# A-code
## Tutorial

Mike Arnautov

mipmip.org/acode

# Table of Contents

This tutorial will guide you through a basic A-code implementation of Roger Firth's nano-adventure "Cloak of Darkness".

# Game Specification

Here is Roger Firth's summary specification of Cloak of Darkness:

- The Foyer of the Opera House is where the game begins. This empty room has doors to the south and west, also an unusable exit to the north. There is nobody else around.
- The Bar lies south of the Foyer, and is initially unlit. Trying to do anything other than return northwards results in a warning message about disturbing things in the dark.
- On the wall of the Cloakroom, to the west of the Foyer, is fixed a small brass hook.
- Taking an inventory of possessions reveals that the player is wearing a black velvet cloak which, upon examination, is found to be light-absorbent. The player can drop the cloak on the floor of the Cloakroom or, better, put it on the hook.
- Returning to the Bar without the cloak reveals that the room is now lit. A message is scratched in the sawdust on the floor.
- The message reads either "You have won" or "You have lost", depending on how much it was disturbed by the player while the room was dark.
- The act of reading the message ends the game.

This implementation of the game also will use non-mandatory object and place description sugegsted by Roger.

# Style and Terminology

A few words on style and terminology.

- Except in text messages to be displayed to players, A-code is completely case insensitive. As a personal convention, I use lower case in actual A-code code, except in declarations, where names of entites being declared are in upper case.
- When explaining code, I will be presenting code snippets in upper case in order to avoid excessive quoting.
- All declarative keywords (a.k.a. major directives) **must** start in column 1 (i.e. with no leadin spaces or tabs). A declaration is terminated by another non-comment) line starting in colum 1, or by the end of a file.
- A-code opcodes (a.k.a. minor directives) as well as any lines in declaration bodies must have at least one blank or tab preceding them. Beyond that, A-code does not care about levels of indentation.
- Comments are permitted everywhere, except within text declarations. The start of a comment is signalled by a hash (#) sign and A-code simply ignores the sign, any preceding blanks or tabs, and the rest of the line. (NB: Dave platt's version of A-code had a different comment convention.)

Once you have an acode source file, you will want to convert it into a game executable. Other than an ANSI C compier, you will find everything necessary in the [acode-12.91.tgz](acode-12.91.tgz) tarball. If you can use the **advbld** bash script, just give it the name of the source file and it will do the rest. If you cannot use a bash script, you'll need to do it "manually". First run the **acdc** translator, giving it the source file name or pathname as an argument. Then use an ANSI C compiler on resulting .c and .h files together with the three kernel files: adv00.c adv01.c and adv0.h.

# Program structure

The game starts with a one-off announcement of the player's arrival at the Opera House. For now, we'll skip the actual arrival message and use the traditional "Hello world!" instead.

```
init
    say "Hello world!"
repeat
    stop
```

That's a fully functional A-code program, which will print its message and stop. The INIT and REPEAT code sections are mandatory. Either or both can be empty, though this would not make sense in any real game.

INIT code sections are used for game initialisation. If there are several, they get executed in the order or their declaration. The REPEAT sections (also executed in the order of their declaration) constitute the main loop of the game. So INIT sections get executed once and then the game repeatedly executes the REPEAT sections in their order, until the stop directive is encountered.

The actual initial message for our game is a multi-line one, with some lines being blank. Nevertheless, we could simply substitute it for "Hello world!" in the program so far:

```
init
    say "
        The Cloak of Darkness

        Hurrying through the rain-swept November night, you're glad
        to see the bright lights of the Opera House. It's surprising
        that there aren't more people about but, hey, what do you
        expect in a cheap demo game...?

    "
repeat
    stop
```

Notice that all text lines are preceded by some blanks – only major directives start at the beginning of a line. Any such leading blanks are ignored and successive non-blank lines are interpreted as a single line to be

wrapped at runtime as appropriate. Within text definitions, blank lines can be completely blank (no spaces or tabs preceding end of line).

However, I reckon this use of multi-line text is untidy and makes code less readable. So I'll use a named text instead.

```
text YOU.ARRIVE
   The Cloak of Darkness

   Hurrying through the rain-swept November night, you're glad
   to see the bright lights of the Opera House. It's surprising
   that there aren't more people about but, hey, what do you
   expect in a cheap demo game...?

init
   say you.arrive
repeat
   stop
```

Note that no double quotes are used in defining named texts. The text is simply terminated by the next declaration (or end of file).

# Player vocabulary

The game specification does not tell us what actions the player must be able to perform. This being a minimalist implementation, we'll declare only the bare essentials. For example the player is originally wearing the cloak; do they need to REMOVE it before dropping it or hanging it on the hook? No. They clearly need to be able to pick the cloak up if they'd dropped it, but there no need to allow them to wear it again – so no WEAR either. Only actions clearly implied by the specification will be catered for.

While A-code player command interface is at present restricted to the ancient verb/noun structure (no adjectives, prepositions or instruments), it also permits apparently complex commands, e.g. "drop everything except the ring and the orb then go out". This is achieved by the parser reducing such complex command to a series of one or two word simple ones. However, in constructing a game one can also cheat: in so far as adjectives are not necessary to identify objects, they can be simply ignored.

All vocabulary words are automatically abbreviable to the smallest unambiguous length. By default, the A-code parser will also do approximate, one-typo matching of player's command words against the vocabulary, but in this tutorial we'll be switching approximate matching off, in order to avoid complications of dealing with ambiguous typos and such-like.

Note that in the absence of any clashing vocabulary words (including objects, which I have not tackled yet), the four direction commands will be automatically abbreviable to their first letter.

So, verbs available to the player:

```
# Game specification mandates just the four cardinal directions
verb NORTH
verb EAST
verb SOUTH
verb WEST
#
verb GET, TAKE    # Both the player and game code can use these interchangeably
verb INVENTORY
verb DROP
```

```
verb READ          # For reading the message in the bar room
verb HANG          # To hang the cloak on the hook
verb LOOK
verb QUIT          # Not in the spec, but every game should allow quitting
#
noise THE, THAT, VELVET, GO     # Player command words to be ignored
```

The NOISE definition tells the game which words in player's command are to be completely ignored. Thus GET THE VELVET CLOAK will be parsed as GET CLOAK, despite the parser's limitation to verb/noun commands. Similarly GO WEST will become simply WEST, which somewhat simplifies game's code.

I sneaked in another feature here: comments. Comments are delimited by the hash sign. Full-line comments (with the # sign being the line's first character) are permitted everywhere – even within message or description texts. They are simply ignored. In-line comments are not allowed in texts (be it game's messages or object/place descriptions).

# Objects

The vocabulary definitions we have so far contain only verbs, but the game will obviously need nouns as well. These are mostly added via object and place definitions. So here's the first object definition:

```
object HOOK
  %A small brass hook is screwed to one wall.
  &It is just a small brass hook[/ with a cloak hanging on it].
```

This defines an object and names it HOOK. By default, object names are automatically added to the player's vocabulary, so that names can be used (synonymously, if there is more than one) both by the player and by game's code. If for whatever reason some object name is to be excluded from the vocabulary, this is achieved by prefixing the name with a minus sign.

An OBJECT declaration line is optionally followed by up to three kinds of object description: an inventory one, a long "here-is" one and a detailed one. The first two are displayed by the SAY directive (we'll get to it later on), depending on whether the object is carried by the player or is just present in the current location. The detailed description can be shown by the DESCRIBE directive, which is used to show the most detailed description available.

In this case the hook can never be picked up by the player, so its definition omits the inventory description, which would have consisted of line(s) of text immediately following the OBJECT declaration line. The "here-is" description starts with a line prefixed with the % sign and can also consist of several text lines. Finally, the detailed description is signalled by the & sign at the beginning of a line and can also feature additional lines.

In the hook's case, the detailed description contains a "text switch", which will display different text fragments depending on the internal state of the object. (All declarative features of A-code texts are covered at some length by [relevant section of A-code documentation](#)). If the hook's state has its default value of zero (or for that matter if it is negative), the description will read "*It is just a small brass hook.*". If it's internal state is 1 (or higher), the

description will include the cloak. We could give these two values symbolic names for code clarity, but I'll leave that for an advanced tutorial.

Now for the cloak...

```
object CLOAK
    A velvet cloak[ (worn)/]
  %A black velvet cloak [/lies on the floor/hangs on the hook].
  &It is a handsome cloak, of velvet trimmed with satin, and slightly
   spattered with raindrops. Its blackness is so deep that it
   almost seems to suck light from the room.
```

This object does have an inventory description, which depends on the object's state, as does the long, "here-is" description. The cloak will have three states, which again could have symbolic names: worn, not worn (can be either carried or lying on the floor) and hanging on the hook.

Finally, the specification says that there are three objects altogether, the message in the bar being the third one. Here it is, but I'll postpone its explanation until later. For the moment, just note that it has just one description with a couple of text switches.

```
object MESSAGE
    The message[ has been carelessly trampled, making it difficult to read.
    You can just distinguish the words/, neatly marked in the sawdust,
    reads]...

    +YOU HAVE [LOST/WON] !!!
```

That's it for objects. Next come locations.

# Game initialisation

A full size A-code game can have quite large initialisation sections, setting up correct attributes for places, objects and texts. You may wonder why this is being done at run-time, rather than statically, as those entities are being declared. The reason is partly history, but there is also a much stronger reason, why this unusual arrangement persists: it is extremely helpful in guaranteeing upward compatibility of saved games, which, if done correctly, permits game players (and testers!) to upgrade the game withuot invalidating any existing game saves.

In this minimalist implementation, though, there are no such attributes to set and this the initialisation code is sweet and short. Here it is, replacing the earlier suggested version:

```
init
    goto foyer              # Move the player to the start location
    apport cloak, inhand    # Give him the cloak
    apport hook, cloakroom  # Put the hook where it belongs
    set message, 4          # Limits the number of moves in the bar
    say you.arrive
```

The minor directive GOTO places the player inthe FOYER.

The first APPORT moves the object CLOAK to the location INHAND, an automatically defined location for objects held or worn by the player. The second APPORT puts the hook into the cloakroom. (Use of the word "apport" is Platt's little joke – it means transporing an object by occult means, without material agency.)

Note, by the way, that all value-bearing entities (objects, places, variables and even some texts) are automatically initialised with value of zero, which for the cloak happens to correspond to being worn by the player.

Finally, the SAY directive displays the previously defined named text. This directive is extremely versitile and cen be applied to any entity (text, object, place, vocabulary word) assiciated with text. It can also be applied to variables which point to any such entity. We'll see use of pointer variables later.

# The main loop – house-keeping

After initialisation comes the main program loop. This can be thought of as consisting of three parts. First comes the general per-turn house-keeping (e.g. determining whether the current location is lit, whether the player is getting thirsty, whether there is some NPC action to be announced...). Secondly, the player's input is obtained and parsed. Thirdly player command is responded to.

For clarity, I prefer to separate them into individual REPEAT sections, which get executed in the order of their declaration. In this game, there isn't much house-keeping required:

```
repeat
   ifflag status, moved   # Has the player moved?
      call describe.here  # If so describe their new location
   fin
```

Brief as it is, this section packs a lot of new stuff. Firstly, STATUS is a global variable. It must be present in any A-code game, so if not declared explicitly in the game's source, it is declared automatically by the **acdc** translator. We shall meet its other uses further on, but for the moment it is enough to know that like all variables (global or local), in addition to carrying a value, it has a set of binary flags associated with it. In this particular case, the relevant flag is called MOVED. This flag is optional – if declared by the game, it is set by the kernel to true if the last simple command resulted in a change of location and to false otherwise. We haven's actually declared that flag yet, but that's all right – A-code allows you to use entities before their declaration. So let's declare it now:

```
flags variable
   MOVED       # Optional STATUS variable flag, maintained by the kernel
```

This declares a flag which can be used with variables (but not places or objects).

Back to house-keeping... As you may guess, the IFFLAG directive in the above code snippet checks whether the MOVED flag of the STATUS variable is set to true and if so, causes execution of the following code, up

to the corresponding FIN directive closing the conditional code block. (Block's indentation is irrelevant and only used to enhance code readability.)

In this case, the conditional code does nothing more than call a procedure, (that is a named chunk of code) called DESCRIBE.HERE. As the name suggests, it will describe the current location. Let's attend to it straight away.

# Named procedure declaration

Here's that procedure for describing locations:

```
proc DESCRIBE.HERE
   local optr        # Local pointer to objects
   say here          # Show description of current location
   itobj optr, here  # Loop through objects at this location
      say optr       # Show such objects
   fin               # Loop terminator
```

Once again, this brief declaration introduces several important concepts.

The LOCAL OPTR statement declares a variable called OPTR, which is local to this code section (i.e. to this procedure). All other variables (whether declared explicitly by the VARIABLE major directive or are are supplied automatically by the A-code kernel) are global in scope. Declaration of local variables forms the sole exception to general rules in that any LOCAL declaration lines **must** be placed at the very beginning of the code section to which they apply.

I've called this variable OPTR because it will be used as a pointer to objects. Pointers are just variables which happen to be referencing some game entity (object, place, text, word or variable). In almost all cases, if a variable containing a pointer value is used in some context where a game entity is expected, the effect is as if the entity pointed to were used. So in our case, the SAY directive will display the description of the player's current location, as if that location was explicitly named as its argument.

The SAY HERE statement displays the description of the current location. This is because HERE is an automatic variable, which is maintained by the kernel and always point to the player's current location. (There is also the companion automatic variable THERE, which always points to the player's location prior to the last location change. We do not need it in this game.)

The ITOBJ directive iterates through game's objects, executing for each one the following code, up to the matching FIN statement. It can be qualified by a location (as in this case) and/or by an object flag or flags, in which case

any non-matching objects are ignored. Thus ITOBJ OPTR, HERE will only consider objects in the current location.

As you may have gathered from the above, SAY OPTR will give the description of the object pointed to. It will not be the inventory description, which would be automatically used if the object in question were carried by the player. And it will not be the detailed description, because that is only shown by the DESCRIBE directive.

# The main loop – player input

Back to the main loop... It is time to obtain and parse theplayer's command.

```
repeat
   set status, no.amatch     # Suppress approximate matching of command words
   input           # Get and parse player's command
   ifeq status, 2
      and
   iflt arg2, 0   # Can't find 2nd command word in the vocabulary
      or
   iflt arg1, 0   # Can't find 1st command word in the vocabulary
      quip "Pardon?"      # Command parse failure -- abort the main loop
   fin
```

The INPUT directive considers the next simple command (a complex one may have been previously given). If the previous command (simple or complex) has been fully processed, it will first prompt the player for a new one. It will parse the verb/noun command thus obtained and will set three automatic variables: STATUS, ARG1 and ARG2.

The STATUS variable gets set to the number of words in the command being parsed, which can be one or two. It cannot be zero, because a null command is simply ignored. Parsing the command includes matching its words against game's vocabulary. If all is well, ARG1 becomes a pointer to the first word of the command and ARG2 becomes a pointer to the second one – or, if only one word is supplied, ARG2 is set to zero. If either of the supplied words cannot be matched, the corresponding ARG variable is set to a negative value indicating the kind of failure involved. In this simple implementation we won't bother with details and simply report parsing failure of some kind.

The conditional directive IFEQ checks for equality of its two arguments, while IFLT checks whether the value if the first argument is smaller than that of the second one. Such conditional statements can be joined by means of AND and OR directives into compound ones. The joint effect is conditional execution of any subsequent code up to the matching FIN directive.

One unusual aspect of A-code should be noted here. Compound IF statements are parsed on the "as-you-go" basis. Or to put it otherwise, AND and OR have the same precedence and compound IFs are processed in the order in which they are given. Thus the above triple IF will report a failure either if a two word command is supplied and the second word cannot be parsed **or** the first word cannot be parsed (regardless of the number of words supplied).

# Generalised error handling

Here is that BAIL.OUT procedure for handling general errors. I'll use it to show two distinct ways of embedding player command words in the game's responses.

```
proc bail.out
   local qualifier          # Local variable initialised to zero
   ifeq status, 1           # If a single word command given
     ifflag arg1, object    # If that word is an object name
       ifnear arg1
       else
         quip "There is no # here!", arg1
       fin
     else
       set qualifier, 1     # The word is not an object name, assume verb
     fin
     quip "You need to say what you want to [do with the /]{arg1}.", qualifier
   fin
   quip "You can't do that!"  # Generic no can do
```

Used as other than a major directive for declaring objects, OBJECT is an automatically maintained entity flag, which is true for objects (or variables pointing at objects) and false otherwise.

If the supplied word refers to an object and the object is not nearby (not carried and not at the current location), the QUIP directive will say so. The # sign in that message is a word holder, which gets replaced by the word pointed at by the ARG1 variable.

If the object referred to is present or the supplied word is not an object, we want to say that not enough information has been given. That response is very similar in either case, so I used the same meassge with an embedded switch, which is qualified by the QUALIFIER variable. The command word is, in this case, echoed via the {} construct, which gets replaced by the text (if any) with the entity pointed by the variable name.

If code execution continues past that second FIN, it means that the command being processed had more than one word (otherwise it would have been handled by one of the preceding QUIPs). That being so, a very generic "don't understand" will do the job.

# Location-specific code

Now for the game code specific to game's locations. This is declared using the major directive AT. Yes, it could have been defined as a part of location definitions, but that's not how Dave Platt structured it. If preferred, individual AT code definitions can be placed immediately after the corresponding PLACE definition. Conventionally, though, all PLACE definitions are grouped together and so do all AT code definitions.

In the foyer, the player can go south or west, attempt to move north or east, look around and attempt to drop the cloak. However, trying to move in a direction in which there are no exits is handled generically by the error handling REPEAT section, so only legal exits need to be catered for. Similarly, looking around and dropping things are generally not location specific, so are better left to verb-associated code to handle later. That's not to say that we can't intercept such actions at location-specific code, if more convenient.

```
at foyer
   move south, bar
   move west, cloakroom
   respond north "You've only just arrived, and besides, the weather outside
      seems to be getting worse."
```

The MOVE opcode is a conditional version of GOTO. It takes a variable number of arguments, of which the last one is the location to move the player to, if the last player command contained one of the words listed before the location name.

The cloakroom code is self-explanatory. Again, other actions will be handled by code specific to individual verbs. The bar is a little bit more complicated.

```
at cloakroom
   move east, foyer
at bar
   move north, foyer
   ifloc cloak, cloakroom
   else                       # The cloak is not in the cloakroom
     sub message, 1           # Count attempted actions in the dark
     quip "Blundering around in the dark isn't a good idea!"
   fin
```

As one might expect, the IFLOC directive checks whether its first argument (which must be an object or a variable pointing at an object) is in one of locations listed by the rest of the directives arguments. In this case we want to do something when the cloak is **not** in the cloakroom. This is done by using the ELSE directive which executes code if the IF test returns false. As usual, a test can have separate code to be executed when the condition is satisfied and when it is not.

What happens at the bar is that if the cloak is not at the cloakroom, the value of the object MESSAGE, which was set to 4 in the game's initialisation (the INIT section) is decremented (you will see why later). Because there is no action that can be performed in the dark bar, it makes sense to ignore the player's command and abort the REPEAT loop with a warning message about blundering in the dark.

# Verb-specific code

All that is left now is to define actions associated with individual verbs. The major directive ACTION is used to define code associated with a particular vocabulary word. Let's start with INVENTORY.

```
action inventory
   local optr  # Local variable which will be used as a pointer to objects
   local count # Counts listed objects, automatically initialised to zero
   itobj optr, inhand # Loop through objects in player's inventory
      say optr        # Show object's description
      add count, 1    # Increment count of listed objects
   fin
   ifeq count, 0                        # If nothing listed, ...
      say "You are not carrying anything." # ... say empty-handed
   fin
   quit        # Command fully handled, so terminate the REPEAT loop
```

We have already met local declarations and comments should make the that code quite self-explanatory.

Having tackled INVENTORY, we'll proceed with TAKE, which normally applies only to objects, but it is traditional to code also for TAKE INVENTORY. That's not a problem, since A-code makes no fundamental distinction between verbs and nouns in handling player commands. So INVENTORY can be used as either.

```
action take inventory
   call inventory     # Execute code associated with the INVENTORY command
```

The ACTION directive takes one or two arguments. The first is the vocabulary word with which subsequent code is to be associated. The second argument is optional. If it is present, then the subsequent code will be executed only if the specified word also features in the player's command. (NB, I've dealt with INVENTORY before TAKE INVENTORY but that was just for clarity of exposition.)

Just like INIT and REPEAT (and also AT and PROC) there can be multiple ACTION sections for a given verb. The only object that can be picked up in this game is the cloak, so there is no need for generic ACTION TAKE – any

other attempt to use TAKE will be handled by the last REPEAT section as an error.

```
action take cloak
   ifhere cloak     # If cloak is at this location
      get cloak
      set cloak, 1  # The cloak is now carried
      set bar, 0    # and the bar is now dark
      quip "You [/pick/take] the cloak[/ up/ off the hook].", cloak
   fin
```

The IFHERE conditional return true if the nominated object is at the same location as the player (which also means not in player's possession!). If the cloak is absent, the error handling REPEAT section will handle it.

The response to success could have been just the generic "Taken" or similar, But I cannot resist showing of the feature of A-code which really caught my eye when I first encountered the language. The cloak has three states: 0 if it is carried/worn, 1 if it is lying on the floor and 2 if it is hanging on the hook. The QUIP directive will only be executed if the state is 1 or 2 and will automatically construct the appropriate response.

The DROP action is very similar:

```
action drop cloak
   ifhave cloak        # True only if the player has the cloak
      ifat cloakroom   # True only if player is at cloakroom
         drop cloak    # Move the cloak from player's inventory to cloakroom
         set cloak, 1  # It is now lying on the floor
         set bar, 1    # The bar is now lit
         quip "You drop the cloak."
      fin
      quip "This is not a good place to leave your cloak."
   fin
```

That handles all cases where the cloak is carried or worn by the player. Other possibilities are already covered by the general error handling in the last REPEAT section.

# More actions

By now you should have no problem following the below code for hanging the cloak. The only directive that needs explaining is IFNEAR CLOAK, which tests for the cloak being present, whether or not it is carried by the player. IFNEAR is a shorthand for IFHAVE followed by OR followed by IFHERE.

```
action hang cloak
   ifnear cloak
      ifat cloakroom      # True only if the player is at the cloakroom
         ifeq cloak, 2   # Already hanging -- nothing to do
           quip "It is already hanging on the hook!"
         fin
         ifhave, cloak   # True only if the cloak is in the players' possession
           drop cloak    # Make sure it is not carried
         fin
         set hook, 1     # Include the cloak in the hook's description
         set cloak, 2    # Set the cloak to have no description
         set bar, 1      # Bar is now lit
         quip "You hang the cloak on the hook."
      fin
   fin
```

That leaves just two more actions to be defined – READ and LOOK:

```
action read
   ifeq status, 1      # Player said READ - we'll default to READ MESSAGE
      or
   ifkey message       # Player actually said READ MESSAGE
      and
   ifat bar            #Player is at the bar
      and
   ifeq bar, 1         # If the bar is lit
      say message      # "Lost" or "won", depends on the message's value
      stop
   fin
```

Recall the as-you-go parsing of compound IFs. The above one says that if the player said READ or READ MESSAGE, and is at the bar, and the bar is lit then the message is displayed and the game terminated. Recall also that the value of message was originally set to 4 and decremented any time the player did something in darkness other than leaving the bar. The message description is a text switch and will give the "you win" win description for values 1 through 4, and the "you lose" one for message values of zero or less.

Finally, LOOK. If no object is given, it just needs to call the DESCRIBE.HERE procedure, which we have already constructed for use by the REPEAT loop. If an object is nominated and is nearby (carried or just present) and the object in question is the message, that's already covered by the READ action – so just CALL READ. Otherwise DESCRIBE ARG2 does the job.

```
action look
   ifeq status, 1
      call describe.here  # The procedure does not QUIT...
      quit                # ... so QUIT explicitly
   fin
   ifnear arg2
      ifkey message   # If the object to be describe is the message...
         call read    # ... just call the code for READ.
      fin
      describe arg2   # Otherwise give long object description
      quit
   fin
```

An that's it. Well, sort of...

# Finishing it off

What we have constructed so far in this tutorial is a full working code of the "Cloak of Darkness" nano-adventure, but any self-respecting game should carry some additional header information, which must come before anything else, such as

```
name Cloak of Darkness
version Tutorial.1.0
author Mike Arnautov
date 24 Feb 2024
style 12
```

All of these five header lines are optional and can occur in any order, but all must come before any other non-comment lines. The information they supply is stored in the games executable and can be displayed by running it with the command option -v (or /v for DOS/Windows). With the exception of the STYLE directive they have no other function. The STYLE line tells the acdc translator which major version of the A-code language is used by the game. It defaults to the current version (style 12).

And with that header, the game is complete. For your convenience, here is the final version, exactly as constructed in this tutorial. I've just added some comment line separators to enhance readability.

As for converting this code into a playeable executable, please see a separate document explaining how to do that. The simplest way is available if you have access to the bash command shell and have downloaded the current A-code tarball, in which case you can use the **advbld** script to do it for you – see the advbld documentation.

```
# Cloak of Darkness tutorial source code
#
name Cloak of Darkness
version Tutorial.1.0
author Mike Arnautov
date 24 Feb 2024
style 12
#----------------------------------------------------------------------
text YOU.ARRIVE
    The Cloak of Darkness

    Hurrying through the rain-swept November night, you're glad
    to see the bright lights of the Opera House. It's surprising
    that there aren't more people about but, hey, what do you
    expect in a cheap demo game...?

#----------------------------------------------------------------------
# Game specification mandates just the four cardinal directions
verb NORTH
verb EAST
verb SOUTH
verb WEST
#
verb GET, TAKE     # Both the player and game code can use these interchangeably
verb INVENTORY
verb DROP
verb READ          # For reading the message in the bar room
verb HANG          # To hang the cloak on the hook
verb LOOK
verb QUIT          # Not in the spec, but every game should allow quitting
#
noise THE, THAT, VELVET, GO     # Player command words to be ignored
#----------------------------------------------------------------------
object HOOK
  %A small brass hook is screwed to one wall.
  &It is just a small brass hook[/ with a cloak hanging on it].
#----------------------------------------------------------------------
object CLOAK
    A velvet cloak[ (worn)/]
  %A black velvet cloak [/lies on the floor/hangs on the hook].
  &It is a handsome cloak, of velvet trimmed with satin, and slightly
   spattered with raindrops. Its blackness is so deep that it
   almost seems to suck light from the room.
#----------------------------------------------------------------------
object MESSAGE
    The message[ has been carelessly trampled, making it difficult to read.
    You can just distinguish the words/, neatly marked in the sawdust,
    reads]...

    +YOU HAVE [LOST/WON] !!!
#----------------------------------------------------------------------
place FOYER
    You are standing in a spacious hall, splendidly decorated in red
    and gold, with glittering chandeliers overhead. The entrance from
    the street is to the north, and there are doorways south and west.
place CLOAKROOM
    The walls of this small room were clearly once lined with hooks,
    though now only one remains. The exit is a door to the east.
place BAR
    [It is too dark here to see anything!/The bar, much rougher than
```

```
        you'd have guessed after the opulence of the foyer to the north,
        is completely empty.

        A message is scratched in the sawdust on the floor.]
    #------------------------------------------------------------------------
    init
        goto foyer                # Move the player to the start location
        apport cloak, inhand      # Give him the cloak
        apport hook, cloakroom    # Put the hook where it belongs
        set message, 4            # Limits the number of moves in the bar
        say you.arrive
    #------------------------------------------------------------------------
    repeat
        ifflag status, moved    # Has the player moved?
            call describe.here  # If so describe their new location
        fin
    #------------------------------------------------------------------------
    flags variable
        MOVED       # Optional STATUS variable flag, maintained by the kernel
    #------------------------------------------------------------------------
    proc DESCRIBE.HERE
        local optr        # Local pointer to objects
        say here          # show description of current location
        itobj optr, here  # Loop through objects at this location
            say optr      # Show such objects
        fin               # Loop terminator
    #------------------------------------------------------------------------
    repeat
        set status, no.amatch      # Suppress approximate matching of command words
        input           # Get player's next command
        ifeq status, 2
            and
        iflt arg2, 0    # Can't find 2nd command word in the vocabulary
            or
        iflt arg1, 0    # Can't find 1st command word in the vocabulary
            quip "Pardon?"      # Command parse failure -- abort the main loop
        fin
    #------------------------------------------------------------------------
    repeat
        ifkey quit      # If the word QUIT occurs in the command
            say "As you wish."
            stop        # Exit the game
        fin
        call here       # Execute code, if any, associated with this location
        respond north, east, south, west, "There is no such exit here."
        call arg1       # Handle player command
        call bail.out   # Command not handled - report an error
    #------------------------------------------------------------------------
    proc bail.out
        local qualifier             # Local varaible initialised to zero
        ifeq status, 1              # If a single word command given
            ifflag arg1, object     # If that word is an object name
                ifnear arg1
                else
                    quip "There is no # here!", arg1
                fin
            else
                set qualifier, 1        # The word is not an object name, assume verb
            fin
            quip "You need to say what you want to [do with the /]{arg1}.", qualifier
        fin
```

```
   quip "You can't do that!"  # Generic no can do
#-----------------------------------------------------------------------
at foyer
   move south, bar
   move west, cloakroom
   respond north "You've only just arrived, and besides, the weather outside
      seems to be getting worse."
#-----------------------------------------------------------------------
at cloakroom
   move east, foyer
at bar
   move north, foyer
   ifloc cloak, cloakroom
   else                     # The cloak is not in the cloakroom
      sub message, 1        # Count attempted actions in the dark
      quip "Blundering around in the dark isn't a good idea!"
   fin
#-----------------------------------------------------------------------
action inventory
   local optr  # Local variable which will be used as a pointer to objects
   local count # Counts listed objects, automatically initialised to zero
   itobj optr, inhand # Loop through objects in player's inventory
      say optr        # Show object's description
      add count, 1    # Increment count of listed objects
   fin
   ifeq count, 0                          # If nothing listed, ...
      say "You are not carrying anything." # ... say empty-handed
   fin
   quit        # Command fully handled, so terminate the REPEAT loop
#-----------------------------------------------------------------------
action take inventory
   call inventory     # Execute code associated with the INVENTORY command
#-----------------------------------------------------------------------
action take cloak
   ifhere cloak     # If cloak is at this location
      get cloak
      set cloak, 1  # The cloak is now carried
      set bar, 0    # and the bar is now dark
      quip "You [/pick/take] the cloak[/ up/ off the hook].", cloak
   fin
#-----------------------------------------------------------------------
action drop cloak
   ifhave cloak       # True only if the player has the cloak
      ifat cloakroom  # True only if player is at cloakroom
         drop cloak   # Move the cloak from player's inventory to cloakroom
         set cloak, 1 # It is now lying on the floor
         set bar, 1   # The bar is now lit
         quip "You drop the cloak."
      fin
      quip "This is not a good place to leave your cloak."
   fin
#-----------------------------------------------------------------------
action hang cloak
   ifnear cloak
      ifat cloakroom      # True only if the player is at the cloakroom
         ifeq cloak, 2    # Already hanging -- nothing to do
           quip "It is already hanging on the hook!"
         fin
         ifhave, cloak    # True only if the cloak is in the players' possession
           drop cloak     # Make sure it is not carried
         fin
```

```
            set hook, 1     # Include the cloak in the hook's description
            set cloak, 2    # Set the cloak to have no description
            set bar, 1      # Bar is now lit
            quip "You hang the cloak on the hook."
         fin
    fin
    #----------------------------------------------------------------------
    action read
       ifeq status, 1    # Player said READ - we'll default to READ MESSAGE
          or
       ifkey message       # Player actually said READ MESSAGE
          and
       ifat bar            #Player is at the bar
          and
       ifeq bar, 1       # If the bar is lit
          say message    # "Lost" or "won", depends on the message's value
          stop
       fin
    #----------------------------------------------------------------------
    action look
       ifeq status, 1
          call describe.here  # The procedure does not QUIT...
          quit                # ... so QUIT explicitly
       fin
       ifnear arg2
          ifkey message   # If the object to be describe is the message...
             call read    # ... just call the code for READ.
          fin
          describe arg2   # Otherwise give long object description
          quit
       fin
    #----------------------------------------------------------------------
```