МЕТАРАРСЕР3

Петр Косых

13 октября 2018 г.

Оглавление

Глава 1

Введение

1.1 Чувство парсера

INSTEAD начался как проект, который пытался избежать подхода CYOA при разработке текстовых квестов и привнести чувство игры как в "настоящем приключении", но только в текстовом виде.

Как вы наверняка знаете, классические текстовые приключения – это парсерные игры, в которых моделируется мир, и герой исследует этот мир под чутким управлением игрока, который описывает желаемые действия с помощью клавиатуры. Тут и клавиатура и способ моделирования мира работают на создание общей цели – у игрока возникает чувство свободы и загадки. Ты свободен перемещаться по миру, взаимодействовать с ним – в этом смысле такая игра и есть настоящая адвенчура (квест).

CYOA игры - это игры с управлением через меню, обычно сводящиеся к выбору вариантов развития событий (убежать, показать пропуск охраннику и т.д.) Здесь способ ввода и способ написания игры также направлены на реализации своей цели. Игра обычно выглядит как сборник параграфов, нередко описывающих целые сюжетные повороты, а не только (и не сколько) локации (места действий, декорации) с переходами между ними. В этом смысле, жанр таких игр больше тяготеет к книгам-играм, хотя, конечно, многое зависит от конкретной игры.

Если же отойти от текстовых игр и посмотреть на классические графические квесты (из золотого фонда Lucas Arts и Sierra), то там мы увидим что-то среднее: либо глаголы, как в Monkey Island, либо манипуляции предметами, как в Goblins, либо что-то смешанное. Тем не менее, эти игры больше похо-

жи на парсерные игры в том смысле, что герой свободно исследует мир, а его способ взаимодействия с миром носит условно свободный характер (набор универсальных действий, свободная манипуляция предметами и т.д.) Фактически, это и есть парсерные игры, которые перешли в графическую плоскость, заодно сильно упростив способ ввода.

INSTEAD начинался как проект по созданию движка для игры, которая будучи текстовой, по своему игровому процессу не сильно отличалась бы от классических графических квестов. Такой первой игрой стала "Возвращение квантового кота".

В этой игре был выбран упрощенный ввод в виде свободного манипулирования предметами, что сильно снижало порог вхождения для потенциального игрока, но вместе с тем оставляло поле для свободы. Потом, по мере развития движка, появились игры с глагольным меню, которые еще больше сблизили текстовые квесты с классикой (правда, такие игры обычно обладали повышенной сложностью). В качестве примеров таких игр можно назвать "Кайлет" (набор глаголов) и "Особняк" (три глагола).

Тем не менее, мне было всегда интересно посмотреть, можно ли сблизить игру с парсером еще теснее, при этом не сильно подняв порог вхождения? Можно ли взять лучшее от INSTEAD подхода и парсера?

Кое-какой опыт по работе с парсером у меня был. Еще перед написанием INSTEAD я изучал Inform и даже писал небольшую игру, поэтому я примерно представлял проблемы и плюсы парсерной игры на русском языке.

Огромное преимущество парсерных игр – глубокое погружение. Во время игры, игрок вынужден читать и думать, это очень сильно отличается от CYOA игр, и все-еще отличается от игр с глагольным меню. Именно эта черта заставляла экспериментировать с игровым вводом. Однако, такие игры сложнее писать, особенно на русском, и (что не менее важно) в них не так просто играть!

Обычно, неявным образом подразумевается, что идеальный парсер – это ИИ, который понимает все, что напишет игрок. Это и сила и слабость одновременно. Я не верю, что компьютер научится думать (по крайней мере, в обозримом будущем), а это значит, что ИИ, который бы мог учитывать в полной мере контекст введенных команд и вообще угадывать то, что игрок хочет сделать в свободной разговорной форме – утопия. Во всех существующих реализациях, парсер пытается притвориться умным с разной степенью успешности, но любой парсер бессильно проигрывает, когда за клавиатуру садится неподготовленный игрок, который воображает, что игра поймет все, что он напишет.

Если же совместить это с богатством русского языка, то получится, что в

русские парсерные игры способны играть далеко не все, да и даже подготовленный игрок с большой долей вероятности натолкнется на ситуацию, где отсутствие привычного ему синонима отобьет всю охоту заканчивать игру.

Первой серъезной попыткой в рамках INSTEAD на пути к обретению чувства парсера стал – "МЕТАПАРСЕР 2".

1.2 МЕТАПАРСЕР 2

Основной идеей для МЕТАПАРСЕРА 2 послужило наблюдение такого факта, что с ростом числа глаголов в глагольном меню, чувство игры начинает приближаться к парсерному варианту. Правда, при этом возникали другие проблемы: захламление меню и все возрастающая сложность написание игры (так как обработчики неудачных действий ничего не знают о смысле действий, о падежных формах и вообще не занимаются анализом, то в целях литературности приходится прописывать реакции на все возможные действия). В тоже время, в парсере указанных проблем нет.

Тогда и возникла идея о создании парсера, который работает одновременно и на принципах меню и на принципах парсера, взяв лучшее из обоих миров. Я подумал, что задачу абсолютной непротиворечивости интеллекта парсера можно решить не за счет его усложнения, а наоборот, за счет его упрощения. Другими словами, игрок не может ввести действия, которые не будут поняты игрой, или же игрок должен постоянно как-то понимать, понимает ли его игра в данный момент.

Итак, нужно было решить две проблемы.

Первая из проблем – парсер должен уметь делать склонения, так как настоящая парсерная игра предполагает десятки и сотни действий, которые автор игры не обязан (да и не способен) прописывать сам. Например, если в игре есть дверь и ее можно открыть и закрыть, то движок сам в состоянии отреагировать на команды "открыть" или "закрыть дверь" на основании указания, что объект "дверь" может открываться. Или предметы, которые можно подбирать и бросать. Фразу - "Я взял яблоко." вполне может сформировать и сам движок. Но гораздо важнее, чтобы движок мог среагировать на непрописанное действие, навроде такого:

```
толкнуть Габриэллу
Габриэлле это не понравится!
```

На самом деле, чтобы научить движок склонять слова, можно пойти двумя путями.

Первый путь, это при описании объекта четко описать его характеристики: число, род, одушевленность. Выделить окончание, и воспользоваться алгоритмом для формирования склонений. У такого подхода есть пару существенных недостатков:

- алгоритм склонений не универсален;
- код игры становится сложным, высокий порог вхождения для автора игры.

Второй путь, это использование словаря. В этом случае, для описания объекта только иногда необходимо указать одушевленность, и все. Тем не менее у этого подхода есть свои недостатки:

- объем словаря велик;
- иногда, в словаре может не быть нужного слова.

Я не пошел по пути алгоритма, главным образом потому, что это сильно затрудняет написание игры (я помнил это еще по моим экспериментом с Inform). В итоге, в метапарсере я использовал словарь, взяв за основу базу с http://aot.ru. К счастью, формат словаря оказался понятным. Кроме того, я нашел его черновое описание, и в итоге, мне удалось сконвертировать его для удобного распознавания изнутри lua. Фактически, за пару недель, решение проблемы склонений было найдено! Словарь в UTF–8 кодировке занимает около 8МБ и для повышения скорости и снижения объемов игры был выбран путь компиляции словаря для конкретной игры. В итоге, получаемый словарь был очень маленьким и быстрым, так как содержал только то, что нужно.

Результат мне очень понравился, ситуации, когда словарь срабатывал неправильно можно пересчитать по пальцам, кроме того, всегда можно было вписать свои варианты склонений (например, имя Габриэллы).

Осталось сделать совсем немного – сам парсер. :)

Идея была в следующем. Парсер понимает некий набор глаголов, с переменным числом параметров. Например, можно написать:

```
бросить лампу в гоблина
```

или просто:

```
бросить лампу
```

Оба действия – это действие “бросить”, но в одном случае мы бросаем лампу в гоблина, а во вторую - избавляемся от нее. Погружаться в детали пока не будем, но суть в том, что парсер всегда знает множество того, что может ввести игрок. То-есть метапарсер в каком-то смысле обратен парсеру. Он не пытается понять, что пишет игрок – он знает что вообще можно написать и следит, какую из веток набирает сейчас игрок.

Такое устройство парсера означает, что в любой момент времени он может подсказать игроку возможные варианты, например, игрок пишет:

```
бросить л_
```

В этот момент парсер знает, что игрок хочет написать какое-то слово, которое начинается на букву л и это предмет из сцены или инвентаря, парсер может *подсказать*.

Подсказки выводятся в каждый момент времени в виде списка слов, которые можно продолжать набирать, или по которым можно щелкать мышкой. Это приводит к тому, что в метапарсер можно играть без клавиатуры.

На МЕТАПАРСЕРЕ 2 было написано несколько отличных игр, и я считал его своим успешным проектом, но все-таки меня не оставляла мысль о том, что можно пойти еще дальше…

1.3 МЕТАПАРСЕР 3

Я хотел понять, а можно ли совместить удачные идеи от МЕТАПАРСЕРА 2 с настоящим парсером? Так появился МЕТАПАРСЕР 3.

Итак, в отличие от своего предшественника МЕТАПАРСЕР 3 это настоящий парсер. Это значит, что если отключить подсказки в виде ссылок - слов игра играется так же как игры на информе. Можно написать: взять яблоко, или яблоко взять или быстро возьми яблоко и это будет работать

Словарь теперь используется на полную катушку. в том числе и для глаголов. Это очень сильно упростило код самого движка. Пример вывода стандартной библиотеки;

```
mp.msg.Enter.INV = [[{#Me} не {#word/могу,#me,нст} зайти в то,
        что {#word/держать,#me,нст} в руках.]]
```

МЕТАПАРСЕР 3 написан на stead3 – код стал понятным и простым для расширения. Код игр *значительно упрощен*.

В МЕТАПАРСЕР 3 использована стандартная библиотека, которая была позаимствована из Inform6. Это очень облегчает моделирование мира игры. Например, можно поставить стол, на нем аквариум (прозрачный и открытый) В нем рыбка. И все будет корректно отрабатывать - включая зоны видимости.

Парсер чуток к игроку. Например, он понимает описки в словах и указывает на них. Подсказывает варианты ввода.

При этом МЕТАПАРСЕР 3 может работать в режиме своего предшественника, в таком случае варианты слов подсвечиваются в виде списка и их можно выбирать мышкой.

Итак, остался только один вопрос. Можно ли сегодня написать парсер, в который будут играть?

Глава 2

Быстрый старт

Перед тем как писать игры на МЕТАПАРСЕРЕ3 вам желательно ознакомиться с STEAD3 и понимать общие принципы программирования игр на INSTEAD.

Для того, чтобы начать свою игру, вам необходимо скачать последнюю версию МЕТАПАРСЕРА (далее МП), который, кроме всего прочего, содержит в себе словарь.

Стабильную версию в виде zip архива можно скачать со странички INSTEAD3[1]. В архив включен модуль МП в виде готовой к запуску демонстрационной игры.

Версию, находящаяся в разработке можно взять из репозитория модулей INSTEAD3[2]. Модули МП расположены в подкаталоге metaparser.

Также, в репозитории в каталоге metaparser/demos есть несколько дополнительных демонстрационных игр. Вы можете скопировать необходимые файлы любой из игр (обычно main3.lua и dict.mrd) в каталог с МП3 и запустить её.

Самая простая игра – heidi. Вы можете изучить ее исходный код и использовать ее как шаблон.

[1] https://instead-hub.github.io/page/metaparser/
[2] https://github.com/instead-hub/stead3-modules

Глава 3

Скелет игры

В качестве примера я буду рассматривать игру heidi. Итак, заготовка игры:

```
--$Name:Хейди$
require ”mp-ru” -- подключаем модули МП
require ”fmt” -- форматирование

game.dsc = [[^Пример простой игры на Inform.
^Авторы: Роджер Фирт (Roger Firth) и Соня Кессерих (Sonja Kesserich).
^Перевод Юрия Салтыкова a.k.a. G.A. Garinson^
^Перевод на МЕТАПАРСЕР 3 выполнил Петр Косых.
^
]]
```

Собственно говоря, это все. Такую игру уже можно запустить. Обратите внимание на game.dsc – это тот текст, который выводится перед началом игры. Конечно, game.dsc может быть функцией.

Глава 4

Словарь

Прежде чем мы начнем создавать мир игры нужно отметить следующую вещь. Как уже было сказано во введении, МПЗ использует словарь.

Полный словарь ($>$ 8Мб) расположен в metaparser/morph/morphs.mrd. При запуске игры движок смотрит за изменением словарного состава игры и *компилирует* персональный (маленький) словарь для конкретной игры. Этот словарь будет расположен в metaparser/dict.mrd.

Словарный состав слов смотрится в файлах *.lua в каталоге игры (без захода в подкаталоги!). Если вам важно размещать исходные коды игры (включая mp-ru.lua) в других подкаталогах, перед включением mp-ru.lua задайте std.SOURCES_DIR:

```
std.SOURCES_DIR = { 'lib', 'game' }
require "mp-ru"
require "fmt"
```

Тогда исходные коды будут анализироваться в этих подкаталогах. Каталог с файлом mp-ru.lua тоже должен быть включен в список!

Компиляция словаря может быть длительным процессом на слабых машинах. Например, на ееерс время компиляции около 20 секунд. На i3 – около 7 секунд. Если вы Unix пользователь я рекомендую собрать INSTEAD с luajit, таким образом вы ускорите время компиляции словаря в 2 раза.

Компиляция происходит тогда, когда в код игры добавляются или удаляются словарные слова (об этом будет сказано далее).

ВНИМАНИЕ: Когда ваша игра готова, вы можете стереть файл полного словаря metaparser/morph/morphs.mrd. Это действительно стоит сделать, так как размер готовой игры уменьшится на 8Мб.

Глава 5

Комнаты и объекты

Давайте добавим комнату в игру.

```
room {
    nam = ”before_cottage”;
    title = ”Перед домом”;
    dsc = ”Ты стоишь около избушки, на восток от которой раскинулся лес.”;
}
```

Здесь пока все понятно и соответствует STEAD3 API. Теперь, чтобы наша игра начиналась с этой локации, а не из комнаты main, добавим функцию init:

```
function init()
    pl.room = ’before_cottage’
    -- то же самое можно было бы записать так:
    -- me().room = ’before_cottage’
end
```

Теперь, если запустить игру, то мы окажемся в локации “Перед домом”.
Что дальше? Добавляем объект.

```
obj {
    -”домик”;
    nam = ”cottage”;
    dsc = ”Домик мал и неказист, но ты очень счастлива, живя здесь.”;
}
```

Здесь мы видим определение словарных слов. Модуль словаря при запуске игры ищет в *.lua файлах строчки вида

```
-”что то”
```

И считает их словарными словами, для которых нужно создать запись в персональном словаре игры. Таким образом, вы можете добавлять слова просто используя комментарии:

```
--”бутылка”
```

На самом деле, в стандартной библиотеке используется этот прием.

Но при создании объектов мы делаем две вещи сразу:

1) определяем имя объекта, по которому его видит игрок; 2) определяем факт того, что это словарное слово.

Чтобы более подробно пояснить этот факт, приведу пример:

```
--”домик”
obj {
    word = ”домик”;
    nam = ”cottage”;
    dsc = ”Домик мал и неказист, но ты очень счастлива, живя здесь.”;
}
```

На самом деле мы сделали тоже самое. Определили отображаемое имя через word = и определили словарные слова, но гораздо проще запись, которая делает эти две вещи сразу:

```
obj {
    -”домик”;
    nam = ”cottage”;
```

Теперь добавим объект в сцену. В МП3 мы можем сделать это несколькими способами.

Через задание obj у комнаты:

```
room {
    nam = ”before_cottage”;
...
    obj = { ’cottage’ };
}
```

Через :with у комнаты:

```
room {
    nam = ”before_cottage”;
...
}: with {’cottage’}
```

Через задание found_in у объекта:

```
obj {
    -”домик”;
    nam = ”cottage”;
    found_in = ’before_cottage’;
-- или found_in = { ’before_cottage’ }
```

found_in может быть функцией! В таком случае предмет будет находиться в тех локациях, в которых found_in вернул true.

Если вы запустите игру, произойдет генерация словаря а затем вы сможете, например, ввести команду: осмотреть домик и получить ответ: ты не видишь в домике ничего необычного.

Как видим, слово “домик” попало в словарь и успешно склоняется движком.

Но что если вы наберете: осмотреть дом? Увы, игра вас не поймет.

Глава 6

Подробнее о словарных словах

Итак, задавая словарное слово в виде: -"домик" мы определили основное словарное слово. Если вы считаете (а обычно это так), что предмет может быть доступен по нескольким именам, у вас есть следующие возможности:

Перечислить варианты через запятую. Внимание! Этот метод работает, если род и число синонимов совпадают. Например:

```
-"домик,дом"
```

При этом в словарь будут добавлены оба слова. Основным словом останется *домик*, а *дом* будет дополнительным синонимом. Так, если игрок напишет: осмотреть дом, то получит сообщение: ты не видишь в домике ничего необычного.

Хорошо, но что если мы хотим, чтобы игра откликалась и на избушку?

Если мы напишем:

```
-"домик,дом,избушка"
```

Движок не сможет корректно склонять слова, так как слова имеют разный род. В таком случае можно воспользоваться альясами:

```
-"домик,дом|избушка"
```

Альяс – это полноценное альтернативное имя объекта. Например, если игрок напишет: осмотреть избушку, то получит: ты не видишь в избушке ничего интересного.

На самом деле есть еще один простой способ заставить игру понимать слово избушка. Этот способ – шаблоны. Например:

```
-"домик,дом,избуш*"
```

При этом, если игрок введет что угодно, начинающееся с избуш – игра воспримет это как сионим домика. Отличия шаблонов от полноценных слов:

1) шаблоны не могут быть 1-ми в списке; 2) шаблоны не автодополняются по клавише TAB; 3) шаблоны не могут быть подсказаны движком.

Использовать шаблоны имеет смысл в каких то сложных случаях как дополнительное средство, например:

```
-”домик,дом|избушка,избу*,терем*,коттедж*,хат*,строени*”;
```

Обычно словарь сам справляется с определением рода, числа и даже одушевленности, однако иногда возникают спорные ситуации, когда одно и то же слово может означать, например, фамилию или качество объекта. В таких случаях вы можете уточнять словарное слово, например:

```
-”домик,дом/мр,ед”
```

Уточнения всегда относятся к одному слову или группе слов, разделенных запятыми. Например:

```
-”домик,дом/мр,ед” -- правильно
-”домик/мр,дом/мр” -- неправильно
-”домик,дом/мр|избушка/жр” -- правильно
```

Список уточнений:

- мр - мужской род;
- жр - женский род;
- ср - средний род;
- мн - множественное число;
- ед - единственное число;
- од - одушевленное;
- но - неодушевленное;
- С - существительное;
- Г - глагол;
- П - прилагательное;

Для комбинации уточнений, используйте запятую. Для отрицания, можно использовать символ ~. Например:

```
-”фрукты/~од”
```

На самом деле, уточнения нужны редко. Используйте их только для проблемных слов. Чаще всего, для решения проблемы склонений вам придется добавлять уточнения одушевленности: “од” или “но”.

В качестве словарных слов вы можете использовать несколько слов, например:

```
-”черный котенок,котенок/мр”
```

В таком случае, обязательно делайте наиболее полным первое – основное описание, а затем указывайте сокращенный вариант. Тогда игрок сможет обратиться и к котенку и к черному котенку. А если на сцене будет несколько котят, то движок сможет сделать подсказку.

Иногда может оказаться, что слова нет даже в полном словаре (или оно неверно склоняется). Например, какая то экзотическая фамилия, или вымышленное слово. В таком случае, придется добавить его в словарь вручную.

У вас есть две возможности: добавить слово в словарь объекта или в словарь игры.

Добавление словарного слова в словарь объекта:

```
obj {
    -”герцог|Гесслер,фогт,Герман”;
    nam = ’governor’;
...
} : dict {
    [”Гесслер/вн”] = ”Гесслера”;
    [”Гесслер/рд”] = ”Гесслера”;
    [”Гесслер/дт”] = ”Гесслеру”;
    [”Гесслер/тв”] = ”Гесслером”;
    [”Гесслер/пр”] = ”Гесслере”;
}
```

Или словарь игры:

```
game : dict {
    [”Гесслер/вн”] = ”Гесслера”;
    [”Гесслер/рд”] = ”Гесслера”;
    [”Гесслер/дт”] = ”Гесслеру”;
    [”Гесслер/тв”] = ”Гесслером”;
    [”Гесслер/пр”] = ”Гесслере”;
}
```

Ну и наконец, даже комнаты могут иметь словарное слово. Например:

```
room {
    -”двор,дворик,лес*”;
    nam = ”before_cottage”;
    title = ”Перед домом”;
    dsc = ”Ты стоишь около избушки, на восток от которой раскинулся лес.”;
}
```

Игрок сможет написать: осмотреть двор, уйти со двора, осмотреть лес.
Но задавать словарное слово комнатам необязательно.

Глава 7

Атрибуты

Если попробовать запустить нашу игру игру с локацией и домом в ней, то в целом она будет выглядеть вполне адекватной. До тех пор, пока вы не возьмете дом командой: взять домик.

Дело в том, что все объекты по умолчанию могут быть взяты.

Модель мира МПЗ заимствована из Inform 6 и предполагает наличие у объекта специальных атрибутов, которые меняют его поведение.

Для задания атрибутов используется метод :attr. Например:

```
obj {
    -”домик,дом”;
    nam = ”cottage”;
    found_in = ’before_cottage’;
}:attr ’scenery’
```

В данном примере мы присвоили домику атрибут scenery, который означает следующее:

- описание предмета не нужно выводить после описания сцены;
- предмет не может быть взят, он является декорацией.

Попробуйте запускать варианты игр в которых атрибут scenery задан и не задан и вы увидите, что когда scenery не задан, в описании игровой ситуации после описания сцены присутствует строчка: Здесь находится домик. Но когда мы задаем атрибут scenery эта строка пропадает. Действительно, мы ведь уже

описали домик в тексте комнаты. Таким образом, атрибут scenery используется главным образом для декораций.

Вы можете задавать несколько атрибутов, разделяя их запятыми. Например:

```
}:attr 'scenery,supporter'
```

Вы можете динамически задавать атрибуты в коде:

```
_'cottage':attr 'scenery'
```

Или снимать их:

```
_'cottage':attr '~scenery'
```

А также проверять на их наличие:

```
-- по имени
if _'cottage':has'scenery' then ...

-- по параметру в обработчике
... = function(s, w)
    if w:has'scenery' then
```

Аттрибутов существует множество, пока опишем несколько:

- scenery – объект является декорацией;
- static – объект статичен, зафиксирован. Его нельзя брать, но описание его присутствует после описания сцены;
- concealed – объект обычный, но его описание скрыто (в инвентаре и сцене);
- light – освещено. В МПЗ по умолчанию все комнаты имеют свет. Если вам нужна комната без света, при создании задайте :attr '∼light'

Глава 8

Свойства

Если игрок попробует осмотреть домик, он получит стандартное сообщение библиотеки, которое не очень захватывает воображение.

Вы можете определить свойство description, которое будет содержать сообщение при осмотре объекта:

```
obj {
    -”домик,дом|избушка,избу*,терем*,коттедж*,хат*,строени*”;
    nam = ”cottage”;
    description = ”Домик мал и неказист, но ты очень счастлива, живя здесь.”;
}:attr ’scenery’
```

Конечно, description может быть функцией.

Выше мы уже использовали другое свойство: dsc. Это свойство отвечает за то, как информация об объекте представлена в сцене. Для комнат это будет описанем комнаты. Для обычных объектов – их описанием в сцене. Например, мы можем убрать scenery (заменив его static) и задать dsc у домика:

```
obj {
    -”домик,дом|избушка,избу*,терем*,коттедж*,хат*,строени*”;
    nam = ”cottage”;
    description = ”Домик мал и неказист, но ты очень счастлива, живя здесь.”;
    dsc = ”Ты находишься в тени домика.”;
}:attr ’static’
```

Описание “Ты находишься в тени домика.” будет выведено после описания комнаты. Для перемещаемых объектов удобным будет свойство: init_dsc, которое задает описание объекта в его первоначальной локации (пока он еще не был ни разу перемещен).

Существует множество свойств, которые мы будем постепенно рассматривать. А пока, создадим остальные локации.

Глава 9

Стороны света

Мы создали только одну локацию. Обычно игра состоит из нескольких локаций, связанных между собой. Традиционно, для ориентирования и перемещения между локациями в адвенчурах используются стороны света. Таким образом, локации связываются с учетом сторон света.

Например:

```
room {
    nam = ”before_cottage”;
    title = ”Перед домом”;
    dsc = ”Ты стоишь около избушки, на восток от которой раскинулся лес.”;
    e_to = ’forest’;
}

room {
    -”чаща|лес”;
    nam = ”forest”;
    title = ”В лесной чаще”;
    dsc = [[На западе, сквозь густую листву, можно разглядеть небольшое строение.^
            Тропинка ведет на северо-восток.]];
    w_to = ’before_cottage’;
}
```

Здесь мы видим две локации. В локации “Перед домом” нам доступен переход на восток (это отражено в тексте локации). При этом, если мы пойдем на восток – мы попадем в лес. Вы уже заметили, что комнаты связаны с помощью свойст e_to (путь на восток) и w_to(путь на запад).

В МП3 как и в Inform 6 определены 8 горизонтальных направлений.

1) n_to на север 2) ne_to на северо-восток 3) e_to на восток 4) se_to на юго-восток 5) s_to на юг 6) sw_to на юго-запад 7) w_to на запад 8) nw_to на северо-запад

Два вертикальных направления:

1) u_to вверх 2) d_to вниз

И два дополнительных:

1) in_to внутрь 2) out_to наружу

Все эти свойства могут быть как строками (и содержать nam комнат-направлений), так и функциями. Во втором случае, при попытке перехода по заданному направлению будет выполнена эта функция.

Если в вашей игре вам не нужен компас, просто отключите его:

```
objs'@compass':disable()
```

Когда у комнаты нет какого-либо направления, а игрок пытается идти в этом направлении, библиотека выдает стандартное сообщений. Если вы зададите у комнаты свойство cant_go, то это свойство будет выполнено в данной ситуации.

```
room {
    nam = "before_cottage";
    title = "Перед домом";
    dsc = "Ты стоишь около избушки, на восток от которой раскинулся лес.";
    e_to = 'forest';
    in_to = function()
        p [[Такой славный денек...
            Он слишком хорош, чтобы прятаться внутри.]];
    end;
    cant_go = "Единственный путь ведет на восток.";
}
```

Обратите внимание, что свойство in_to задано в виде функции. Если бы это была просто строка, движок бы попытался найти комнату с таким nam. Теперь же, если игрок напишет: идти внутрь, ему будет показано наше сообщение.

cant_go тоже может быть функцией. В таком случае, в качестве параметра передаётся направление в виде текстовой константы:

```
cant_go = function(s, to)
    if to == 's_to' then
        p "Ты не хочешь идти на север."
    else
        p [[Там нет ничего интересного.]]
    end
end
```

Глава 10

Добавляем новые объекты

В игре Хейди мы должны будем спасти птенчика выпавшего из гнезда. Вот как определяется птенчик:

```
obj {
    -”птенчик,птенец|птица,птичка|детёныш”;
    nam = ”bird”;
    description = ”Слишком мал, чтобы летать, птенец беспомощно попискивает.”;
}: attr ’~animate’
```

Здесь нам многое уже знакомо. Описаны словарные слова. Задано свойство description. Но есть и нечто новое.

С помощью записи :attr ‘∼animate’ отменяется атрибут animate. animate это признак того, что персонаж является живым персонажем игры. В МПЗ используется словарь, который обычно автоматически считает все одушевленные объекты персонажами (а птенчик одушевлен). Но почему мы отменяем это в данном случае? Дело в том, что стандартная библиотека по умолчанию не дает брать персонажей. А птенчика мы должны уметь забрать, так что в данном случае признак персонажа нам мешает. Попробуйте убрать отмену атрибута animate и посмотреть, что скажет игра на попытку взять птенчика.

Теперь гнездо:

```
obj {
    -”гнездо|мох|прутики,прутья”;
    nam = ”nest”;
    description = function(s)
```

```
        p ”Гнездо сплетено из прутиков и аккуратно устлано мхом.”;
        mp:content(s)
    end;
}: attr ’container,open’
```

Снова новые атрибуты.

- container - указывает на тот факт, что данный объект может содержать внутри себя другие объекты;
- open - контейнер открыт. В противном случае, мы не смогли бы положить птенчика в гнездо.

Таким образом, если игрок напишет: положить птенчика в гнездо – птенчик окажется в гнезде.

Теперь рассмотрим свойство description. Это свойство реализовано в виде функции и кроме сообщения-описания гнезда содержит:

```
mp:content(s)
```

Дело тут вот в чем. Если у объекта нет функции description, то при его осмотре стандартная библиотека выведет описание его содержимого (если это контейнер). Но если мы определили свое свойство description, то за вывод информации отвечает уже наша игра.

mp:content(s) – это метод метапарсера который показывает содержимое объекта. На самом деле есть другой способ достичь той же цели, это вернуть из description false:

```
obj {
    -”гнездо|мох|прутики,прутья”;
    nam = ”nest”;
    description = function(s)
        p ”Гнездо сплетено из прутиков и аккуратно устлано мхом.”;
        return false
    end;
}: attr ’container,open’
```

В этом случае мы даем сигнал библиотеке продолжать стандартный ход выполнения цепочки вызовов и библиотека опишет содержимое контейнера за нас.

Еще один эксперимент, добавьте гнезду атрибут прозрачности ‘transparent’:

```
obj {
    -”гнездо|мох|прутики,прутья”;
    nam = ”nest”;
    description = function(s)
        p ”Гнездо сплетено из прутиков и аккуратно устлано мхом.”;
        return false
    end;
}: attr ’container,open,transparent’
```

И осмотритесь в сцене. Теперь, при осмотре сцены нам будет сразу показано содержимое гнезда. “Здесь есть гнездо. В гнезде находится птенчик.”

В игре про птенчика игрок должен подняться на дерево и положить гнездо с птенцом на ветку. Посмотрим как это сделано.

В локации полянка присутствуют объекты “гнездо” и “дерево”, а также задано направление вверх:

```
room {
    -”полянка,поляна”;
    nam = ”clearing”;
    title = ”Полянка”;
    dsc = [[Посреди полянки стоит высокий платан.
        Тропинка вьется меж деревьев, уводя на юго-запад.]];
    sw_to = ’forest’;
    u_to = ’top_of_tree’;
    obj = { ’nest’, ’tree’ };
}

obj {
    -”платан|дерево|ствол”;
    nam = ’tree’;
    description = [[Величавое дерево стоит посреди поляны.
        Кажется, по его стволу будет несложно влезть наверх.]];
} : attr ’scenery’
```

А в локации “верхушка” находится объект ветка:

```
room {
    -”верхушка”;
    nam = ’top_of_tree’;
    title = ”На верхушке дерева”;
    dsc = ”На этой высоте цепляться за ствол уже не так удобно.”;
    d_to = ’clearing’;
    obj = { ’branch’ };
}

obj {
    -”сук|ветка”;
    nam = ’branch’;
    description = [[Сук достаточно ровный и крепкий, чтобы
        на нем надежно держалось что-то не очень большое.]];
    each_turn = function(s)
        if _’bird’:inside’nest’ and _’nest’:inside’branch’ then
            walk ’happyend’
        end
    end
}:attr ’supporter,static’
```

Рассмотрим объект ветку подробнее. Во первых появился новый атрибут supporter. Этот атрибут означает, что объект может содержать на себе другие объекты. Типичные supporter объекты это мебель. В данном случае, мы должны смочь положить гнездо на ветку (а не в ветку), поэтому мы задаем атрибут supporter.

Далее, у объекта задано свойство each_turn. Этот метод будет вызываться после каждого хода игры, пока ветка находится в зоне доступности игрока. В данной функции мы проверяем условие выигрышной ситуации с помощью метода объекта inside.

Сцена happyend лаконична:

```
room {
    nam = ’happyend’;
    title = ”Конец”;
    dsc = [[Поздравляем! Вы прошли игру.]];
    noparser = true;
}
```

Обратите на свойство noparser. Так мы отключили игроку возможность “играть” на финальной сцене.

Глава 11

Улучшаем игру

Продолжим рассматривать Хейди. По замыслу авторов игры, игрок не должен забраться на дерево держа в руках и птенчика и гнездо.

Проще всего это сделать, задав свойство capacity у игрока, например:

```
function init()
    pl.word = -”ты/жр,2л”
    pl.room = ’before_cottage’
    pl.description = ”Здесь нет зеркала.”
    pl.capacity = 1 -- задать максим размер инвентаря
end
```

Свойство capacity есть и у supporter и у container объектов. Как видим из примера, мы задали description у игрока, и теперь при вводе команды: осмотреть себя, мы получим сообщение о зеркале.

И наконец, мы задали у игрока его словарное слово. По умолчанию словарное слово игрока это “ты” в мужском роде. Но в данной игре главный герой это женщина (или девушка) и мы явно это указали. Заодно указав и лицо.

Птенчик это такой персонаж игры к которому игрок наверняка проявит интерес. Что произойдет, если игрок попробует послушать птенчика? Или погладить его? Стандартная библиотека сообщит нам довольно сухо что то вроде: никаких необычных звуков нет. Исправим эту ситуацию.

```
obj {
    -”птенчик,птенец|птица,птичка|детёныш”;
    nam = ”bird”;
    description = ”Слишком мал, чтобы летать, птенец беспомощно попискивает.”;
```

```
    before_Listen = [[Жалобный писк испуганной птички разрывает
    тебе сердце.^Надо помочь!]];
}: attr '~animate'
```

Здесь мы определили свойство before_Listen. Рассмотрим этот момент подробнее.

Когда игрок вводит команду, например, послушать птенчика, МПЗ ищет в библиотеке подходящий глагол и формирует *событие*.

Параметрами события становятся объекты. Для команды "послушать" параметр только один – птенчик. Существуют команды с двумя параметрами. Например "отпереть дверь ключом".

Когда аргументы сформированы запускается цепочка обработки события. Эта цепочка – последовательность вызовов функций (или взятия строк) которая выглядит следующим образом (на примере Listen):

1) before обработчики mp; 2) before обработчики game; 3) before обработчики here(); 4) before обработчики объекта (птенчика); 5) обработчки Listen mp; (стандартное действие библиотеки) 6) after обработчики объекта (птенчика); 7) after обработчики объекта here(); 8) after обработчики объекта game(); 9) after обработчики объекта mp (обычно, стандартное текстовое сообщение библиотеки);

Если обработчик на любой стадии возвращает false, цепочка продолжает свое выполнение. В противном случае она завершается на этом шаге.

Таким образом, определив свойство before_Listen у птенчика мы вывели свою реакцию и прекратили выполнение цепочки (не дали библиотеке вывести стандартное сообщение.)

Как выглядит вызов обработчиков на любом из шагов? Для примера возьмем шаг 4. Пусть w – это объект "птенчик", тогда последовательность вызовов выглядит таким образом:

1) w:before_Any(event) – если не false, продолжать 2) w:before_Listen() – только если у объекта определено свойство before_Listen 3) w:before_Default(event) – если у объекта не определено свойство before_Listen

event – это имя события. Обратите внимание, что этот параметр передается только для Any и Default.

Для наглядности, вот как выглядит определение этих свойств:

```
room {
    -"буфет";
```

```
    nam = 'bar';
    before_Default = function(s, ev, w)
        if ev == 'Exit' then -- это Exit?
            return false -- Нет, продолжаем цепочку
        end
        -- в w -- объект над которым совершается действие
        if not s:has 'light' then
...
```

Итак, определение before_ или after_ свойств дает возможность изменить стандартные реакции библиотеки.

Давайте посмотрим как исправить еще одну проблему.

Если на стартовой локации игрок попробует выполнить: войти в дом, то получит стандартную фразу о том, что в дом войти нельзя. "Войти" это глагол который создает событие Enter.

```
obj {
    -"домик,дом|избушка,избу*,терем*,коттедж*,хат*,строени*";
    nam = "cottage";
    description = "Домик мал и неказист, но ты очень счастлива, живя здесь.";
    before_Enter = [[Такой славный денек...
                    Он слишком хорош, чтобы прятаться внутри.]];
}:attr 'scenery'
```

Отлично! Теперь и на команду "идти внутрь" (см. in_to) и на "зайти в дом" мы получаем верные реакции игры!

В локации, где находится дерево, мы можем пойти вверх. Но что если игрок наберет "залезть на дерево" или "взобраться по дереву"? Снова стандартная реакция! Исправляем:

```
 obj {
    -"платан|дерево|ствол";
    nam = 'tree';
    description = [[Величавое дерево стоит посреди поляны.
        Кажется, по его стволу будет несложно влезть наверх.]];
    before_Climb = function(s)
```

```
        move(me(), 'top_of_tree');
    end
} : attr 'scenery'
```

Climb – это событие которое возникает при попытки игрока взобраться на какой либо объект. (Или используя этот объект.)

ОБратите внимание на функцию move(). В МПЗ все перемещения объектов (включая игрока!) выполняются этой единственной функцией.

Альтернативный вариант вызова move:

```
    before_Climb = function(s)
        me():move 'top_of_tree';
    end
```

Что если находясь на верхушке дерева мы бросим гнездо вниз? К сожалению, оно останется в локации "верхушка" – таково стандартное поведение библиотеки. Ведь только автор знает что мы находимся на дереве и тут нет твердой почвы под ногами.

Взгляните на цепочку вызовов, нам поможет 7й шаг.

```
room {
    -"верхушка";
    nam = 'top_of_tree';
    title = "На верхушке дерева";
    dsc = "На этой высоте цепляться за ствол уже не так удобно.";
    d_to = 'clearing';
    after_Drop = function(s, w)
        move(w, 'clearing')
        return false
    end;
    obj = { 'branch' };
}
```

Когда мы сбросим гнездо вниз (надеюсь, без птенчика?) будет вызвано свойство after_Drop у текущей комнаты (см. 7й шаг) в котором мы переместим объект к подножию дерева и вернем false. Тем самым, дав стандартной библиотеке сообщить игроку о том, что предмет выброшен.

Глава 12

Двери и объекты в/на которые можно заходить

Перемещение с использованием сторон света может показаться неестественным, но оно вполне удобно. Тем не менее, бывают ситуации когда необходимо уметь заходить в/на объекты сцены, проходить в двери или перемещаться по миру используя какие-то объекты (например: подойти к скале).

На самом деле, в отличие от STEAD3 API в МПЗ игрок может заходить внутрь объектов или на них. Для этого у объекта должен быть задан атрибут enterable.

Если enterable задан, и объект является container или supporter, то вы можете спокойно попасть в/на него. Например:

```
obj {
    -”стол”;
}:attr ’enterable,supporter’;
```

Этого достаточно, чтобы игрок мог выполнить “залезть на стол”.

Когда герой попадает внутрь или на объект, МПЗ учитывает зоны видимости для того, чтобы описать сцену. Так, находясь на supporter, скорее всего вы будете в состоянии видеть комнату в которой находитесь, но если это непрозрачный контейнер – нет.

Если вы хотите, чтобы ситуация нахождения игрока в/на объекте отражалась как то иначе (по умолчанию эта информация отображается в заголовке сцены), вы можете задать свойство inside_dsc:

```
obj {
    -”стол”;
```

```
    inside_dsc = [[Ты стоишь на столе.]];
}:attr 'enterable,supporter';
```

Событие входа в/на объект называется Enter. А выхода с/из объекта – Exit. Это значит, что вы можете добавлять before и after свойства любым объектам для изменения стандартной реакции библиотеки.

Если игрок идет по направлению к объекту, создается событие Walk. Например:

```
obj {
    -"скала";
    before_Walk = function(s)
        move(pl, 'скала')
    end;
}
```

Что если вы хотите определить несколько свойств-синонимов? Допустим, мы хотим отработать команду "идти к скале" и "войти в скалу" одинаково? В таком случае воспользуйтесь следующей записью:

```
obj {
    -"скала";
    ['before_Walk,Enter'] = function(s)
        move(pl, 'скала')
    end;
}
```

Как уже было сказано, при обработке события вызывается в том числе свойства комнат. При этом в качестве параметра передается само событие:

```
room {
    nam = 'main';
    ['before_Walk,Enter'] = function(s, w)
        p ("Событие: ", ev)
        p ("Попытка идти к/в ",w)
        return false
    end;
}
```

Если вы при этом попробуете пойти на какую то сторону света, то увидите сообщение, что игрок попытался идти в объект ‘@compass’. Дело в том, что все перемещения по сторонам света реализованы через Enter и Walk этого объекта. Если вам нужно определить направление движения, воспользуйтесь mp:compass_dir:

```
room {
    nam = ’main’;
    [’before_Walk,Enter’] = function(s, w)
        if mp:compass_dir(w) == ’n_to’ then
            pn (”Игрок пытается идти на север.”)
        end
        p (”Событие: ”, ev)
        p (”Попытка идти к/в ”,w)
        return false
    end;
}
```

Чаще всего для переходов лучше всего использовать двери. Надо понимать, что дверь – это абстрактное понятие. Она не обязательно должна выглядеть как дверь. Просто это объект, который служит для перемещения игрока в другую комнату. Дверь создать очень легко:

```
door {
    -”дверь,дверца”;
    door_to = ”комната”;
}: attr ’open’
```

Обратите внимание на атрибут open. В закрытую дверь нельзя войти.

Написав “идти в дверь” игрок может войти в дверь и переместиться в локацию “комната”.

door_to может быть функцией, тогда эта функция будет вызвана при попытке войти в дверь. Функция может вернуть локацию, в которую нужно осуществить переход. Тем самым, легко делать двусторонние двери:

```
door {
    -”дверь,дверца”;
    door_to = function(s)
        if here() ^ ’улица’ then
```

```
            return ”комната”;
        else
            return ”улица”
        end
    end;
    found_in = { ’улица’, ’комната’ };
}: attr ’open’
```

Чтобы дверь можно было открывать и закрывать, добавьте openable атрибут:

```
...
    found_in = { ’улица’, ’комната’ };
}: attr ’open,openable’
```

Теперь игрок может открывать и закрывать дверь. А если нужно сделать дверь, которую можно запирать и отпирать?

```
...
    with_key = ’ключ’;
    found_in = { ’улица’, ’комната’ };
}: attr ’openable,lockable,locked’
```

lockable – дверь можно отпирать чем-либо. locked – дверь в данный момент заперта. with_key – свойство, которое определяет ключ, которым может быть открыта дверь.

События, которые относятся к рассмотренным действиям:

- Open: открыть;
- Close: закрыть;
- Lock: запереть чем либо (два параметра, сам объект и чем закрываем);
- Unlock: отпереть чем либо.

Вы можете переопределять поведение библиотеки, как обычно:

```
    before_Unlock = function(s, w)
        if w ^ ’ключ’ then
            return false -- пропустить
        end
        p [[Что то не подходит.]] -- остановить цепочку
    end;
}: attr ’openable,lockable,locked’
```

Вы можете описывать объект в сцене с помощью свойства dsc, однако для дверей вы можете задать отдельное описание для случая закрытой и открытой двери:

```
    when_locked = [[Здесь есть закрытая дверь.]];
    when_open = [[Дверь открыта.]];
}: attr ’openable,lockable,locked’
```

Да, мы чуть не забыли защититься от того, чтобы игрок не мог взять дверь с собой и использовать ее как телепортатор. Добавим атрибут static:

```
    when_locked = [[Здесь есть закрытая дверь.]];
    when_open = [[Дверь открыта.]];
}: attr ’openable,lockable,locked,static’
```

Двери могут быть связаны с направлением. Для этого, просто задайте в свойстве-направлении ту дверь, с которым направление связано:

```
room {
    n_to = ’дверь’;
}
```

Тогда и при заходе в дверь и при переходе на север игрок попадет в локацию, в которую ведет дверь. (Если она не закрыта).

Глава 13

Свет

в МПЗ (в отличие от Inform 6) все комнаты по умолчанию имеют свет. Если вы хотите создать комнату без света, задайте в атрибутах ~light.

```
room {
    title = -”комната”;
    nam = ’main’;
}: attr ’~light’
```

В такой комнате игрок не будет видеть ничего, кроме сообщения о кромешной тьме. Из такой комнаты игрок может только вернуться назад (или включить каким то способом свет).

Если вам не нравится сообщение по умолчанию, воспользуйтесь dark_dsc:

```
room {
    title = -”комната”;
    nam = ’main’;
    dark_dsc = [[Хоть глаз выколи!]];
}: attr ’~light’
```

Если вы хотите, чтобы во тьме какие то предметы были видимы, дайте им свет (задав атрибут light). Если light задан у игрока (std.me() или pl), то все считается освещенным.

Глава 14

Выключатели

Если у объекта задан атрибут switchable, то предполагается что такой объект можно включать и выключать. Эти действия соответствуют событиям SwitchOn и SwitchOff. Признак включенного объекта это атрибут ‘on’. Давайте добавим выключатель в темную комнату.

```
room {
    title = -”комната”;
    nam = ’main’;
    obj = {’выключатель’};
}:attr ’~light’

obj {
    -”выключатель|свет”;
    nam = ’выключатель’;
    after_SwitchOn = function(s)
        here():attr’light’ -- дали свет комнате
        pl:need_scene(true) -- свет! показать сцену!
        return false -- дать библиотеке завершить дело
    end;
    after_SwitchOff = function(s)
        here():attr’~light’
        pl:need_scene(true)
        return false
    end;
} : attr ’light,switchable,static’;
```

Аналогично дверям, у выключателей есть свойства when_on и when_off, которые позволяют задавать описатель в сцене для разных состояний выключателя.

Глава 15

Катсцены

Для вывода текстовых сюжетных вставок или диалогов удобно воспользоваться специальным типом сцены: cutscene.

cutscene выдает текст порциями. Для продолжения игрок может нажать ввод или ввести “дальше”. Остальные глаголы в этот момент не действуют.

```
cutscene {
    text = {
        ”Это пример катсцен.”;
        ”Из трех страниц.”;
        ”Это последняя...”;
    };
}
```

Если в катсцене присутсвует свойство next_to, то это так комната в которую герой переместится после того как прочтет весь текст.

Если такого свойства нет – герой переместится в ту комнату, в которой был до этого.

Глава 16

Диалоги

Существует несколько способов организовать общение в игре. Самый простой, с использованием глагола “поговорить с”, которому соответствует событие Talk.

При этом, если у персонажа задано свойство talk_to, то произойдет переход на эту комнату, диалог или cutscene. Например:

```
dlg {
    nam = 'dlg1';
    phr = {
        [[-- Привет, главный герой.]];
        {
            'Что?',
            [[-- Ничего!]],
        },
        {
            'Поговорим!',
            function() p [[-- А мне не хочется разговаривать!]]; walkout() end;
        },
    };
}

obj {
    -"персонаж";
    nam = 'npc';
    talk_to = 'dlg1';
```

```
}

room {
    nam = -”комната”;
    obj = { ’npc’ };
}

function init()
    pl.room = ’комната’
end
```

Диалоги STEAD3 работают обычным образом в МПЗ, только выбор вариантов осуществляется с клавиатуры.

Другой вариант это глаголы Tell, Ask, AskFor и AskTo. Сказать, спросить про, попросить у, сказать что то. При этом в обработчики приходят строка, которую ввел пользователь в нормализованной форме (строчные, обычно без буквы ё).

Таким образом, можно реагировать на какие-то фразы:

```
[’before_Ask,AskTo,AskFor,Tell’] = function(s, w)
    if w:find ”привет” then
        p ”Привет тебе, тоже!”
        return
    end
    return false
end
```

Однако, как видно, в русском языке разница между событиями весьма условна. Поэтому этот способ не является рекомендованным. Если вы используете его, лучше всегда делайте свойства-синонимы. К счастью, AskTo и AskFor по умолчанию превращаются в событие Ask. Поэтому достаточно определить реакцию на Ask и Tell:

```
[’before_Ask,Tell’] = function(s, w)
    if w:find ”привет” then
        p ”Привет тебе, тоже!”
        return
    end
    return false
end
```

Глава 17

Сложные объекты

Вы можете создавать сложные объекты, состоящие из нескольких частей. Когда объект состоит из нескольких частей, вы не можете взять составные части. Например.

```
obj {
    -”девушка,девиц*,красавиц*”;
    nam = ’девушка’;
    obj = {
        obj {
            -”глаза,глаз*”;
            description = [[Глаза цвета моря!]];
        };
        obj {
            -”нос”;
            description = [[Прелестный носик!]];
        };
    }
}
```

При попытке взять глаза девушки, МПЗ напишет “Глаза являются частью девушки.”.

Как видим, составные объекты просто включают в себя свои “запчасти”, без определения container или supporter.

Глава 18

Классы

Иногда возникает необходимость создавать в игре однотипные объекты. У этих объектов могут быть какие-то определенные атрибуты и/или свойства. В МПЗ в таких случаях вы можете создавать классы.

Например:

```
window = Class {
    word = -”окно”;
    description = ”В окне белым-бело.”;
    before_Listen = ”Ты слышишь как воет ветер за окном.”;
}:attr ’static’

window {
    nam = ”окно1”;
    found_in = ’scene1’;
}
window {
    nam = ”окно2”;
    found_in = ’scene2’;
}
```

В текущей реализации на классы накладываются следующие ограничения (которые, вероятно, будут устранены в следующих версиях).

1) Класс не может содержать в себе определение изменяемых переменных. Если вам нужна переменная, придется определить ее при создании экземпляра объекта.

```
window = Class {
    word = -”окно”;
    description = ”В окне белым-бело.”;
    before_Listen = function(s)
        p ”Ты слышишь как воет ветер за окном.”;
        s.listen = true;
    end
}:attr ’static’

window {
    listen = false; -- определили переменную в объекте, а не классе
    nam = ”окно1”;
    found_in = ’scene1’;
}
```

2) Если в классе какое-то свойство определено как функция (а не строка), то и в объектах класса эти свойства придется определять функцией (при необходимости переопределения этого свойства).

```
window = Class {
    word = -”окно”;
    description = ”В окне белым-бело.”;
    before_Listen = function(s)
        p ”Ты слышишь как воет ветер за окном.”;
        s.listen = true;
    end
}:attr ’static’

window {
    nam = ”окно1”;
    found_in = ’scene1’;
    before_Listen =  [[Так не получится переопределить свойство!]]
}

window {
    nam = ”окно1”;
    found_in = ’scene1’;
```

```
    before_Listen = function() p [[А так -- получится!]] end;
}
```

Вы можете наследовать одни классы от других:

```
window2 = Class({ -- наследуемся от окна
    before_Push = [[Скрипит!]];
}, window)
```

Глава 19

Список событий

Для более подробного изучения библиотечных событий, вы можете посмотреть файл mp-ru.lua

Событие	Описание
Walk	Переход по компасу или подход к объекту
Enter	Войти во что-либо
Exit	Выйти из чего-либо
Exam	Осмотреть
Search	Поиск внутри объекта
LookUnder	Поиск под объектом
Consult	Прочитать про что то в чем то, найти что то в книге и т.д.
Open	Открыть
Close	Закрыть
Unlock	Отпереть (чем либо)
Lock	Закрыть на ключ
Inv	Посмотреть инвентарь
Take	Брать
Drop	Выбросить
PutOn	Положить на
Insert	Положить внутрь
Remove	Извлечь что-то из чего-то
ThrowAt	Бросить в кого-то
Wear	Надеть
Disrobe	Снять с себя
SwitchOn	Включить
SwitchOff	Выключить
Eat	Есть
Taste	Попробовать
Drink	Пить
Push	Толкать
Pull	Тянуть
Transfer	Переместить что-то на/в что-то
Turn	Вращать

Событие	Описание
Wait	Ждать
Rub	Тереть
Sing	Петь
Touch	Гладить, трогать
Give	Отдать
Show	Показать
Burn	Жечь
WakeOther	Разбудить кого-либо
Wake	Проснуться
Kiss	Целовать
Think	Думать
Smell	Нюхать
Listen	Слушать
Dig	Копать (возможно, чем-то)
Cut	Резать (возможно, чем-то)
Tear	Разрывать, рвать, срывать
Tie	Привязать
Blow	Дуть
Attack	Напасть
Sleep	Спать
Swim	Плавать
Fill	Наполнить, налить
Jump	Прыгать

Событие	Описание
JumpOver	Перепрыгнуть
WaveHands	Махать руками
Wave	Махать чем-то
Climb	Взобраться
GetOff	Слезть
Buy	Купить
Talk	Поговорить
Tell	Сказать
Ask	Спросить про
AskFor	Попрость что-то у персонажа
AskTo	Попросить что-то персонажа
Answer	Ответить
Yes	Да
No	Нет

Глава 20

Вспомогательные свойства life

В стандартной библиотеке реализовано множество глаголов. Часть событий относится к взаимодействию с персонажами. Как уже было сказано, по умолчанию в МПЗ персонажем считаются одушевленные сущности или те объекты у которых задан атрибут animate.

При взаимодействии с персонажами вызываются специальные свойства life персонажа. Например, если мы отдадим предмет персонажу, то кроме цепочки события Give будет вызвано свойство life_Give у персонажа. При этом в качестве параметра будет передан предмет.

Например:

```
obj {
    -”старик,дед,дедушка”;
    life_Give = function(s, w)
        if w ^ ”дробовик” then
            p [[Спасибо, сынок!]]
            return
        end
        return false -- стандартный ход события
    end;
}
```

Ниже перечислены события, для которых вызываются life свойства:
ThrowAt, Give, Show, WakeOther, Kiss, Attack, Talk, Tell, Ask, Answer

Глава 21

Псевдо-событие Receive

Когда игрок перемещает объект: кладет его на другой объект или помещает внутрь другого объекта, то для удобства движок вызывает свойства before_Receive и after_Receive для объекта, который выступает в роли приемника.

Ниже перечисляются события, которые порождают Receive псевдо-события:

Insert, PutOn

В качестве примера рассмотрим объект, который может вести себя и как supporter и как container.

```
obj {
    -”аквариум”,
    nam = ’аквариум’,
    before_Receive = function(s, w)
        if mp.xevent == ’PutOn’ then
            move(w, ’#поверх’)
            p (”Ты кладешь ”, w:noun’вн’, ” на крышку аквариума.”)
        else
            return false
        end
    end;
    obj = {
        ’рыбка’,
        obj {
            -”аквариум”;
```

```
            nam = '#поверх';
            dsc = function(s)
                mp:content(s)
            end;
        }:attr 'supporter';
    }
} : attr 'container,transparent,open'
```

Глава 22

Псевдо-событие ThrownAt

Когда мы бросаем предмет в персонажа или объект, кроме события ThrowAt создается псевдо-событие ThrownAt. Свойство before_ThrownAt вызывается о объекта, в который осуществлен бросок.

Глава 23

Псевдо-событие LetGo

Когда предмет перемещается и покидает свое место пребывания (supporter ил container), у объекта, который содержал предмет, вызываются свойства before_LetGo и after_LetGo. Это позволяет контролировать событие извлечение предметов более удобным способом, чем слежение за каждым из предметов.

Глава 24

Список дополнительных методов объектов МПЗ

МПЗ добавляет некоторые методы у объектов, которые расширяют стандартное STEAD3 API.

Для вызова метода используется запись: объект:метод(параметры).

Если объект задан по имени, то: _"имя":метод(параметры)

Метод	Описание
attr "строка"	Задать/снять атрибуты
has "атрибут"	Проверить наличие атрибута
hasnt "атрибут"	Проверить отсутствие атрибута
daemonStart	Добавить в список фоновых событий
daemonStop	Убрать из списка фоновых событий
once	Выполнить условие только 1 раз
noun(склонение)	Видимое имя предмета в требуемом склонении
Noun(склонение)	То же самое, но с заглавной буквы
it(склонение)	Информация о предмете в виде местоимения
inside(где)	Находится ли этот предмет внутри другого?
hint(уточнение)	Проверка свойств и уточнений слова, например, if w:hint'мн' then
for_plural(функция)	Вызвать функцию для всех объектов, попавших в выборку

Глава 25

Список переменных МПЗ

Переменные МПЗ доступны как mp.переменная. Например: mp.autohelp = false

Ниже перечислены все переменные с кратким описанием.

Переменная	Описание
mp.first	Первый объект события (обычно предмет действя)
mp.second	Второй объект события (если есть)
mp.event	Имя события
mp.xevent	Имя события до смены события (для вспомогательных событий)
mp.errhints	Показывать ли при ошибках ввода подсказки [true]
mp.detailed_inv	Детализированный инвентарь [false]
mp.autohelp	Интерактивные подсказки [false]
mp.autohelp_limit	Максимальное число подсказок [1000]
mp.autohelp_noverbs	Не показывать все доступные глаголы при пустом вводе [false]
mp.compl_thresh	При каком минимальном количестве символов в слове предоставлять автодополнение [0]
mp.togglehelp	Включать подсказки по f1 [true]
mp.autocompl	Автодополнение по TAB [true]
mp.cursor	Вид курсора [fmt.b("|")]
mp.prompt	Приглашение ["> "]
mp.clear_on_move	Очищать экран при переходах [true]

Глава 26

Список методов и функций МПЗ

Методы доступны как: mp:метод(). Ниже перечислены все методы с кратким описанием.

Кроме того, для простоты в МПЗ определены некоторые глобальные функции.

Метод	Описание
mp:clear()	Очистить вывод окна МП
mp:content(w)	Вывести содержимое объекта
mp:xaction(ev, …)	Сменить цепочку на новое событие
mp:subaction(ev, …)	Выполнить новую цепочку без прерывания текущей
mp:runmethods(тип, метод, …)	Выполнить свойство (напр. mp:runmethods('before', 'LetGo', wh, w))
mp:runorval(объект, имя, …)	Взять значение или выполнить функцию
mp:move(что, куда, [force])	Перемещение объекта
mp:inside(что, где)	Проверка на то, что объект находится внутри объекта
mp:thedark()	Находится ли игрок во тьме
mp:visible_scope(где)	Найти максимальный охват (объект) зоны видимости
mp:trace(где, функция)	Трассировка объекта наружу (проход по всем родителям)
mp:offerslight(что)	Освещен ли объект
mp:check_touch()	Доступны ли предметы (находсятся в зоне видимости?) mp.first и mp.second? Вернет true, если надо прервать цепочку.
mp:check_held(что)	В руках предмет? Если нет – попытка взять. Вернет true, если надо прервать цепочку.
mp:check_worn(что)	Надет предмет? Если да – попытка снять. Вернет true, если надо прервать цепочку.
mp:check_live(что)	Персонаж? Если да – стандартное сообщение. Вернет true, если надо прервать цепочку.
mp:compass_dir(что)	Проверка на направление компаса
mp:it(что, склонение)	Информация о предмете в виде местоимения
mp:It(что, склонение)	То же, но с заглавной буквы

Функция	Описание
inside(что, где)	Аналог mp:inside
move(что, куда)	Аналог mp:move(что, куда, true)
parent(что)	Аналог where()
content(что)	Аналог mp:content()

Глава 27

Список всех атрибутов

После краткого обзора, приведем список всех свойств МПЗ.

Свойство	Описание
animate	Признак персонажа.
clothing	Одежда. Ее можно надевать и снимать.
concealed	Невидимый объект (но действующий).
container	Содержит другие объекты
cutscene	Признак cutscene
door	Признак двери (ставится автоматически классом door)
edible	Можно есть (съедобно)
enterable	Можно заходить в/на объект
light	Источник света
luminous	Светящийся объект (видимый в темноте, но не являющийся источником света)
lockable	Можно запирать и отпирать
locked	Заперто на ключ
moved	Объект перемещался (с помощью move)
on	Включено
open	Открыто
openable	Можно открывать и закрывать
scenery	Декорация, не выводить описание объекта
static	Статичный объект
supporter	На объект можно что то класть
switchable	Можно включать/выключать
transparent	Прозрачный
visited	Сцена была посещена (ставится автоматически)
worn	Надето

Глава 28

Список всех свойств

Надо иметь в виду, что кроме свойств МП3 остались свойства STEAD3 API, например: onexit/onenter, exit/enter.

Свойство	Описание
after_xxxxx	Действия после обработки события
before_xxxxx	Действия до обработки события
cant_go	Сообщение, когда игрок пытается двигаться в недоступном направлении
capacity	Как много объектов может содержать supporter, container, игрок или персонаж
compass_look	Вызывается у комнаты при попытке осмотреть сторону света. На вход приходит сторона света в виде ‘n_to’, ‘s_to’ и т.д.
d_to	Комната или дверь для перехода “вниз”. Или функция.
daemon	Функция, которая выполняется в конце каждого хода. DaemonStart(w)/DaemonStop(w) для управления
dark_dsc	Описание для темной комнаты
default_Event	Событие, которое создаётся при вводе объекта без глагола. По умолчанию – “Exam”.
description	Длинное описание объекта (при осмотре)
dsc	Для комнаты - описатель комнаты. Для предметов - то, что будет выведено после описания комнаты
door_to	Куда ведет дверь. Может быть функцией.
e_to	Комната или дверь для перехода “восток”. Или функция.
each_turn	Функция, которая выполняется каждый ход, когда предмет в зоне доступа
found_in	Список мест, где находится объект или функция, возвращающая true тогда, когда объект д.б. в сцене
gfx	В комнате определяет графическое изображение, которое будет внедрено в текст.

Свойство	Описание
in_to	Комната или дверь для перехода “внутрь”. Или функция.
init_dsc	Как dsc, но только если объект не перемещался (первоначальное описание)
inside_dsc	Показывается когда игрок внутри этого объекта
inv	Как выглядит предмет в инвентаре
life_xxxx	Действия для ситуаций затрагивающих персонажей. Например: life_Give
n_to	Комната или дверь для перехода “север”. Или функция.
ne_to	Комната или дверь для перехода “северо-восток”. Или функция.
nw_to	Комната или дверь для перехода “северо-запад”. Или функция.
out_to	Комната или дверь для перехода “наружу”. Или функция.
s_to	Комната или дверь для перехода “юг”. Или функция.
scope	Список или функция, добавляющий объекты в список доступности.
se_to	Комната или дверь для перехода “юго-восток”. Или функция.
sw_to	Комната или дверь для перехода “юго-запад”. Или функция.
talk_to	Комната, диалог или cutscene для перехода по событию Talk (поговорить с).
u_to	Комната или дверь для перехода “вверх”. Или функция.
w_to	Комната или дверь для перехода “запад”. Или функция.
when_closed	Описание объекта в сцене, когда он закрыт.
when_open	Описание объекта в сцене, когда он открыт.
when_on	Описание объекта в сцене, когда он включен.
when_off	Описание объекта в сцене, когда он выключен.
with_key	Ключ, которым может быть открыт запертый объект.
word	Словарные слова объекта (word = -“что то”)

Глава 29

Создание своих глаголов и расширение существующих

В стандартной библиотеке реализованы различные глаголы. Но что делать, если требуемого глагола нет? Прежде чем ответить на этот вопрос, давайте рассмотрим как именно создаются глаголы в библиотеке. В качестве примера, возьмем события Wave и WaveHands.

Итак, определение глагола:

```
Verb {
    ”#Wave”, -- тег глагола, может отсутствовать
    ”мах/ать,помах/ать,помаш/и”, -- список слов
    ”WaveHands”, -- шаблон 1
    ”~ руками : WaveHands”, -- шаблон 2
    ”{noun}/тв,held : Wave” -- шаблон 3
}
```

Итак, для создания глагола используется Verb {}. Определение глагола состоит из следующих частей:

1) Необязательного тега. Тег позволяет выключать глагол или расширять его, поэтому все библиотечные глаголы содержат тег.

2) Список слов, которые соответствуют глаголу. Слова пишутся через запятую. Первое слово в списке считается основным (используется при подсказках). Допустима запись, содержащая /. Так, мах/ать – запись, которой удовлетворят все слова, которые начинаются на мах. Также поддерживаются префиксы. Например:

"[|по]мах/ать,маш/и"

Обратите внимание на запись [<пробел>|по], она означает что и махать (пустой префикс) и помахать (префикс "по") – допустимы.

3) Список шаблонов. Шаблоны - это варианты глагола. Шаблон состоит из двух частей: шаблона параметров и события. Эти части разделяются символом ":". Если у варианта нет параметров – то задается только событие.

Рассмотрим шаблоны нашего примера.

Шаблон 1 соответствует ситуации, когда игрок ввел просто "махать" или "помаши". Аргументов нет, поэтому в шаблоне задан только шаблон события: WaveHands. То-есть, будет порождено событие WaveHands.

Шаблон 2 соответствует ситуации, когда игрок написал "помахать руками". Символ "~" в начале шаблона говорит о том, что шаблон является *вспомогательным*. Что это значит? Это значит, что помахать руками мы можем в форме шаблона 1, но шаблон 2 – это просто добавление еще одного варианта. На самом деле, если вы не собираетесь использовать графические подсказки в своей игре, вы можете не помечать символом "~" дополнительные глаголы. Эта информация используется для разгрузки меню. Дополнительные шаблоны не будут добавляться в меню.

Шаблон 3 содержит запись вида: {noun}/тв,held Как вы уже вероятно догадались, данная запись означает "слово в винительном падеже". Запись {} это так называемый token генератор. Это функция, которая определена как mp.token.[имя генератора] и которая возвращает список из возможных вариантов.

Генератор mp.token.noun генерирует список всех доступных объектов в заданном падеже. held - это уточнение, которое используется МПЗ для графической помощи, а также для формирования событий. Существуют следующие уточнения:

- held - объекты в инвентаре
- scene - объекты на сцене
- container - объекты container
- supporter - объекты supporter
- enterable - объекты в которые можно зайти
- edible - съедобно

- inside - container или supporter
- live - персонажи
- holder - объект содержит в себе второй (или первый) параметр

Теперь рассмотрим как реализованы обработчики этих событий:

```
-- файл mplib.lua
mp.msg.WaveHands = {}
function mp:WaveHands()
    p (mp.msg.WaveHands.WAVE)
end

mp.msg.Wave = {}
function mp:Wave(w)
    if mp:check_touch() then
        return
    end
    if mp:check_held(w) then
        return
    end
    p (mp.msg.Wave.WAVE)
end
-- файл mp-ru.lua

--”помахать”
mp.msg.WaveHands.WAVE = ”{#Me} глупо {#word/помахать,прш,#me} руками.”
mp.msg.Wave.WAVE = ”{#Me} глупо {#word/помахать,прш,#me} {#first/тв}.”
```

Как видим, функции довольно простые, но вот сообщения mp.msg представляют интерес.

Здесь используется запись, содержащая {#} - это называется сокращением. Каждое сокращение реализовано в виде функции: mp.shortcut.[имя shortcut] и возвращает текстовую строку.

Существует множество сокращений, которые вы можете посмотреть в mp-ru.lua. Перечислим основные (каждое сокращение м.б. задано со строчной или заглавной буквы):

- #me – главный герой
- #first – первый объект события
- #second – второй объект события
- #firstit – первый объект события, местоимение
- #secondit – второй объект события. местоимение
- #where – местоположение героя
- #if_has – условное сокращение (проверка на атрибут)
- #if_hint – условное сокращение (проверка на словарное уточнение)
- #word – слово из словаря с требуемыми уточнениями, в качестве уточнений могут быть указаны #me, #first, #second – тогда слово будет согласовано с этим существительным

Как вы уже догадались, вы можете писать свои сокращения. Но это тема для отдельной документации.

На самом деле, в своей игре вы можете не использовать сокращений вовсе:

```
function mp:WaveHands()
    p [[Ты глупо машешь руками.]]
end
function mp:Wave(w)
    if mp:check_touch() then
        return
    end
    if mp:check_held(w) then
        return
    end
    p (”Ты глупо машешь ”, w:noun’тв’, ”.”)
end
```

Однако сокращения хорошо использовать для библиотечных функций, ведь тогда вывод будет адекватным для разного рода, числа и лица главного героя.

Мы рассмотрели глагол с одним существительным. Ситуация несколько усложняется, когда существительных два. Рассмотрим, например, глагол "отдавать":

```
Verb {
    "#Give",
    "дать,отда/ть,предло/жить,предла/гать,дам,даю,дадим",
    "{noun}/вн,held {noun}/дт,live : Give",
    "~ {noun}/дт,live {noun}/вн,held : Give reverse",
}
```

Тут уже вам должно быть все понятно, кроме записи : Give reverse. На самом деле, она означает просто обратный порядок объектов. То-есть обе формы записей:

отдать яблоко принцессе

отдать принцессе яблоко

Создадут одинаковое событие Give(яблоко, принцесса). Именно для этого указано слово reverse во втором шаблоне.

В шаблонах могут встречаться также конструкции "|", означающее "или". Например:

```
"{noun}/вн на|в|во {noun}/вн : Transfer",
```

Итак, с полученной информацией вы должны смочь создавать свои внутриигровые глаголы. В демонстрационных играх "Вильгельм Телль" и "Алиса" есть примеры таких глаголов.

Существует возможность *расширить* уже существующий глагол несколькими шаблонами. Для этого используется VerbExtend:

```
VerbExtend {"#Exam",
        "{noun}/вн,scene в {noun}/пр,2,scene : Reflect",
        "~ на {noun}/вн,scene в {noun}/пр,2,scene : Reflect",
        "~ в {noun}/пр,2,scene на {noun}/вн : Reflect reverse"
}
```

Мы расширили существующий в библиотеке глагол.

Для удаления глагола используйте VerbRemove:

```
VerbRemove '#Exam'
```

ВНИМАНИЕ! Глаголы нельзя модифицировать динамически! Вы можете создавать и менять их только в общем контексте вашей игры!

При рассмотрении глаголов не была упомянута возможность иметь свои наборы глаголов для комнат и/или игроков.

На самом деле, при добавлении/изменении/удалении глагола вы можете указать второй параметр – игрока или комнату. На самом деле, cuscene реализована как комната с единственным глаголом:

```
Verb ({'#Next', "дальше", "Next" }, mp.cutscene)
```

Глава 30

Интерактивные подсказки

Не смотря на то, что МПЗ это настоящий парсер, в нем остается возможность использовать интерактивные подсказки в стиле МЕТАПАРСЕРА 2.

Однако, число глаголов по-умолчанию настолько велико, что вам нужно приложить некоторые усилия, чтобы игра могла играться комфортно в режиме с подсказками.

Во первых, вам следует определить минимальный базис глаголов вашей игры:

```
game.hint_verbs = { "#Exam", "#Walk", "#Push", "#Take",
    "#Drop", "#Search", "#Give", "#Touch" }
```

В определенных комнатах, вы можете определить дополнительные глаголы к базису:

```
room {
    nam = "чулан";
    hint_verbs = { "#Burn" };
}
```

Кроме того, существует возможность задать глаголы в комнате, исключая все остальные:

```
room {
    nam = "вопрос";
    hint_verbs_only = { "#Yes", "#No" };
}
```

Если условие необходимости глагола более сложное, вам следует определить это условие в виде функции:

```
VerbHint ( ”#Burn”, function(v)
    return have ’спички’
end)
```

Вы можете задавать функцию hint_noun у объекта, для контроля появления его в подсказках:

```
obj {
    -”фонарь,фонарик”;
    hint_noun = function(s)
        return here()^’чердак’
    end -- подсказка только на чердаке
    ...
}
```

Для исключения подскзок для этого объекта, задайте: hint_noun = false.

Чтобы включить подсказки используйте:

```
mp.autohelp = true
```

Для разрешения динамического включения/выключения подсказок клавишей F1, воспользуйтесь:

```
mp.togglehelp = true
```

Глава 31

Отладка

Если INSTEAD запускается с опцией -debug, то в метапарсере становятся доступными некоторые отладочные команды.

Команда	Описание
_трассировка	Включает трассировку событий
_дамп	Дамп состояния окружающего мира
_слово СЛОВО[_свойства]	Дамп слова из словаря (напр. елка_вн)
автоскрипт	Выполнить команды из файла autoscript

Функция автоскрипта (доступна также по клавише F6 в режиме -debug) позволяет разрабатывать сценарии тестирования.

Типовой сценарий использования функции автоскрипт выглядит так:

1. Вы проходите игру с включённым режимом транскрипт (команда “транскрипт”), при этом все события игры записываются в файл logXXXX.txt

2. Далее, на основе log-файла вы создаёте файл autoscript, который содержит только команды прохождения, и это становится системой тестирования.

3. При необходимости, вы запускаете игру заново, прогоняете автоскрипт и убеждаетесь, что игра проходима. При этом лог прохождения записывается в logXXXX.txt и вы можете сравнить его с эталонным.

Глава 32

Послесловие

В первой редакции документации тут был написан текст, который содержал размышления о творчестве и об актуальности игр с текстовым вводом в 2018 году.

Но я вовремя понял, что все это – лишнее.

Достаточно того, что вы сейчас добрались до последних абзацев и готовы написать свою историю. :) Какая она будет, эта история, зависит только от вас.

Творчество (и искусство) – не продукт и не товар. Хотя, кажется, весь мир пытается убедить нас в обратном. Творчество не нуждается в оправдании или обосновании.

Так что в этом небольшом послесловии я могу только пожелать вам удачи и…

До встречи!