value to a constant variable. Unlike normal variables, constant expressions are evaluated *each time they are referenced* within a table.

Usage:

```
Define: var_name=value
```

Where *var_name* is the name of your constant, and *value* is the value you're assigning to the constant. Note that everything after the equals sign is assigned to the constant. So if you put a space between the equals sign and the value, that space will end up as part of the constant value!

For more information, see Variables and Constants 15.

3.18 EndTable

The EndTable command tells the program that data for the previous table is finished. This is not a command you'd normally need to use. But, it exists in case you want to assign application-wide variables at the bottom of a file (which otherwise would be evaluated as items in a table).

Usage:

```
EndTable:
```

Nothing is required after the colon in the command.

3.19 Footer

The Footer command assigns text to display at the end of the table application's output.

Usage:

```
Footer: SomeFooter
```

Where SomeFooter is text to display at the end of the table application's output.

Example:

```
Footer: Some information here is used under the Open Gaming...
```

The Footer command does not evaluate expressions, so a command like:

```
Footer: My name is [@somename]
```

will not be evaluated.

3.20 Formatting

The Formatting command tells the program which encoding method to use when the formatting filters (bold, italic, underline) are used. If used, this command should be placed at the top of your generator file before any tables.

Usage:

```
Formatting: text
```

or

```
Formatting: html
```

Normally the default formatting, html, is all that's desired, so most of the time you don't need to use this command at all. But, in cases where you want to generate text-only results (for example, via the command line version of the program), you can use this command.

When this command is set to 'html', the formatting filters (bold, italic, underline) wrap the result with corresponding HTML formatting tags (, etc.).

When this command is set to 'text', no encoding is performed. Instead, the text is transformed in some way in plain text (converted to all upper case for the bold filter, for example).

The value of this setting can be accessed by using the {$formatting} built-in variable 19.

See Filters 30 for more information.

3.21 Header

The Header command assigns text to display at the top of the table application's output.

Usage:

```
Header: SomeHeader
```

Where *SomeHeader* is text to display at the beginning of the table application's output.

Example:

```
Header: Random Spell Book
```

The Header command does not evaluate expressions, so a command like:

```
Header: My name is [@somename]
```

will not be evaluated.

3.22 MaxReps

Sets a limit on the number of repetitions that can be generated for a table application.

Usage:

```
MaxReps: n
```

Where *n* is the maximum number of repetitions that can be run against the table. If the number of repetitions selected from the repetitions drop down is greater than *n*, the table will only be executed n times.

Use MaxReps to prevent people from running very large tables in a way that cause the program to hang for a number of minutes. For example, if you are generating a complete army in such a way that hundreds of soldiers are generated in each repetition, set MaxReps to 1 so that the program doesn't take 20 minutes to generate 50 different armies!

3.23 Prompt

The Prompt command asks the user for input. Prompts are displayed in the Generator Option box.

Usage:

```
prompt: <prompt-label> { <prompt-options>} <prompt-default-value>
```

Where <*prompt-label*> is the label to display for the prompt, <*prompt-options*> is a pipe, |, delimited list of pick list options (if any), and <*prompt-default-value*> is the initial value for the prompt.

If <*prompt-options*> is left empty, the prompt created is a free-text entry prompt allowing the user to type in whatever they want.

If <*prompt-options*> contains one or more items separated by a pipe, the prompt created is a pick list allowing the user to select from one of the pre-defined options.

Example:

```
Prompt: Name {} Thorgrum
Prompt: Character Class {Fighter|Thief|Mage|Cleric} Fighter

table: prompt_example
You selected - Name of {$prompt1}, class of {$prompt2}
```

Prompt commands are ignored by the command line and CGI versions of the program.

3.24 Set

The Set command assigns a value to a variable.

Usage:

```
Set: var_name=value
```

Where *var_name* is the name of your variable, and *value* is the value you're assigning to the variable. Note that everything after the equals sign is assigned to the variable. So if you put a space between the equals sign and the value, that space will end up as part of the variable value!

For more information, see Variables and Constants 15.

3.25 Shuffle

The Shuffle command tells the program to shuffle the specified table before a table is rolled on.

Usage

```
Shuffle: table
```

Where *table* is the table to be shuffled.

This command must be placed within a table definition (i.e., after a Table command). When the table/subtable is rolled on, if a shuffle command exists for it, the specified tables will be shuffled before the table is rolled on. This allows you to shuffle any subtables that might be needed to support items in a table. Any number of tables can be shuffled within the calling table - use one Shuffle command per table.

Tables always start in a shuffled state each time a table application (a table file) is run.

See Deck Picks 11 for more information.

3.26 Table

The Table command tells the program that the lines below the command are part of a table.

Usage:

```
Table: TableName
```

Where *TableName* is the name of your table.

3.27 Title

The Title command sets the HTML page title for the generated HTML output when using the CGI/ command line version of the program.

Usage:

```
Title: SomeTitle
```

Where *SomeTitle* is the title of the HTML page. This will assign a <title></title> value to the html output when using the CGI interface. This command has no effect when running the Inspiration Pad Pro application.

3.28 Type

The Type command tells the program what type of table is being loaded. Place the Type command after a Table command.

Usage:

```
Type: Lookup
   or
Type: Weighted
   or
Type: Dictionary
```

Tables of type *Lookup* are tables where range values are assigned for each item in the table (1-3: Orc, for example).

Tables of type *Weighted* are tables where either no weight is assigned, or where weight values are assigned to table items (3: Orc, for example).

Tables of type *Dictionary* use non-numeric values, such as names, as keys for the items in the table. See Dictionary Tables 14 for more information.

The default value for a table's Type is 'Weighted'. So, if you have a weighted table, this command can be skipped.

3.29 Use

The Use command tells the program to load another generator file so that the current generator can call tables in the other file. By allowing this functionality, you can store commonly used tables (such as monster lists, random names, etc) and call them from other generator files.

Usage:

```
Use: filename
```

Example:

```
Use: nbos/names/DwarfNames.ipt
```

Where *filename* is the name of the generator file to include. All files included with a Use command should be stored within the program's 'Common\' directory, or stored in a sub directory underneath it. You may specify one or more Use commands in your generators - one for each file being used. Use commands may be located anywhere within the generator file, though its recommended that you place them at the top of the generator file so that you can see what generators are using what files at a glance.

If variables are defined or set in the generator that is being Use'd, they will be initialized when you run the table application.

See Using external tables 13 for more information.

3.30 [When] / [When Not] (conditionals)

The When and When Not commands are used to branch expression evaluation when a certain condition exists.

Usage:

```
[when]condition[do]expression[end]
[when]condition[do]expression[else]expression2[end]
[when not]condition[do]expression[end]
[when not]condition[do]expression[else]expression2[end]
```

Where *condition* is a conditional expression in the form of: 'a > b', 'a = b', 'a < b', or 'a <> b' . Alternatively, if no operator is given (<, >, =) then the conditional is evaluated as true if it contains non-blank space, and false if the conditional is empty.

When used in '[when]' form, if *condition* is evaluated to true, *expression* is evaluated. Otherwise no evaluation is performed. If the optional [else] tag is used, *expression* is evaluated if *condition* is true, otherwise *expression2* is evaluated.

When used in the negative '[when not]' form, it's the opposite. The do expression is evaluated if *condition* is false, and the else condition (if given) is evaluated when the condition is true.

Both conditions and expressions can contain variables, dice rolls, and other table picks. Conditionals cannot be nested.

For more information see Conditionals 22.

3.31 With

The With command may be used inside a sub-table tag to specify parameters that are passed into the sub-table roll.

Usage:

```
[@sometable with param1, param2,...,paramN]
```

Where *sometable* is the name of a table, and the comma separated list *param1* through *paramN* are the parameters to pass. Within the table call the parameters are being passed in to, the parameters are converted to variables based on their position in the parameter list. The first parameter is stored in {$1}, the second in {$2}, and so on. Any number of parameters are supported.

For more information, see Table Parameters 10.

3.32 {<expression>} (expressions)

The {<expression>} tag represents a mathematical or textual expression.

For more information, see Expressions 20.

3.33 {!math} (math expressions)

This tag, while still supported for backwards compatibility, is no longer required. See Expressions 42 for more information.

3.34 {$variable} (variables)

This tag, while still supported for backwards compatibility, is no longer required. See Variables and Constants 15 for more information.

The {$variable} tag inserts the value of a variable or constant at that point in a table item.

Usage:

```
{$variable}
```

Where *variable* is the name of your variable or constant.

3.35 {nDn+n} (dice rolls)

This tag, while still supported for backwards compatibility, is no longer required. See Dice Rolling 7 for more information.

3.36 Filter Reference

3.36.1 At

The At filter outputs the position of a sub-string within a table result. The parameter to the At filter is the text to find. If the sub-string is found, the filter outputs the index at which the string starts. If the sub-string is not contained in the text, the filter outputs 0.

Example:

```
[ abcdefghijklmnopqrstuvwxyz >> At t]
```

Would output:

```
20
```

Since the 't' is the 20th character in the text. The sub-string can be text of any length:

```
[ the wizards have a lot of gold to spend >> At gold]
```

Would output:

```
26
```

Since the word 'gold' starts at the 26th character (leading and trailing spaces are ignored).

```
[ the wizards have a lot of gold to spend >> At silver]
```

Would output:

```
0
```

Since the word 'silver' does not appear in the text.

3.36.2 Bold

The Bold filter converts the results to bold text in a platform independent manner. In most cases this simply means automatic wrapping of the text with necessary HTML tags. But in cases where HTML rendering is not available (for example, when used on a command line), bold text is returned capitalized.

Example:

```
[ this is bold >> Bold]
```

Would output

```
This is bold
```

See the Formatting 37 command for more information.

3.36.3 Each

The Each filter passes each result of a table roll into a specified table as a parameter. This allows you to process each item in a table call that returns more than one item.

For example, the following would return 4 different items from a sub-table:

```
table: getmonster
[!4 monsters]

table: monsters
orc
goblin
bugbear
kobold
lizardman
elf
ogre
hill giant
```

The result might be:

```
goblin bugbear kobold orc
```

To process each returned item, you can use the Each filter which will pass the results to another table as a parameter 10. To do this, use the Each filter, along with the name of the table that will process each result. Each result from the first table will be passed to that second table as the first parameter, accessible by referencing the {$1} variable. The result of that second table call will then replace the original text of the item that was sent into it.

```
table: eachtest
[!4 monsters >> each describe >> implode]

table: monsters
orc
goblin
bugbear
kobold
lizardman
elf
ogre
hill giant

table: describe
a big {$1}
a small {$1}
a hungry {$1}
```

The results might then look like:

```
an angry ogre, a hungry kobold, a hungry orc, a big goblin
```

(Note the use of the Implode 46 filter to separate the results with a comma)

You can see that each result from the *monsters* table is sent to the *describe* table as a parameter. This is similar to calling the *describe* table like:

```
[@describe with goblin]
```

The difference is the Each filter will do that for you automatically for each result in a table roll.

3.36.4 EachChar

The EachChar filter allows you to process each character in a piece of text. It does this by passing each character in the text into a specified table as the first parameter[10], and then replacing that character with the results of that second table call.

To use the EachChar filter, specify the table that each character is passed into.

Example:

```
table: eachcharexample
[ Necromancer >> eachchar mangle]

table: mangle
[{$1} >> lowercase]
[{$1} >> uppercase]
```

This processes each character in the word "Necromancer", replacing it with either a lower or upper case version of the letter. In this case, it looks at each letter, and calls the *mangle* table using the letter as the first parameter. In the *mangle* table, the letter is accessible using the {$1} variable.

The results might look like:

```
NEcROmAnceR
necROmAnceR
nECroMaNCer
NecrOMaNCER
necRoMANcEr
NeCrOmancEr
NecroMANceR
NecromANCER
NECROmANCeR
```

Another example:

```
table: example
[abc >> eachchar getsubst]

table: getsubst
[#{$1} convert], \z

table: convert
type: dictionary
```

```
a:Ankheg
b:Blinkdog
c:Cockatrice
d:Dryad
e:Eettin
```

Would return

```
Ankheg, Blinkdog, Cockatrice,
```

3.36.5 Implode

The Implode filter joins the results of a multiple repetition table roll with a 'glue' string. The default glue is a comma, but something else can be used if desired.

For Example (without filter):

```
[@3 humanoids]
```

Might output:

```
orc goblin kobold'
```

While (with the filter):

```
[@3 humanoids >> implode]
```

Might output:

```
orc, goblin, kolbold
```

Which separates each result with a comma.

If you wish to use something other than a comma as the glue string, place the string after 'implode'.

For example, to separate each result using HTML table cell tags:

```
<td> [@3 humanoids >> implode </td><td>] </td>
```

Might output:

```
<td> orc</td><td>goblin</td><td>kolbold </td>
```

3.36.6 Italic

The Italic filter converts the results to italicized text in a platform independent manner. In most cases this simply means automatic wrapping of the text with necessary HTML tags. In cases where HTML rendering is not available (for example, when used on a command line), text is returned surrounded by asterisks.

Example:

```
[This is italic >> Italic]
```

Would output:

```
This is italic
```

See the Formatting 37 command for more information.

3.36.7 Left

The Left filter outputs a substring of one or more characters from the start of a table result. The default number of characters to return is 1. To return more than one character, pass the number along as a parameter to the filter.

Example:

```
[ abcdefghijklmnopqrstuvwxyz >> Left]
```

Would output:

```
a
```

That is, the left most character in the text.

Alternatively:

```
[ abcdefghijklmnopqrstuvwxyz >> Left 5]
```

Would output:

```
abcde
```

That is, the first five characters in the text.

3.36.8 Length

The Length filter returns the length, in characters, of a table result. The Length filter ignores leading and trailing spaces when calculating the length of text.
Example:

```
[ how long is this? >> Length]
```

Would output:

```
17
```

Since the text is 17 characters long.

3.36.9 Lower

The Lower filter converts text to all lower case.

Example:

```
[THIS IS LOWER CASE >> Lower]
```

Would output:

```
this is lower case
```

3.36.10 LTrim

The LTrim filter trims leading spaces from a table result.

3.36.11 +- (PlusMinus)

The +-, or PlusMinus filter allows you to format numbers with a plus sign if the number is zero or greater. This is useful for formatting skill and roll modifiers.

Example:

```
Table: calcmodifier
[{1d40-20} >> +-]
; or
[{1d40-20} >> plusminus]
```

This rolls 1d40-20, and formats the result as a modifier using the +- filter. Note that you can use either `+-` or `plusminus` as the filter name.

This might return:

```
-10
+1
+19
-3
+18
-7
+0
-17
-17
-13
-9
-6
-17
+14
+11
-14
```

3.36.12 Proper

The Proper filter capitalizes the first letter in each word.

Example:

```
[the town of brekville >> Proper]
```

Would output:

```
The Town Of Brekville
```

3.36.13 Replace

The Replace filter allows you to replace the occurrences of one piece of text with another. When you use this filter you pass in two additional pieces of information - the text you want to find, and the text you want to insert as the replacement. The find and replace pieces are delimited with a forward slash, not a space. The format to use this filter is:

```
[input-text >> replace /text-to-find/text-to-replace/]
```

Where *input-text* is the result of a table roll or literal text, *text-to-find* is the text you want to find, and *text-to-replace* is the text you want to insert as the replacement.

Example:

```
table: replace_example
Set:a=you see orcs attacking
[{$a} >> replace /orcs/ancient red dragons/]
```

Would output:

```
you see ancient red dragons attacking
```

As you can see from the example, spaces can be included in the find and replace expressions.

All nested expressions are evaluated first *before* the filter is executed. So while it's certainly possible to use variables and table calls within a replace expression, take care to ensure that the end result of the nested calls output the proper format for the replace filter.

To replace a forward slash, escape the forward slash by placing a back-slash in front of it. For example:

```
[slash / can be escaped  >> replace /slash \/ can/question mark doesn't have to/]
```

Would output:

```
question mark doesn't have to be escaped
```

3.36.14 Reverse

The Reverse filter reverses the text being passed into it.

Example:

```
table: reverse_example
Set:a=esrever ot txet si siht
[{$a} >> reverse]
```

Would output:

```
this is text to reverse
```

3.36.15 Right

The Right filter outputs a substring of one or more characters from the *end* of a table result. The default number of characters to return is 1. To return more than one character, pass the number along as a parameter to the filter.

Example:

```
[ abcdefghijklmnopqrstuvwxyz >> Right]
```

Would output:

```
z
```

That is, the right most character in the text.

Alternatively:

```
[ abcdefghijklmnopqrstuvwxyz >> Right 5]
```

Would output:

```
vwxyz
```

That is, the last five characters in the text.

3.36.16 RTrim

The RTrim filter removes trailing spaces from a table roll result.

3.36.17 Sort

The Sort filter alphabetically sorts the results of a table roll. Obviously, this only applies when you call a table roll with more than one repetition.

Example:

```
[@4 spells >> sort]
```

This would select 4 spells, and return them sorted.

3.36.18 Substr

The Substr filter extracts a specified part of a table result (a *sub-string*). There are two parameters to the Substr filter. The first is the start index of the text, and the second is the length of the sub-string

(number of characters) to return. If the length is omitted, length is assumed to be 1. If length is set to 0, then the entire text is returned starting at starting index.

Examples:

```
[ abcdefghijklmnopqrstuvwxyz >> Substr 5]
```

Outputs:

```
e
```

(The 5th letter in the text, 'e')

```
[ abcdefghijklmnopqrstuvwxyz >> Substr 5 5]
```

Outputs:

```
efghi
```

(outputs the five characters starting at the 5th character in, 'e')

```
[ abcdefghijklmnopqrstuvwxyz >> Substr 5 0]
```

Outputs:

```
efghijklmnopqrstuvwxyz
```

(returns the rest of the text, starting at 'e')

3.36.19 Trim

The Trim filter trims leading and trailing spaces from a table roll result.

3.36.20 Underline

The Underline filter converts the results to underlined text in a platform independent manner. In most cases this simply means automatic wrapping of the text with necessary HTML tags. In cases where HTML rendering is not available (for example, when used on a command line), text is returned surrounded by quotes.

Example:

```
[This is underlined >> Underline]
```

Would output:

```
This is italic
```

See the Formatting 37 command for more information.

3.36.21 Upper

The Upper filter converts the results of a table roll to upper case.

Example:

```
[This is upper case >> Upper]
```

Would output:

```
THIS IS UPPER CASE
```