Part 5 – disambiguation

First, a little background information on how XVAN parses user input. After the user enters a command string, the parser tries to translate it into an action, subjects, specifiers, etc. The parser looks for nouns and adjectives in the user input and compares them to the d_sys descriptions from objects and locations that are in scope.

It may occur that more than one object or location can be mapped to the user input. Let's take the following example: we now have two toasters in the kitchen, a red toaster and a blue toaster. The player enters the command "get toaster". In this situation, the parser has 2 objects that qualify as a subject: the red toaster and the blue toaster. The player has issued an ambiguous command. In such a situation, the parser needs more information from the player and would print a message like "Which toaster do you mean? The red toaster or the blue toaster?". The player would, for example, reply with "blue" and the parser would map the blue toaster as the subject. This works well.

Now consider the following. The player is in the kitchen. He carries the blue toaster and the red toaster is on the floor. The player enters "get toaster". Again, the parser will find two objects that qualify, the red toaster and the blue toaster, so it will ask the player which toaster he means.

But… considering the fact that the player is already carrying the blue toaster and that he issued a get command, it is very likely that he means to get the red toaster. The same goes for the "drop toaster" command: since he's only carrying the blue toaster it is safe to assume he wants to drop the blue toaster.

It is important to realize that the parser can only map nouns/adjectives to XVAN objects/locations. The parser has no knowledge about the context(the action) like the interpreter has. After the parser has finished the mapping, the interpreter will execute the trigger or verb default code with the objects/locations it got from the parser. So, to help the parser become better in resolving ambiguities, we must make it aware of the action.

From XVAN version 2.1 each section of default code can be extended with disambiguation rules. In case the parser finds more than one candidate when mapping the user input, it will check the disambiguation rules section for the verb syntax that it is parsing. The disambiguation rules give points to objects who – given the context - qualify better than other objects. So in our "get toaster" example, the red toaster would get points for not being carried and the blue toaster would not.

An example for the "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]"
  DISAMBIGUATION_RULES
   If not(owns(o_actor, o_subject)) then score(5) endif
  END_RULES
  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.")
ENDVERB
```

The disambiguation rules will be evaluated for each possible subject. In our toaster example this will be the red toaster and the blue toaster. The red toaster will be awarded5 points because it's not held by the player. Therefore the red toaster 'wins'. The score() function is the only function that can be used after the 'THEN' statement in the disambiguation rules section.

Keep in mind that disambiguation rules are only consulted in case multiple objects qualify for mapping user input. If the user is carrying the blue toaster and enters "get blue toaster", the parser will map the subject to o_blue_toaster. There is no disambiguation here. When executing the command, the interpreter will find – when executing the "get [o_subject" default code – that the user is already carrying the blue toaster and will print the error message. So, disambiguation rules and trigger/verb default code operate at different levels.

No Such Thing

In part 3 we already saw o_nst, the no-such-thing object. As explained in the previous section, possible candidates get awarded points by the disambiguation rules. If all candidates end up with the same amount of points, the parser still has no clue and will fall back to asking the player which object he means.

All candidates start with zero points. In our toaster example, if both the red and blue toaster are on the floor and the player says "get toaster", both will end up with five points after applying the disambiguation rules and the parser will ask the player which toaster he means, which is what we want.

But if the player carries both toasters and he says "get toaster" both toasters will end up with 0 points and the parser will ask the player which toaster he means and then tell him he's already carrying it. Which is *not* what we want.

This is where the no-such-thing object comes in. o_nst is contained in the player object and it always participates in disambiguation. Where all objects start with 0 points in disambiguation, o_nst will have 1 point by default.

o_nst object

```
$OBJECT o_nst
 # this object must always be present
```

```
 # it may be modified but removal will cause a compiler error
DESCRIPTIONS
 d_no   "no such thing"
 d_any "any such thing"
CONTAINED in o_player
TRIGGERS
  t_entrance
    agree()
END_OBJ
```

In case all objects end up with 0 points, o_nst will win. The interpreter must check for this and can print an appropriate message.

verb "get" that considers o_nst

```
$VERB get SYNONYM take SYNONYM grab
 PROLOGUE
   If equal(o_subject, %none) OR equal(o_subject, o_nst) 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]"
   DISAMBIGUATION_RULES
     if not(owns(o_actor, o_subject)) then score(5) endif
   END_RULES
if equal(o_subject, o_nst) then
     printcr("There is [o_nst.d_no] to get.")
     disagree()
   endif
   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.")
ENDVERB
```

As an example, consider the toaster scenario where the player is holding both the blue and the red toaster and says "drop toaster".

```
Transcript from the tutorial game
XVAN transcript for: XVAN tutorial
version: 1.0
> l
Kitchen
This is the kitchen. There is not much here. The hallway is to the south.
There's a red toaster here.
There's a blue toaster here.
To the north is a door that leads to the garden.
> get toaster
Which toaster do you mean?
The red toaster or the blue toaster?
> blue
blue toaster: taken.
> l
Kitchen
This is the kitchen. There is not much here. The hallway is to the south.
There's a red toaster here.
To the north is a door that leads to the garden.
> get toaster
Red toaster: taken.
> i
You are carrying:
  a blue toaster
  a red toaster
> get toaster
There is no such thing to get.
> transcript
Turning off transcript mode.
```

Note: we used the transcript command described in section 2 to copy all screen output to file transcript.txt.

The first "get toaster" command was issued when both the red and the blue toaster were on the floor. Disambiguation rules did not help here (both toasters got 5 points an o_nst had 1) so the parser had to get back to the player for more info.

The second "get toaster" command was given when the player had the blue toaster and the red one was on the floor. Disambiguation rules gave 5 points to the red toaster, 0 to the blue and o_nst already had 1. So, the red toaster was the winner here.

The third "get toaster" command was given when the player held both toasters. So both the red and the blue toaster ended up with 0 points and o_nst had the 1 it started with and thus won. The verb code detected o_nst was the subject and printed the "There's no such thing to get." message.

Again, keep in mind that the disambiguation rules will only be consulted by the parser when t cannot map user input to a single object.

This is the end of part 5. Everything we did is in the files part5-end.xvn and part5-end.lib..