

GAMING HACKS™

*100 Industrial-Strength
Tips & Tools*



O'REILLY®

Simon Carless
With a foreword by Marc Laidlaw

HACK
#85

Download, Compile, and Create an Inform Adventure

Return to the roots of adventure gaming, where the story's the thing.

A time was when only programmers played computer games, for only programmers had access to computers. Hackers created the first adventure games, deep in the warrens of SAIL and the stygian halls of MIT. The first was Adventure, or Colossal Cave; the second was Dungeon, or Zork.

A time was, a few years later, when ordinary people could *own* computers—TRS-80, Apple II, C-64—for only a few thousand dollars. Those MIT hackers packed up Zork, divided and expanded it, and sold it as Zork 1, 2, and 3. Along the way, they made other fantasy adventure games, mysteries, science fiction, historical adventures, and less classifiable experiments.

Their company was Infocom. Their games, in the days of their glory, were entirely text, which understood natural-language commands and produced prose output. They even adopted a new term, *interactive fiction*, or IF. Through the early- and mid-1980s, critics called Infocom's work the most literate, sophisticated, and entertaining in the computer-gaming world.

Gaming changed after that. Today a “computer game” means flashy graphics, almost without question. The computer market grew enormously, but text adventures didn't grow in proportion.

Yet text adventures have not gone away. When gaming markets wouldn't support them and big gaming companies wouldn't sell them, hobbyists and amateurs wrote them instead. Today, they create dozens of text adventures every year and distribute most for free (<http://lifarchive.org/>). Experimentation and artistic striving dominate the industry, not marketing and the Christmas hit season. Interactive fiction has matured, perhaps beyond where the commercial game industry can ever go.

Modern IF Tools

Every Infocom fan from the '80s must have dreamed of creating his own games. Sometimes it seems as though two-thirds of them invented IF development tools. There are several complete IF systems available today, most distributed for free. In these hacks, we'll focus on Graham Nelson's Inform (<http://inform-fiction.org/>).



For a comparison between Inform and other popular IF systems, see the following sites:

<http://www.brasslantern.org/writers/howto/chooselang.html>
<http://www.firthworks.com/roger/cloak/>

Inform is a compiled language, rather like C in syntax, with many features devoted specifically to text adventure creation. Inform programs, like Java programs, compile into a portable game format that requires a runtime interpreter in order to function. In fact, Inform compiles to Z-code—the very game format Infocom invented for their games. (Infocom’s source language looked nothing like Inform, however.) Z-code interpreters exist for just about every computer and PDA used today; the interpreters, like the compiler, are free.

Inform’s standard library contains the basics of a text adventure. It has a full-featured parser, which is the code that reads player input, figures out what the player means, and executes the command. The standard library also comes with a large stock of common commands: movement, taking and dropping objects, talking to other characters, and so on. Furthermore, all these standard features are easy to customize, extend, or replace, if the needs of a particular game so require.

Before we begin, download the appropriate version of Inform for your computer from <http://inform-fiction.org/>. The current version (as of this writing) is Inform 6.30. You will also need to download Inform library 6/11. Finally, you’ll need a runtime interpreter. Go to <http://www.ifarchive.org/indexes/if-archiveXinfocomXinterpreters.html>. For Windows, I recommend *WindowsFrotz2002* in the *frotz* folder. On MacOS Classic, *MaxZip* (under *zip*) is good. For Mac OS X, try *Zoom*. On Unix systems, including Mac OS X, you can build Frotz or several other interpreters as command-line applications.

Unpack all of these files. Now let’s test the system by creating a world.

Creating a World

Interactive fiction is like any other sort of fiction. It has a setting, a story, characters, scenes, a resolution—and maybe even footnotes and silly chapter titles. We will not create a complete game, but we can create a scene, which could be part of a game. So we’ll need a setting.

Our setting will fit the title of the book: a virtual cyberworld in which the player hacks a secure network. We won’t be the first IF creators to venture into cyberspace, though. Let’s give it a twist: the player is a purely electronic intelligence—a native inhabitant of the network with his own agenda.



Is a hacker one who uses hacks, or one who cracks security? You see, that's a meat-person distinction. To an AI, all action is hacking, and security is just a current in the ocean. Information wants to be free, and can say so....

This will let us have some fun with the world. Seen through a program's eyes, every program is a fellow being. Every tool is a living thing. Every bit of data is a droplet in the living ocean. The humans outside the data-sea ... Well, they're meat-people—useful, if a bit thick.

Enough philosophy. Let's create the first element of our program. The player will start in Router NP-462—a humble network router in a low-security part of cyberspace.

Create a file named *inhack.inf* containing these lines:

```
Constant Story "INSIDE HACK";
Constant Headline "^An Inform sample adventure.^";

Include "Parser";
Include "VerLib";

[ Initialise;
  location = Router;
];

Object Router "Router NP-462";

Include "Grammar";
```



What have you typed? Inform code. The witty, complete, and intensely detailed *Inform Designer's Manual* (<http://inform-fiction.org/manual/html/>) fully documents the Inform language and libraries. Since we're hacking here, we can skip the operator precedence tables and dive into our example.

The first two lines define string constants—messages—the game will print at appropriate points. *Story* is the game's title. *Headline* is a subtitle; it can also contain other information, such as the author's name and copyright notice. The caret characters (^), by the way, represent line breaks.

The two *Include* lines load in two-thirds of the standard Inform library.

Next comes the definition of the *Initialise* function. The game will call *Initialise* exactly once, after the game loads and before the player's first command. It can contain any setup code that you want. It must also set the

Download, Compile, and Create an Inform Adventure

location global variable to a room object. This determines the player's initial location.



Yes, `Initialise` is spelled with an `s`. The creator of Inform is British. Keep a stiff upper lip. If U.K. web designers can cope with color in HTML, you can cope with `Initialise` in Inform.

After `Initialise`, you define the object called `Router`, which is the starting room of the game. Inform is an object-oriented language, so game objects, rooms, players, and other characters are all objects in the Inform program. Refer to this room object as `Router` in your code. Players will see its name as `Router NP-462`.

Finally, the line `Include "Grammar";` loads in the final third of the Inform library. Put it after the object definitions, for annoying technical reasons that are not interesting now, nor ever.

Compile `inhack.inf` with the Inform compiler. If you encounter errors, make sure your semicolons are in the right places.



On Windows, the easiest way to run the compiler is to put the compiler (`inform.exe`) in the same directory as all the library files (`English.h`, `Grammar.h`, etc.) and your source file (`inhack.inf`). Open a command prompt window and `cd` to this directory. Then type `inform inhack.inf`. If there are no errors, the compiler generates a game file called `inhack.z5`.

If you understand Windows command paths, feel free to move `inform.exe` to a more elegant location. You can also move the library files to their own directory and refer to them by typing `inform +library\path\name inhack.inf`.

The First Room

You now have a playable game file named `inhack.z5`. Load it into your interpreter and you'll see the following:

```
INSIDE HACK
An Inform sample adventure.
Release / Serial number 040305 / Inform v6.30 Library 6/11 SD
```

Darkness

```
It is pitch dark, and you can't see a thing.
```

```
>
```

The game banner prints properly, with a title and headline, but it's dark; you can't even see which Router you're in.

Inform has a sophisticated lighting system—suitable for spelunking into dark caves, guided by sputtering handheld lanterns. This game has no darkness, though; an AI needs no photons to see. So let's just declare this room to be fully lit. While we're at it, we'll write a room description.

Change the definition of Router to look like this:

```
Object Router "Router NP-462",
    has light,
    with
        description
            "This little data interchange is run-down, shabby, and
            rather sad. You can't see any traffic -- besides yourself --
            and cobwebs of noisy static hang in the dusty corners. The
            meat-side that this router serves must be a bare hallway,
            almost no hardware.^
            A broad network connection leads east, and a serial line runs
            south. The network connection is barred by a fierce-eyed access
            controller.";
```

Let's break this down. You already know that the first line declares an Object called Router, which is visible to players as Router NP-462. Then there is a comma, since the object definition continues.

The `has` line contains a list of *attributes*. The standard library defines a generous list of attributes that affect the behavior of objects in various ways. Attributes are yes-or-no; an object has an attribute or it doesn't. This room object has just one attribute, `light`, which causes it to be fully lit.

The `with` line introduces a list of *properties*. The standard library defines many properties as well. A property is more interesting than an attribute. Each property has a value, which can be a number, string, function, or another object.

The Router has one property, `description`. The description of a room object is what the player sees when she looks around. In this case, it is a fairly long string—two paragraphs. Recall that the `^` indicates a line break. Inform games usually use blank lines to separate paragraphs, so use `^^` to indicate a paragraph break. There's no need for `^` at the end of the description. It's implicit.

Now, compile this. As before, be sure to put the commas in the right place and the semicolon at the end of the whole Object definition.

Run the bytecode to see that there is now a working room:

```
INSIDE HACK
An Inform sample adventure.
```

Release 1 / Serial number 040305 / Inform v6.30 Library 6/11 SD

Router NP-462

This little data interchange is run-down, shabby, and rather sad. You can't see any traffic -- besides yourself -- and cobwebs of noisy static hang in the dusty corners. The meat-side that this router serves must be a bare hallway, almost no hardware.

A broad network connection leads east, and a serial line runs south. The network connection is barred by a fierce-eyed access controller.

Take a moment to mess around in this one-room game. You can't do much, because there's nothing in the room, but all the standard text adventure verbs are in place. If you type **north**, the game says You can't go that way. If you type **get**, the game correctly asks What do you want to get? (The only possible answer is myself, which produces a wry comeback.) All this functionality—including the disambiguation and the comeback—are part of the standard Inform library.

Incidentally, this room description may be a simple string, but it's doing real work. It's the introductory text for the scene, maybe for the whole game, and it has to set the player's mind in motion.

This description tells so much. This router is both a location and a data interchange; the game world is somehow electronic, and sections of the networks are somehow places. You can't see any traffic besides yourself, so are you network traffic? A program, maybe? An access controller sounds like a program, perhaps a firewall, but it has character. It's fierce-eyed, not a program as you know programs. "Meat-side" is a "hallway"—that must be the real world; but "meat-side" has a sneer in it. You know something about how the protagonist views people.

The Second Room

Let's add a second room. This will be the Scanner room. It lies south of the Router, along that serial line.

Compass directions may not make sense in a cyberworld, but they're so convenient that nearly every IF game uses them, regardless of setting. Changing the movement system might be more consistent, but it would interfere with gameplay, so don't do it.

To create the connection from Router to Scanner, let's add a second property to Router. Edit the object description:

```
Object Router "Router NP-462",
  has light,
  with
  description
```

```
"This little data interchange is run-down, shabby, and
rather sad. You can't see any traffic -- besides yourself --
and cobwebs of noisy static hang in the dusty corners. The
meat-side that this router serves must be a bare hallway,
almost no hardware.^
A broad network connection leads east, and a serial line runs
south. The network connection is barred by a fierce-eyed access
controller.",
```

```
s_to Scanner;
```

Note that an `s_to` property has been added. Its value is the location you reach by going south from the Router room. In this case, that's the Scanner room, and it needs a definition. Put this after the Router object:

```
Object Scanner "Scanner South-9",
  has light,
  with
  description
    "The firmware here is tight and clean; this device, at
    least, is well-maintained. You can see image-analysis nets
    strung in their neat rows around the sensor fountain which
    dominates this space. The only exit is the serial line to the
    north.",
  n_to Router;
```

The format here is the same. Don't forget the comma between the two properties or the semicolon at the end. This room has an `n_to` property, indicating the location you'll reach by going north—back to the Router. It's a two-way connection, so you have to set up both ends like that. (Yes, you can add `e_to`, `w_to`, `u_to`, `d_to`, and even `ne_to`, and so on. But not today.)

Behold: two rooms. That's enough to set the scene.