About this tutorial

This tutorial teaches:

- setting up the basic things for a story;
- vocabulary;
- working with objects, locations and timers.

Basic things

XVAN was developed based on the idea that the story author must have flexibility in writing his stories. This means very little is predefined, so the author initially must invest some time to take care of some basic things (moving around, inventory, save, restore, etc). But this needs only be done once, because the basics can be reused in other stories.
Note: the basic things are also covered in the XVAN Library, but as this is a tutorial the Library is not used here

Vocabulary

The vocabulary is one of the basic things to set up, but it's important enough to have its own section in the tutorial. XVAN comes with one predefined word: "go". All other words must be defined by the author. The vocabulary is meant to be story independent so it can be reused and grow with next stories. It is advised to have the vocabulary in a dedicated file, separate from the story file. The Starter Kit comes with a predefined vocabulary.

Object, locations and timers

As described in the introduction document, an XVAN story file is a collection of locations, objects and timers. The tutorial shows how to define these and how to model the behavior.

Structure of this tutorial

In this tutorial we will write a sample story through a set of exercises. At the start and end of each part we will have predefined vocabulary and story files. Of course you can create your own files, but for the tutorial we need predefined file sets that match with the tutorial text.

Note: the XVAN compiler requires unformatted text files. Copy/pasting source code from other file types may introduce invisible control characters that generate compiler errors.

Part 1 – the vocabulary

So, we are going to think about the vocabulary before we designed our story? Yes, we want to be able to reuse the vocabulary for other games so there's no problem making a first version without having a story yet.
We start with file "part 1 start.lib". The sections for nouns, adjectives, prepositions, adverbs, articles, question words and conjunctions are straightforward: the more words you put here, the more words the interpreter will know. A word can be in more than 1 section. E.g. 'orange' can be an adjective

(color) but also a noun (fruit). The parser will combine word types in a sentence until it has a valid English syntax in the user input (or not).
For the remainder of this section we will focus on verbs and directions.

Verbs (and default code)

A verb can be defined as a single word:
$VERB take ENDVERB

But a verb can also be used as a last resort when a user command cannot be handled by a location or an object. As described in the XVAN introduction document, the user input is offered to all locations and objects in scope. If none of them can process the user input, we can fall back to the verb default code to respond to the message. Likewise, actions that will be always be handled in the same way (e.g. score) can be coded in the verb defaults.

For our tutorial game we want at least the following verbs:

- look or l;
- examine or x;
- get or take;
- drop;
- inventory or I;
- and the predefined verb 'go'.

Note: technically speaking, inventory is not a verb, but by defining it as one, the user can enter it as a single word on the command line.

look

With verb default code we are only interested in default replies, when none of the locations or objects have responded (which is very unlikely for 'look', by the way).

So, what kind of messages would do as default? The user could have entered different look commands, like "look", "look at something", "look in/on/under/… something". The verb syntax allows to define responses for all these situations. Responses could be "You see nothing special.", "You see nothing special about the something." and "There's nothing in/on/under/… the something."

In XVAN code:

```
$VERB look
  "look"
    printcr("You see nothing special.")
  "look at [o_subject]"
    printcr("You see nothing special about [the] [o_subject].")
  "look [prepos] [o_subject]"
    printcr("There's nothing [prepos] [the] [o_subject].")
ENDVER
```

A little bit about [o_subject], [prepos] and [the]. These are called wildcards. During play, they are replaced with the user input. If the user types "look behind the couch", [prepos] will be replaced by 'behind' and [o_subject] by 'couch'. [the] means that the interpreter must print the article in case there is one defined with the subject. There also is [a] which will print either "a" or "an".

But ‘at’ also is a preposition. Because the “look at [o_subject]” line is defined before the “look [prepos] [o_subject]” line, this line will be executed first. If the lines were reversed, “look at” would never be executed because the interpreter stops after a matching line is found.

But we’re not there yet. What if the user enters a look-sentence with a syntax that we didn’t cover here?, like for example “look carefully” (carefully being an adverb)? We cannot cover everything, so we have the DEFAULT section. This usually prints a very generic response that fits all user input like “I only understood you as far as wanting to look.”

Finally, suppose we’re in the dark and can’t see anything. Than all off the above is useless. We need some sort of initial test possibility, before anything is done with the verb. This is the verb prologue. The prologue is executed right after the user input is parsed, even before the input is handed over to the objects or locations.

Our verb ‘look’ with DEFAULT and PROLOGUE sections:

```
$VERB look
  PROLOGUE
    if not(islit(o_player)) then
      printcr(“It is pitch black.”)
      disagree()
    else
      agree()
  “look”
    printcr(“You see nothing special.”)
  “look at [o_subject]”
    printcr(“You see nothing special about [the] [o_subject].”)
  “look [prepos] [o_subject]”
    printcr(“There’s nothing [prepos] [the] [o_subject].”)
  DEFAULT
    printcr(“I only understood you as far as wanting to look.”)
ENDVERB
```

The function disagree() tells the interpreter to stop further execution of the user input. Likewise, agree() tells to continue. By default, the compiler will add agree() at the end of code.

examine

The verb examine differs from look in the way that it always requires a subject. If the user only enters ‘examine’, we want to ask for additional information about the subject to be examined. XVAN has the getsubject() function to do this.

In XVAN code:

```
$VERB examine SYNONYM x
  “examine”
    printcr(“What do you want to examine?”)
    getsubject()
  “examine [o_subject]”
    printcr(“You see nothing special about [the] [o_subject].”)
  DEFAULT
    printcr(“I only understood you as far as wanting to examine something.”)
ENDVERB
```

The SYNONYM keyword tells the interpreter that ‘examine’ and ‘x’ are identical.

There's no prologue here to check if we are in the dark. Why? The parser will try to map the user input to a subject to be examined. In this process it will take into account whether the subject is visible to the player. So, when the user refers to a subject he cannot see, the parser will generate a "you don't see that here" message. With the 'look' verb, there was no subject so we had to check it ourselves.

get

The verb 'get' is similar to 'examine', but we can make it a bit more complicated. We have to make a design decision for "get something":

- Either we say that all get actions should be defined by the objects and the verb will print a default message like "you can't get the [o_subject]."
  or
- We say that default behavior for "get something" is that it will be picked up and that any exceptions must be handled by the objects.

To make things a bit more interesting, let's implement the second option, so we give more intelligence to the verb.

First of all, before each execution of a get command, we must make sure the subject can be picked up. This can be implemented in a prologue:

```
PROLOGUE
  If equal(o_subject, %none) then
    agree()
  else
    if not(testflag(o_subject.f_takeable)) then
      printcr("[the] [o_subject] is not something that can be taken.")
      disagree()
  endif
```

Some new things here.

In the else part we test whether the subject can actually be picked up. We do this by testing the flag f_takeable. If it is set, the subject can be carried. If not, the verb will tell the interpreter to stop by calling the disagree() function.

As we will see in the story file, f_takeable is a common flag, each object and location has it. Common as opposed to local flags that are specific to an object or location. Testing a wildcard object (like o_subject) or location for a local flag will generate a compiler error, because it is not guaranteed at compile time that the subject will indeed have the local flag.

In the if part we test if the user actually entered a subject with his command. Suppose he only entered "take", then testing for flag o_subject.f_takeable would not be possible. We know that the user did not enter a subject when the o_subject wildcard has value %none.

Our get verb

```
$VERB get SYNONYM take SYNONYM grab
  PROLOGUE
    If equal(o_subject, %none) then
      agree()
```

```
  else
    if not(testflag(o_subject.f_takeable)) then
      printcr("[the] [o_subject] is not something that can be taken.")
      disagree()
  endif
 "get"
  printcr("What do you want to get?")
  getsubject()
 "get [o_subject]"
  If not(owns(o_actor, o_subject)) then
    move(o_subject, o_actor)
    setflag(o_subject.f_bypass)
    printcr("[o_subject]: taken.")
  else
    printcr("But [o_actor] already [o_actor.r_have] [the] [o_subject].")
 DEFAULT
  printcr("I only understood you as far as wanting to get something.")
END_VERB
```

The verb will generate an error response if the actor already carries the subject.

But what's with this o_actor thing? Why not use o_player? Well, there's a good explanation for that. Usually, the player will be the actor, but we can also give commands to an npc (non-player character). After parsing the user input, the interpreter will set o_actor to the entity who must perform the command.

Examples:

- "get lamp", o_actor = o_player
- "Fred, get lamp" , o_actor = o_fred

We don't know beforehand who the actor will be and we want the verb default code to work with all actors. Hence the [o_actor]. As a side effect we must also define an r_have attribute to store the actor's correct conjugation for the verb "to have" for printing (see part 2 about attributes).

But you don't use o_actor with the verbs Look and Examine? That's right, in my opinion the you-replies will do, even for other actors. It's my design decision, but feel free to change it.

There's one other flag we have not seen before: f_bypass. This is also a common flag and it tells the interpreter to bypass the visibility check for this subject. Remember from the examine verb that the interpreter will only consider objects that the player can see? Sometimes we want the player to be able to refer to objects he cannot see. Suppose he enters a dark location and wants to turn on the lamp in his inventory? We don't want a "You don't see a lamp here"-like response.

Our design decision is that all items in the player's inventory must be usable in darkness, so we set f_bypass when he picks something up and we clear it when he drops something.

drop

The drop verb is similar to get. But it will sometimes be the case that we want to put things *in* other things. To prevent us from having to code this functionality for each object we create default verb code that will not print a default "You cannot …"message but that will handle the "put something *in*

something else” command. We must define a new common flag f_container that tells whether an object can contain other objects.

Actions to put or drop things under/on/besides/… other objects must be handled by the objects themselves. For these scenarios, the verb default code will print a rejection message. We could also have decided to handle these actions in the verb code but for this tutorial we will only do the “in” preposition.

Our drop verb, *almost finished*

```
$VERB drop SYNONYM put
 "drop"
   printcr("What do you want to drop?")
   getsubject()
 "drop [o_subject]"
   move(o_subject, owner(o_actor))
   clearflag(o_subject.f_bypass)
   printcr("[o_subject]: dropped.")
 "drop [o_subject] in [o_spec]"
    if not(testflag(o_spec.f_container)) then
      printcr("[the] [o_spec] cannot contain things.")
      disagree()   # exit
    endif
    if testflag(o_spec.f_locked) then
      printcr("[the] [o_spec] is locked.")
    else
      if not(testflag(o_spec.f_open)) then
        printcr("[[opening [the] [o_spec] first]")
        setflag(o_spec.f_open)
        clearflag(o_spec.f_opaque)
      endif
      clearflag(o_spec.f_bypass)
      move(o_spec, o_subject, in)
      printcr("[the] [o_subject] is now in [the] [o_spec].")
 "drop [o_subject] [prepos] [o_spec]"
   printcr("[the] [o_subject] is not something that can be put [prepos] [the] [o_spec].")
 DEFAULT
   printcr("I only understood you as far as wanting to drop something.")
ENDVERB
```

As you see, dropping the subject is moving it to the object (or location) that contains the player.

The verb also checks if the container is locked or closed and will open the container automatically if possible. Opening the container will also clear the common flag f_opaque so the player can see what's in the container.

o_spec is the specifier wildcard.

We see that we have rules for “drop [o_subject] in [o_spec]” and “drop [o_subject] [prepos] [o_spec]”. The rules are matched top to bottom, so it is important that the … in … rule is before the … [prepos] … rule. If not, then ‘in’ would be caught by the ‘prepos’ rule and the ‘in’ rule would never be executed.

We still forgot two things: handling the case where the user wants to drop a subject he is not carrying and putting an object into itself. We will handle both situations in the prologue. And yes, for the “get” verb we did not handle the getting-something-you-already-have scenario in the prologue. But because this is a tutorial we want to show different ways of doing things and hence we show how to handle the drop-something-you-don’t-have in the prologue.

Our final drop verb

```
$VERB drop SYNONYM put
  PROLOGUE
    if equal(o_subject, %none) then
      Agree()
    else
if not(owns(o_actor, o_subject)) then
       printcr(“But [o_actor] [o_actor.r_be] not holding [the] [o_subject].”)
       disagree()
     endif
     if not(equal(o_spec, %none) then
       if equal(o_subject, o_spec) then
         printcr("You cannot put something into itself.")
       endif
     endif
    endif    # endifs at the end are not necessary
  "drop"
    printcr("What do you want to drop?")
    getsubject()
  "drop [o_subject]"
    move(o_subject, owner(o_actor))
    clearflag(o_subject.f_bypass)
    printcr("[o_subject]: dropped.")
  "drop [o_subject] in [o_spec]"
     if not(testflag(o_spec.f_container)) then
       printcr("[the] [o_spec] cannot contain things.")
       disagree()   # exit
     endif
     if testflag(o_spec.f_locked) then
       printcr("[the] [o_spec] is locked.")
     else
       if not(testflag(o_spec.f_open)) then
         printcr("[[opening [the] [o_spec] first]")
         setflag(o_spec.f_open)
         clearflag(o_spec.f_opaque)
       endif
       clearflag(o_spec.f_bypass)
       move(o_spec, o_subject, in)
       printcr("[the] [o_subject] is now in [the] [o_spec].")
  "drop [o_subject] [prepos] [o_spec]"
    printcr("[the] [o_subject] is not something that can be put into [the] [o_spec].")
  DEFAULT
    printcr("I only understood you as far as wanting to drop something.")
ENDVERB
```

As with to ‘get’ where we needed the r_have attribute, with ‘drop’ we need an r_be attribute to store the actor’s correct conjugation for to be.

inventory

Inventory illustrates XVAN’s object oriented architecture. Each object in the player’s possession will respond to the inventory command by printing its own description as we will see later in the story file.

The verb prologue will print the “You are carrying:” message. In case the player is carrying nothing (no object will react), the verb default code will print the “nothing, you are empty-handed.” Message.

Our inventory verb

```
$VERB inventory SYNONYM i
  SCOPE player_only
  PROLOGUE
    printcr(“You are carrying:“)
    indent(2)
  EPILOGUE
    Indent(-2)
  DEFAULT
    Indent()
    printcr(“Nothing, you are empty-handed.”)
END_VERB
```

Some new things here:

SCOPE player_only means that the user input is only offered to the player and the objects that are contained in the player. Other scope values are:

- curr_loc (player’s current location and all contained objects). This is the default scope value;
- all_locs (every location and object in the game).

The indent function, when called with a parameter increases or decreases the indent level. When called without a parameter the function prints the a number of spaces equal to the indent level.

The EPILOGUE is similar to the prologue, but is executed as the last item of processing the user input. Here we use it to reset the indent level to its old value.

go

The actual functionality of the go verb is defined in the story file with the player object. The go default code only prints the “you cannot go there” message.

Our go verb

```
$VERB go
  # do not remove this verb
  “go [dir]”,
  “go to [dir]”
    printcr(“You can’t go that way.”)
  DEFAULT
    printcr(“I don’t understand that sentence.”)
END_VERB
```

What's new here?

[dir] is the wildcard for an arbitrary direction. Whenever the user enters only a direction, the interpreter will prefix it with the verb 'go', which is predefined by the compiler.

There are 2 command strings followed by 1 piece of code. This means that both commands will execute the same code. So, both "go [dir]" and "go to [dir]" will print "You can't go that way." as a default message when no objects reply to the command.

Directions

As the last activity of part 1 of the tutorial, we will define the directions that our story will understand.

Any word can be used as a direction and there can be as many or as few as you want. Compass directions, left, right, in, out, up, down, all are possible. Left and right are a bit tricky because they require knowledge about the direction the player is facing.

The actual story map will defined in the story file. For each location we will indicate which direction leads where.

For our tutorial we will use compass directions and up/down.

```
$DIRECTIONS
north    SYNONYM n,
south    SYNONYM s,
east     SYNONYM e,
west     SYNONYM w,
northeast        SYNONYM ne,
northwest        SYNONYM nw,
southeast        SYNONYM se,
southwest        SYNONYM sw,
up       SYNONYM u,
down     SYNONYM d
```

When printing a word that has synonyms, the interpreter will use the first word for printing. So, advice is to not use the abbreviations as the first words.

End of part 1

This is the end of part 1 of the tutorial. Everything we did is in file part1-end.lib. This file is the starting point for part 2 of the tutorial.

This file won't compile to an XVAN game, because we have not made the story file yet. In part 2 and part 3 we will create the story file.