MuCho - Multiple Choice Adventure Engine

Jari Komppa

November 2, 2016

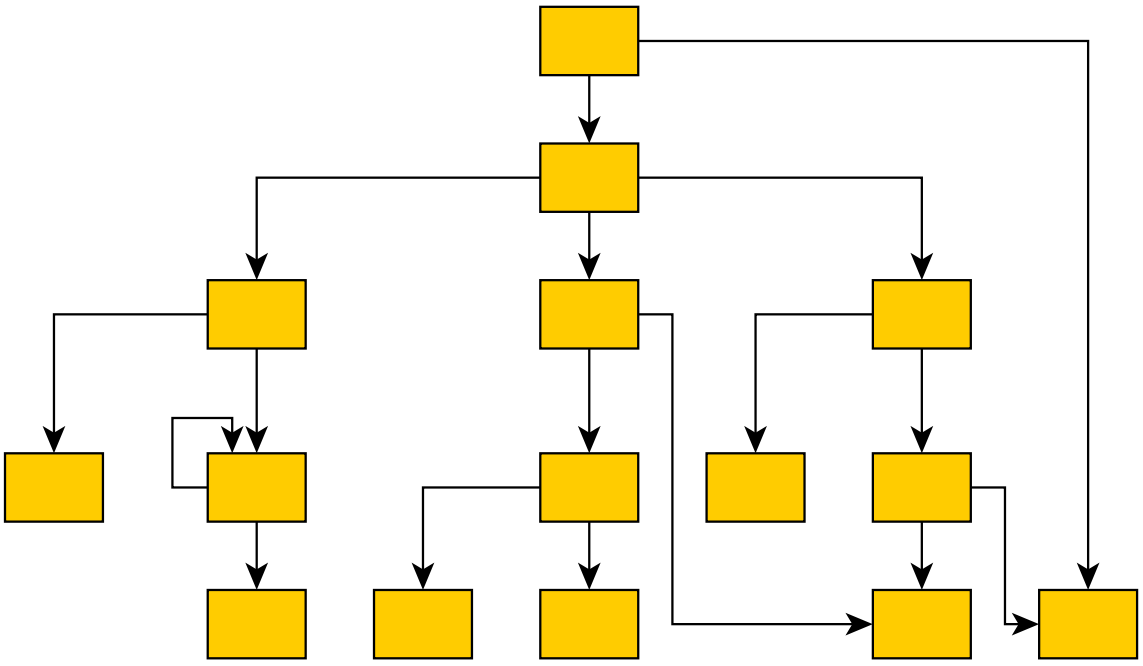
Contents

CHAPTER 1

Introduction

MuCho is a engine for making multiple choice adventure games (also known as choose your own adventure (CYOA), or gamebook) for the 48k ZX Spectrum.

Using the engine requires no actual programming, but having some technical background may help.



Think of each box as a page in a book, and at the bottom of each page you'll find something in the lines of, "Turn to page 32 if you want to..".

1.1 Development Kit Contents

The structure of the development kit is as follows:

```
dividers\           – Example custom divider images
    divider_checkerline.png
    divider_crosser.png
    divider_dither.png
    ...
selectors\
    selector_barwave.png
    selector_sin.png
    selector_pitchfork.png
    ...
fonts\              – Example custom font images
    font_consolas.png
    font_courier_new.png
    font_lucida_console.png
    ...
manual\             – This MuCho manual in different formats
    mucho.pdf
    mucho.html
    mucho.mobi
    mucho.epub
examples\           – Some example MuCho adventures
    simple\
        simple.txt
        build.bat
    complex\
        ...
    dukes\
        ...
    ...
crt0.ihx            – The game engine itself, compiled z80 code
mc.exe              – MuCho compiler
mackarel.exe        – Packager and .tap builder
```

The fonts directory also contains a photoshop template for fonts. The MuCho compiler turns adventure text files into data usable by the engine.

The source code to the game engine can be found in github at:

`https://github.com/jarikomppa/speccy/tree/master/mucho`

1.2 Legalese

TL;DR: MuCho is based on free software and is free software, feel free to use it however you see fit, including commercial usage. (If it doesn't work, you're pretty much on your own, though).

-

MuCho was written in C and compiled with SDCC

`http://sdcc.sourceforge.net/`

-

Uses BeepFX by Shiru, public domain

`https://shiru.untergrund.net/software.shtml`

-

Uses ZX7 compression by Einar Saukas

`http://www.worldofspectrum.org/infoseekid.cgi?id=0027996`

-

The rest of MuCho is released under Unlicense, which is basically public domain.

`https://github.com/jarikomppa/speccy/tree/master/mucho`

any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to

`http://unlicense.org/`

CHAPTER 2

MuCho Story Definition Language

The MuCho story definition language may sound scary, but it is actually very simple. The MuCho scripts are plain text files, such as those that can be created with Windows Notepad, although the author recommends using something more advanced, such as UltraEdit, Sublime Text or Notepad++.

Not all of the features of the language are needed to write adventures; this document is structured so that more complicated features are described later, so you won't need to read all of it to get started. If something feels complicated, don't worry about it - you can come back to it later.

2.1 Structure

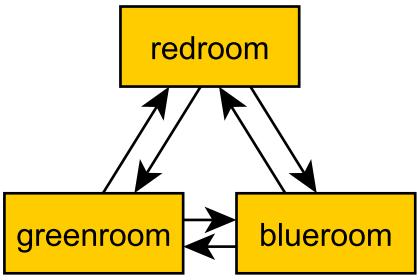
Structure of a multiple choice adventure is based on pages (other used terms include rooms, locations, nodes). Each page consists of description part as well as zero or more choices for the user (also known as answers, selections, options).

Consider the computer asking the user a question, and offering several possible answers for the question.

The script for a simple page looks like this:

```
$Q roomname
The description of the room
$A otherroomname
Go to other room
```

The choices link pages together. Here we'll describe three rooms and let the user travel between them:




```
$Q redroom
You're in the red room.
$A greenroom
Go to the green room
$A blueroom
Go to the blue room

$Q greenroom
You're in the green room.
$A blueroom
Go to the blue room
$A redroom
Go to the red room

$Q blueroom
You're in the blue room.
$A greenroom
Go to the green room
$A redroom
Go to the red room
```

Everything between two $Q lines goes to a single page. The first word after the $Q and $A is the label of the room. In programming terms, you could think of it as GOTO label. Two rooms may not have the same label, but any number of $A lines may point to the same label. Labels may not include spaces (i.e, they need to be a single word).

It is also perfectly legal for a room to have an choice which goes back to the same page. This is actually rather useful.

If a room has no $A lines, it is considered an “end node”, and the adventure ends if player ends up on that page.

2.1.1 Text

The text in the page descriptions is automatically formatted, removing any unnecessary “white space” like line breaks or additional spaces. To start a new paragraph, add an empty line in your text.

```
All of
these go
on the
same line.

This is a new paragraph.

    Adding   more   spaces   does   nothing.
```

To add an empty line, use the $P line (see below).

2.1.2 Comments

You can add comments in the script which do not end up in your adventure file by starting a line with the hash character.

```
# This is a comment
This is not a comment
```

2.1.3 Images

Images can be added to a page’s description portion by using an $I line.

```
$Q fountain
You arrive at a picturesque fountain.
$I fountain.scr
Water from the fountain sparkles in the sunlight.
```

The first word after the $I is considered to be a file name. The files should be standard spectrum screen files (6912 bytes).

Empty space at the bottom of the .scr image is trimmed. A maximum of 14 character rows, or 112 pixels is used. If bigger images are desired, you have to use several images.

If you use the same image several times in your adventure, the image data is still only stored once.

Images can not be used after a $A line.

2.1.4 Paragraph Breaks

While the text is often expected to flow, sometimes empty lines may be desirable. To output an empty line, use the $P statement.

```
$Q fountain
You arrive at a picturesque fountain. Assume there's a long description here.
$P
You can see some mushrooms here.
```

Technically the $P is not a real statement and cannot be used with commands or flags (see below). Its effect is part of the text, and thus will not break an $O line segment (again, see below). This may be relevant later on, but for the time being just consider $P something that adds an empty line in your text.

The $P line cannot be used after a $A line, as text after $A must fit on a single line.

2.2 Flags

In addition to the basic structure of the pages, flags can be used to introduce more complicated behavior. Every time the player goes to a page, the flag with the page's name is turned on. Using a $O line (as in Oh, uppercase o, not 0, zero) an optional part can be added to a page's description.

```
$Q alleyway
You're in a dark alley. There's a door.
$O backyard
You remember seeing the bad guys inside when you were at the back yard.
```

In this example the "You remember.." line is only shown if the player has visited the backyard page before ending up here.

If you wish to have optional part of the text between two parts of text that is always shown, you can reset the optionality by using an empty $O line.

```
Text always shown
$O daytime
Text shown only during daytime
$O
Another text always shown
```

2.2.1 Checking Flags

You can check if a flag is on by using its label. You can also check if a flag is not on by prefixing the flag's label with an exclamation point (!), like !backyard.

```
$O daytime
The sun is shining.
$O !daytime
The moon is bright tonight.
```

Alternate syntax for this is to use the has: and not: prefixes.

```
$O has:daytime
```

```
The sun is shining.
$O not:daytime
The moon is bright tonight.
```

If you wish something to happen randomly, you can use the rnd: prefix with a value.

```
$O rnd:64
Thorin sits down and starts singing about gold.
```

The maximum value is 255, so a value of 128 gives about 50% chance, 64 gives 25%, 32 gives 12.5%, etc.

The flag checks can also be used with the $Q, $I and $A lines.

If flag check is used with the $I line, the image is only shown if the flag check succeeds.

```
$I sun.scr has:daytime
$I moon.scr not:daytime
```

If flag check is used with the $A line, the choice is only given to the player if the flag check succeeds.

```
$A mine has:cleared
Enter the mine through the cleared tunnel.
```

If flag check is used with the $Q line, any other commands on that same line are only executed if the flag check is positive. (See below for the other commands).

2.2.2 Setting Flags

To set a flag, use the set: prefix with a flag label.

```
$A alleyway set:trapped
Place the trap just outside the door.
```

If used with the $A line, the command is performed if the player picks the line. If used on a $Q, $I or $O line, the command is performed if the line's flag check succeeds.

Setting a flag that is already on is legal, but has no effect.

2.2.3 Clearing Flags

You can also clear flags, including the ones set by visiting a page. This is done with the clear: or clr: command.

```
$A alleyway clear:trapped
Change your mind, and clear the trap from the door.
```

Clearing a flag that is not on is legal, but has no effect.

2.2.4 Toggling Flags

Flags can be toggled with the toggle: command. This way you don't need to know which state the flag is in, if you wish to switch between two states.

```
$Q busystreet toggle:trafficlights
$O trafficlights
The traffic lights are red.
$O !trafficlights
The traffic lights are green.

$A busystreet
Wait for a while
$A sleepytown !trafficlights
Cross the street
```

2.3 Numbers

Sometimes it's useful to handle numbers instead. Many gamebooks have concept of hit points, for instance.

There can be a maximum of 32 numeric variables, and each holds a number from 0 to 255.

2.3.1 Setting Numbers

To set the value of a number, use the = operator.

```
$A fountain hitpoints=7
Drink from the fountain
```

You can set a number variable to the value of another variable or to a fixed value.

```
$A tavern temp=player_money player_money=stranger_money stranger_money=temp
Swap purses with the stranger
```

2.3.2 Adding and Subtracting Numbers

Adding and subtracting are done with the - and + operators.

```
$A fountain hitpoints-1
Eat a mushroom
$A fountain hitpoints+1
Eat a biscuit
```

Again, fixed values or other variables can be used.

```
$A fountain hitpoints+potion potion-1 potion>0
Drink from the healing potion
```

Note that the variables can over- and under-run. This means that if you subtract 1 from 0, you'll get 255, and if you add 1 to 255, you'll get 0.

If you prefer, you can use -= and += instead of - and +:

```
$A fountain score+=3 darts-=1 darts>0
Throw a dart at the board
```

2.3.3 Comparing Numbers

Numeric variables can be compared in various ways, to each other and to fixed numbers.

```
$O a==42
a is 42
$O a!=42
a is not 42
$O a>42
a is bigger than 42
$O a>=42
a is bigger or equal to 42
$O a<42
a is smaller than 42
$O a<=42
a is smaller or equal to 42
```

Note that a=1 means "assign 1 to a", while a==1 means "is a equal to 1".

As you might expect, it's also possible to compare two variables.

```
$O a==b
a is equal to b
$O a!=b
a is not equal to b
$O a>b
a is bigger than b
$O a>=b
a is bigger or equal to b
$O a<b
a is smaller than b
$O a<=b
a is smaller or equal to b
```

Here again, a=b means "assign a to value of b" and a==b means "is a equal to b".

2.3.4 Printing Numbers

It is also possible to print out the values of the numeric variables, by simply putting the variable name between << and >> in the text. Note that there must be no spaces between these characters.

```
$Q store
The shopkeeper polishes an apple while he's waiting for you
```

```
to make a selection. You currently have <<gold>> gold.
$A store gold>5 gold-5 set:dagger
Buy the dagger for 5 gold
```

2.3.5 Mixing Flags and Numbers

Numbers and flags do not mix. If you try to assign flag to a numeric variable, for instance, it won't do what you expect:

```
$O set:parrot
$O bird=parrot
```

This will create a new numeric variable called "parrot", which will live alongside the flag "parrot". The compiler will warn you if you try to do this.

2.4 Display Attributes

It is also possible to change the display attributes (e.g, text color) as well as border color.

2.4.1 Text Attributes

To change the color of text or background (INK and PAPER in spectrum terms), use the attr: command.

Color	Ink	Paper
Black	0	0
Blue	1	8
Red	2	16
Magenta	3	24
Green	4	32
Cyan	5	40
Yellow	6	48
White	7	56

You can also use the extra bits of the attribute for bright and flashing text:

Extra	Value
Bright	64
Flash	128

Simply sum the values together that you desire. For bright white on black, use 7+64=71. For red on yellow background, 2+48=50. For blinking bright red, 2+64+128=194, etc.

```
$Q launchroom
The text on the console reads:
$O attr:194
SAFETY OFF — DO NOT PRESS THE BUTTON
```

```
$O attr:7
You wonder what to do. There's a button here.
```

Note that the attr: command sets the attribute for the next bit of text. To set the attribute for the whole screen, you need to clear the screen after setting the attribute. See cls:, later on.

2.4.2 Interface Attributes

You can also change the color of the divider line (the line between the choices and the page text) as well as the color of the interface (where the choices are listed).

The divider attribute is set with dattr: command. It actually gets changed when the bottom is cleared the next time, which occurs when page is changed or when the bottom is cleared with and cls: command (see below).

```
$O dattr:5
```

The interface attribute is set with iattr: command.

```
$O iattr:56
```

2.4.3 Clearing the Screen

To clear the screen (primarily to set the color to the whole screen, as the screen is also cleared whenever a new page is drawn), use the cls: command.

What to clear	Value
Everything	0
Page	1
Interface	2

In most cases you can just use cls:0.

```
$Q darkroom attr:2 cls:0
```

2.4.4 Changing the Border Color

The border color can be changed with the border: command.

Color	Value
Black	0
Blue	1
Red	2
Magenta	3
Green	4
Cyan	5
Yellow	6
White	7

The color is changed instantly, so if you change the color several times during a page, the border color also changes multiple times. Playing sound may also change the border color.

```
$O trafficlight border:2
$O !trafficlight border:4
```

2.5 Go and Gosub

Sometimes it is useful to interrupt the normal page flow and do something else for a change.

For example, if your game has a hit point mechanism, it would be wasteful to add checking if the player has died on every single page.

```
$Q healthcheck
$O hitpoints==0 go:dead
$O hitpoints<5
You're not feeling too good.

$Q fountain
The fountain.
$O gosub:healthcheck
You find yourself near a marble fountain in the forest
clearing. There are some mushrooms nearby.
```

In the example above, whenever the player arrives at the "fountain" page, the game will load the healthcheck page, which will first check if player is dead, and if so, will turn to the "dead" page immediately. If the player is still alive, the page will output the "You're not feeling too good" message if hitpoints are low. Otherwise the processing of that sub-page is done and drawing of the "fountain" page resumes from where we were at.

Only one subpage can be used at a time, so you can't "call" subpages from subpages. If you try to gosub to a subpage from a subpage, the results will be undefined, which is programmer speak for "bad things will happen". Subpages can also not have any $A statements; if any exist, they will be ignored.

2.6 More Technically Speaking

Here's some a bit more technical notes which may be useful in problematic cases.

2.6.1 Command Execution Order

To get a little bit more nitty-gritty, here's a few words about command execution order.

Commands are, generally speaking, executed in the order they're set. So, to clear the screen with a new attribute, first set the attribute, then clear the screen.

```
$O attr:7 cls:0
```

However, flag check is always performed first, and only if the whole flag check succeeds, the other commands are executed. Thus, if you write something as convoluted as:

```
$O set:flaggy flaggy clear:flaggy toggle:flaggy
```

what happens is:

```
If flaggy is on:
    Set flaggy
    Clear flaggy
    Toggle flaggy
```

Another example, just to be sure:

```
$O attr:7 apple cls:1 banana rnd:64 orange set:strawberry
```

This becomes:

```
If apple is on, and banana is on,
and random is bigger than 64 and orange is on:
    Set attribute to 7
    Clear the page
    Set strawberry
```

To reiterate: first everything that affects whether the line should be executed is evaluated, and only then the rest are executed, assuming all of those things turn out to be true.

If any of the checks fail, the rest are not executed.

2.6.2 Go and Gosub Revisited

The exact point at which the go and gosub commands are executed is at the end of the statement, meaning that:

```
$Q mysubpage
Hello

$Q normalpage
$O gosub:mysubpage
World
```

..will actually output World Hello instead of Hello World. To get the result you might expect, simply add another $O statement:

```
$Q mysubpage
Hello

$Q normalpage
$O gosub:mysubpage
$O
World
```

Using the go: command will send the player to a new page just as if they had selected an option: the room's flag will be set, the screen cleared, etc.

2.6.3 The $P Statement

Unlike other $-lines the $P line is completely processed in the compiler, by adding empty line to the previous set of text lines. The game engine will never see that a $P line was here. This has a few implications:

First, the $P line can not be predicated, i.e, it may not have any commands associated with it. Second, it does not break the previous block.

```
$O !light
It's pitch black here.
$P
You can hear music in the distance.
```

In the above example the "You can hear music in the distance" message is part of the $O line block, and thus the player won't see the text if the "light" flag is on.

2.6.4 Limits

A compiled room must not take more than 4096 bytes. The MuCho compiler will tell you when you cross the limit.

Maximum size for the compiled and compressed data is 29952 bytes (a bit over 29kB). All data is compressed, images and code blocks are only stored once even if they are used several times, and labels (flags, variable names, room labels) are turned into numbers, so label length doesn't matter.

Images may be up to 112 pixels high. If more is needed, split images in pieces.

There's a maximum of 1024 flags. You will most likely run out of space before running out of flags.

There's a maximum of 32 numeric variables. The numbers in numeric variables can range from 0 to 255.

Maximum of 16 active options at the same time. If there are more, the game will simply stop showing more options.

2.7 Example Snippets

Here's a few example snippets of more complicated behaviors that are possible with flag manipulation.

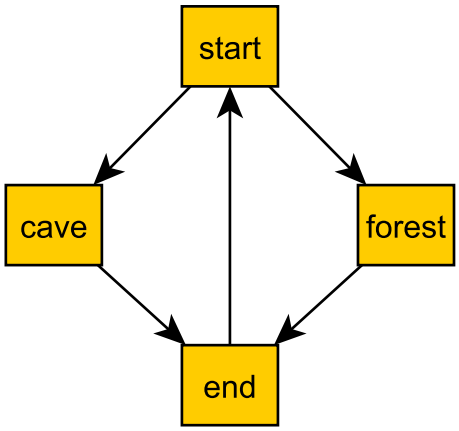
2.7.1 Button Toggle

```
$Q room
$O !light
Light is off
$O light
Light is on
$A room toggle:light
Toggle light
```

This example has a single room with a light that can be on or off. The player's option always points back to the same room, and the optional text blocks show whether the light is on or off.

The same flag could be used for other things, such as revealing other things to do if the light is on.

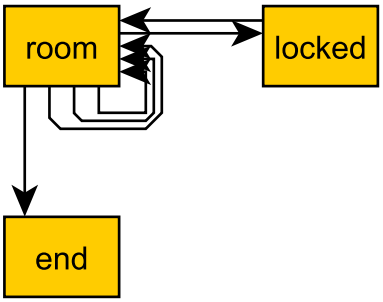
2.7.2 Using Room Flags to Change Flavor Text




```
$Q start
Go to cave or forest?
$A cave
Go to cave
$A forest
Go to forest

$Q cave
Well, go on..
$A end
Exit cave

$Q forest
Well, go on..
$A end
Exit forest

$Q end
$O cave clr:cave
As you exit the cave, you find...
$O forest clr:forest
As you exit the forest, you find...
$O
...the end of the example.
$A start
Restart
```

This example shows two different routes to a room, and the room's description reacts to where you came from. Note that the room flags are also cleared when the flavor text is shown, so it won't be shown again, should the player end up back on the same page from some other direction later on.

2.7.3 Lock and Key




```
$Q room
You're in a room with a door.
$O !key
There is a key here.
$O open
The door is open.

$A room !key set:key
Get key
$A room key !unlocked set:unlocked
Unlock door
$A locked !unlocked !open
Open door
$A room unlocked !open set:open
Open door
$A end open
Exit

$Q locked
The door is locked.
$A room
Dang it!

$Q end
You went through the door.
```

In this example the player needs to take a key, unlock door, open door, and leave. When the player initially reaches this page, the view looks like this:

```
You're in a room with a door.
There is a key here.
----
Get key
Open door
```

The “open door” option shown here does not actually open the door, but sends the player to a page that just says “The door is locked” with the only option of returning back to the page.

If the “Get key” option is picked, the player is sent back to the same page, but with the “key” flag enabled. Now the page looks like this:

```
You're in a room with a door.
---
Unlock door
Open door
```

The “Open door” option is still the same “The door is locked” option. Hitting the “Unlock” option again sends the player back to the same page, now with the “unlocked” flag on.

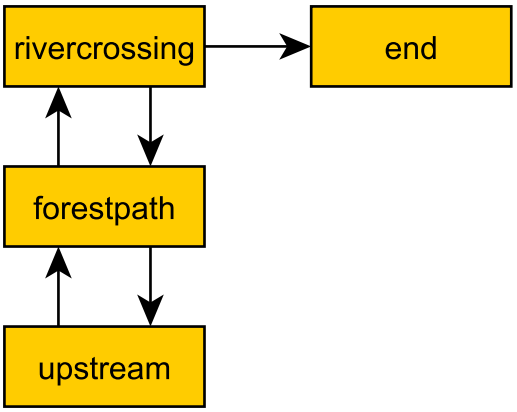
```
You're in a room with a door.
---
Open door
```

Now the “Open door” option does something different (it’s a different “Open door”, after all); sending the player back to the same page with the “open” flag on.

```
You're in a room with a door.
The door is open.
---
Exit
```

Finally, the player is given the “Exit” option, which sends the player to the end node, where the game ends.

2.7.4 Separate Gate




```
$Q rivercrossing
River flows north to south. There's signs of people
having crossed the river here.
$O !dam
The flow is too strong for you to pass.
$A forestpath
Go north
$A end dam
Cross the river

$Q forestpath
You're on a north-south forest path. There's a river
to the west.
$A rivercrossing
Go south
$A upstream

$Q upstream
This is as far north as you can go. There's a forest
path south, and a river to the west.
$O !dam
It seems you could easily roll a big stone to the river,
$O dam
Huge rock is blocking the river flow.
creating a dam.
$A forestpath
Go south
$A upstream set:dam
Roll that rock.

$Q end
You crossed the river.
```

This example uses several locations. The player has to travel upstream to create a dam in order to be able to cross the river.

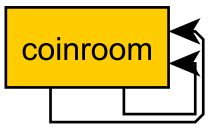
2.7.5 Dialogue

The description-choices structure can also describe dialogue with a non-player character.

```
$Q start
Well hello there. What do you want to know about?
$A name
Name
$A jobs
Jobs
$A virtue
Virtue
```

2.7.6 Sticky Randoms

In order to keep the result of a random decision, set the result in a flag.




```
$Q coinroom
$O !toss clr:result
$O !toss rnd:64 set:result
$O set:toss
$O result
The coin shows heads.
$O !result
The coin shows tails.

$A coinroom
Look again.
$A coinroom clear:toss
Toss again.
```

In pseudocode, the above becomes:

```
If "toss" is not set:
    Clear "result"
If "toss" is not set, AND random is less than 64:
    Set "result"
Set "toss"
If "result" is set
    Print "The coin shows heads."
If "result" is not set:
    Print "The coin shows tails."
```

2.7.7 Nerdy Boolean Logic

Freely ignore this if you don't consider yourself a nerd.

In order to AND two flags, simply check both of them at the same time:

```
$O this that
```

In order to OR two flags, check them separately and set a third flag

```
$O this set:thisorthat
$O that set:thisorthat
$O thisorthat
```

In order to exclusively-or two flags, you can use toggle.

```
$O clr:thisxorthat
$O this toggle:thisxorthat
$O that toggle:thisxorthat
```

2.7.8 Number Overflow

The 8-bit numeric variables overflow (and underflow) rather easily. You can check for this by using a temporary variable and seeing if the results are impossible:

```
$O temp=myvar temp+5
# Temp should now be 5 more than myvar
$O temp<myvar
Overflow!
$O temp>myvar myvar=temp
Business as usual—
```

Same goes for underflow.

```
$O temp=myvar temp—5
# Temp should now be 5 less than myvar
$O temp>myvar
Underflow!
$O temp<myvar myvar=temp
Business as usual.
```

2.7.9 Swapping Values

A temporary variable can also be used to swap contents of two numeric variables.

```
$A tavern temp=player_money player_money=stranger_money stranger_money=temp
Swap purses with the stranger
```

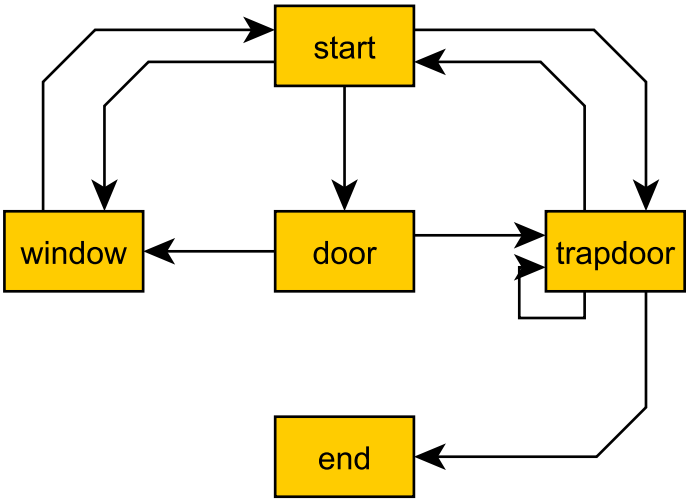
Swapping the state of two flags can also be done through a temporary flag, but can not be done on one line.

```
$O clr:t
$O a set:t clr:a
$O b set:a clr:b
$O t set:b
```

2.8 Examples

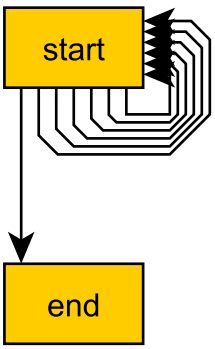
Here's short overview of the examples included in the kit.

2.8.1 Simple



The simple example has four pages, with links between them. None of the more advanced features of MuCho are used.

2.8.2 Complex

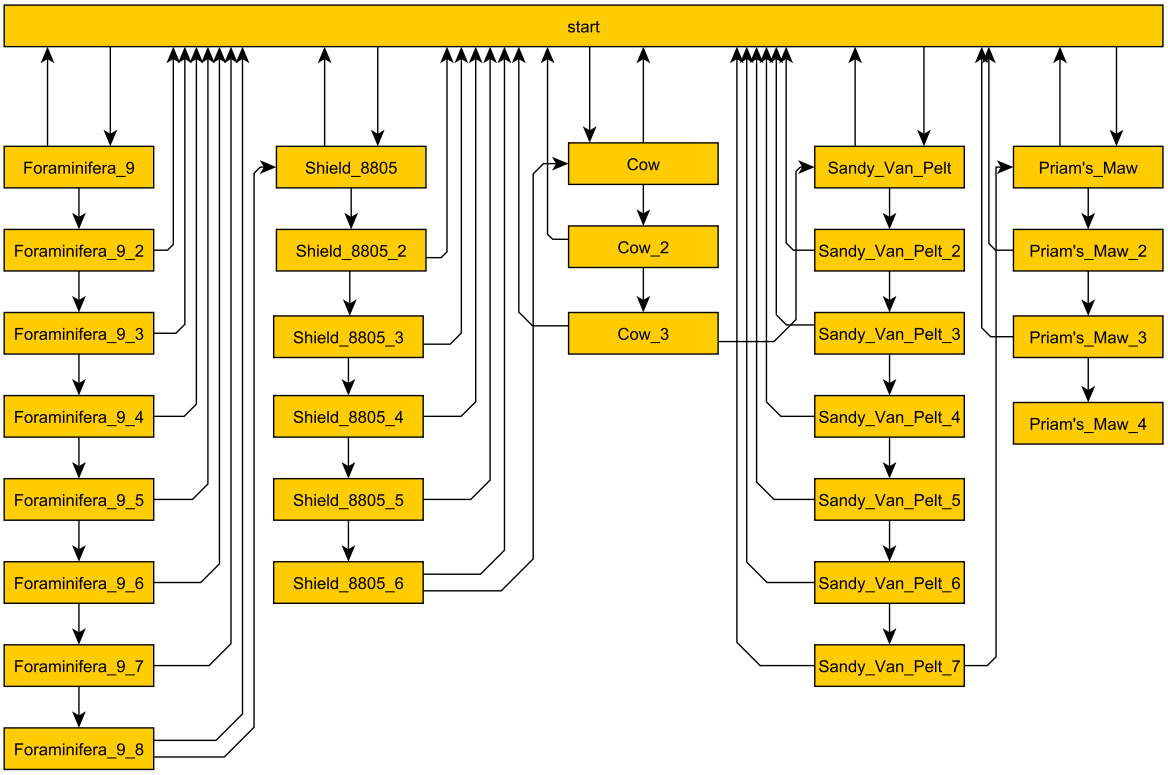


The complex example concentrates on the more advanced features of MuCho, including flag manipulation, attributes, colors, images and sound, and primarily has one heavily self-referential page.

The player needs to put on a light in order to see a key, which the player must pick up, unlock a door, open the door, and then leave.

While all of this is going on, the player sees the screen border change color based on the neon lights blinking outdoors, the page attributes change based on whether the light is on or off, and various actions trigger sound effects.

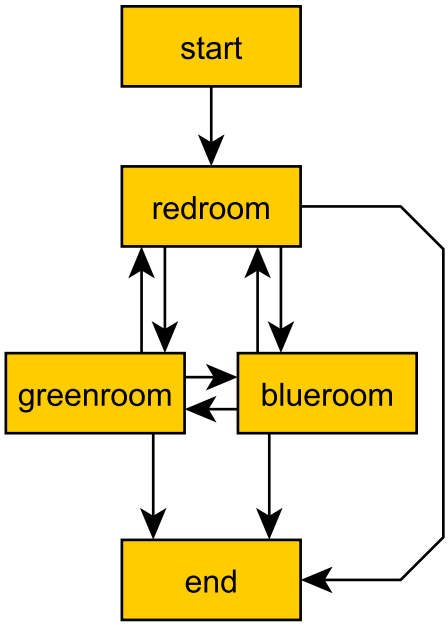
2.8.3 Dukes



The Dukes example's flow chart may look scary, but it's simply 29 pages in a chain, with the possibility to hop back to the beginning after each page, as well as possibility to hop to the start of any of the five chapters from the start page.

The Dukes example is a stress test: it's Frederik Pohl's "The day of the boomer dukes" novella (about 46kB of text, public domain) in MuCho format.

2.8.4 Traveller



In the traveller example the player is expected to run after a fleeing creature, which hops from one room to the next randomly, with a small chance of it staying put.

This effect is created through a relatively complicated logic. Each room consists of the following kind of structure:

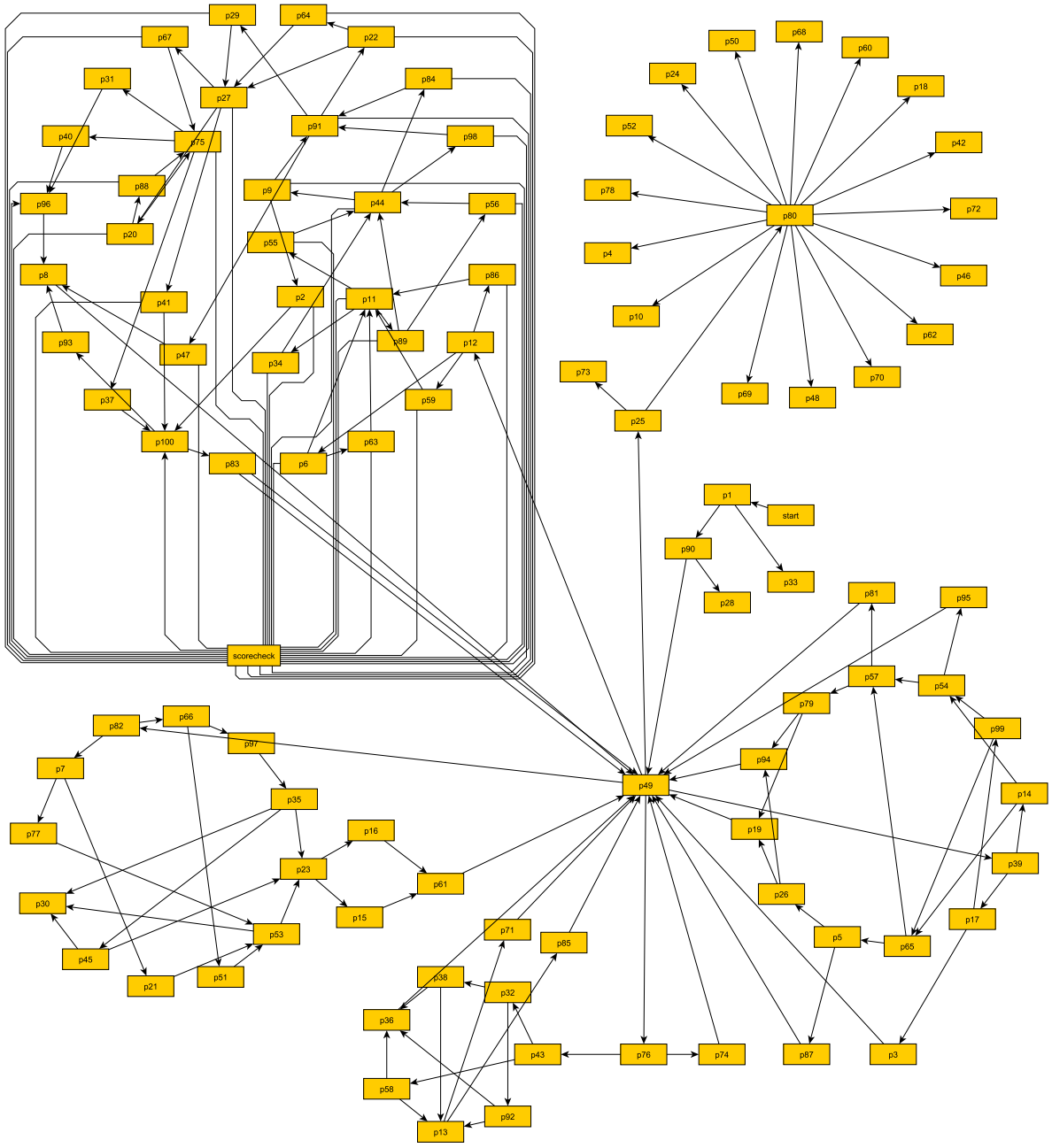
```
$O clr:flip clr:stay
$O rnd:127 set:flip
$O rnd:10 set:stay
$O red !stay flip clr:red set:green
You see Yellow run towards the green room.
$O red !stay !flip clr:red set:blue
You see Yellow run towards the blue room.
$O red
You see a wild Yellow here.
```

First, “flip” and “stay” flags are cleared. Next, “flip” is set 50% of the time, and “stay” is set rather rarely.

Then, if the creature is in this room and is not staying, depending on whether flip is on, the creature is moved from this room to one of the other ones.

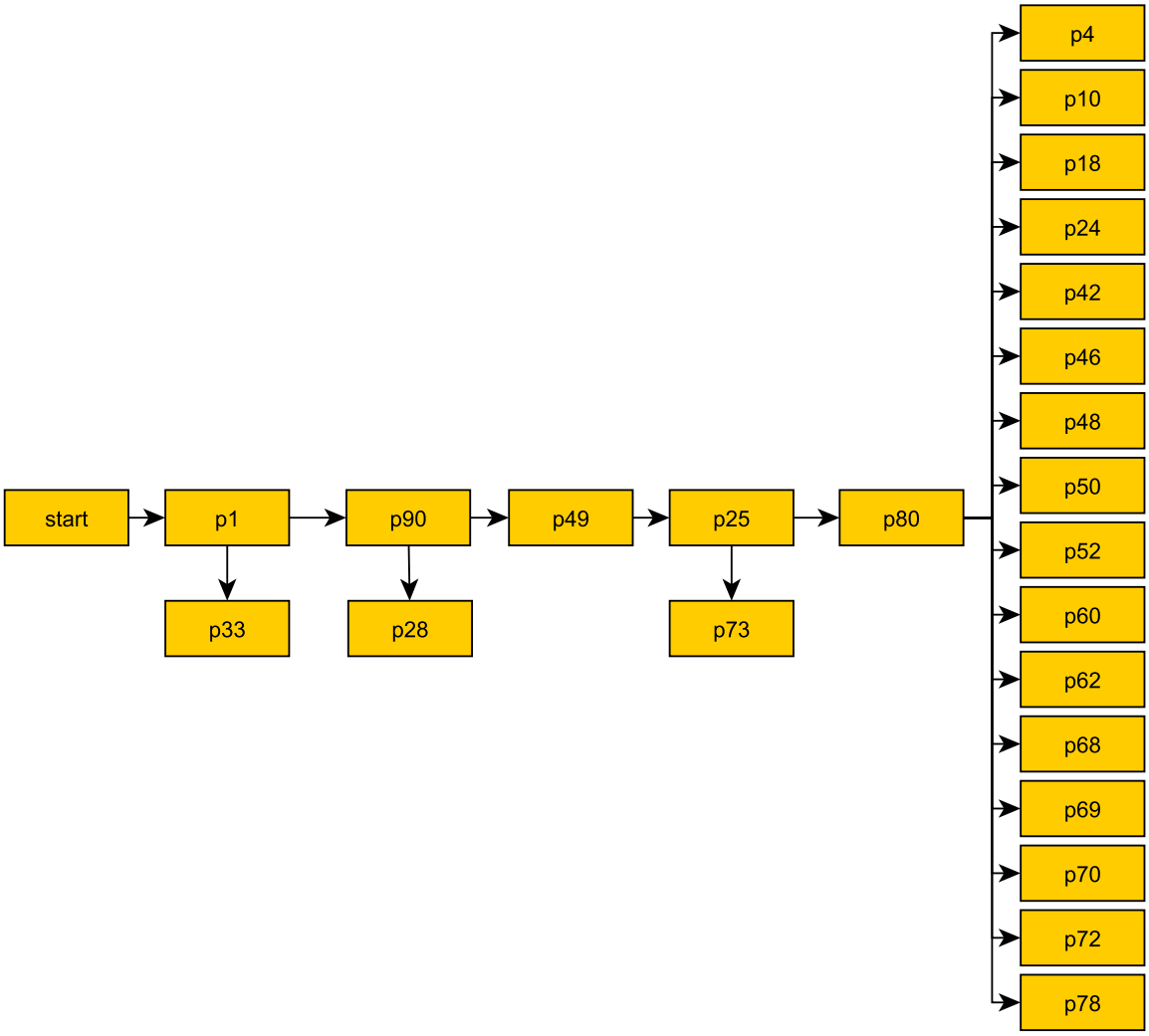
If the creature is still in this room after those checks, we announce it.

2.8.5 Waiting

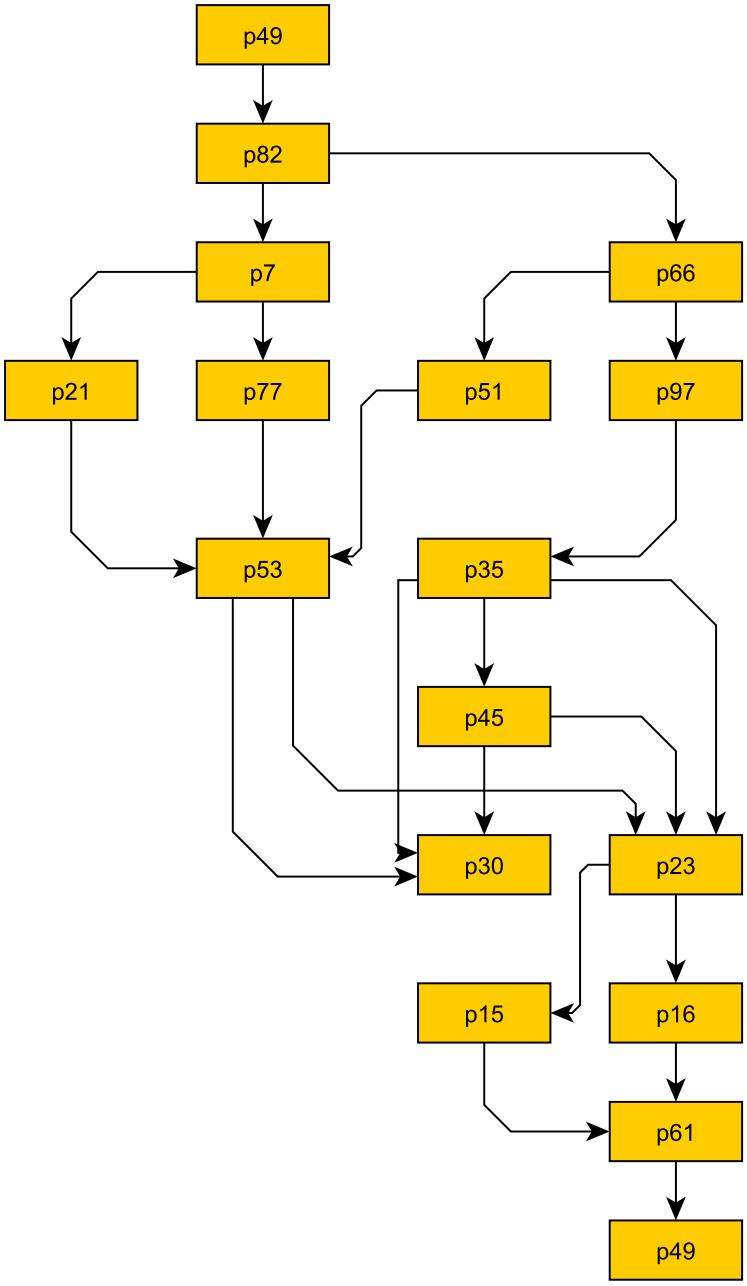


This is the complete “Waiting for the Light” gamebook by Kieran Coghlan, consisting of a hundred locations, flags and numbers, converted to MuCho. The MuCho version was done with persmission of the author.

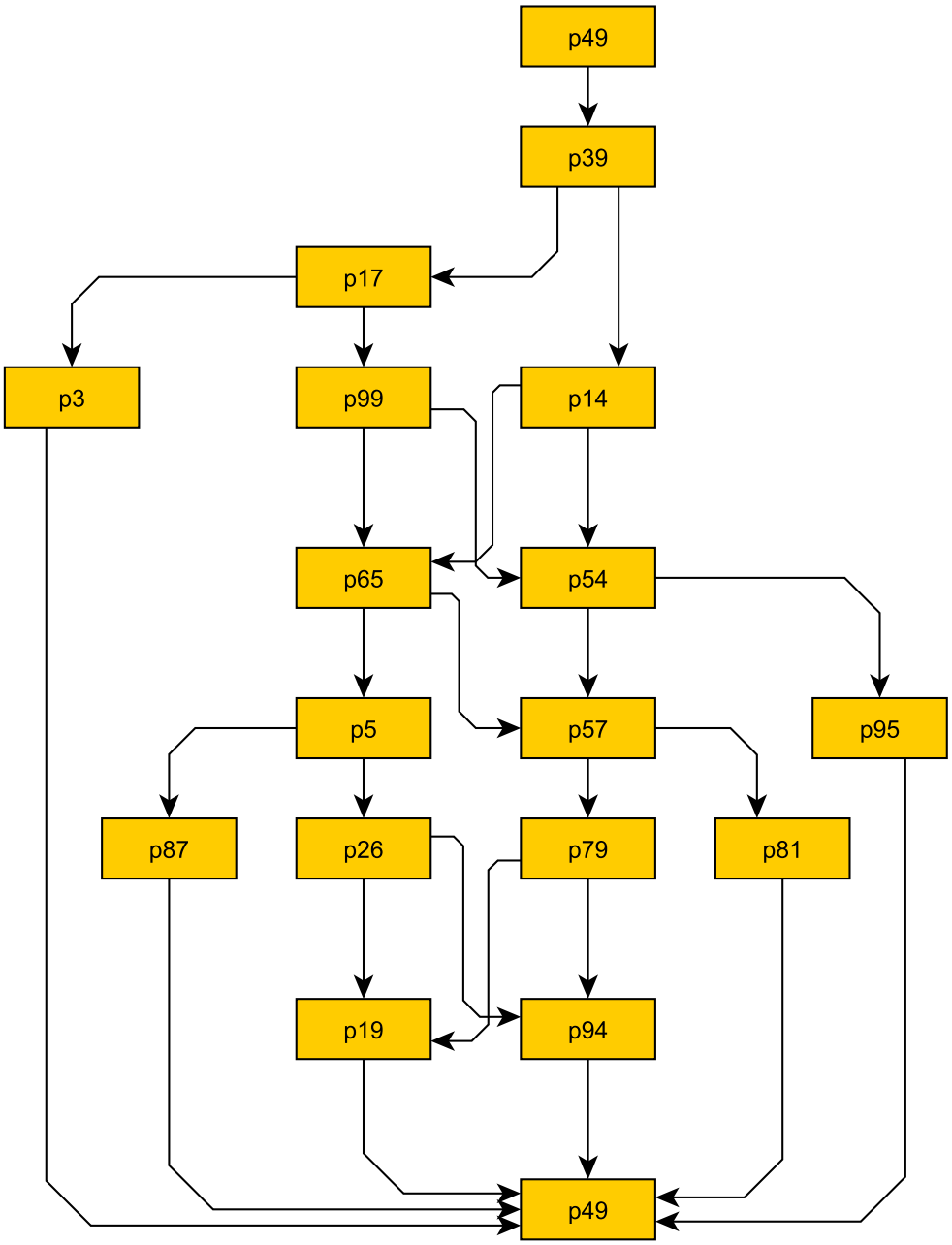
To understand the structure, let’s look at some sub-graphs.



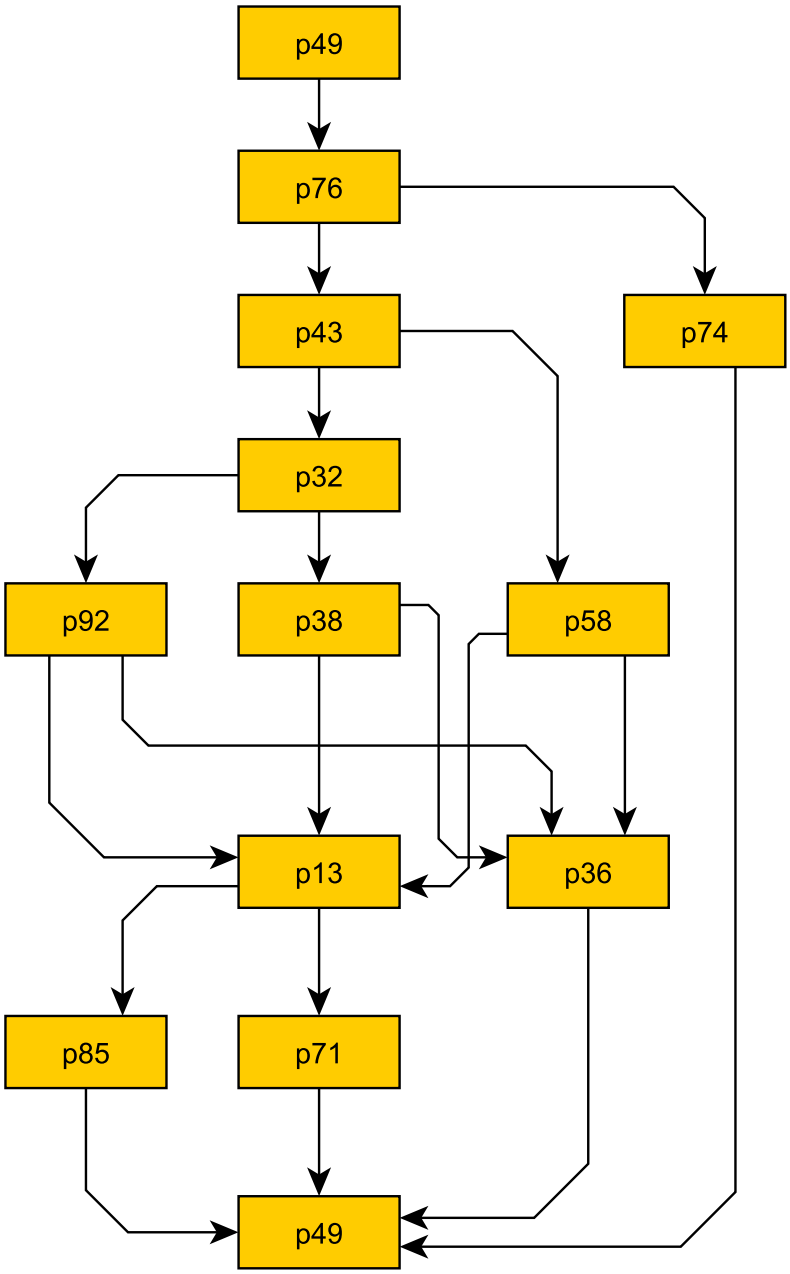
The base structure is as follows: we start, have a couple simple trap choices, then arrive at the hall of lights (page 49), visit some other segments but eventually pick the white light (page 25) which checks if we've gone through all the other segments (via checking the various items), and then presents one of the 16 endings depending on the player choices in the other colored light segments.



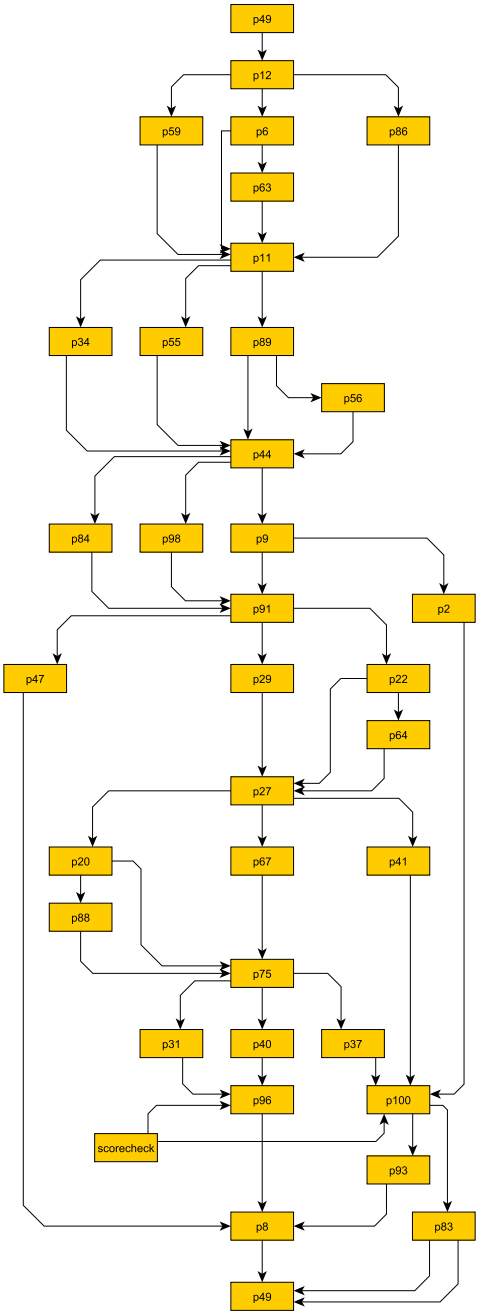
The green segment is relatively simple (in context of this gamebook). Note that the player can die here in a few ways (p30). In the end, the player can either be kill the beast or not (pages 15 and 16).



The pink segment represents a relatively complex dialogue. Basically you get to recommend various options, one being a good one and the rest are less so.



The red segment is again relatively simple, in the end you get to pick whether to be aggressive or not.

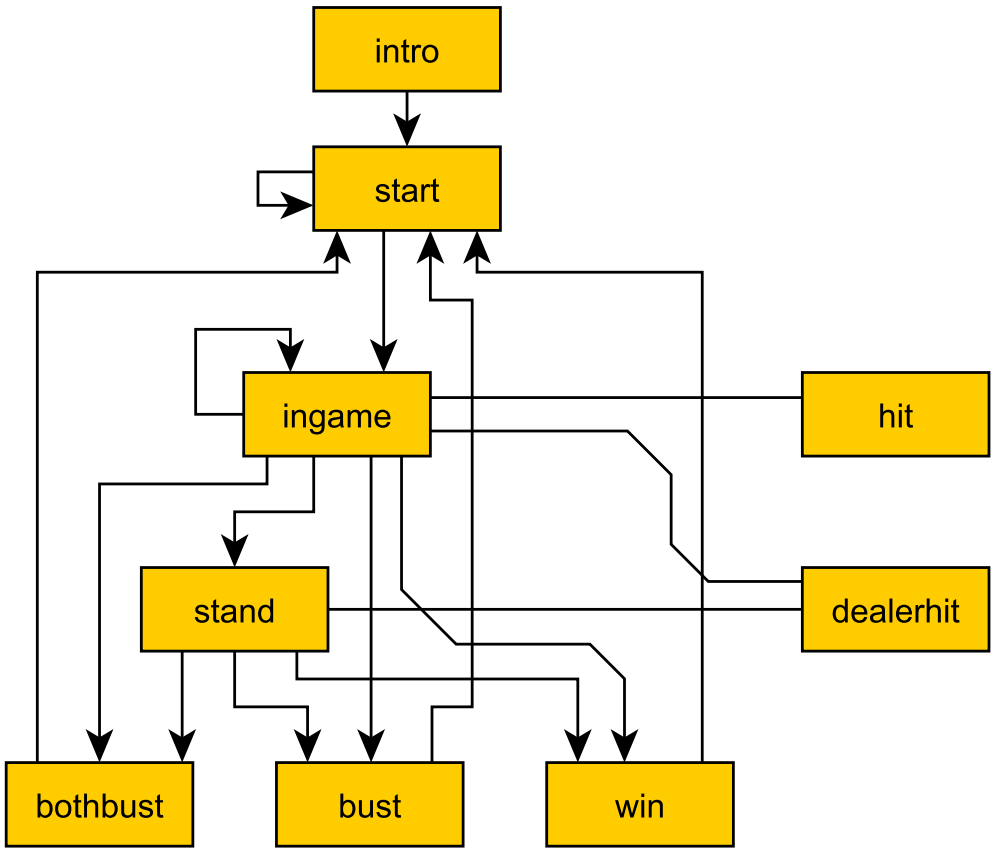


The yellow segment is super complicated: it's a game show. Not only can you answer right or wrong, or let the opponent answer instead, if you get the answers right, you can either take a point or take a card that has various results. And if either you or your opponent reaches 3 points, you hop to an ending.

This uses a subpage to check (and display) the scores and a couple numeric variables to keep the score.

The spectrum version is actually more difficult than the paper version because the player keeps track of the variables in the book version; the spectrum version hides them, so you don't know which result you're getting from your choices until the end.

2.8.6 Blackjack



The Blackjack example demonstrates some complex handling of numeric variables. The rules are a bit simplified as compared to actual blackjack; there's no actual card deck, and cards are valued from 0 to 10.

After the intro page the player arrives at the start page where they can choose how much to bet. Note that the game makes sure the player never bets more than they have.

Once a non-zero amount is bet, the player can proceed to the ingame page.

The ingame page does sub-page calls to hit and dealerhit, checks for busts, and asks if player wants to hit or stand, if neither player has gone over 21. If one (or both) players have gone over 21, the player is sent to win, bust, or bothbust pages.

In the stand page there are multiple calls to dealerhit, to make sure the dealer has reached 17 or more. Then the results are checked, and the player is sent to win, bust or bothbust pages. Bothbust is used in case of a tie.

In the bothbust page the player gets the bet back, and is sent back to start to bet again. In the bust page the bet is lost, and we check if player has run out of money, and if so, end the game. There's no separate game over page, we simply won't show the option to start over.

In the win page the player gets their money back twice, and we very, very carefully check if

the sum goes over maximum. If it does, we show the good ending text and end the game there. There's no separate game over page in this case either.

The hit and dealerhit pages generate 4 random flags, and then produce a value between 0 and 10 to add to either the player's, or the dealer's, score. Since there are 16 options, 3 of the options give 10 points and 4 of the options give 0 (9+4+3=16). (There's no looping in MuCho (yet, anyway), so we can't just ignore the zero results and randomize again).

And there you have it - something that approximates blackjack in MuCho.

CHAPTER 3

Compiling Stories

In order to run your story on the ZX Spectrum (or, more likely, an emulator thereof), you need to compile it. The compilation is performed in two steps. First, the MuCho compiler, mc.exe, is used to translate the story script into a binary form that can be used by the game engine. Then the game engine is combined with the compiled data to generate a .TAP file, which can be either loaded to an emulator, or played as audio to an actual spectrum.

The easiest way to make your own adventure is to modify one of the examples and use its build.bat file, but that won't help you if you make mistakes (and trust me, everybody makes mistakes).

3.1 MuCho Compiler

The MuCho compiler is meant to be run from the command line. It may be convenient to run it from a batch file directly from Windows, but then you'll miss any error messages it may generate.

3.1.1 Getting Help On Parameters

If you run mc.exe without parameters, it gives brief help:

```
C:\mucho>mc
MuCho compiler, by Jari Komppa http://iki.fi/sol/
mc <input> <output> [font image [divider image [selector image]]] [flags]
Optional flags:
-v   verbose (useful for debugging)
-q   quiet (minimal output)
```

The help means that at minimum, the compiler expects input and output file names as a parameter. In addition, you may include a custom font image, custom divider image, custom selector image, as well as one of the two optional flags.

For the font, divider and selector image, please see the advanced topics for custom fonts, dividers and selectors below.

3.1.2 Compiling a Story

If we run the complex example through mc, we get the following output:

```
MuCho compiler, by Jari Komppa http://iki.fi/sol/
Trainer data: 436 bytes
|---------1---------2---------3---------4---------5---------6----|
 the door is not as opressive You are in a small silver key glitt
 ers room with The room. ---- --- -- - to it. With A glows red. O
 utside light. sign neon window, light switch next lights turned
 on, for turns. it felt darkness. Lightning flashes window. open.
  door, single has street. down blue. looking window dark lightni
 ng's have key. Press Get Unlock Open Close Exit through Wait whi
 le end. played left on. remembered turn off lights.
Compressing..
start end zx7: 1015 ->  399 (39.310%), 0x5b0c

                dark.scr (09) zx7: 2593 ->  537 (20.710%), 0x5c9b
               light.scr (10) zx7: 2881 ->  703 (24.401%), 0x5eb4
                  beepfx.ihx zx7: 1701 ->  879 (51.675%), 0x6173

Token Hits Symbol
    0    7 "start"
    1    2 "end"
    2    3 "neon"
    3   12 "light"
    4    4 "lightning"
    5    6 "open"
    6    6 "key"
    7    4 "unlocked"

Token Hits Number
    0    2 "turns"

Token Hits Image
    0    1 "dark.scr"
    1    1 "light.scr"

Token Hits Code
    0    1 "beepfx.ihx"

Memory map:

         5         10        15        20        25      29
---------.---------|---------.---------|---------.--------
IIC.....................................................

Page data :   411 bytes
Image data:  1240 bytes
Code data :   879 bytes
Free      : 27865 bytes
Trainer   :   436 bytes (used to improve compression by "training" it)
```

The first bit after the compiler header is "trainer data". In order to maximize compression, the MuCho compiler figures out what are the most commonly used words in your story, and then builds most likely chains from these words. Sometimes this produces rather interesting beatnik poetry, but that's merely a side effect.

After this, the MuCho compiler tries to figure out which if your pages compress the best together, and packs them as tightly as possible. In this case you can see that the pages get compressed down to 39% of their original size, largely since pretty much all of the text can be found in the trainer.

Next the MuCho compiler packs any images and code blocks.

There you can see the two images, with number of character rows of their height; so the light.scr above is 80 pixels tall. Empty space tends to compress well, which explains the rather impressive compression ratio. This is followed by the beepfx code which gets compressed as well. It'll be uncompressed before it's executed during game play.

Following this are four tables. First is table of symbols found in the script, their internal representation token number, how many times the symbol was referenced in the script, and the symbol's name. If you ever typo a label, you'll probably find out on this table. Just look for symbols with only one hit.

Second table is for numeric variables. Same deal here.

The third table is for images. In this game, each image is only used once, but you can use images as many times as you wish, they'll only be stored once.

The fourth table contains the code blocks. You can re-use your code blocks as many times as you wish.

Next up is a simple visual representation of how much space you've used, and how much is left. Thanks to compression, this doesn't translate directly to number of words you can still fit in, but it's some kind of a guideline. Images take a lot of space compared to text, so in this example the text takes insignificant amount of space.

Finally there's the same data in numeric form.

If the compiler finds something to complain about, like a clear mistake in the script, it will abort compilation and tell you the reason.

3.1.3 Further Debugging

If you want to suppress even this minimal information, use the -q flag. If, however, you need more information, you can use the -v flag. It will output massive amount of information about your story. For the "complex" example the output is so large that it would make no sense to reproduce it here, but we'll take snippets and explain what they mean.

If you're getting an error message and don't know what it refers to, running with the -v flag may help, as it shows what the compiler has been doing when it throws the error.

```
Most frequent words in source material:
0. "the"        (19)
1. "You"        (5)
2. "in" (5)
3. "a"  (5)
4. "door"       (5)
5. "room"       (4)
6. "The"        (4)
7. "key"        (3)
8. "small"      (3)
```

First part of the verbose listing is the top list of the most frequent words in the story. The more often some word gets used, the better it will compress. Theoretically.

```
Making word chains:
the[4]->door[1]->is[1]->not[1]->as[1]->opressive # You[1]->are[1]->
in[1]->a[1]->small[2]->silver[2]->key[2]->glitters # room[1]->with #
```

Next up, the compiler builds word chains. The process takes the most frequent word, looks for what most frequently follows that (and is not already in the list), and so on and so forth. This is all done to improve compression. The # marks show where chains get broken (no suitable next word was found).

After the word chains there's the finished trainer data, which we already covered in the previous chapter.

```
Predicated section
    Opcode: HAS(light)
  Command buffer 'O' with 3 bytes payload (about 1 ops)
  lit 5: "  With the lights turned on, the room is not as opressive"
  lit 6: "as it felt in the darkness."
End of section
```

The next part of the output is more detailed listing of what the compiler finds in the script, including opcodes and lines of text.

```
          **************          row 0, live 1
          **************          row 1, live 1
          **************          row 2, live 1
          **************          row 3, live 1
          **************          row 4, live 1
          ***********  **         row 5, live 1
           **********  **         row 6, live 1
           **********  **         row 7, live 1
          **************          row 8, live 1
          **************          row 9, live 1
                                  row 10, live 0
                                  row 11, live 0
                                  row 12, live 0
                                  row 13, live 0
                                  row 14, live 0
```

The second part shows analysis of the images (if any). If the compiler complains about your images being too tall, but you're sure the bottom is cleared out, this representation may help you figure out what the compiler is seeing that you think it should not be seeing.

3.2 Mackarel Packager

TL;DR: Mackarel builds your .tap file.

Mackarel is a tool that takes compiled z80 code (in Intel hex, or .ihx) format, and turns them into ZX Spectrum .tap files, alongside additional data file. It also compresses everything (if possible) and handles loading screens, generating one if necessary.

To build the .tap file, the command line is a bit complicated, but most of the time you won't need to play around with it (unless you want to customize things).

If you're curious, you can run mackarel without parameters to get parameter help.

For example, to build the complex example, we would run:

```
mackarel patched.ihx complex.tap Complex –nosprestore –noei
–lowblock complex.dat 0x5b00
```

(That should all be in one line. Line was split to not blow up documentation. Don't panic).

To understand what this does, let's look at the parameters separately:

```
mackarel IHXFILE TAPFILE APPNAME –nosprestore –noei –lowblock GAMEDATA 0x5b00
```

The IHXFILE is either the crt0.ihx that hosts the game engine's compiled z80 code, or (like in this case) patched.ihx which the MuCho compiler produces when compiling the data. If you're not using custom fonts, selectors or dividers, it doesn't matter which one you use.

The TAPFILE is the output file name.

The APPNAME is the name the user first sees when running LOAD "". The APPNAME is also used in the generated loading screens.

The -nosprestore -noei parameters are mandatory for MuCho games. They basically tell Mackarel that we're not going to return to BASIC and we don't want interrupts enabled. (Removing these won't make it possible to return to BASIC).

The -lowblock GAMEDATA 0x5b00 parameters tell Mackarel that we want to load the GAMEDATA file to address 0x5b00.

Typically you'll want to change the TAPFILE, APPNAME and GAMEDATA file names when running Mackarel.

Unless something totally weird happens, Mackarel should always succeed in building MuCho .TAP files. However, it will give descriptive error messages if it notices something wrong (such as missing files).

When run, Mackarel gives out the following kind of report, which you can largely ignore:

```
Mackarel 2.1 by Jari Komppa, http://iki.fi/sol
Progname set to "Complex   "

"patched.ihx":
        Exec address : 57344 (0xe000)
        Image size   : 7501 bytes
        Compressed to: 4695 bytes (62.592%) by ZX7

Generated loading screen
        Image size   : 6912 bytes
        Compressed to: 134 bytes (1.939%) by RCS

"complex.dat":
        LowBlock addr: 23296 (0x5b00)
        LowBlock size: 29952 bytes
        Compressed to: 3023 bytes (10.093%) by ZX7

Boot exec address: 57527 (0xe0b7)
BASIC part       : 51 bytes
Screen unpacker  : 122 bytes
App bootstrap    : 123 bytes (69 codec, 54 rest)
"complex.tap" written: 8202 bytes
Estimated load time: 64 seconds (7 secs to loading screen).

Memory : 0       2       4       6       8       10      12      14      16
         |-------|-------|-------|-------|-------|-------|-------|-------|
On load: rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
         sssssssssssssssssssssssss..b.............................
         ........................................................
         ............................BLLLLLLLLLLLLCCCCCCCCCCCCCCCCCC

On boot: rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
         sssssssssssssssssssssssssLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
         LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
         LLLLLLLLLLLLLLL.............CCCCCCCCCCCCCCCCCCCCCCCCCCCC..

Key    : r)om s)creen b)asic L)ow block C)ode block
         H)igh block .)unused -)reserved
```

The report tells how well different parts (code, loading screen and game data) get compressed. If the game data is big enough, it won't get compressed - in this example most of the game data is actually empty space.

Following this, there's some statistics, followed by approximate memory maps on load and on game boot time. Not all of the memory can be filled by game code, since we need somewhere to decompress the game data on the fly, as well as space for some buffers (such as stack).

CHAPTER 4

Advanced Topics

And finally we're here, some advanced tidbits for additional modification of the game, as well as more information about error messages, should you ever see any.

4.1 Using Custom Code

You can bundle custom code blocks inside MuCho stories using the $C statement. Like with images, the same code block can be called several times but will only be stored once. The code will be loaded to address 0xd000 and must be 4096 bytes or smaller. The $C statements can also be predicated like any other normal statement.

```
$C beepfx.ihx 47 !alarm_disabled
```

The code can be either Intel hex or raw binary. In case of raw binary, the start address is expected to be 0xd000.

The second parameter is loaded to HL before calling. The code is expected to preserve registers. You can expect to have at least 100 bytes of stack available.

After the call, the memory is overwritten with something else, so don't expect any changes to stick.

4.1.1 Playing Sound Effects

MuCho comes with BeepFX 1.11 which can be used through the code loading system.

The sounds are played wherever the code is run, and some sounds are rather long, so if they are played at the middle of the page, the page drawing is paused while the sound is played.

The sound effects are basically most of the BeepFX v1.11 demo sounds, except those that take large amount of memory.

Id	Sound	Id	Sound
0	Shot 1	28	Item 1
1	Shot 2	29	Item 2
2	Jump 1	30	Item 3
3	Jump 2	31	Item 4
4	Pick	32	Item 5
5	Drop 1	33	Item 6
6	Drop 2	34	Switch 1
7	Grab 1	35	Switch 2
8	Grab 2	36	Power off
9	Fat beep 1	37	Score
10	Fat beep 2	38	Clang
11	Fat beep 3	39	Water tap
12	Harsh beep 1	40	Select 1
13	Harsh beep 2	41	Select 2
14	Harsh beep 3	42	Select 3
15	Hit 1	43	Select 4
16	Hit 2	44	Select 5
17	Hit 3	45	Select 6
18	Hit 4	46	Select 7
19	Jet burst	47	Alarm 1
20	Boom 1	48	Alarm 2
21	Boom 2	49	Alarm 3
22	Boom 3	50	Eat
23	Boom 4	51	Gulp
24	Boom 5	52	Roboblip
25	Boom 6	53	Nope
26	Boom 7	54	Uh-huh?
27	Boom 8	55	Old computer

4.2 Patching the Code

Please note that using custom fonts, dividers and selectors requires the MuCho compiler to patch the Z80 code. After patching the crt0.ihx with your custom graphics assets, the compiler outputs patched.ihx file.

So if you're not seeing your customizations when playing the game, you're probably not giving the generated patched.ihx file to Mackarel.

4.3 Custom Fonts

The fonts MuCho uses are proportional, meaning they are not fixed width. They are generated from bitmap image files which are 8 pixels wide and 752 pixels tall. Each of the 94 characters is stored within a 8x8 pixel block, on top of each other.

MuCho accepts most image formats, but lossless formats are preferred (i.e, while you CAN use .jpg, it's much better to use .png or .psd). When loaded, empty space to either side of the glyphs is removed.

A bunch of sample fonts are included in the kit, under "fonts" directory. There's also a photoshop .psd template.

The characters in the font are in ASCII order, starting from the space character:

```
  !"#$%&' ()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
 OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
```

In order to use a custom font, simply add the font file name after the other parameters to the MuCho compiler:

```
mc story.txt story.dat myfont.png
```

Note that you need to use the "patched.ihx" file instead of the "crt0.ihx" in order for your customizations to show up.

4.4 Custom Divider

The divider between the player's choices and the page text is a 8 pixel tall band where a pattern repeats every 8 pixels. I.e, the divider pattern is 8 by 8 pixels.

A bunch of sample dividers are included in the kit, under "dividers" directory.

In order to use a custom divider, simply add the font file name after the other parameters to the MuCho compiler:

```
mc story.txt story.dat myfont.png mydivider.png
```

Please note that you must also have a custom font in order to use a custom divider.

Also note that you need to use the “patched.ihx” file instead of the “crt0.ihx” in order for your customizations to show up.

4.5 Custom Selector

Selector is the player’s arrow with which the player knows which option to pick. It’s also a 8 by 8 pixel pattern. The way most fonts work, it’s best to bias the selector above the middle line, for example:

```
........
*...*...
**..**..
***.***.
**..**..
*...*...
........
........
```

A bunch of sample selectors are included in the kit, under “selectors” directory.

In order to use a custom selector, simply add the font file name after the other parameters to the MuCho compiler:

```
mc story.txt story.dat myfont.png mydivider.png myselector.png
```

Please note that you must also have a custom font and custom divider in order to use a custom selector.

Since both selectors and dividers are 8x8 patterns, you can swap between them.

Note that you need to use the “patched.ihx” file instead of the “crt0.ihx” in order for your customizations to show up.

4.6 Custom Loading Screen

Adding custom loading screens requires a modification to how you call Mackarel to generate the .TAP files.

```
mackarel IHXFILE TAPFILE APPNAME LOADINGSCREEN –nosprestore
–noei –lowblock GAMEDATA 0x5b00
```

(That should all be in one line. Line was split to not blow up documentation. Don’t panic).

Add the loading screen .scr file name after the APPNAME, and you’re set.

4.7 Error messages

Here’s a list of MuCho compiler error messages, and short explanations of what they mean.

```
Too many symbols, line ...
```

You've tried to use more symbols than we have room for. Congratulations! You must be doing something super complicated.

```
Max buffer size overrun. This should never happen, line ...
```

So yeah, there may be bugs. You can try to bug the author about this, maybe he'll figure it out.

```
Invalid character "..." found near line ...
```

Sorry, only ASCII characters are supported.

```
Room ... data too large; max 4096 bytes, has ... bytes
```

One room, in uncompressed form, may take a maximum of 4096 bytes. You'll probably need to reduce text. Massive amounts of text need to be split into separate pages.

```
Syntax error — too many operations on one statement, line ...
```

You've put a crazy amount of commands on one line. You should probably split them to several lines.

```
Parameter value out of range, line ...
```

You've used a value that's out of range for some command, like doing cls:5, where the only legal values are 0, 1 and 2.

```
Invalid GO parameter: symbol "..." is not a room, like ...
Invalid GOSUB parameter: symbol "..." is not a room, like ...
```

You've most likely typoed the room name. Or are trying to jump to another dimension. Not sure.

```
Syntax error (op=null), line ...
```

This probably should never happen. But who knows!

```
Syntax error (op starting with '...') "...", line ...
```

Labels may not start with a :, >, <, =, +, - or a !=. Probably a typo.

```
Syntax error (op with more than one instruction) "...", line ...
```

Again, probably a typo. The compiler found more than one operator (a : or a numeric one) in one command.

```
Syntax error: unknown operation "...", line ...
```

Likely a typo again, like using atr:7 instead of attr:7

```
Parse error near "..." ("..."), line ...
```

While trying to figure out numeric values, something went wrong. Check for typos.

```
Statement A must have exactly one line of printable text (... found)
(Multiple lines may be caused by word wrapping; see verbose output
to see what's going on), near line ...
```

Like it says, it found an $A line with more than one line of text, possibly caused by word wrapping. Write shorter.

```
Syntax error — statement P may not be included in statement A, line ...
```

The $P lines may only happen between $Q and $A lines, not after $A.

```
Syntax error — statement P may not include any operations, line ...
```

The $P lines are special in that way that they are not "actual" statements, and as such can't be predicated or run commands.

```
Syntax error — statement O may not be included in statement A, line ...
```

The $O lines may only happen between $Q and $A lines, not after $A.

```
Syntax error — statement I may not be included in statement A, line ...
```

The $I lines may only happen between $Q and $A lines, not after $A.

```
Syntax error — statement C may not be included in statement A, line ...
```

The $C lines may only happen between $Q and $A lines, not after $A.

```
Syntax error: unknown statement "...", line ...
```

Another probable typo. You've started a line with $, but followed up with a character that's not Q, A, I, P, C or O. Note that O is not the number zero, but uppercase o.

```
Error parsing numeric output on line ..., near column ...
```

You've probably made a typo, like "Player has < gold" instead of "Player has <> gold".

```
File "..." not found.
```

You've probably mistyped the file name.

```
syntax error: room id "..." used more than once, line ...
```

Rooms must have unique labels. Otherwise we have no idea which of the rooms you want to send the player to!

```
Image "..." not found.
```

You've probably mistyped the file name.

```
Image "..." wrong size (has to be 6912 bytes).
```

Only standard .scr files supported.

```
Image ... data too large; max 4096 bytes, has ... bytes (... lines)
```

This also probably should never happen, as we crop the images, but hey, better safe than sorrier.

```
Code "...\" not found
```

Guess what? You've probably mistyped the file name.

```
Code ... data too large; max 4096 bytes, has ... bytes
```

Sorry, we can only fit 4KB of code in there.

```
Code ... start address not 0xd000; ... found
```

Sorry, the code must be loaded at address 0xd000, no exceptions.

```
Can't open "..." for writing.
```

For some reason, MuCho can't open the destination file name. Is the directory read only?

```
Can't find data to patch in crt0.ihx!
```

The crt0.ihx is probably corrupted. Download a fresh copy.

```
crt0.ihx not found
```

The mc.exe can't find crt0.ihx in the current directory, nor in the directory where it resides. mc.exe is confused and sad. Don't let mc.exe be confused and sad.

```
unable to load "..."
```

You've probably typoed yet another file name.

```
Bad image dimensions; should be 8x752 (found ...x...)
```

We're rather picky about the font image size.

```
divider pattern image not 8x8 (found ...x...)
```

We're rather picky about the divider image size.

```
Selector pattern image not 8x8 (found ...x...)
```

We're rather picky about the selector image size.

```
ERROR Room "..." didn't get included
```

Our super-intelligent content packer forgot to include a room again. Contact the author. He may have sympathy for you.

```
Too many symbols in use (... > 1024)
```

You've managed to use too many symbols. You're probably making something super complicated.

```
Too many numeric variables in use (... > 32)
```

Yeah, so, this is a multiple choice adventure engine, not a general purpose processing system. You're probably doing something weird. Do tell us what it is.

```
Total output size too large (29952 bytes max, ... found)
```

You've been busy! You've hit the limit of how large a game in MuCho can be. Maybe trim some images?

```
Unknown parameter "..."
```

You've given a parameter to MuCho compiler that it doesn't recognize. Run without parameters for help.

```
Invalid parameter "..." (input, output, font, divider and
selector image files already defined)
```

Mucho compiler figures you've given it too many file names as a parameter.

```
Invalid parameters (run without params for help)
```

What are you reading this document for? Do what it says.