NICK MOFFITT

MAKING STORY GAMES WITH PUNYINFORM

Adventures in Text

> But if you could talk straight to that imagination and cut out all the senses, then... it would be impossible to ignore it. You couldn't say, "Oh, that's just an image of a dragon." That would **be** a dragon. And if there was some kind of technology which could enable you to talk straight to the imagination...
>
> Well, there is: it's called text. And it's been around several thousand years.
>
> And I have seen people **leap out of their chairs** when a line has said in front of them, "There is an immense fire-breathing dragon here."
>
> Richard Bartle, interviewed in *Get Lamp*[1]

[1] https://archive.org/details/getlamp_bartle

In the 21st century, computer games are capable of astonishing graphics. We can generate 3D virtual worlds to explore, and many of these are works of art. But even the best equipment pales in comparison to a reader's imagination.

Whether you call them "text adventures" or "interactive fiction"[2], the ability for your mind to fill in the most fantastic of details can make simple typed-in text the most direct of interfaces. Story games take text in from your keyboard, and use that to decide what text to send back to you.

[2] There is some debate over the meaning of "Interactive Fiction". Some authors used the term to try and suggest that they were doing something more *important* than simple adventure games. And some use it now as a large category that includes works that have no "adventuring" inside. To avoid conflict, we will call them "story games" in this book.

Players control story games by typing imperative commands, such as `go north` or `take the key`. More advanced games may allow you to `set the controls for the heart of the sun` or `please put the landlord hat on the landlord`. The types of actions needed to play a game are limited, but it can respond with paragraphs of description that set the scene and lead players to imagine the world their character inhabits.

The PunyInform Accelerator

> Theorising that one could time travel within his own lifetime, Doctor Sam Beckett stepped into the Quantum Leap accelerator and vanished... He woke to find himself trapped in the past, facing mirror images that were not his own, and driven by an unknown force to change history for the better.
>
> The opening narration to the television serial, *Quantum Leap*

Playing a story game can be slightly disorienting, at first. Players find themselves inhabiting the perspective of a character they have not had time to get to know. A game may begin *in medias res*, which means "dropped into the middle of things". Although you may react with a surprised "oh boy...", do not give up: you can always `restart` a game to try again, or `save` and `restore` attempts to play through a tricky spot.

Building a story game can feel a little *in medias res* as well[3], but this book is here to help you. To begin with, you will need three tools:

[3] "Oh boy..."

1. A "Z-Machine" interpreter, which plays story games in the format invented by Infocom in the 1970s.
2. The Inform compiler, which turns source code you write into a game playable on the interpreter.
3. The PunyInform library, which does the hard part of writing Inform code to handle things that are common to nearly all story games.

To install these tools on your computer, please refer to the reference card for your specific machine. But if you are just looking to get started quickly, there is an easier option.

Borogove

The easiest way to get started is to use an online development tool called Borogove. Just follow these steps:

1. Go to https://borogove.app
2. Under the `Inform 6` card, select a new project using PunyInform[4].
3. When you're ready to build and play your game, click the button marked `Go` and explore it in the interpreter.

[4] As of this writing, the option says `PunyInform 3.6`, but there may be a new version out by the time you read this.

If you're just looking to explore, there is also an option to begin with a `Library of Horror` example game. Feel free to play around with it before moving on to the next chapter. We'll be writing our own story games, next.

Our First Leap

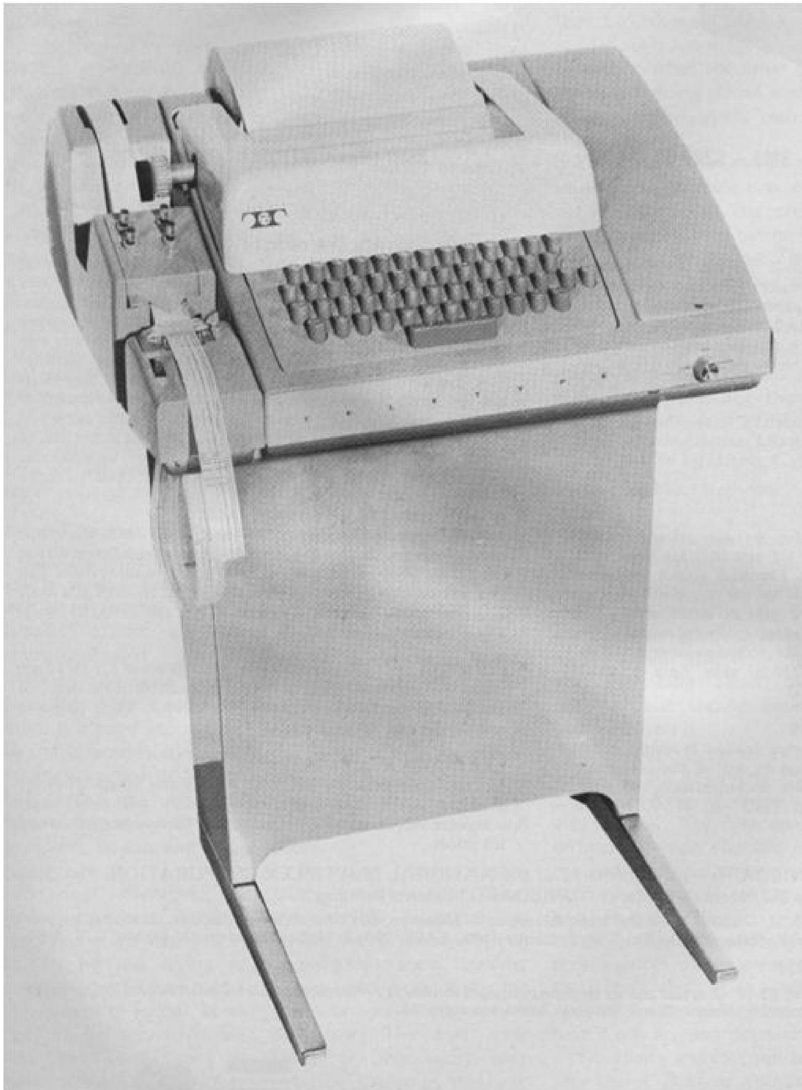

Figure 1: From the 1940s to the 1970s, people used computers through **teletypes**. These were noisy mechanical typewriters that sent binary codes over wires or radio waves. This was the type of device that Will Crowther used to write a game called *Colossal Cave Adventure* on a PDP-10 computer, while working at a company called BBN.

Some things to try:

- `look at self`
- `look on the teletype`
- `take the transcript`
- `get office`
- `get teletype`

- `inventory`
- `drop transcript`
- `put the transcript on the teletype`
- `get all`
- `go east`
- `read transcript`

Your first mission is to go back in time to 1975, and read a transcript of what we believe is the very first story game ever made with a computer. To do this, open a new PunyInform project and replace any source code you may already have with the following:

```
Constant Story      "BBN";
Constant Headline   "^Mission for Colossal Cave.^";
Constant INITIAL_LOCATION_VALUE = Office;

Include "globals.h";
Include "puny.h";

[Initialise;
  "Stepping into the PunyInform Accelerator, you vanish into
    another life, and another time...";
];

Object Office "Your Office"
with
  description "This is the quiet office where you wrote
    Colossal Cave Adventure, while coming to terms with the
    divorce that estranged you from your caving group.",
has light;

Object -> teletype "ASR-33 TeleType"
with
  name 'asr-33' 'asr33' 'teletype',
  description "This is a model ASR-33 teletype. It's an
    electric typewriter, connected by wires to the PDP-10
    down the hall.^It has a paper tape puncher/reader to
    store data for later, and everything the computer has
    displayed is left printed in block capitals on an
    endless spool of paper.",
has static supporter;

Object -> -> transcript "game transcript"
with
  name 'printout' 'game' 'transcript',
  description "This is what you came for: a printout of an
    early game of Colossal Cave Adventure your kids played
    with you. It is currently at a particularly descriptive
    section:^~YOU ARE IN A SPLENDID CHAMBER THIRTY FEET
    HIGH. THE WALLS ARE FROZEN RIVERS OF ORANGE STONE. AN
```

```
    AWKWARD CANYON AND A GOOD PASSAGE EXIT FROM EAST AND
    WEST SIDES OF THE CHAMBER.~",
  after [; Examine: deadflag = GS_WIN; ];
```

Go ahead and play around with the game for a bit. There is only one room and a couple of objects, so you can see it all very quickly. We've included some suggestions for things to try in the sidebar on the right.

Once you've explored how the game works, it's time to look at how we made it.

Source Code

The text you pasted in is written in the Inform 6 language[5]. It has a lot of special words, which are shown in **bold**. There is also lots of text the game will listen for or print out, which is underlined.

[5] Pay close attention to the punctuation and capitalisation: the little details can make a big difference! Look for semicolons (`;`), commas (`,`) and both types of inverted commas (`'` and `"`).

Let's look at each section, and find out what it does.

```
Constant Story      "BBN";
Constant Headline   "^Mission for Colossal Cave.^";
Constant INITIAL_LOCATION_VALUE = Office;
```

The first section defines **constants**[6], which link names to values that cannot change while the game runs. Here we name the game, and tell it to start in the `Office`.

[6] But don't forget that `Constant` must have a capital `C`!

Did you notice the use of `^` in the `Headline` string? Those are turned into **newlines**, so the `Headline` is printed on a line of its own. We'll talk more about this later.

When we surround text in `"` characters, we call that a **string**. **Strings** are pieces of text that we can print out during the game.

```
Include "globals.h";
Include "puny.h";
```

This next section loads in PunyInform, by **including** two files. The first one, named with the string `"globals.h"`, has lots of **Constants** like the ones you set earlier. The second file, `"puny.h"`, loads in all of the PunyInform code to let you explore the world you're about to create.

```
[Initialise;
  "Stepping into the PunyInform Accelerator, you vanish into
    another life, and another time...";
];
```

Inform requires every game to have a special **routine** called `Initialise`[7]. A **routine** is a way of packaging up instructions to make them easier to use later. Fortunately, Inform will understand that a **string**, on its own in a routine, means that we want to display the text to the player and finish the **routine** successfully.

[7] Americans may be tempted to write this as "Initialize". Inform was written in Oxford, UK, so we often need to use British spelling!

```
Object Office "Your Office"
with
  description "This is the quiet office where you wrote
    Colossal Cave Adventure, while coming to terms with the
    divorce that estranged you from your caving group.",
has light;
```

The world we explore in our game is made of **objects**. Everything from items you can hold, to rooms you can walk through, and even the player, are all defined with the `Object` statement.

Each **object** has two names:

1. The word used to name it in your code[8]
2. The name that is displayed to the player

[8] For this game, we have decided to capitalise the code names for rooms, but leave other objects in lower-case.

So we have a room referred to as `Office`, which is what we put in the **constant** called `INITIAL_LOCATION_VALUE`. This means our player will begin the game standing in a room which displays its name as "Your Office".

The words **property** and **attribute** sound like they should mean the same thing, don't they? In Inform, a **property** in an **object** has a value, which you can set to data or code. An **attribute** is just a tag that an **object** either `has` or `hasnt`!

We used the `with` statement to give this room a `description` **property**, which is a **string** that is displayed when the player uses the `look` command while standing inside the room.

We also used the `has` statement to give it an **attribute** called `light`, which means the `look` command won't just tell the player that it is pitch dark.

```
Object -> teletype "ASR-33 TeleType"
with
  name 'asr-33' 'asr33' 'teletype',
  description "This is a model ASR-33 teletype. It's an
    electric typewriter, connected by wires to the PDP-10
    down the hall.^It has a paper tape puncher/reader to
    store data for later, and everything the computer has
    displayed is left printed in block capitals on an
    endless spool of paper.",
has static supporter;
```

Our next **object** is the `teletype`. The `->` means that it will start the game located in the room we just defined.

In addition to the `description` **property**, we added a `name` **property**, with three words. When we surround words with `'` characters, we call that a **dictionary word**. This means that the game can understand these words when the player types them in. So the player can refer to this **object** as `teletype` or `asr33` if they want.

We also gave this **object** two **attributes**:

1. `static`, which means the player can not pick it up.
2. `supporter`, which means that other objects may be placed on top of it.

```
Object -> -> transcript "game transcript"
with
  name 'printout' 'game' 'transcript',
  description "This is what you came for: a printout of an
    early game of Colossal Cave Adventure your kids played
    with you. It is currently at a particularly descriptive
    section:^~YOU ARE IN A SPLENDID CHAMBER THIRTY FEET
    HIGH. THE WALLS ARE FROZEN RIVERS OF ORANGE STONE. AN
    AWKWARD CANYON AND A GOOD PASSAGE EXIT FROM EAST AND
    WEST SIDES OF THE CHAMBER.~",
  after [; Examine: deadflag = GS_WIN; ];
```

Did you notice the use of `~` in the `description`? Tildes are turned into `"` when printed, because we can't put `"` in our **strings** without ending them. Again, we'll talk more about special **string** characters later.

The last **object** is the `transcript`, which is used to win the game. We used the double arrows `-> ->` to state that it begins the game located in the most recently-defined **object** that is inside a room. Since in our case that is the `teletype`, and the `teletype` is defined with `has supporter`, that means the `transcript` is *on* the `teletype`.

The final line of the game is the one that lets you win. It has a lot of punctuation, so here is just that line in a friendlier format:

```
Object transcript
with
  after [;
    Examine:
      deadflag = GS_WIN;
  ];
```

Good story games anticipate what the player will try. Many players get frustrated when a word that makes sense to them isn't recognised by the game. PunyInform tries to link common **verbs** together so that `read` and `look at` are the same as `examine`.

We'll show you how to avoid "guess the **verb**" in your own games, later on.

We have given this object a **property** called `after`. This looks a bit like the **routine** we defined at the top, but it has no name between the `[` and the first `;`. The Inform code we put here will be sent a **message** every time PunyInform finishes successfully letting the player do something with the `transcript`. The **message** will be

the **verb** the player used on the `transcript`.

In this case, we want the game to be won after the player has read this **object's** `description`. The game ends when a special **variable**, called `deadflag`, is changed from `0` to something else. Here we say that `after` the player successfully finishes using `Examine` on the `transcript`, we will set `deadflag` to the **constant** `GS_WIN`[9]. This will tell PunyInform to congratulate the player for a game well-played!

[9] The value of `GS_WIN` happens to be `2`, in case you're curious!

A Larger World

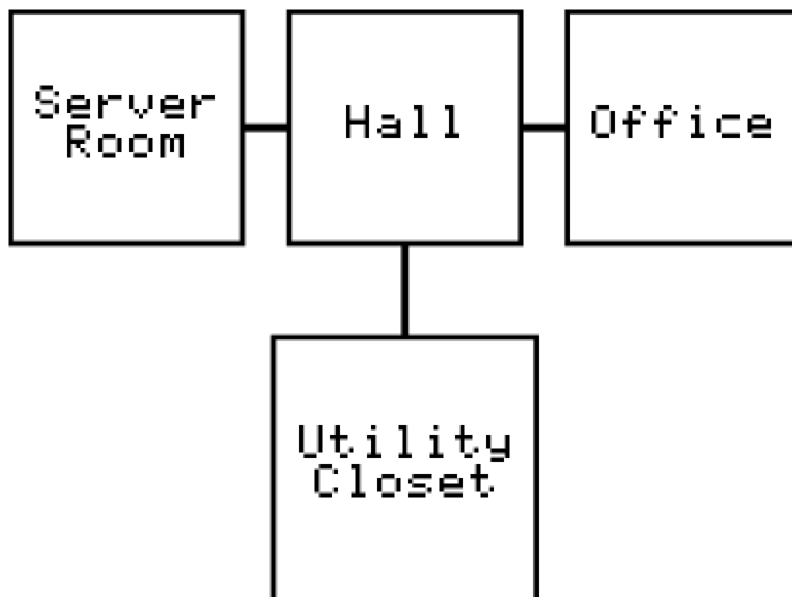

Figure 2: A map of the rooms for our small game.

There are many brilliant games that use only one room, but most story games involve exploring a map of interconnected locations. We tend to sketch these maps out on paper with boxes representing rooms, and the paths between them as lines.

The player types compass directions to move from room to room. This is such a common thing to do that `go east` may be entered as `east`, or even simply `e`.

Our second game will be set in the same building, but with more rooms and a bit of a puzzle to solve. We'll take a look at it in pieces.

Notice that `Statusline score;` could have been on a line of its own. The semicolon (`;`) separates lines of Inform code.

```
Constant Story      "IMP";
Constant Headline   "^An ARPANET quest.^";
Constant STATUSLINE_SCORE; Statusline score;
Constant OPTIONAL_SCORED;
Constant INITIAL_LOCATION_VALUE = ServerRoom;

Include "globals.h";
```

```
Include "ext_cheap_scenery.h";
Include "puny.h";

[Initialise;
  "The crackling energy of the leap subsides, and you blink
    your eyes as you get your bearings in a new situation.";
];
```

Cheap Scenery

We've added some Constants to keep track of the player's score, and added this score to the status line. We have also `Include`d an extension for "cheap scenery".

Remember when we said that good games anticipate what the player will try? Well, one thing that can frustrate players is when they try to examine something that was mentioned in a room's `description`, only to be told that it does not exist! A polished game should describe as many words as possible.

Unfortunately, we have a limited number of **objects** we can create[10]. If we gave each "scenery" word an **object** with a `description` and `has static`, our game would become very large and run out of **objects** for rooms and items.

[10] PunyInform can produce three different sizes of game. The smallest of these, which can be played on the widest variety of computers, can only hold 255 **objects**.

PunyInform comes to our rescue with the "cheap scenery" extension! This lets us give our room a `cheap_scenery` **property** which takes two **dictionary words** and one **string**. When the player tries to `examine` either of the words, the descriptive **string** is displayed.

Note that we defined a new **constant** to store the **string** that describes the PDP-10 computer. This let us make two entries in `cheap_scenery` that share the same long **string**.

We can now use any of four words to refer to the PDP-10, but kept our game size down.

Note that the final comma indicates the *end* of the `cheap_scenery` data. Each "cheap" description is separated with spaces or a new line.

```
Constant PDP_10 "This is the computer your group is using to
  develop the ARPANET.  It is the size of three or four
  refrigerators, and covers one wall.^Unfortunately, it isn't
  much use with the IMP offline.";

Object ServerRoom "BBN Server Room"
with
  description "The din of cooling fans reverberates off the
    metal racks of computer equipment.  One wall is covered
    by an enormous PDP-10 computer system, and the other by
    the ARPANET IMP.",
```

```
  cheap_scenery
    'cooling' 'fans' "Transistors are much cooler than
      thermionic valves, but the heat from all this equipment
      must still be vented off with forced air coming from
      the raised floor."
    'metal' 'racks' "Each computer in this room takes at
      least four square metres of floor space, housed in
      industrial metal cabinets."
    'noise' 'din' "The cooling fans make such a loud,
      whooshing, whirring noise. It's really not healthy to
      stay in here for too long without ear protection!"
    'pdp-10' 'pdp' PDP_10
    'computer' 'system' PDP_10
    'internet' 'arpanet' "The IMP breaks data into
      ~packets~, which it can send around the ARPANET, and
      reassemble from other IMPs. It's very clever, and some
      day all business will be conducted over a network like
      this."
    'substantial' 'lock' "The IMP's power switch is behind a
      military-grade lock, which requires a heavy metal key
      to operate.",
  e_to Hall,
has light;
```

In addition to all of the description properties, we have a new one: `e_to`. In this case, that means that when the player is standing in the `ServerRoom` **object**, they can type `go east`, `east`, or even just `e`, to move to the `Hall` **object**.

Switching Objects On and Off

We have also given the `imp` two new description **properties**: `when_on` and `when_off`. These are special descriptions for any **object** that `has switchable`.

The `when_off` description will be displayed when the `imp` is switched off and the `ServerRoom`'s `description` is printed.

Would `when_on` *ever* be printed?

```
Object -> imp "Interface Message Processor"
with
  name 'imp' 'arpanet' 'router',
  description "The IMP is a special-purpose computer that
    breaks data up into ~packets~ to send over a new internet
    called The ARPANet.^There are now 110 computers you can
    connect to!^^Or at least, you could if the thing were
    turned on...",
  when_on "The IMP hums away happily, routing packets.",
  when_off "The IMP is currently off, which means no one can
    receive any e-mail from outside right now.",
```

Figure 3: An IMP next to a teletype. The IMP was a special networking computer used to develop the ARPANET.

```
  with_key impkey,
  before [;
    SwitchOn:
      if (self has locked)
        "Unfortunately, the IMP's power switch is protected
          with a substantial lock";
  ],
  after [; SwitchOn: deadflag = GS_WIN; ],
has static lockable switchable locked;
```

Note that the `cheap_scenery` describing aspects of the IMP is listed in the the `ServerRoom` **object**. That is because `cheap_scenery` belongs in the location where the player is when they type `examine`.

The `imp` **object** begins with some familiar **properties**:

- The `->` places it in the `ServerRoom`
- It has a `description` **string** that uses `~` for inverted commas[11] and `^` for newlines.

[11] Often called "quotation marks" or "speech marks" in the US.

But the `imp` isn't just an inert piece of scenery. It is a machine that can be switched on and off, provided that it has been unlocked. That's why it `has static lockable switchable locked`.

The goal of this game is to switch it back on. Just as in the last chapter, we define a **routine** for the `after` **property**. This one reacts to the `SwitchOn` message, such as when a player tries `turn on imp`, by declaring the game won.

Locks

So how do we prevent the player from turning it on while it's still `locked`? To do this, we need to step in *before* the `SwitchOn` action succeeds. So we define a **routine** in the `before` property[12] that responds to the `SwitchOn` message.

[12] We will use `before` and `after` a lot. Just remember that `after` only runs after an **action** has *succeeded.* The `before` **property** can decide whether or not the **action** takes place.

Our **routine** makes a choice:

- *If* the IMP is locked, it will end the action with a message that describes why it failed.
- Otherwise, it will do nothing, and allow the normal `SwitchOn` action to proceed.

The statement `if (self has locked) "...";` prints the **string** and exits *if* the **object**[13] has the `locked` **attribute**. Since unlocking an object *removes* that attribute, we can stop here. The unlocked object will not pass the `if` test, so the player can switch it on.

[13] The special **object** name `self` refers to whatever object the **routine** is a **property** of. This lets us re-use **routines** the same way we re-used that `PDP-10` description string.

But how does the player unlock the `imp`? We have set a special **property**, called `with_key`, to the **object**[14] that the player can use to `unlock` the `imp`.

[14] We specified that the `imp` lock can be opened with `impkey`. Until we define an `impkey` **object**, we won't be able to build this game. Instead, we will get an error message: `No such constant as "impkey"`.

Another Room

Players expect pathways to work both ways. If the `Hall` is east of the `ServerRoom`, then we need to let the user go west to get back. So we've added the `w_to ServerRoom`.

This hallway is a central "branching-off point" that connects to many other rooms. We'll define those rooms later, but building the game will produce an error until we have.

We gave the `ServerRoom` an exit east toward a room called `Hall`. Let's make that room now.

```
Object Hall "Hallway"
with
  description "You are in a hallway in the BBN offices in
    Cambridge, MA.  To the west is a noisy server room, a
    storage closet is to the south, and your office lies to
    the east.",
  w_to ServerRoom,
  s_to Closet,
  e_to Office,
has light;
```

Now let's put a broken vending machine in the hallway. If the player has the `vmkey` object,

```
Object -> vending "Vending Machine"
with
  name 'vending' 'machine' 'soda',
  description "This is a large soda machine.  It has a large
    ~OUT OF ORDER~ light.",
  with_key vmkey,
has static openable lockable locked container;
```

Note that we've turned the `describe` property into a **routine**. This lets us return a different description depending on the position of the switch.

This is similar to the lock we gave the `imp`: we test `if (self has on)` to return the lit description, and otherwise return the unlit description.

We've also given the torch a `scored` **attribute**. This will make the player's score go up by 4 points when they take it. Scores in story games are less about a player's skill, and more an indication that the game is on the right track. When the player's score goes up, that means they've advanced the plot in some way.

Now let's create an electric torch, which is a `switchable` object that provides `light` when it's on.

```
Object -> -> torch "electric torch"
with
```

```
  name 'electric' 'torch' 'flashlight',
  describe [;
    if (self has on)
      "^The torch projects a weak circle of light.";
    "^The torch is switched off.";
  ],
  after [;
    SwitchOn: give self light;
    SwitchOff: give self ~light;
  ],
has switchable scored;
```

The `after` **property** lets us react to the `SwitchOn` and `SwitchOff` **messages**, using `give self` to change the `torch` to either `has light` or `hasnt light`. If an object `has light`, then the player can see objects and descriptions even if the current room is dark.

We'll make a dark room next.

We've given `Closet` the `scored` **attribute**. This means that the player will gain 5 points when they first see the room's description.

Since we left out `has light`, this room is dark. The score can't increase until the torch lights it.

```
Object Closet
with
  description "A storage closet, but unfortunately the lights
    have gone out.",
  n_to Hall,
has scored;

Object -> impkey "heavy metal key"
with
  name 'heavy' 'metal' 'key',
  description "This is a solid metal key on a stout chain.",
has scored;
```

Now we have an idea of how to play through the game:

1. Get the vending machine key.
2. Unlock the vending machine.
3. Open the vending machine.
4. Take[15] the torch.
5. Switch on the torch.
6. Take the IMP key.
7. Unlock the IMP.
8. Switch on the IMP.

[15] Some players use `take torch`, while others prefer `get torch`. Which one do you use?

But we haven't made the vending machine key. Let's do that now.

```
Object Office "Your Office"
```

```
with
  description "This is the quiet office where you wrote Colossal
    Cave Adventure.",
  cheap_scenery
    'asr33' 'teletype' "This is a model ASR-33 teletype. It's
      an electric typewriter, connected by wires to the
      PDP-10 down the hall.^It has a paper tape
      puncher/reader to store data for later, and everything
      the computer has displayed is left printed in block
      capitals on an endless spool of paper.",
  w_to Hall,
has scored light;

Object -> vmkey "round key"
with
  name 'round' 'key',
  description "This is a cylindrical key on a lanyard.",
has scored;
```