XVAN 2.6

-- introduction --

-- everything is a location, an object or a timer --

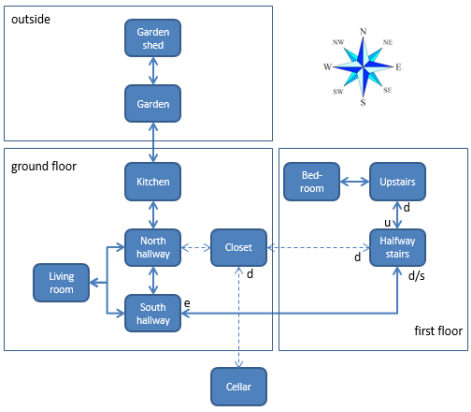


Table of Contents

Introduction

This document gives a general overview of the XVAN Interactive Fiction Authoring System. It describes the basic concepts in terms of locations, objects and timers and also addresses the vocabulary and actions. Examples are throughout the text.

A tutorial that describes how to create an XVAN story from scratch is available in a separate document.

Apart from the tutorial there are (of course) a Cloak of Darkness implementation and a medium-size sample story called Escape!. Escape! Was entered in the Sprint Thing 2019 competition and received badges for best NPC and best puzzles.

About XVAN…

XVAN is an interpreter-based[1] interactive fiction authoring system. It is a compiler, an interpreter and an authoring language.

The XVAN compiler takes a story source file written in the XVAN language and creates a binary game file. The interpreter reads the game file and lets the user play the story.

Technical

The XVAN *compiler* is a console application with a command line interface. The XVAN *interpreter* comes in 3 versions: a console version, a Glk version and as a back-end that is used with a separate GUI: IFI-XVAN.

The first version of XVAN was built in the 90's of the previous century, when internal computer memory was not sheer unlimited (my Atari has 1 MB of RAM which was a lot back then), I therefore built it to be low on memory requirements. A flag, for example, occupies 1 bit of internal memory and XVAN has a mechanism for swapping objects and locations in and out of internal memory.

XVAN is written in C. The console version of compiler and interpreter does not use any external libraries or interfaces other than the standard C include files.

The Glk version of the interpreter uses the Glk libraries. Glk is mainly intended for applications with text user interfaces (like interactive fiction). It gives more possibilities than a console or terminal window. More information on Glk can be found at http://www.eblong.com/zarf/glk/.

As of version 2.3.4, IFI-XVAN is available. This version of the interpreter uses the IFI (Interactive Fiction Interface) to connect to the Brahman GUI, which supports in-game graphics, graphical map display, clickable links etc. Both desktops and mobile devices are supported. More information on IFI and Brahman can be found at www.strandgames.com.

[1] As of version 2.4, XVAN also has a choice mode.
[2] o_nst, o_all and o_it are part of the Library. When the Library is used, these objects are already defined.

Platforms – Linux, Windows and macOS

XVAN started its life on my Atari 1040st back in the 90s. After the Atari went obsolete, I ported XVAN to Windows. I ran it on W95 and XP, and currently on W7 and W10.

I also made a version for Linux. I used the Linux Mint Rebecca distribution.

I made macOS versions on High Sierra.

As of version 2.3.4, there is IFI-XVAN that runs on Windows, Linux, and macOS.

Binaries and save files are portable between the operating systems. What you compile or save on one platform will work on the others.

Screen readers

My sample game Escape! Was tested on windows with NVDA by a vision-impaired person. I tested Escape! On linux Mint with the Orca screenreader and it seems to work.

Design

XVAN is designed around Locations, Objects and Timers. In XVAN story source files there is no main flow, it's just the collection of a number of locations, objects and timers in one or more text files. At playtime, the user input is offered to each location and object that is in scope and each decides whether and how to respond to the input.

Locations and objects have a number of artifacts available that they can use and manipulate:

- descriptions - text strings;
- flags - to remember yes/no situations;
- attributes - to store values;
- and triggers - tiny programs that are executed based on the player's input.

These will be explained in more detail in this document.

Locations, objects and Timers

Locations

The XVAN world model is made up of a number of locations that can be connected, so the player can travel through the world. Locations cannot contain other locations but they can contain objects (for objects, see below). Anything tangible that is not contained in something else is a location.

Example:

```
$LOCATION l_kitchen
 DESCRIPTIONS
   d_sys   “the kitchen”, “the dining room”
END_LOC
```

l_kitchen is the *location identifier* that is used in the source code to refer to the kitchen. d_sys is the so called *system description*, a list of friendly names for the location.

Objects

Objects are items in the world that can be manipulated. Objects can be moved around and may contain other objects. At the top level, an object is contained in a location.

Example:

```
$OBJECT o_lamp
 DESCRIPTIONS
   d_sys   “the lamp”, “the flashlight”, “the brass lantern”
CONTAINED on o_table
END_OBJ
```

o_lamp is the *object identifier* that is used in the source code to refer to the lamp. The *contained* line tells which object or location holds the lamp. In this example, the lamp object is *on* the table object. The table object is defined elsewhere in the source and is referred to by its *object identifier* here.

Predefined Objects

XVAN has five predefined objects:

- o_player;
- o_nst (**n**o **s**uch **t**hing);
- o_all;
- o_it;
- o_unbound.

These objects must be defined in the story file(s) by the author[2]. The compiler will check if they are present. If not, the compiler will throw an undefined object error message.

Timers

Timers are used to trigger events at certain moments. The moment is determined by a threshold value. At the end of each turn, all timers are updated according to the instructions stored with the timer. If the threshold value is reached, the timer will trigger an associated chain of actions. *Timer ids* have format m_timername.

An example of a timer is the battery timer for the lamp object:

```
m_battery
  value 100                 # initial power is for 100 moves
  step 1                    # change value by 1 each time
  direction down            # count down
  interval 1                # update this timer every 1 moves
  state go                  # this timer is active
  trigger_at 0              # when timer value reaches 0, execute the trigger
  execute o_lamp.t_empty    # trigger t_empty from object o_lamp must be executed
```

This timer will count down until it reaches zero and then executes the lamp's t_empty trigger.

XVAN has no default start routine when the interpreter starts , but a timer can be used to kick off the story. Create a timer with value 0 that triggers at value 1. After the interpreter starts the story, the timers get updated and the timer fires a series of actions that display the opening screen etc.

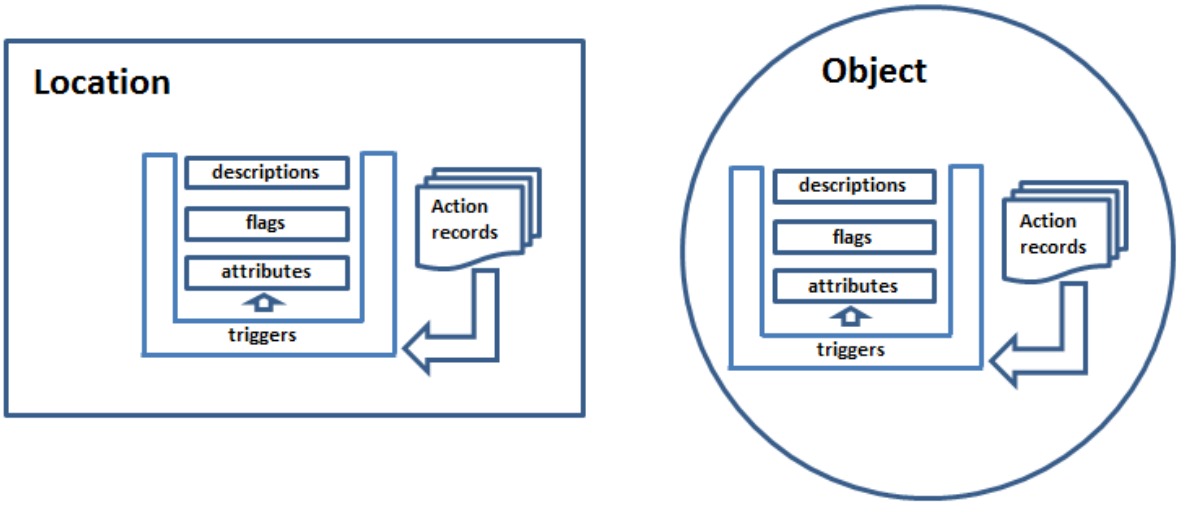
```
m_init
  value 0                   # initial value is 0
  step 1                    # change value with 1 each time
  direction up              # count up
  interval 1                # update this timer every 1 moves
  state go                  # this timer is active
  trigger_at 1              # when timer value reaches 1, execute the trigger
  execute o_player.t_init   # trigger t_init from object o_player must be executed
```

Trigger t_init will print the game's opening message, start the moves counter and perform other initial activities for the story.

[2] o_nst, o_all and o_it are part of the Library. When the Library is used, these objects are already defined.

Location and object Artifacts

In order to run the game, locations and objects use Descriptions, Flags, Attributes and Triggers.



Action records will be described later on in this document.

Descriptions

Descriptions are blocks of text. A *description identifier* is used to refer to the description in the source code. *Description identifiers* have format d_descriptionname.

Example:

```
d_exa_locked    "The old chest seems to be locked."
```

About text strings

So, as we are creating Interactive Fiction works, text will be an important part of the story. Text strings can be long, which makes the story's source code harder to read. XVAN has a number of mechanisms to format text strings so the story source is easier to read, without affecting the way how the string is printed.

Each string must end with a " or a /:

All carriage returns and spaces after a '/' will be ignored up to the next non-<cr>-or-space character.

Example:

```
d_long_descr        "This is a very very long description that /
                      goes on and on and on over several /
                      lines in the source file, but it will print /
                      as one line on the screen."

Will print as:
This is a very very long description that goes on and on and on over several lines in the source file,
but it will print as one line on the screen."
```

The same effect can be achieved by ending the string with an end quote and start a new string on the next line. Consecutive strings will be combined to 1 string.

Example:

```
d_long_descr        "This is a very very long description that"
                    " goes on and on and on over several"
                     " lines in the source file, but it will print"
                    " as one line on the screen."

Will print as:
This is a very very long description that goes on and on and on over several lines in the source file,
but it will print as one line on the screen.
```

String formatting characters:
The following string formatting characters are available:

- \n inserts a carriage return in the string;
- \t inserts a tab in the string;
- \" inserts a quote in the string (as a character, not an end quote);
- \\ inserts a \ in the string.

Example:

```
d_descr             "This is a string with a \n, a \t, a \" and a \\ in it."

Will print as:
This is a string with a
, a         , a " and a \ in it.
```

Flags

A flag can be either up (true) or down (false). Flag identifiers have format f_flagname.

Examples of situations where a flag would be used by a location or an object to keep track of things are:
- Lamp turned on or off;

- Box open or closed;
- Has the player been in the location before?

Flags are used to remember true/false situations.

Attributes

Attributes are used to store information that is dynamically generated during the game (i.e. not known at compile time) and that must be used at a later moment in the game.

Attributes can be used to remember different types of information:
- location names;
- object names;
- timers;
- other attributes;
- descriptions;
- numerical values;
- directions;
- words.

Attribute identifiers have format r_attributename.

An attribute can remember all of the above types, but it will keep track of the type that you put in. When an attribute is used as a parameter in a function, the interpreter will check if the attribute type matches with the function parameter.

Example:

```
r_some_item = o_lamp
move(r_some_item, o_player)    # moves the lamp into the player's inventory.

r_some_item = 10
move(r_some_item, o_player)    # will cause a runtime error
```

Both lines will be accepted by the compiler because at compile time it is unknown what will eventually be put in the attribute. It's the author's responsibility to prevent runtime errors.

Examples of the use of attributes are:
- tied_to: denote to which other object the object is tied to;
- dial_value: remember the number to which a dial has been set;
- teleport_location: remember the location where the other half of the teleport system has been placed.

Attributes and system description

As explained in the locations section, the system description is a location's or object's friendly name. The player uses it to refer to the location or object. By using an attribute in the system description, we can change the friendly name during runtime.

An example of a system description with an attribute

```
$OBJECT o_wall
 DESCRIPTIONS
  d_sys   "the [r_color] wall"
  d_exa  "The wall is painted [r_color]."
 CONTAINED in l_room
 ATTRIBUTES
  r_color = white   # initial color of the wall.
.....
END_OBJ
```

Initially, the wall is white. By changing the value of the r_color attribute, we can change the wall's system description.

Triggers

Triggers are small programs – written by the story author or in a library – that allow locations and objects to manipulate their flags and attributes and print information. Locations and objects have triggers depending on the tasks they must perform. *Trigger identifiers* have format t_triggername.

An example of a trigger defined with an object:

```
$OBJECT o_lamp
 DESCRIPTIONS
  d_sys   “the lamp”, “the flashlight”, “the brass lantern”
  d_exa  “An ancient lamp made of brass.”
 CONTAINED on o_table
 FLAGS
  f_lit = 1
 ATTRIBUTES
  r_power_left = 100
 TRIGGERS
   “light [o_lamp]”  -> t_light

t_light
  # the user has indicated he wants to light the lamp
  IF testflag(f_lit) THEN
    printcr("But the lamp is already lit!")
  ELSE
    setflag(f_lit)
    printcr("Ok, the lamp is now lit.")
  ENDIF
END_OBJ
```

Words like printcr, testflag and setflag are functions that perform specific tasks. For example, the function setflag() sets the indicated flag to value 1. XVAN has several functions that can be used to write the story.
There is a separate document with detailed descriptions of the functions that are available in XVAN. The functions document is quite large, but it is intended as reference when you need information on how a specific function works and what parameters it expects. It is not for reading cover to cover, you may just flip through it to see what’s available.

The same t_light trigger, but now combined with the earlier described timer m_battery (*blue italic* parts), to make the behavior of the lamp object a bit more realistic:

```
t_light
  # the user has indicated he wants to light the lamp
  IF testflag(f_lit) THEN
     printcr("But the lamp is already lit!")
  ELSE
     IF equal(m_battery, 0) THEN
       printcr("Nothing happens. The battery must be empty.")
     ELSE
       setflag(f_lit)
       starttimer(m_battery)  # start draining the battery
       printcr("Ok, the lamp is now lit")
     ENDIF
ENDIF
  agree()
```

Common Descriptions, Flags, Attributes and Triggers

There are descriptions, flags, attributes and triggers for which it is obvious that they will be necessary for all locations and objects. For example, a description d_exa that contains the text to be printed when an object is examined. Or a trigger t_look that allows an object or location to print a description about itself when the user types 'look'.
Such common artifacts need only be defined once in a dedicated section of the story file. The compiler will then add them to each object and location. The definition is a default definition for all locations and objects. When necessary a common definition can be simply redefined by defining it again locally in a location or object. This *local* definition will then replace the *common* definition.

Wildcards with (common) triggers and verbs

Locations and objects use triggers to respond to certain events. The actual data for the event (which objects, which location, which direction, etc) will usually be available during play time. XVAN has the possibility to use wildcards in trigger code that will be filled in with the actual data when the trigger is executed.

Following wildcards are available:

- l_location the player object's current location.
- o_subject the object of the current move (e.g. a chest).
- o_specifier additional object that is involved in the move (e.g. a key to open the chest).
- o_actor the object or location that performs the current action.
- %action the current action.
- %prepos the preposition in the current move.
- %this the location or object that executes the current trigger.
- %dir direction in the current move.
- %value value in the current move.
- %ord ordinal value in the current move.
- %the to be used in strings, will print "the" if the following object or location has an article defined with it.

-%a to be used in strings, will print "a" or "an" if the following object or location has an article defined with it.

The %-character is to allow XVAN to tell the wildcards from normal vocabulary words when used as function parameters. When used in a string, wildcards must be put in [] and the '%'-sign may be omitted.

Examples:
move(o_subject, %this)
"There is [a] [this] here."

The following function example is not valid:
move(o_subject, this), this must be written as move(o_subject, %this).

An example of using wildcards

```
t_wildcard_example
# This trigger illustrates the use of wildcards
# This trigger is fired by all user input with syntax “ask [o_fred] about [o_specifier]”

# Suppose the user has entered the command ‘Ask Fred about the dial, then
# during execution of the trigger [o_specifier] will be substituted with ‘dial’.
# This makes that we only need to define the trigger once to work for all objects.

 IF equal (o_specifier, o_dial) THEN
    printcr(“The keys with 3, 5, 6 and 9 are a bit more worn than the others.“)
 ELSE
    printcr("I don't know anything about the [o_specifier]!")
 ENDIF
agree()    # let other objects react
```

Predefined descriptions, flags, attributes and triggers

XVAN has a number of predefined descriptions, flags, attributes and triggers. They are used internally by the interpreter but are also available to the story author.

Following description is predefined:
d_sys: system description. The friendly name of a location or an object. Used by the interpreter to match user input to objects and locations. Default value is "".

Following flags are predefined:

f_lit: tells whether an object or location is lit. Default 0;

f_opaque: tells whether an object or location is opaque. Default 0;

f_bypass: tells the interpreter to bypass the visibility check for objects3. Default 0;

f_hidden: tells whether an object is hidden. Default 0.

f_swap: in case an object or location has more than 1 system description the interpreter will use the description that the player used to when printing the location or object name. Default 0.

f_json_mode: used by the interpreter for the IFI version of XVAN.

Following attributes arepredefined:

r_preposition: default "in".

r_it: used by the interpreter to remember what “it” in user input refers to.

r_unbound: used by the interpreter to handle user input that cannot be linked to real life objects (see section on unbound objects further in this document).

Following triggers are predefined:

t_entrance: Common trigger. No action by default;

t_exit: Common trigger. No action by default.

t_default: Common trigger. When redefined within an object or location, it will be called when no other triggers are called for this object or location.

t_choice: Trigger that must be executed when in choice mode (see section on choice mode further in this document).

t_unbound: used to handle user input that cannot be linked to real life objects (see section on unbound objects further in this document).

Vocabulary

To interact with the player, XVAN needs a vocabulary. It is best to store XVAN’s vocabulary of words in one or more separate files, so it can be used with multiple stories by including the vocabulary files in the story file. The idea is that this vocabulary file grows with each new story that you write so XVAN will learn more words with each new story and existing words can be reused.

Following sections are distinguished in the vocabulary:

- verbs;
- nouns;
- adjectives;
- adverbs;
- articles;
- prepositions;
- directions;
- conjunctions.

[3] for example, in case the player wants to take off his cloak in the dark, he must be able to refer to it and not get a "you don't see a cloak here" message.

A word section starts with the section keyword, followed by the words. E.g:

```
$PREPOSITIONS
at, behind, in, on, under
```

A word section – and other sections as well – can be in any file, as long as it starts with the section keyword.

XVAN has an understanding of the syntax of English (and Dutch) sentences so it is able to evaluate the user's input.

Multiple types
XVAN allows words to be of more than one type. For example 'light' can be:
- a noun (as in lamp);
- a verb (as in ignite something);
- an adjective (as in not heavy).

When parsing user input, the parser starts with the first available type for each word. In case a syntax error is encountered during the parsing process, XVAN has a trial-and-error mechanism to deduce a valid input. It will go back one step and see if the conflicting word has another type that might fit. So, for example, the sentence "light light light" is valid user input (meaning "ignite the not so heavy lamp").

Verbs

For verbs extra functionality can be coded. Verbs may have their own actions, comparable to triggers. These actions will be used as a default scenario. In case none of the locations or objects responded to the user input, the verb's actions will be searched for a match.
It is good practice to not refer to specific locations or objects but use wildcards (this, actor, subject, specifier, etc) to ensure that the vocabulary remains independent of specific stories.

Verb Prologue

A verb can have a prologue. Before the user input is offered to the locations and objects, the interpreter checks if there is a prologue for the verb. If so, the prologue will be executed. A possible result of the prologue is that the input will not be offered to the locations and objects.

Example of a prologue for the verb 'look':

```
PROLOGUE
 If not(islit(o_player)) then
    printcr("It is pitch black.")
    disagree()   # stop further processing user input
 ELSE
    agree()      # let locations and objects react
```

Verb Epilogue

A verb's epilogue (if present) will be executed after all locations and objects have had the opportunity to respond to the user's input.

Example of an epilogue for the verb ‘close’:

```
EPILOGUE
 # they may have closed the object with the light source in it
 IF not(islit(o_actor)) THEN
    printcr(“It is now pitch black.”)
 ENDIF
 agree()
```

Verb Default

In case none of the locations and objects respond to the user input and there are no default actions defined with the verb, the verb may have a *default* default action defined. The purpose of this default is to have a sort of last resort response available so the user doesn’t get the feeling the author forgot to implement something.

In case the verb does not have a default section, the interpreter will not print a response.

Example of default section for the verb ‘charge’:

```
DEFAULT
 printcr(“I only understood you as far as wanting to charge something.”)
 agree()
```

Here is a full example of the verb ‘Inventory’ with Prologue, Epilogue and Default sections.

```
$verb inventory SYNONYM i      # 'i' may be used as a synonym for inventory
 PROLOGUE
  printcr(“You are carrying:”)
  indent(2)                          # from now on, the indent() function will print 2 white spaces
  agree()

# after the prologue and before the epilogue, all locations and objects will respond to the command

EPILOGUE
 indent (-2)                    # remove indent

DEFAULT
  # in case there were no responses, the default section will be executed.
  printcr(“Nothing, you are empty-handed.”).
ENDVERB
```

The location's and object's responses to the "inventory" command could be coded in the story file as the following common trigger:

```
t_i
  IF owns(o_player, %this) THEN
      printcr("[a] [this]")
  ELSE                      # player is not carrying this object
     nomatch()              # Return nomatch() to inform the interpreter to forget this object had
                            # a matching action record. Otherwise the verb default code will not
                            # be executed.
```

The above example shows that there's no loop or similar construction to list the inventory. The prologue prints an opening message and then each location and object will decide to contribute or not. If nobody responds, the verb default section will respond.

There is some hierarchy, however. The location will always be the first to receive the user input and containing objects go before their contained objects.

The Library

XVAN wants to give full control to the author. But if you don't want to create your own framework and build everything from scratch, there is the Library that contains predefined verbs, flags, triggers and vocabulary words to handle most common tasks.

The Library is described in separate documents.

Language versions

As of version 2.3.2, XVAN also supports the Dutch language (next to English). There are two keywords that you can use in story files to select the language:

XVAN_LANGUAGE tells the compiler the language of the programming interface. If set to Dutch, all XVAN keywords like if, then, else, prologue and functions like IsLit(), CanSee(), Owner() are replaced by Dutch translations. Also, the error messages are translated.

STORY_LANGUAGE tells the compiler and interpreter in which language the story must be played. The story language is stored in the compiled file, so the interpreter knows which state machine to select to parse input text.

An example:

```
# XVAN language sample

TITLE             "Sample to show language selection"
VERSION           "1.0"

XVAN_LANGUAGE     english         # may also be eng or engels
STORY_LANGUAGE    nederlands      # may also be nl or dutch
```

Why is there an option to change the XVAN programming language? I would not use it myself but I think it may be useful for educational purposes. Suppose you're a teacher and you want to teach your class about programming concepts. It might be handy if the keywords and messages are in their native language. And in some countries it's a best practice to just translate everything :-)

How does XVAN actually work?

XVAN consists of a compiler and an interpreter (and a language definition).

Compiler

The compiler needs an input file with the story (locations, objects, timers etc) and the vocabulary (verbs, nouns, adjectives etc). The compiler supports multiple files, but one file must be leading. It is advised to use separate files for things that can be reused with other games. E.g, create a separate vocabulary file with generic verb definitions that can be reused with new stories. Game specific code is put in the main story file. In earlier versions from XVAN the separate vocabulary file was mandatory.
From the input file(s), the compiler creates an output file that contains the complete story in binary format. The output file can be played using the interpreter.

Interpreter

The interpreter reads the output file created by the compiler and offers the user a prompt ('>') to enter commands to play the story. For each turn, the user will enter a sentence at the command line. E.g. "Open the mailbox", "Take the leaflet", etc. This sentence will eventually cause locations and objects to execute certain triggers to progress the game.

How do locations and objects know which trigger corresponds to the user's input? This is where action records come in.

Action records

When the story was compiled and the output file was created, the compiler created and stored so called action records for the objects and locations.

An action record is a standard set of data with information about:
- the actor (who must do it)
- the action (what must be done)
- the subject (that the action must be performed on)
- the specifier (to perform the action with)
- the trigger that must be executed for this action record
-etc, etc

The action record is created from the information that the author entered in the story file for a specific location or object..

Such information is of the format:

"unlock [o_chest] with [o_key]" -> t_unlock

Basically, what this says is: if the player enters something like "unlock the old wooden chest with the rusty key", you must execute the t_unlock trigger.

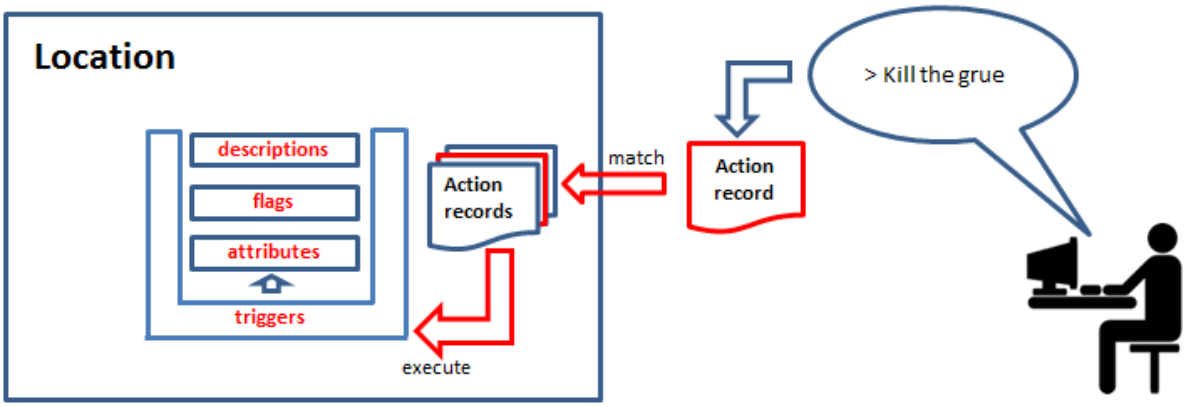
In the above example the compiler will create an action record in the output file with:

```
action      : unlock;
subject     : o_chest;
specifier   : o_key;
preposition : with
```

and link it to the trigger t_unlock.

This action record will be stored with the location or object.

When playing the game, the interpreter 'disassembles' (parses) the user's input line and creates an action record from it. The interpreter will offer the action record to each location and object. The locations and objects will search through their own set of action records and will try to find a matching record. If a matching record is found, this record will contain information on the trigger that must be executed.
With this mechanism, XVAN can turn the user's input into a series of triggers that will be executed by locations and objects.



The above is a simplified description of the mechanism; it explains the basics, but there's more to it. To elaborate a little bit, the interpreter doesn't actually offer the action record to all locations and objects from the story, only to those that qualify (are in scope). An object may tell the interpreter to stop offering the action record to other objects, an object or location may tell the interpreter to ignore that it had a matching action record so a default trigger will be called, etc, etc.

Verbs also have action records. In case none of the locations or objects respond to an action record, the interpreter will check the applicable verb whether it has a matching action record with default code to handle the user input in a more general way. It is advised to keep the verb code as general as possible (default messages), so a verb can be used with different story files.

Evaluation priority

From the above examples, it shows that the interpreter imposes priorities in executing the action record created from the user input. A graphical representation of the evaluation priority mechanism is depicted in the Annex at the end of this document.

In a bullet list, the evaluation priority is as follows:

The interpreter will:

1. create an action record from the user input;
2. in case of ambiguities when mapping user input to objects, use the disambiguation rules, if any are defined (see next section);
3. check if the verb has a Prologue and execute it
4. by default, offer the action record to the current location and all its contained objects. Depending on the current scope, the action record can be offered to other locations and objects.
5. if none of locations/objects reply, offer the action record to the verb to execute it;
6. if the verb does not reply, check if the verb has a DEFAULT section and execute it;
7. if the verb has no DEFAULT section, print nothing;
8. check if the verb has an Epilogue and execute it.

Depending on the outcome of a step execution of further steps may be cancelled.

Ambiguities in user input

Sometimes the user input will contain insufficient information for the parser to determine which object(s) the user is referring to: an ambiguous command. In such a case, the interpreter will ask for additional information so it can map the command to the right objects.

Consider this:

```
> take cube
Which cube do you mean? The red cube, the green cube or the blue cube?

> red
Red cube: taken
```

Nothing wrong with that. But now consider the following:

```
> look
There is a red cube here.
There is a green cube here.

> i
You are holding:
  a blue cube.
  a yellow cube.
```

```
 a lamp (providing light)

> drop cube
Which cube do you mean? The red cube, the green cube, the blue cube or the yellow cube?
```

This makes less sense. The interpreter also asks about the red and green cubes although the player isn't holding them. We want some way to tell the interpreter that an item must be held before it can be dropped. This is where disambiguation rules come in.

Disambiguation rules are defined in the vocabulary file in the sections with verb default code.

```
VERB drop
….
"[o_actor], drop [o_subject]"
 DISAMBIGUATION_RULES
    if owns(o_actor, o_subject) then score(5)
 END_RULES
 if not(owns(o_actor, o_subject)) then
     printcr("But [the] [o_actor] is not holding [the] [o_subject].")
 else
     move(o_subject, owner(owner(o_subject))
 endif
…..
ENDVERB
```

In case the parser finds it has more than 1 candidate for an actor, a subject or a specifier, it will query the verb to see if it has disambiguation rules.
If there are disambiguation rules, the parser will create one action record for each possible actor, subject and specifier. In our cube example, 4 action records will be created with different subjects. Each action record is then ran by the disambiguation rules. An action record can earn points if it complies with the rules. In the end, the action record with the most points wins.
The number of created action records can be large. Suppose we have 2 possibilities for the actor, 3 for the subject and 3 for the specifier then the parser will create 2 x 3 x 3 = 18 action records.

In the example, the disambiguation rules tell the parser that a situation where the actor holds the subject gets 5 extra points. If there are more winners, the interpreter will ask the user, but only for the winners (in our example only for the blue and yellow cubes).

It is important to realize that the disambiguation rules will only be executed in case the interpreter cannot decide which item the user means. Again, in our example, if the user would type 'drop red cube' although he is not holding it, the interpreter would not consult the disambiguation rules,

because it can map the text 'red cube' to one unique object. That's why we still need the code after the END_RULES keyword in the verb default code.

Choice mode

As of version 2.4, XVAN has a choice mode. In choice mode, the player is presented with a number of options to choose one from. This contrary to interpreter mode, where the player must enter sentences. As an example, you may use choice mode to limit the number of interactions with an npc.

Choice mode uses the following functions and artifacts:

- Function playmode();
- Function addchoice();
- Trigger t_choice.

At the end of each turn the interpreter checks the playmode. If the playmode is CHOICE, the interpreter calls trigger t_choice for all objects and locations in scope. Typically, in their t_choice trigger the objects will add choices relevant in the current situation by calling function addchoice(). Next, the interpreter prints all choices and ask the player to select one.

An example:

```
LOCATION l_room
# in this room we only allow choices
….
    t_entrance
        print(“Entering the room…\n”)
        playmode(choice)
    t_exit
        print(“Leaving …\n”)
        playmode(interpreter)
 t_choice
    addchoice(“ Check out this room”, “look”)   #execute “look” if they select this
    addchoice(“Leave”, “go north”)              # execute “go north” if they select this
….
END_LOC
```

The output would be like this:

```
> s
Entering the room…

What do you want to do next?

  1. Check out this room
  2. Leave

Enter a number from 1 to 2:

> 2
Leaving …
```

This example only has one location. If there are objects in the locations they may have their own t_choice trigger and add options to the choice menu.

Plurality

From version 2.5, XVAN supports plurality. Plurality implementation is based on the following starting points:

- By default, plural words end with “s” or “es” (Dutch: “n” or “en”). We call this regular plurality;
- Regular plurality words are not in the dictionary, unless they correspond to an object in the game that must be referable as one entity;
- When the parser encounters an unknown word in user input it will check for a plural by stripping off “e” or “es” from the end. If the remaining word is a known word, the parser will continue, but mark the word as plural, so it can be mapped to more than one object.
- The matched objects will be ran by the disambiguation rules to rule out less likely objects;
- The parser will create multiple action records, one for each object that matches the plural.
- For irregular plurality, the plural word must be in the dictionary after the singular form. E.G.

```
$NOUNS
…, ox PLURAL oxen SYNONYM cattle, …
```

This defines oxen and cattle as plural for ox. All actions released on the oxen/cattle will be split up into actions for each individual ox. E.g, in case there are three oxen, “examine oxen” would be similar to “examine white ox, grey ox, black ox”.

An example with our cubes again:

```
> look
There is a red cube here.
There is a green cube here.

> i
You are holding:
  a blue cube.
  a yellow cube.
```

```
 a lamp (providing light)

> get cubes
red cube: taken
green cube: taken

> drop cubes
blue cube: dropped
yellow cube: dropped
red cube: dropped
green cube: dropped
```

The disambiguation rules exclude the blue and yellow cubes from the ‘get’ action.

Unbound objects

The interpreter will always try to bind nouns and adjectives from the user input to objects and locations in the story. If it cannot find a matching object or location it will print a “You cannot see that here…” message.

But what if we have a valid expression whose noun cannot be bound to an object? E.g “Get some rest”, “hit the road” (go on your way) or the like?

We don’t want something like:

```
> get rest
you don’t see that here.

>
```

So, in order to deal with this we have:

- object o_unbound, with
- attribute r_unbound and
- trigger t_unbound.

Whenever the interpreter finds a noun that it cannot bind to an object, it will store the noun (actually its word id) in attribute r_unbound and then calls t_unbound. In the t_unbound trigger for object o_unbound we can check the noun and decide whether we need to do something.

An example

```
  t_unbound
    if equal(%action, hit) and equal(r_unbound, road) and equal(o_spec, %none) then
      printcr("Please use compass directions to move.")
      agree()
    endif
    if equal(%action, get) and equal(r_unbound, rest) and equal(o_spec, %none) then
      printcr("Resting costs you 2 moves.")
      wait(2)
      agree()
    endif
    disagree()
>
```

Finally

This ends the explanation of XVAN's basic concepts. In the Syntax document, examples of a larger location, object and verb are included, as well as the layouts for story files.
A detailed description of all of XVAN's available functions is available in a separate document.

Annex – XVAN interpreter flow