

Foyer of the Opera House

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.

> i

You have no possessions. You're wearing a velvet cloak.

> (now) (#hook is #heldby #player)

> i

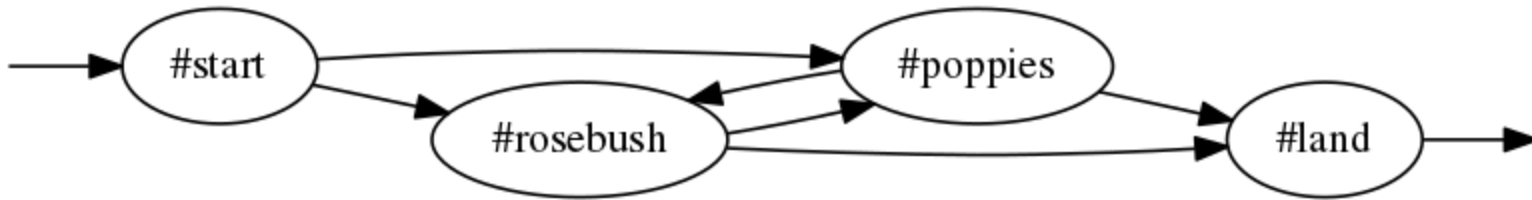
You have a small brass hook. You're wearing a velvet cloak.

> i

The source code has been modified. Merging changes into the running program.

You have a small brass hook. You're wearing a flamboyant yellow hat and a velvet cloak.

> █



(try \$)

(refuse \$)

*(before \$)

(refuse \$)

(instead of \$)

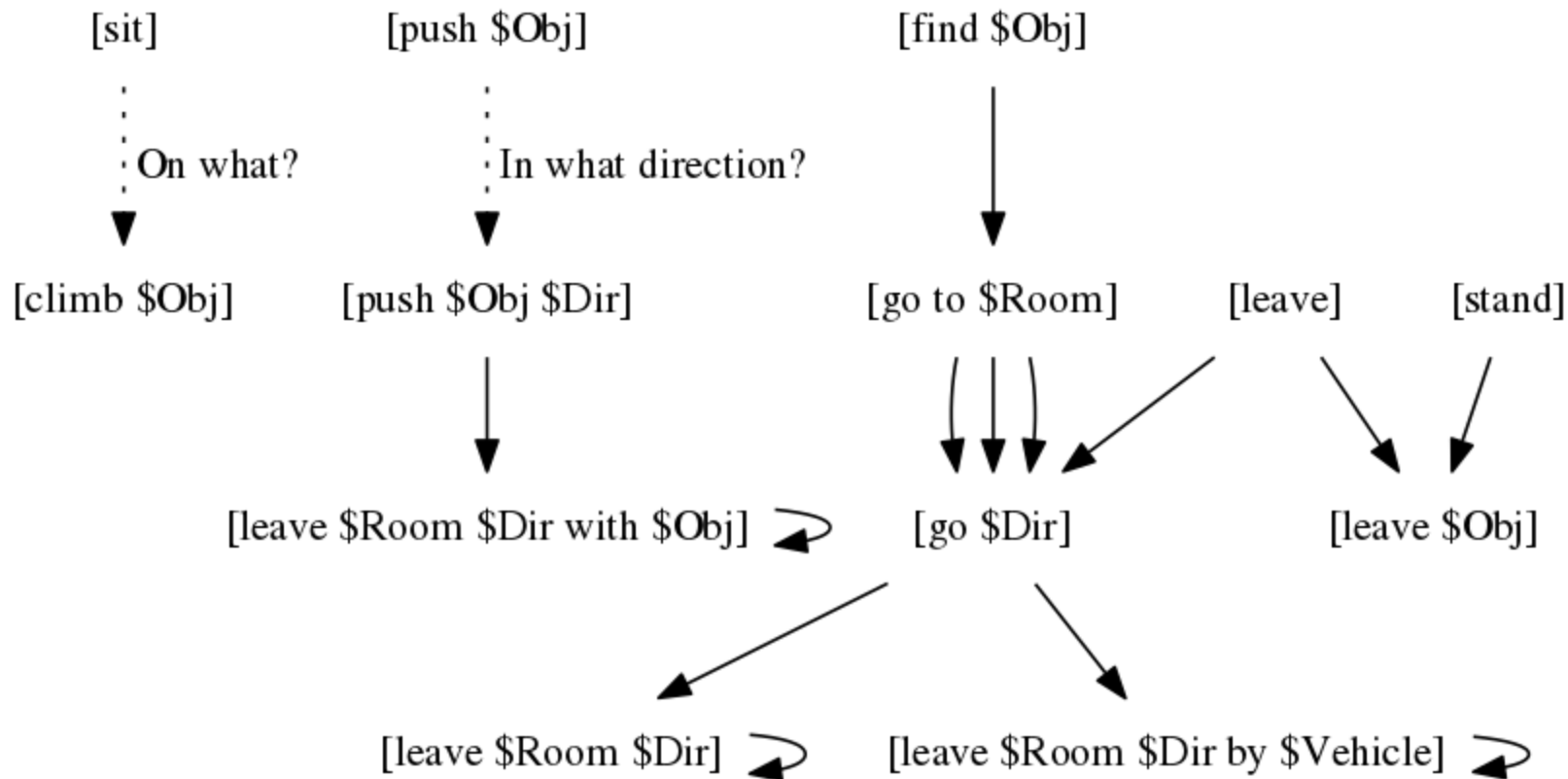
(prevent \$)

(perform \$)

*(after \$)

Action-specific predicates
e.g. (descr \$Obj) or
(narrate leaving \$Room \$Dir)

Time →



(excluded from all \$)



(out of reach \$)

(not here \$)

(fine where it is \$)

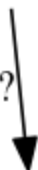
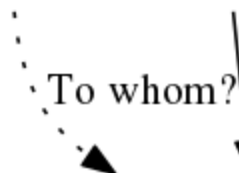
(intangible \$)

(topic \$)

[call]



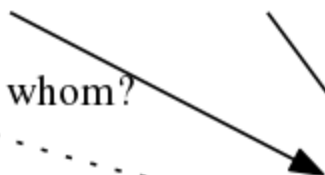
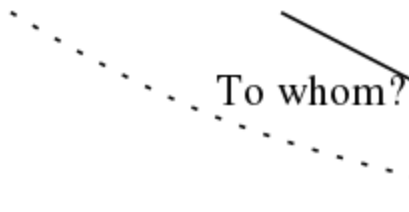
[shout] [call \$Obj] [tell \$Obj to l \$Action] [ask \$Obj about \$Topic] [tell \$Obj about \$Topic]



If \$Action is [greet]



[talk] [shout to \$Obj] [greet \$Obj] [talk to \$Obj about \$Topic] [ask \$Obj] [tell \$Obj]



[talk to \$Obj]

(container \$)

(supporter \$)



(actor container \$)

(seat \$)

(actor supporter \$)



(room \$)

(in-seat \$)

(on-seat \$)

(door \$)

(vehicle \$)

(direction \$)

(relation \$)

(opaque \$)



(animate \$)

(openable \$)

(item \$)



(female \$)

(male \$)

(lockable \$)

(wearable \$)

(potable \$)

(edible \$)

(sharp \$)

(consultable \$)

(pushable \$)

(switchable \$)

Chapter 4: More control structures

([If-statements](#) • [Negation](#) • [Selecting among variations](#) • [Closures](#) • [Stoppable environments](#))

If-statements

We have seen how execution can be directed into different branches based on arbitrary conditions, with the help of conjunctions and disjunctions. It is certainly possible to code up a simple if-then-else routine using these constructs alone:

(eat \$Obj)

You take a

```
(eat $Obj)
You take a large bite, and conclude that it is
{
  { (fruit $Obj) (or) (pastry $Obj) }
  sweet
  (or)
  ($Obj = #steak) (player eats meat)
  savoury
  (or)
  inedible
}.
```

[\[Copy to clipboard\]](#)

But this quickly becomes unreadable as the code grows in complexity. Dialog provides traditional if, then, elseif, else, and endif keywords:

(if)

<i>condition</i>

```
(if) condition (then)
statements
(elseif) other condition (then)
statements
(elseif) other condition (then)
statements
...
(else)
statements
(endif)
```

[\[Copy to clipboard\]](#)

The body of code between (if) and (then) is called a *condition*. If the condition succeeds, possibly binding variables in the process, then any choice points created by the condition are discarded, and execution proceeds with the statements in the corresponding then-clause. Should the condition fail, no variables are bound, and Dialog considers the next (elseif) condition, and so on, eventually falling back on the else-clause if none of the conditions were successful. The (elseif) and (else) clauses are optional; if there is no else clause, it is assumed to be empty (i.e. succeeding). The entire if-statement succeeds if and only if the chosen branch succeeds. Note that if the chosen (then) or (else) block creates choice points, those remain in effect. It is only the conditions that are limited to a single solution.

(eat \$Obj)

You take a

```
(eat $Obj)
You take a large bite, and conclude that it is
(if) (fruit $Obj) (or) (pastry $Obj) (then)
  sweet
(elseif) ($Obj = #steak) (player eats meat) (then)
  savoury
(else)
  inedible
(endif).
```

[\[Copy to clipboard\]](#)

There are subtle differences between the if-statement above and the disjunction shown earlier: An if-condition is evaluated at most once, even if it creates choice points. As soon as one of the then-clauses is entered, all the remaining then-clauses and the else-clauses become inaccessible. And, finally, if-statements without else-clauses succeed when none of the conditions are met (i.e. a blank else-clause is assumed).

In the disjunction-based version of the rule, there are several lingering choice points, so if a failure is encountered further down in the rule (or even in the calling rule, if this was a multi-query), then execution might resume somewhere in the middle of this code, printing half-sentences as a result. When that happens, it is almost always due to a bug elsewhere in the program.

Technically the disjunction-based version works just as well as the if-based version. But in the spirit of defensive programming, it's generally a good idea to stick to if-statements when writing code with side-effects, such as printing.

Negation

Is there a way to test whether a query fails? We can certainly do it with an if-else construct:

```
(if) (my little
query) (then) (fail)
```

(if) (my little query) (then) (fail) (endif) My little query failed.

[\[Copy to clipboard\]](#)

But Dialog provides a shorthand syntax for this very common operation. By prefixing a query with a tilde character (~, pronounced “not”), the query succeeds if there is no solution, and fails if there is at least one solution. The following code is equivalent to the if-statement above:

```
~(my little
query) My little
```

~(my little query) My little query failed.

[\[Copy to clipboard\]](#)

It also works for blocks:

```
~{ (my little
query) (my other
```

~{ (my little query) (my other little query) }

At least one of the little queries failed.

[\[Copy to clipboard\]](#)

which is equivalent to:

```
(if) (my little
query) (my other
```

(if) (my little query) (my other little query) (then) (fail) (endif)

At least one of the little queries failed.

[\[Copy to clipboard\]](#)

Dialog also allows us to define rules with negated rule heads. When such a rule succeeds, the query fails immediately, and none of the remaining rules are considered. Negated rules could be thought of as having an implicit (just) (fail) at the end.

```
(fruit #apple)
(fruit #banana)
```

(fruit #apple)

(fruit #banana)

(fruit #orange)

(fruit #pumpkin)

(sweet #cookie)

~(sweet #pumpkin) %% Equivalent to: (sweet #pumpkin) (just) (fail)

(sweet \$Obj)

(fruit \$Obj)

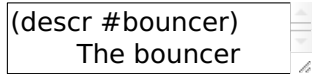
[\[Copy to clipboard\]](#)

Selecting among variations

When writing interactive fiction, it can be nice to be able to add a bit of random variation to the output, or to step through a series of responses to a particular command. Dialog provides this functionality through a mechanism that is

respectfully stolen from the Inform 7 programming language.

To select randomly among a number of code branches, use the expression (select) ...*alternatives separated by (or)*... (at random):



```
(descr #bouncer)
  The bouncer
  (select)
    eyes you suspiciously
  (or)
    hums a ditty
  (or)
    looks at his watch
  (at random).
```

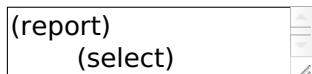
```
(descr #bouncer)
  The bouncer
  (select)
    eyes you suspiciously
  (or)
    hums a ditty
  (or)
    looks at his watch
  (at random).
```

[\[Copy to clipboard\]](#)

Note that (or) just revealed itself to be an overloaded operator: When it occurs immediately inside a select expression, it is used to separate alternatives. When it is used anywhere else, it indicates disjunction.

Select-at-random never picks the same branch twice in succession, to avoid jarring repetitions in the narrative. If a uniform distribution is desired, e.g. for implementing a die, an alternative form is available: (select) ... (purely at random).

To advance predictably through a series of alternatives, and then stick with the last alternative forever, use (select) ... (stopping):



```
(report)
  (select)
    This is printed the first time.
  (or)
    This is printed the second time.
  (or)
    This is printed ever after.
  (stopping)
  (line)
  (program entry point)
```

```
(report)
  (select)
    This is printed the first time.
  (or)
    This is printed the second time.
  (or)
    This is printed ever after.
  (stopping)
  (line)
  (program entry point)
  (report)
  (report)
  (report)
  (report)
```

[\[Copy to clipboard\]](#)

The output of that program is:

```
This is printed the first time.
This is printed the second time.
This is printed ever after.
This is printed ever after.
```

A combination of predictability and randomness is offered by the following two forms, where Dialog visits each alternative in turn, and then falls back on the specified random behaviour:

(select) ...*alternatives separated by (or)*... (then at random)

(select) ...*alternatives separated by (or)*... (then purely at random)

To advance predictably through a series of alternatives, and then start over from the beginning, use:

(select) ...*alternatives separated by (or)*... (cycling)

The three remaining variants from Inform 7 are currently not supported by Dialog.

Closures

A *closure* is an anonymous bit of code that can be kept as a value, and invoked at a later time.

A closure definition in curly braces can appear at any place where a value is expected. Once a closure has been created, it can be invoked using the built-in predicate (query \$).

Example:

```
($X = { Hello,
world! })
```

```
($X = { Hello, world! })
```

%% Nothing is printed yet, but \$X is bound to the code in braces.

```
(query $X)    %% This will print "Hello, world!"
```

[\[Copy to clipboard\]](#)

A closure captures the environment surrounding its definition. This means that the same local variables are accessible both inside and outside of the brace-expression. In the following example, the closure is created with a reference to the local variable \$X. Afterwards, the same variable is bound to a dictionary word.

```
(program entry
point)
```

```
(program entry point)
```

```
($Closure = { Hello, $X! })
```

```
($X = @world)
```

```
(query $Closure)
```

[\[Copy to clipboard\]](#)

The output is:

Hello, world!

It is possible to make multi-queries to closures. The following program:

```
(program entry
point)
```

```
(program entry point)
```

```
(exhaust) {
```

```
*(query { Veni (or) Vidi (or) Vici })
```

```
!
```

```
}
```

[\[Copy to clipboard\]](#)

produces the following output:

Veni! Vidi! Vici!

It is also possible to invoke a closure with a single parameter, using an alternative form of the query builtin: (query \$Closure \$Parameter). The parameter is accessible from within the closure, by means of the special variable \$_. Here is an example:

```
(program entry
point)
```

```
(program entry point)
```

```
($Greeter = { Hello, $_! })
```

```
(query $Greeter @world)
```

```
(query $Greeter @indeed)
```

[\[Copy to clipboard\]](#)

The output is:

Hello, world! Hello, indeed!

Under the hood, closures are actually lists. The first element is a number, assigned by the compiler to differentiate between the various closure definitions appearing in the program. The remaining elements, if any, are bound to local variables from the environment surrounding the closure definition.

Thus, there is no way to check at runtime whether a value is a closure, or just an ordinary list that happens to begin with a number.

Stoppable environments

Dialog provides a mechanism for non-local returns, similar to exceptions in other programming languages. By prefixing

a statement (such as a query or a block) with the keyword (stoppable), that statement will execute in a *stoppable environment*. If the built-in predicate (stop) is queried from within the statement, at any level of nesting, execution immediately breaks out of the (stoppable) environment. If the statement terminates normally, either by succeeding or failing, execution also resumes after the (stoppable) construct; (stoppable) never fails. Regardless of how the stoppable environment is left, any choice points created while inside it are discarded.

Stoppable environments can be nested, and (stop) only breaks out of the innermost one. A stop outside of any stoppable environment terminates the program.

Here is a convoluted example:

(routine)

this (stop) (or)

(routine)
this (stop) (or) that
(program entry point)
{ Let's (or) now. (stop) }
(stoppable) {
take
(routine)
another
}
shortcut
(fail)

[\[Copy to clipboard\]](#)

The printed output is:

Let's take this shortcut now.

The standard library uses stoppable environments to allow action-handling predicates to stop further actions from being processed. For instance, TAKE ALL may result in several actions being processed, one at a time. If taking the booby-trapped statuette triggers some dramatic cutscene, the code for that cutscene can invoke (stop) to prevent spending time on taking the other items.

Onwards to “[Chapter 5: Input and output](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 5: Moving around

([Rooms and map connections](#) • [Floating objects](#) • [Regions](#) • [Light and darkness](#) • [Reachability, visibility, and scope](#) • [Doors and locks](#) • [Moving the player character](#) • [Path finding](#))

Rooms and map connections

So far, all our example games have had just one room. To add more, we declare some objects that have the (room \$) trait. Room-to-room connections are defined using the (from \$ go \$ to \$) predicate. The first parameter is the present room, the second parameter is a compass direction, and the third parameter is the neighbouring room in that direction.

The standard library provides twelve pre-defined directions: #east, #northeast, #north, #northwest, #west, #southwest, #south, #southeast, #up, #down, #in, and #out. Story authors can add more directions (such as starboard and port) by copying the rule definitions for one of the standard directions from the library, and editing them.

Here is a small, working game with three rooms:

#library
(room *)

#library
(room *)
(name *) library
(from * go #east to #foyer)
#foyer
(room *)
(name *) foyer
(from * go #west to #library)
(from * go #south to #study)
#study
(room *)
(name *) study
(from * go #north to #foyer)
#player
(current player *)
(* is #in #library)

[Copy to clipboard]

Try navigating by compass directions, as well as EXITS and e.g. GO TO STUDY.

Rooms have names, and optionally dict synonyms, like any other object. In Dialog games, neighbouring rooms are in scope by default, although the player can't do all that much with them, except ENTER them. But commands such as EXITS have to be able to print the name of the room as an ordinary noun that's part of a sentence.

Of course, you are free to make use of the linguistic traits to affect how the room name is presented in various contexts. In particular, (singleton \$) and (proper \$) can be used to give room names a more rigid appearance:

#library
(room *)

#library
(room *)
(singleton *)
(name *) university library
(from * go #east to #foyer)
#foyer
(room *)
(singleton *)
(name *) grand foyer
(from * go #west to #library)
(from * go #south to #study)
#study
(room *)
(proper *)
(name *) Professor Stroopwafel's study

(from * go #north to #foyer)

#player

(current player *)

(* is #in #library)

[\[Copy to clipboard\]](#)

When the player is inside a room, the name of the room is usually displayed in the status bar, and as a bold header before the room description. This piece of text is called the *room header*. The default room header is simply the room name, with the first character converted to uppercase. But it is possible to override the (room header \$) rule:

```
#belowcliff
(room *)
```

#belowcliff

(room *)

(singleton *)

(name *) area below the cliff

(room header *) Below the cliff

[\[Copy to clipboard\]](#)

This will make “Below the cliff” the room header, even though the room turns up as “the area below the cliff” in e.g. EXITS listings.

Sometimes, two or more directions point towards the same exit. For instance, from a rooftop, there might be a ladder leading down along the eastern wall of the building. Both DOWN and EAST should take the player to the location below the building, but the exit should only appear once in the EXITS listing. To achieve this, we regard one of the directions as secondary, and *redirect* it onto the other. Example:

```
(from #rooftop go
#down to
```

(from #rooftop go #down to #parkinglot)

(from #rooftop go #east to #down)

[\[Copy to clipboard\]](#)

which could lead to the following exchange:

> EXITS

Obvious exits are:

Down to the parking lot.

> E

You climb down.

Redirecting #in and #out is particularly useful.

When there's no exit in a particular direction, Dialog allows you to specify that an object is located that way. This makes it possible for the player to e.g. LOOK NORTH to examine a large mural on the wall there. This functionality is implied by the (from \$Room go \$Direction to \$Target) predicate, whenever \$Target is neither a room, a direction, nor a door (to be described shortly). Thus:

```
(from #rooftop go
#up to #sky)
```

(from #rooftop go #up to #sky)

[\[Copy to clipboard\]](#)

Such objects, like redirections, do not appear in EXITS listings.

What you are describing with (from \$ go \$ to \$) are the so called *obvious exits* from a room. What actually happens when a player tries to move in a particular direction, depends on how the [leave ...] family of actions are handled. We will return to that in the chapter on [actions](#); for now just keep in mind that the act of moving in a compass direction can be intercepted by story code, in order to block obvious exits, allow movement in non-obvious directions, trigger cutscenes, or anything else. But in the absense of any such intercepting code, the default behaviour of the [leave ...] actions (and [exits], and [go to \$]) is to make use of the obvious exits.

Floating objects

Floating objects are objects that appear to exist in several rooms at once. This is an illusion, created by moving the floating objects whenever the player character moves. The movement is handled by the standard library, based on what you declare using the (\$Room attracts \$Object) predicate. Thus:

#wallpaper	
(name *)	

#wallpaper
(name *) wallpaper
(descr *) Brown and austere.
(#library attracts *)
(#foyer attracts *)
(#study attracts *)

[\[Copy to clipboard\]](#)

Objects around the perimeter of a room, such as doors and e.g. #sky in (from #rooftop go #up to #sky), are attracted automatically. Thus:

#floor	
(name *)	floor

#floor
(name *) floor
(singleton *)
(descr *) Vinyl, with a marble pattern.
(from #library go #down to *)
(from #foyer go #down to *)
(from #study go #down to *)

%% The following rule definitions aren't necessary:
%% (#library attracts *)
%% (#foyer attracts *)
%% (#study attracts *)

[\[Copy to clipboard\]](#)

Regions

Rooms can often be classified into a number of conceptual *regions* (possibly by geographical proximity), such as “outdoors” or “in the dungeon”. Rooms that belong to the same region tend to share properties, such as what floating objects they attract.

In Dialog, regions are modelled using traits. Thus, we might create a trait (indoors room \$) that inherits most of its behaviour from the (room \$) trait, but also attracts a certain set of floating objects:

%% Every indoors-room is a room.	
----------------------------------	--

%% Every indoors-room is a room.
%% Phrased differently, an object is a room given that it's an indoors-room:
(room *(indoors-room \$))

#wallpaper
(name *) wallpaper
(descr *) Brown and austere.
((indoors-room \$) attracts *)

#floor
(name *) floor
(singleton *)
(descr *) Vinyl, with a marble pattern.
(from (indoors-room \$) go #down to *)

#foyer
(indoors-room *)
(name *) grand foyer
(singleton *)
(from * go #south to #study)

#study
(indoors-room *)
(name *) Professor Stroopwafel's study
(proper *)
(from * go #north to #foyer)

(from * go #out to #north)

#player

(current player *)

(* is #in #library)

[\[Copy to clipboard\]](#)

If you try it out, you'll find that it's possible to walk around and examine the floor and wallpaper from within either room.

For very simple regions, another option is to use [slash expressions](#):

```
(#foyer/#study/#library attracts
```

```
#wallpaper)
```

```
(from #foyer/#study/#library go #down to #floor)
```

[\[Copy to clipboard\]](#)

Light and darkness

Light travels up and down the object tree. It can pass between a child and its parent, unless the parent is opaque and the child is #under it, or the parent is closed and opaque and the child is #in it. Openable objects are opaque by default.

An object is *illuminated* by another object if the latter provides light, and light can pass between the objects.

By default, rooms are assumed to contain ambient light, so they act as light sources. Hence, most objects are illuminated by default. But it is possible to disable the ambient lighting for any room, by adding a rule to the (inherently dark \$) predicate:

```
#cave
(name *)    cave
```

```
#cave
```

```
(name *)    cave
```

```
(room *)
```

```
(inherently dark *)
```

[\[Copy to clipboard\]](#)

Now, if the player enters that room, they will be in darkness unless the room is illuminated by some other object that provides light:

```
#lamp
(name *)    lamp
```

```
#lamp
```

```
(name *)    lamp
```

```
(item *)
```

```
(* provides light)
```

[\[Copy to clipboard\]](#)

Rule definitions for the (\$ provides light) predicate often contain conditions in the rule body. For instance, a flashlight might provide light when it is switched on:

```
#flashlight
(name *)
```

```
#flashlight
```

```
(name *)    flashlight
```

```
(item *)
```

```
(switchable *)
```

```
(* provides light) (* is on)
```

[\[Copy to clipboard\]](#)

Note: The standard library makes a [multi-query](#) to (\$ provides light), in order to iterate over every object that currently provides light. Be sure to add asterisks to the rule body as required, for instance if you define a trait for light-providing objects:

```
($Obj provides light)
*(lamp $Obj)
```

```
($Obj provides light)
```

*(lamp \$Obj) %% The asterisk is crucial.
(\$Obj is on)

[\[Copy to clipboard\]](#)

The standard library provides a predicate, (player can see), that succeeds when the current player character is illuminated.

Reachability, visibility, and scope

An object is within *reach* of another object (such as the player character) when there is a path between them, via child-parent relations in the object tree, that doesn't pass through a closed object. Objects that are nested #under an object that is #wornby somebody other than the current player character are also considered out of reach. Finally, objects may explicitly be declared (out of reach \$).

An object is *visible* to another object (such as the player character) when they are both illuminated, and there is a path between them, via child-parent relations in the object tree, that doesn't 1. pass through a closed, opaque object, or 2. pass underneath an opaque object. Openable objects are opaque by default.

Neither reach nor visibility extends across room boundaries, but doors and other objects that are located at the perimeter of the current room, using (from \$ go \$ to \$), are automatically moved into the room.

To check whether an object is currently visible to the player, use (player can see \$).

There is currently no simple, generic way to check whether an object is visible to some other object (e.g. a non-player character), because of the way floating objects and moving light sources are handled. But for a given story, it is often sufficient to make a pragmatic approximation, such as whether the observer and the object are in the same room.

Most actions require reachability. Of the ones that don't, some (e.g. LOOK IN) explicitly require visibility. Normally, anything that is reachable is also visible (but something that is visible might be in a closed, transparent container, and hence not reachable). But in darkness, objects tend to be reachable but not visible.

Under certain circumstances, when a player looks through a door (e.g. by looking in a compass direction), the name of the room on the other side is printed. But that is handled separately from the formal concept of visibility described here.

At any given time, a subset of the objects in the game world are considered to be *in scope*. These are the only objects that the player may currently refer to, i.e. the only objects that the parser will understand. The predicate (\$ is in scope) can be used to check whether a given object is in scope, or, with a [multi-query](#), to backtrack over every object in scope.

The default scope is everything that the player can see or reach, plus objects that are marked out of reach but would be reachable otherwise. If the current room is in scope and the player can see, neighbouring rooms are also added to the scope.

If the player cannot see, the intangible object #darkness (responding to DARKNESS and DARK) is automatically added to the scope. By default, the player can't do much with this object except examine it, which invokes (narrate darkness), printing “You are surrounded by darkness”.

It is possible to add other objects to the scope using the predicate (add \$ to scope), typically with some condition, like this:

(add #mother to scope)

(add #mother to scope)
(current room #phonebooth)

[\[Copy to clipboard\]](#)

That rule allows the parser to recognize e.g. CALL MOTHER when the player is in the phone booth.

Doors and locks

Map connections can also involve *doors*. A door is a gatekeeper object (representing a physical door, an opening, or something else entirely) that either blocks or allows passage.

Whether a door admits passage or not, and whether it's possible to peek at the room on the other side, is determined by the predicates (\$ blocks passage) and (\$ blocks light), respectively. In the standard library, they are defined as follows:

(\$Door blocks passage)

(\$Door blocks passage)
(\$Door is closed)
((opaque \$Door) blocks light)
(\$Door is closed)

[\[Copy to clipboard\]](#)

Openable objects are closed and opaque by default. If you are implementing a physical door, remember to declare it openable, (openable *), and optionally to specify that it starts out open, (* is open).

The standard library provides two mechanisms for setting up door connections. The low-level method involves setting up two rules, one for the predicate (from \$Room go \$Direction to \$Door) and one for its companion (from \$Room through \$Door to \$Target). The high-level method is to use an access predicate that defines both rules in one go: (from \$Room go \$Direction through \$Door to \$Target). Let's build a door using the high-level method:

```
#foyer
(room *)
```

```
#foyer
(room *)
(name *) grand foyer
(singleton *)
(from * go #south through #door to #study)
```

```
#study
(room *)
(name *) Professor Stroopwafel's study
(proper *)
(from * go #north through #door to #foyer)
(from * go #out to #north)
```

```
#door
(door *)
(openable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.
```

```
#player
(current player *)
(* is #in #library)
```

[\[Copy to clipboard\]](#)

Doors usually appear in (from \$ go \$ to \$) rules, and are therefore automatically treated as floating objects. So in the above game, you'll be able to EXAMINE DOOR, OPEN DOOR, CLOSE WOODEN etc. from either side of the door.

Doors can be *locked*. An object that is locked, (\$ is locked), can't be opened (by the default behaviour of the [open \$] action). But a *lockable* object, (lockable \$), can be locked or unlocked with the right key. By trait inheritance, lockable objects are also openable. They start out locked and closed, unless you specify otherwise.

Keys are associated with lockable objects using the predicate (\$ unlocks \$).

```
#door
(door *)
```

```
#door
(door *)
(lockable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.
```

```
#key
(item *)
(name *) small key
(* unlocks #door)
```

[\[Copy to clipboard\]](#)

The standard actions are set up so that an attempt to walk through a closed door first triggers an automatic [open \$] action, which in turn may trigger an automatic [unlock \$ with \$] action if the door was locked. But the latter only happens if the player is holding the right key at the time.

Now that we know about locked doors and keys, we can create a small, playable puzzle game:

```
#library
(room *)
```

```
#library
(room *)
(singleton *)
```

```

(name *) university library
(look *)  What a strange library. There's just a rug in here.
          (notice #rug)
          The exit is east.
(from * go #east to #foyer)
(from * go #out to #east)

#rug
(name *) rug
(* is #in #library)

#key
(item *)
(name *) small key
(descr *) It's a small key, of the kind that unlocks doors.
(* is #under #rug)
(* unlocks #door)

#foyer
(room *)
(singleton *)
(name *) grand foyer
(look *)  It's a grand, grand foyer.
          The library is west from here, and a
          (if) (#door is locked) (then) locked (endif)
          door leads south.
(from * go #west to #library)
(from * go #south through #door to #study)
(from * go #in to #south)

#study
(room *)
(name *) Professor Stroopwafel's study
(look *)  You solved the mystery of the locked door!
          (game over { You win! })

(proper *)
(from * go #north through #door to #foyer)
(from * go #out to #north)

#door
(door *)
(lockable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.
          It is currently
          (if) (* is locked) (then)
          locked.
          (else)
          unlocked.
          (endif)

#player
(current player *)
(* is #in #foyer)

```

[\[Copy to clipboard\]](#)

Moving the player character

The standard library uses a few [global variables](#) internally, of which (current player \$) is particularly noteworthy. Story code may query this variable at any time, but mustn't update it directly using (now); that would confuse the library. The proper way to change the current player character is to make a query to (select player \$).

But while you're not allowed to modify the (current player \$) variable directly from within story code, you are expected to supply an initial value for it:

```
(current player #me)
```

(current player #me)

[\[Copy to clipboard\]](#)

Likewise, it is straightforward to define the initial location of the player character:

(#me is #in #study)

(#me is #in #study)

[\[Copy to clipboard\]](#)

But the location of the current player character must be changed with a query to either (move player to \$Relation \$Parent) or (enter \$Room). The latter also prints the description of the new room (by invoking the [look] action).

Another global variable used by the library is (current room \$). From the point of view of the story author, this could just as well have been a regular predicate that traverses the object tree in order to find the room that's currently enclosing the player character. But it is a global variable for performance reasons. When the player character is moved properly, by querying (move player to \$ \$) or (enter \$), the value of this variable is updated to reflect the new location. Another thing that happens is that floating objects are moved into position.

The library uses a helper predicate called (update environment around player) to carry out the updates described above. Occasionally, it can be useful to query this predicate directly from story code.

Path finding

The standard library predicate (shortest path from \$Room1 to \$Room2 is \$Path) computes the shortest path from \$Room1 to \$Room2, by considering the obvious exits listed in (from \$ go \$ to \$) and (from \$ through \$ to \$). The result is a list of directions.

The predicate (first step from \$Room1 to \$Room2 is \$Direction) computes the same path, but returns only the first step. This is functionally equivalent to (shortest path from \$Room1 to \$Room2 is [\$Direction | \$]), but slightly faster and more memory efficient.

The computed path only includes visited rooms, and doesn't pass through closed doors. But it is straightforward to modify the library to relax those conditions.

Onwards to “[Chapter 6: Actions](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

The Dialog Manual

This is Revision 31 of the Dialog Manual. It describes **version 0m** of the Dialog language, and **version 0.46** of its standard library. Dialog is currently in its *beta* stage, which means that the language may still undergo significant changes.

Preface

[Introduction](#)

Overview • Structure of the manual • Acknowledgements

[Software](#)

The compiler • The interactive debugger • Building from source code

Part I: The Programming Language

[Chapter 1: Flow of execution](#)

Predicates and rules • Printing text • Parameters, objects, and wildcards • Success and failure

[Chapter 2: Manipulating data](#)

Local variables • Values • Unification • Partial lists and recursion

[Chapter 3: Choice points](#)

Disjunctions • Backtracking • Multi-queries • Visiting all solutions • Collecting values • Collecting words • Accumulating numbers • Just • Infinite loops

[Chapter 4: More control structures](#)

If-statements • Negation • Selecting among variations • Closures • Stoppable environments

[Chapter 5: Input and output](#)

Divs, spans, and style classes • Case conversion and inline styles • Status areas • Visualizing progress • Clearing the screen • Input • Hyperlinks • Resources • Debugging • Determining objects from words

[Chapter 6: Dynamic predicates](#)

Global flags • Per-object flags • Global variables • Per-object variables • Has parent

[Chapter 7: Syntactic sugar](#)

Access predicates • The current topic • Nested queries in rule heads • Alternatives in rule heads • Automated object generation

[Chapter 8: More built-in predicates](#)

Checking the type of a value • Numbers and arithmetic • List-related predicates • Manipulating dictionary words • System control

[Chapter 9: Beyond the program](#)

Story metadata • Interfaces • Runtime errors • Some notes on performance • Limitations and the future of Dialog

[Appendix: Quick reference](#)

Part II: The Standard Library

[Chapter 1: Getting started](#)

Architecture of the library • Running the code examples • A minimal story

[Chapter 2: Objects and state](#)

Populating the game world • Descriptions, appearances, and synonyms • Defining new predicates • Object locations • Dynamic predicates • Hidden objects

[Chapter 3: Traits](#)

Custom traits • Linguistic predicates and traits • Full names • Standard traits for categorizing objects

[Chapter 4: Items](#)

Pristine and handled objects • Plural forms • All about appearances • Pristineness of nested objects • Clothing

[Chapter 5: Moving around](#)

Rooms and map connections • Floating objects • Regions • Light and darkness • Reachability, visibility, and scope • Doors and locks • Moving the player character • Path finding

[Chapter 6: Actions](#)

Introduction to actions • How actions are processed • Stopping and ticking • Instead of: Prevent, perform, after • Narration predicates • Diversion • Refuse and before • Group actions

[Chapter 7: The standard actions](#)

Core actions • Actions that reveal information • Actions that print a message • Diverting actions • Communication • Navigation • Miscellaneous actions • Debugging actions

[Chapter 8: Ticks, scenes, and progress](#)

Timed code • Cutscenes • The intro • Keeping score • The status bar • Game over • Choice mode

[Chapter 9: Non-player characters](#)

Movement • Other NPC actions • Taking orders • Ask and tell • Choice-based conversation

[Chapter 10: Understanding player input](#)

Grammar definitions • Adding actions • Adjusting the likelihood of actions • Links and default actions • Example: Defining a new action • How the parser works • Custom grammar tokens

[Chapter 11: Miscellaneous features](#)

Pronouns • List manipulation • Object tree manipulation • Directions and numbers • Predicates for debugging • Common checks and complaints • Asking simple questions • Identical objects

[Appendix: Predicate index](#)

Chapter 10: Understanding player input

([Grammar definitions](#) • [Adding actions](#) • [Adjusting the likelihood of actions](#) • [Links and default actions](#) • [Example: Defining a new action](#) • [How the parser works](#) • [Custom grammar tokens](#))

Actions originate from the *parser*, a central part of the standard library. This chapter describes how to extend the parser to understand new kinds of sentences.

Grammar definitions

Player input is mapped to actions using (grammar \$ for \$). For instance, the following definition causes TAKE A NAP to be recognized as an attempt to perform the [sleep] action:

```
(grammar [take a  
nap] for [sleep])
```

```
(grammar [take a nap] for [sleep])
```

[\[Copy to clipboard\]](#)

There is no automatic rule for recognizing the words that make up the internal name of the action, so the following definition is probably also desired:

```
(grammar [sleep] for  
[sleep])
```

```
(grammar [sleep] for [sleep])
```

[\[Copy to clipboard\]](#)

But in this case, there is also a shorthand form:

```
(grammar [sleep] for  
itself)
```

```
(grammar [sleep] for itself)
```

[\[Copy to clipboard\]](#)

[Slash-expressions](#) are useful:

```
(grammar  
[sleep/nap/dream])
```

```
(grammar [sleep/nap/dream] for [sleep])
```

[\[Copy to clipboard\]](#)

The left-hand side of a grammar definition may contain special *tokens*, standing in for objects. Thus, [object] represents an arbitrary object in scope, or even many objects (separated by commas or AND). On the right-hand side, a wildcard (\$) indicates the place in the action where the object is supposed to go:

```
(grammar  
[buy/purchase
```

```
(grammar [buy/purchase [object]] for [buy $])
```

[\[Copy to clipboard\]](#)

Alternatively, [single] represents a single object in scope. Attempts to specify multiple objects are then rejected with:

(You're not allowed to use multiple objects in that context.)

A grammar definition can contain multiple tokens, which are then mapped to the wildcards in left-to-right order. It is recommended to restrict all except one of them to a single object, to prevent combinatorial explosion. Thus:

```
(grammar [read  
[object] to [single]]
```

```
(grammar [read [object] to [single]] for [read $ to $])
```

[\[Copy to clipboard\]](#)

or:

```
(grammar [read  
[single] to [object]]
```

```
(grammar [read [single] to [object]] for [read $ to $])
```

[\[Copy to clipboard\]](#)

Sometimes you want the tokens mapped to wildcards in right-to-left order. Use (grammar \$ for \$ reversed) for that:

```
(grammar [lend  
[object] to [single]]
```

```
(grammar [lend [object] to [single]] for [lend $ to $])
```

```
(grammar [lend [single] [object]] for [lend $ to $] reversed)
```

[\[Copy to clipboard\]](#)

The basic set of grammar tokens are:

[object]

One or more objects in scope.

[single]

A single object in scope.

[any]

A single object, not necessarily in scope. The object must be located in a visited room, and cannot be marked as hidden.

[direction]

One or more directions, such as NORTH, DOWN, or OUT.

[number]

A number, such as FIVE or 1981.

[topic]

A topic (see [Ask and tell](#)).

In addition to the above, you can guide the parser towards certain classes of objects, to help resolve ambiguities as well as decide what the word ALL refers to. The following constraints are not enforced strictly—you must use (prevent \$) rules for that.

[animate]

A single, animate object in scope.

[any animate]

A single, animate object, not necessarily in scope. For e.g. CALL MOTHER.

[single held]

A single object currently in the player's inventory.

[held]

One or more objects currently in the player's inventory. Special feature: If the player supplies ALL here, then the single object matched by the *next* grammar token (if any) is excluded from the set of objects.

[worn]

One or more objects currently worn by the player.

[takable]

One or more items in scope, whose relation to their parent is neither #heldby, #wornby, or #partof. Furthermore, they shouldn't be nested below an object that's held or worn by a non-player character, or nested below the current player (regardless of relation).

Finally, a list of words that isn't recognized as a special token will match any of the words in the list. This results in slightly faster and more compact code than a slash expression, but either syntax is fine, really. The following two definitions are functionally equivalent:

```
(grammar  
[throw/toss [held]
```

```
(grammar [throw/toss [held] at/on/to/in/into/onto [single]] for [throw $ at $])
```

```
(grammar [throw/toss [held] [at on to in into onto] [single]] for [throw $ at $])
```

[\[Copy to clipboard\]](#)

Grammar rules must start with a regular word, typically a verb. Tokens and other lists are allowed everywhere else, just not at the very beginning.

Adding actions

```
(grammar [yodel] for  
itself)
```

(grammar [yodel] for itself)

[\[Copy to clipboard\]](#)

Here the story author has introduced a new action, [yodel], triggered when the player types YODEL. But so far, there is no code to handle the action. The fallback implementations of the six [action-handling predicates](#) will cause the action to succeed without printing anything at all. This is a no-no in parser games, so the story author should at least add a perform-rule:

```
(perform [yodel])  
  Your voice
```

(perform [yodel])

Your voice soars over the mountain tops, bringing tears to many eyes.

[\[Copy to clipboard\]](#)

Action descriptions

The standard library might have to print the name of an action, perhaps as part of a disambiguation question, or during debugging if the ACTIONS ON command has been issued. This is done with a query to the predicate (describe action \$).

Usually, there is no need to add an explicit rule for describing every new action; the default implementation of (describe action \$) simply visits each element of the action (which is a list) in turn, printing dictionary words verbatim, and calling (the full \$) for anything else. But sometimes a better wording is desirable:

```
(describe action  
[switch on $Obj])
```

(describe action [switch on \$Obj])

switch (the full \$Obj) on

[\[Copy to clipboard\]](#)

Commands

Commands are system-level actions, such as SAVE or TRANSCRIPT OFF, that do not consume any time in the game world. The predicate (command \$) decides whether an action is a command or not. Thus, to define a new command called HINT, we could write:

```
(grammar [hint] for  
itself)
```

(grammar [hint] for itself)

(command [hint])

(perform [hint])

Try yodeling a lot.

[\[Copy to clipboard\]](#)

There is also a short form that combines the first two rule definitions into one:

```
(understand  
command [hint])
```

(understand command [hint])

(perform [hint])

Try yodeling a lot.

[\[Copy to clipboard\]](#)

Asking for clarification

Some actions are designed to require objects, but it makes grammatical sense to use the verb alone (intransitively), or with fewer objects than the author had in mind. For instance, a grammar definition could be added to recognize PLAY VIOLIN WITH BOW as the action [play #violin with #bow]:

```
(grammar [play  
[single] with/using
```

```
(grammar [play [single] with/using [held]] for [play $ with $])
```

[\[Copy to clipboard\]](#)

But now, players who type PLAY VIOLIN (or just PLAY) will be met by an unhelpful message about not understanding what they wanted to do. In this case, it's a good idea to add partial actions that nudge the player towards the full sentence. These actions can ask the player for clarification, and set up an *implicit action* using one of the predicates (asking for object in \$) and (asking for direction in \$). The parameter is an action, with an empty list [] marking the position of a blank slot. If the player now types in the name of an object (or, optionally, USE followed by the name of an object), this will be understood as the implicit action, with that object in the slot. Thus:

```
(grammar [play  
[single]] for [play $])
```

```
(grammar [play [single]] for [play $])
```

```
(perform [play $Obj])
```

```
With what?
```

```
(asking for object in [play $Obj with []])
```

```
(grammar [play] for itself)
```

```
(perform [play])
```

```
Play what?
```

```
(asking for object in [play []])
```

[\[Copy to clipboard\]](#)

Be aware that (asking for object in \$) and (asking for direction in \$) will automatically invoke (stop) to prevent any subsequent actions: We've asked the player a question, so we have to give them an opportunity to respond.

Of course, it is also possible to override your own action-handling rules for this kind of intermediate actions, in specific situations where no additional object is required:

```
(perform [play  
#piano])
```

```
(perform [play #piano])
```

```
You plink away at the Maple Leaf Rag, only to get stuck in the trio.
```

[\[Copy to clipboard\]](#)

Just remember that rules are tried in program order, so the rule for playing the piano must appear before the generic perform-rule that asks for a second object. One approach is to organize the story file as a large bulk of object-specific rule definitions, followed by a smaller section at the end where new actions are defined.

A note on rule ordering

You are encouraged to define plenty of synonyms using slash-expressions and multiple grammar definitions. Here is an example from the library:

```
(grammar  
[leave/exit [single]]
```

```
(grammar [leave/exit [single]] for [leave $])
```

```
(grammar [get/jump/go [out off] of [single]] for [leave $])
```

```
(grammar [get/jump/go off [single]] for [leave $])
```

[\[Copy to clipboard\]](#)

Such definitions can appear in any order. However, if you define multiple grammar rules that begin with the same words, but produce distinct actions, then you should put the longest rule first:

```
%% Understand  
PLAY VIOLIN WITH
```

```
%% Understand PLAY VIOLIN WITH BOW, or PLAY VIOLIN, or PLAY:
```

```
(grammar [play [single] with/using [held]] for [play $ with $])
```

```
(grammar [play [single]] for [play $])
```

```
(grammar [play] for itself)
```

[\[Copy to clipboard\]](#)

This makes a difference when the player has typed something that the parser doesn't understand. When that happens, the library constructs an error message from the **first** grammar rule that is a partial match. Given the above code, if the player types PLAY SPICCATO WITH BOW, the response will be:

```
(I only understood you as far as wanting to play something with the bow.)
```

But if the first two rules were swapped, the parser would match SPICCATO WITH BOW with the sole parameter of

[play \$], and the following message would be printed instead:

(I only understood you as far as wanting to play something.)

Adjusting the likelihood of actions

When the player's input can be understood in multiple ways, it is up to the game to weigh the different interpretations against each other, and select the one most probably intended by the player. This is achieved by looking at the actions from a semantical point of view, and discarding the unlikely ones, as determined by the predicate (unlikely \$):

```
(unlikely [open
$Object])
```

```
(unlikely [open $Object])
~(openable $Object)
```

```
(unlikely [open $Object])
($Object is open)
```

[\[Copy to clipboard\]](#)

If that's not enough, and several equally likely (or unlikely) interpretations remain, the library will ask the player a disambiguating question.

Thus, if the player is located in a room with a wooden door (open), and holds a wooden box (closed), and attempts to OPEN WOODEN, that will be understood as a request to [open #woodenbox]. The alternative, [open #woodendoor], gets discarded due to the second rule above. But if both the door and the box are open, both actions are deemed equally unlikely, and the game resorts to asking the player what they meant.

Story authors may override (unlikely \$) to influence this procedure. For instance, if a room contains a red lever and a red indicator light, it's up to the author to specify:

```
(unlikely [pull
#redlight])
```

```
(unlikely [pull #redlight])
```

[\[Copy to clipboard\]](#)

which makes PULL RED do the expected thing.

Sometimes it is necessary to override (unlikely \$) with a negated rule, when a more general rule would identify it as unlikely by default. For instance, suppose a location contains a wall-mounted ladder, and the story author wants the game to understand CLIMB LADDER as going up. The functionality itself is implemented by redirecting [climb #ladder] to [go #up]:

```
(instead of [climb
#ladder])
```

```
(instead of [climb #ladder])
(current room #ladderroom)
(try [go #up])
```

[\[Copy to clipboard\]](#)

But [climb #ladder] is still considered unlikely by the parser, because (we assume) the ladder is not an actor supporter, i.e. it is not possible to be located #on the ladder. Now, if the player were to attempt to CLIMB LADDER while also holding the ladder instruction manual, the game would ask which one of the objects to climb. To prevent that slightly surreal question, a negated rule can be defined:

```
~(unlikely [climb
#ladder]) %%
```

```
~(unlikely [climb #ladder])    %% Climbing the ladder is not unlikely after all.
```

[\[Copy to clipboard\]](#)

Very unlikely actions

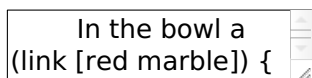
In Dialog, the current room and its neighbours are in scope by default. But rooms are often named by some conspicuous object contained inside them, so that e.g. an engine might be located in the engine room. To avoid a lot of disambiguating questions, any action that explicitly mentions a room is considered (very unlikely \$) by default, unless it's one of the few actions that might involve a room (such as [enter \$] or [go to \$]).

This predicate rarely needs to be touched outside of library code. But if you ever add a new action that involves a room object directly, make sure to adjust the rules for (very unlikely \$) as well as (unlikely \$).

Links and default actions

When a Dialog program is compiled to run on the Å-machine, the text may contain [clickable links](#) that resolve into text input. Selecting a link has the same effect as typing the words of the link target and pressing return. This can simplify text entry on mobile devices.

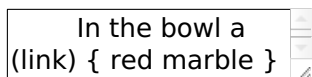
Links are created using the special (link \$) ... syntax:



In the bowl a (link [red marble]) { red marble } glistens in the sunlight.

[\[Copy to clipboard\]](#)

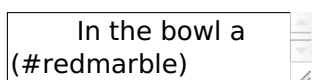
Often, as above, we want both the link target and the clickable text to be the same. In this case a short form is available:



In the bowl a (link) { red marble } glistens in the sunlight.

[\[Copy to clipboard\]](#)

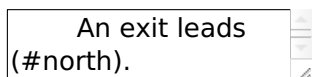
If we want both the link target and the clickable text to be the printed name of an object, we can use the tersely named predicate (\$) from the standard library:



In the bowl a (#redmarble) glistens in the sunlight.

[\[Copy to clipboard\]](#)

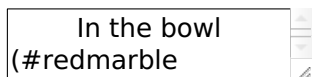
The same predicate can be used for exits:



An exit leads (#north).

[\[Copy to clipboard\]](#)

To use the printed name of an object as a link target, but supply a different text, use the predicate (\$ \$). The first parameter is an object, and the second is a [closure](#):



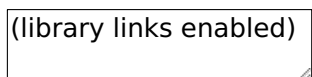
In the bowl (#redmarble {something red}) glistens in the sunlight.

[\[Copy to clipboard\]](#)

Be aware, however, that hyperlinks are an optional feature of the Å-machine, and not every interpreter will support them. While it can be tempting to create a jarring effect by having links resolve into unexpected input text, some players will simply not see it.

By default, library-generated messages never contain hyperlinks. The behaviour of the library should be consistent with the rest of the story, and whether or not to sprinkle room descriptions with clickable links is a decision best left to the author.

To enable clickable links in disambiguation messages, the game over menu, default [appearances](#), and queries to (a \$) and (A \$), add the following rule definition to the source code:

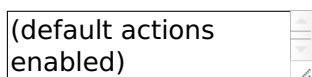


(library links enabled)

[\[Copy to clipboard\]](#)

Since link targets are appended to the current line of input, readers who are playing on a touchscreen device can type a verb using the on-screen keyboard, and then complete the sentence by tapping on a recently printed object name. Compass directions can of course be used directly as commands.

However, when we turn our nouns into hyperlinks, players will (understandably) attempt to click on them without first typing a verb. To handle this situation, the standard library provides an optional feature called *default actions*. It is enabled like this:



(default actions enabled)

[\[Copy to clipboard\]](#)

When this feature is enabled, the parser will understand noun-only input as a request for the default action, which is examine. The default action can be changed, and may depend on the object, as in the following example:

```
(default action
(animate $Obj) [talk
```

```
(default action (animate $Obj) [talk to $Obj])
```

[\[Copy to clipboard\]](#)

Example: Defining a new action

Story-specific actions are typically defined towards the end of the source code file. This allows object-specific rules, defined earlier in the file, to override them.

The following example relies on a special property of the [held] token: When the first token of a grammar rule is [held] and the player uses ALL in that position, then whatever matches the second token is excluded from the set of objects. Thus, if the player types PEEL ALL WITH PEELER, the ALL will expand to every held object except the peeler. Note that this may still not be what the player intended (because in addition to fruit, they might be holding a brass key and a lamp), but at this stage we are primarily interested in grammar, not semantics.

Remember, put the longest grammar definition first:

```
(grammar [peel
[held] with/using
```

```
(grammar [peel [held] with/using [single]] for [peel $ with $])
```

```
(grammar [peel [held]] for [peel $])
```

```
(perform [peel $Obj])
```

```
With what?
```

```
(asking for object in [peel $Obj with []])
```

[\[Copy to clipboard\]](#)

Either variant is deemed unlikely for non-edible objects:

```
(unlikely [peel $Obj |
$]) %% Match both
```

```
(unlikely [peel $Obj | $])    %% Match both variants.
```

```
~(edible $Obj)
```

[\[Copy to clipboard\]](#)

The likelihood of an action helps resolve ambiguities, but it won't prevent the action from being attempted: If the player unambiguously tries to peel the kitchen floor, that request is going to go through, unlikely or not. Thus we also need:

```
(prevent [peel $Obj |
$])
```

```
(prevent [peel $Obj | $])
```

```
~(edible $Obj)
```

```
That's not something you can peel.
```

[\[Copy to clipboard\]](#)

Likewise, because we specified [held], the library will try to satisfy the parser rules using objects that are held by the player, so that e.g. PEEL FRUIT will prioritize held fruit over non-held fruit. But an unambiguous PEEL BANANA will be understood even when the banana isn't held.

Thus, we need a rule to prevent the peeling of a non-held object. The standard library provides a number of handy [when-predicates](#) that check for common conditions, and print appropriate responses when the conditions are met.

```
(prevent [peel $Obj |
$])
```

```
(prevent [peel $Obj | $])
```

```
(when $Obj isn't directly held)
```

```
(prevent [peel $ with $Obj])
```

```
(when $Obj isn't directly held)
```

[\[Copy to clipboard\]](#)

But, out of the kindness of our hearts, we might decide to pick up the mentioned objects automatically before attempting the peel action:

```
(before [peel $Obj | $])
```

```
(before [peel $Obj | $])
```

```
%% This will invoke (first try [take $Obj]) if necessary:
```

```
(ensure $Obj is held)
```

```
(before [peel $ with $Obj])
```

```
(ensure $Obj is held)
```

[\[Copy to clipboard\]](#)

Finally, there needs to be a default response for the [peel \$ with \$] action (we already have one for the [peel \$] action):

```
(perform [peel $Obj with $Tool])
```

```
(perform [peel $Obj with $Tool])
```

```
After an extended period of fumbling, you conclude that you don't know
```

```
how to peel (the $Obj) with (the $Tool).
```

```
(tick) (stop)
```

[\[Copy to clipboard\]](#)

Of course, the story should also contain a couple of objects that would make the peel action succeed. The following object-specific rules must be defined before the generic rules described above, otherwise they will never match:

```
(edible #apple)
(edible #peeled-
```

```
(edible #apple)
```

```
(edible #peeled-apple)
```

```
(perform [peel #apple with #peeler])
```

```
You peel the apple without cutting yourself even once.
```

```
(#apple is $Rel $Loc)
```

```
(now) (#apple is nowhere)
```

```
(now) (#peeled-apple is $Rel $Loc)
```

[\[Copy to clipboard\]](#)

Finally, we could smoothen gameplay by implicitly assuming that if the player is holding the peeler, that's probably their tool of choice for peeling:

```
(instead of [peel $Obj])
```

```
(instead of [peel $Obj])
```

```
(current player $Player)
```

```
(#peeler is #heldby $Player)
```

```
\( with the peeler \) (line) %% Tell the player what's going on.
```

```
(try [peel $Obj with #peeler])
```

[\[Copy to clipboard\]](#)

How the parser works

This section provides an outline of how the parser works, and describes advanced techniques for understanding arbitrary turns of phrase that cannot be represented by ordinary grammar definitions. Most story authors do not need to dig this deeply, and can safely skip ahead to the next chapter.

The parser makes queries to (understand \$ as \$), a predicate that is normally defined by the library, but which can also be extended by the story author. One of the library-defined rules for this predicate is responsible for querying the table of grammar definitions. But there are also rules for special cases like *GO TO ROOM*, or *ACTOR*, *COMMAND*. We will discuss how to add such special cases.

During parsing, the standard library works with an intermediate representation of actions, called *complex actions*. Like regular actions, complex actions are lists of dictionary words and objects, but the following subexpressions are also allowed in them:

```
[+ #object1 #object2 ...]
```

The player referred to multiple objects here.

```
[a #object]
```

The player referred to a non-specific object that should be printed with “a” rather than “the”. [a ...] subexpressions may

be nested inside [+ ...] subexpressions.

[]

The input contained one or more words that couldn't be parsed. When the complex action is printed, this part will appear as “something”.

[1]

The input contained one or more words that couldn't be parsed, and an animate object was expected. When the complex action is printed, this part will appear as “someone”.

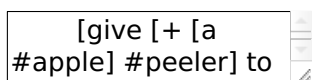
[,]

The input contained multiple objects in an illegal place.

[all]

The input contained an ALL-expression in an illegal place.

Thus, a complex action might be:



[give [+ [a #apple] #peeler] to [1]]

[\[Copy to clipboard\]](#)

and its printed representation makes an appearance in the following message:

(I only understood you as far as wanting to give an apple and the peeler to someone.)

The parsing process

The following chart illustrates the overall parsing process, starting with the player input as a list of words, and ending with a set of actions. The list of words is first split into a sequence of sublists by the word THEN or the full stop. If such a sublist cannot be parsed, it is in turn split by the first AND or comma. This allows the player to type multiple commands on one line, such as: N, U THEN DROP ALL, D.

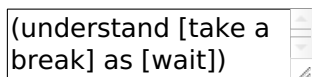


Parsing actions

When the library needs to parse an action, it makes a [multi-query](#) to (understand \$ as \$). A multi-query is made in order to collect every possible interpretation of the player's input, which could be ambiguous.

The first parameter is the input: a list of dictionary words. The second parameter is the output: a complex action.

Story authors can easily add rule definitions to this predicate, in order to add support for new verbs or set phrases (although in this case, a normal grammar definition would also work):

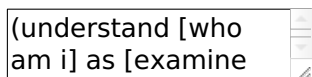


(understand [take a break] as [wait])

[\[Copy to clipboard\]](#)

Note that the multi-query to (understand \$ as \$) may backtrack over several possible interpretations, e.g. [wait] and [take #break] if an object called “break” is [in scope](#).

Understand-rules may of course have rule bodies:



(understand [who am i] as [examine \$Player])

(current player \$Player)

[\[Copy to clipboard\]](#)

Parsing object names

Many actions involve objects. The rule for understanding such an action will typically query a library-provided predicate for parsing a list of words as an object. There are several to choose from, but the most basic one is (understand \$Words as non-all object \$Object), which can be used like this:

```
(understand
[transmogrify |
```

```
(understand [transmogrify | $Words] as [transmogrify $Object])
```

```
*(understand $Words as non-all object $Object)
```

[\[Copy to clipboard\]](#)

Note that a [multi-query](#) must be used, because the words may be ambiguous. Suppose a red box and a blue box are in scope. TRANSMOGRIFY BOX will cause the above rule header to match, binding \$Words to the single-element list [box]. Since there are two boxes, (understand [box] as non-all object \$Object) will return twice, binding \$Object to #redbox the first time, and to #bluebox the second time. Consequently, the rule for understanding the action will return twice, binding its output parameter to [transmogrify #redbox] the first time, and [transmogrify #bluebox] the second time.

Some actions involve two (or even more) objects, usually separated by a keyword such as a preposition. Dialog provides a handy built-in predicate for searching a list for a set of keywords, and splitting the list at the position where a match was found. Consider the following example, where a new “read something to somebody” action is created:

```
#book
(proper *)
```

```
#book
(proper *)
(name *) To Kill A Mockingbird
```

```
#bird
/animate *)
(name *) mockingbird
```

```
(understand [read | $Words] as [read $Object to $Person])
  *(split $Words by [to] into $Left and $Right)
  *(understand $Left as non-all object $Object)
  *(understand $Right as non-all object $Person)
```

[\[Copy to clipboard\]](#)

Again, the consistent use of multi-queries helps with disambiguation. If the player attempts to READ TO KILL A MOCKINGBIRD TO MOCKINGBIRD, \$Words will be bound to [to kill a mockingbird to mockingbird]. The split-predicate first separates it into [] and [kill a mockingbird to mockingbird]. The empty list is not a valid object name, so the subsequent query to (understand \$Left ...) fails, and the split-predicate proceeds with the next occurrence of the keyword: Now it separates \$Words into [to kill a mockingbird] and [mockingbird], which makes the rest of the rule body succeed.

Still, the name of the second object (MOCKINGBIRD) is ambiguous, so the final invocation of *(understand \$Right ...) returns twice. The parser will end up asking the player whether they wanted to read the book to the bird, or the book to the book. One way to address this problem is to indicate that the second noun is supposed to be animate:

```
(understand [read |
$Words] as [read
```

```
(understand [read | $Words] as [read $Object to $Person])
*(split $Words by [to] into $Left and $Right)
*(understand $Left as non-all object $Object)
*(understand $Right as single object $Person preferably animate)
```

[\[Copy to clipboard\]](#)

The library provides a set of object-parsing predicates that favour objects with certain common traits, or limit the selection in some other way. The object-parsing predicates are:

- A predicate that accepts multiple objects, but not the word ALL:

```
(understand $Words as non-all object $Output)
```

- Predicates that accept multiple objects, including ALL:

```
(understand $Words as object $Output preferably held)
(understand $Words as object $Output preferably held excluding $ExcludeObj)
(understand $Words as object $Output preferably worn)
(understand $Words as object $Output preferably takable)
(understand $Words as object $Output preferably child of $Parent)
```

- Predicates that only accept a single object (possibly implied by ALL):

```
(understand $Words as single object $Output)
(understand $Words as single object $Output preferably held)
(understand $Words as single object $Output preferably animate)
```

(understand \$Words as single object \$Output preferably supporter)
(understand \$Words as single object \$Output preferably container)

• Predicates that accept any (single) object, even if it's currently out of scope, as long as it's located in a visited room and not [hidden](#):

(understand \$Words as any object \$Output)
(understand \$Words as any object \$Output preferably animate)

Some of the variants above are primarily there to provide context for the word ALL. For instance, TAKE ALL should only select *takable* objects (items not already held), while DROP ALL should operate on held objects. But the preferably specifier is also used to carry out some initial disambiguation, so that e.g. FEED BIRD might be understood as an intention to feed the bird (animate), but not the bird cage.

Three further variants allow the story author to specify an arbitrary condition using a [closure](#):

(understand \$Words as object \$Output preferably \$Closure)
(understand \$Words as single object \$Output preferably \$Closure)
(understand \$Words as any object \$Output preferably \$Closure)

The closure takes a candidate object as parameter. Here is an example of how to parse an object name while favouring objects that can be picked up, but not eaten:

```
(understand
$Words as object
```

```
{
(item $_)
~(edible $_)
})
```

[\[Copy to clipboard\]](#)

The output of all of these object-parsing predicates is either an object or a list that represents a complex object (e.g. [+ \$Obj1 \$Obj2 \$Obj3], or [a \$Obj1], or []) For the as single object rules, the output is guaranteed to be either an object or one of the values that indicate a parse error.

Directions, numbers, and topics

Some actions involve a named direction, such as SOUTHWEST, OUT, or UP. To parse a direction, use the predicate:

```
(understand $Words
as direction $Dir)
```

```
(understand $Words as direction $Dir)
```

[\[Copy to clipboard\]](#)

As when parsing objects, \$Dir is potentially a complex expression: When the player types PUSH CART SOUTHWEST, OUT AND UP, the words [southwest , out and up] will be understood as the complex direction [+ #southwest #out #up].

To parse a number, typed using decimal digits or spelled out as a word, use:

```
(understand $Words
as number $N)
```

```
(understand $Words as number $N)
```

[\[Copy to clipboard\]](#)

The output parameter \$N is a [number](#), and thus limited to the range 0-16383.

Topics

Some actions, e.g. [ask \$ about \$] and [tell \$ about \$], involve *topics of conversation*. As we saw in the chapter about [non-player characters](#), topics can be regular objects, topic objects, or dictionary words. To parse a topic, use the predicate:

```
(understand $Words
as topic $Topic)
```

```
(understand $Words as topic $Topic)
```

[\[Copy to clipboard\]](#)

It is possible to add rules to that predicate in order to add new topics to a game:

```
(understand [my
childhood] as topic
```

```
(understand [my childhood] as topic @childhood)
(understand [growing up on planet zyx] as topic @childhood)
```

[\[Copy to clipboard\]](#)

However, that would result in a parser that is very picky about the exact wording of ask/tell commands, so it is not generally recommended. A better (but potentially slower) approach is to look for keywords or key phrases like this:

```
(understand $Words
as topic @childhood)
```

```
(understand $Words as topic @childhood)
($Words contains sublist [growing up])
(or) ($Words contains one of [childhood planet zyx])
```

[\[Copy to clipboard\]](#)

The default implementation of (understand \$ as topic \$) tries to strike a balance between performance and flexibility by using a system of simple keywords. Keywords are defined with (topic keyword \$ implies \$):

```
(topic keyword
@childhood implies
```

```
(topic keyword @childhood implies @childhood)
(topic keyword @zyx implies @childhood)
```

[\[Copy to clipboard\]](#)

A short form is available when the keyword equals the topic value:

```
(topic keyword
@childhood)
```

```
(topic keyword @childhood)
```

[\[Copy to clipboard\]](#)

All of these variants can of course be combined. For instance, the keyword approach could be employed as a fall-back that often works well enough, and specific understand-rules could use the [\(just\) keyword](#) to overrule the keyword system when it would otherwise misfire:

```
(understand [your
childhood] as topic
```

```
(understand [your childhood] as topic #doctor)
(just)
(topic keyword @childhood)
(topic keyword @yourself implies #doctor)
```

[\[Copy to clipboard\]](#)

The (just) keyword can also be used to selectively disable the behaviour where objects in scope are understood as topics. For instance, in an aquarium, the word FISH might be accepted as a synonym for every individual fish in the room. But suppose we want the last word of ASK CLERK ABOUT FISH to be understood unambiguously as being in reference to the general subject of fish. That is, suppose we don't want the game to ask the player if they meant to ask about fish in general, the zebrafish, or the neon tetra. To obtain the desired behaviour, we just have to:

```
(understand [fish] as
topic @fish)
```

```
(understand [fish] as topic @fish)
(current room #aquarium)
(just)
```

[\[Copy to clipboard\]](#)

Printed representations of topics

Topics are supposed to have printed representations, accessible via the (describe topic \$) predicate. The default implementation of this predicate delegates to (the full \$) when the topic is an object; otherwise it just prints the word "something". Story authors are strongly recommended to override this predicate for non-object topics:

```
(describe topic
@childhood)
```

```
(describe topic @childhood)
```

your childhood
[\[Copy to clipboard\]](#)

When you add new actions that involve topics, remember to add corresponding (describe action \$) rules as well. That's because the default implementation of (describe action \$) is rather crude: It looks at each element of the action list, printing full descriptions of any objects, and printing dictionary words as they appear. But if the dictionary word happens to be a topic, the proper thing to do is to query (describe topic \$) to print it. Thus:

```
(understand
[complain about |
```

```
(understand [complain about | $Words] as [complain about $Topic])
*(understand $Words as topic $Topic)
(describe action [complain about $Topic])
  complain about (describe topic $Topic)
```

[\[Copy to clipboard\]](#)

Rewriting

Before the player's input is handed to the action-parsing predicate (understand \$ as \$), it undergoes *rewriting*: The predicate (rewrite \$Input into \$Output) is queried once (i.e. neither iteratively nor with a multi-query), and may transform the list of words in any way it sees fit before parsing.

```
(rewrite [please |
$Words] into
```

```
(rewrite [please | $Words] into $Words)
```

[\[Copy to clipboard\]](#)

Rewriting is not used by the library itself, but it offers a powerful way for story authors to override the default behaviour of the parser.

Custom grammar tokens

This section explains how to add new grammar tokens, like [worn] or [single held]. It's an advanced topic, and most story authors can safely skip ahead to the next chapter.

Let's create a [spell] token, to be used like this:

```
(grammar [cast
[spell]] for [cast $])
```

```
(grammar [cast [spell]] for [cast $])
(grammar [look up [spell]] for [consult #spellbook about $])
(understand [xyzy] as spell #xyzy)
(understand [plugh/abracadabra] as spell #plugh)
```

[\[Copy to clipboard\]](#)

Recall (from [How the parser works](#)) that the library defines a generic (understand \$ as \$) rule that queries a table of grammar definitions. This table is called (grammar entry \$ \$ \$), and it is constructed at compile-time from instantiations of the (grammar \$ for \$) access predicate.

A definition like the following:

```
(grammar [give
[held] to [animate]]
```

```
(grammar [give [held] to [animate]] for [give $ to $])
```

[\[Copy to clipboard\]](#)

is transformed into the following table entry:

```
(grammar entry
@give [22 to 11]
```

```
(grammar entry @give [22 to 11] [give $ to $])
```

[\[Copy to clipboard\]](#)

The first parameter of the grammar entry is the first word of the grammar rule. This helps the compiler create efficient lookup code. The second parameter corresponds to the rest of the grammar rule, with numeric values instead of symbolic tokens (e.g. 22 instead of [held]). The third parameter is the action template, exactly as supplied in the grammar definition.

There are two reasons for translating the grammar tokens into numbers: It makes the code more compact (and therefore faster on old systems), and it prevents the grammar tokens from cluttering the game dictionary.

Grammar tokens are converted to numbers using a set of access predicate definitions in the standard library. For instance, here is the rule for converting [direction] to 50:

```
@(grammar
transformer
```

```
@(grammar transformer [[direction] | $Tail] $SoFar $Verb $Action $Rev)
(grammar transformer $Tail [50 | $SoFar] $Verb $Action $Rev)
```

[\[Copy to clipboard\]](#)

It operates by removing one element from the incoming list (first parameter), tacking on a new element to the outgoing list (second parameter), and leaving three more parameters intact.

Numbers in the range 90-99 are reserved for story authors. To create our new [spell] token, we add a similar rule to map the token to an unused number in this range:

```
@(grammar
transformer [[spell] |
```

```
@(grammar transformer [[spell] | $Tail] $SoFar $Verb $Action $Rev)
(grammar transformer $Tail [90 | $SoFar] $Verb $Action $Rev)
```

[\[Copy to clipboard\]](#)

Next, we supply a rule for what to do when the number 90 is encountered in the grammar table:

```
(match grammar
token 90 against
```

```
(match grammar token 90 against $Words $ into $Obj)
*(understand $Words as spell $Obj)
```

[\[Copy to clipboard\]](#)

In the above example, we delegate to a separate custom predicate for handling spells. Under other circumstances, we might have queried an existing predicate such as `*(understand $Words as object $Obj preferably { ... })`.

The third parameter of `(match grammar token $ against $ $ into $)` is rarely used. It contains matches from later grammar tokens (they are parsed from right to left). These can be objects, plus-prefixed lists like `[+ #foo #bar #baz]`, or anything else; it depends on the type of the next token. This feature allows you to craft rules for e.g. PUT ALL IN BAG, where the bag should not be included in “all”.

Onwards to “[Chapter 11: Miscellaneous features](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Software

([The compiler](#) • [The interactive debugger](#) • [Building from source code](#))

An archive containing the latest version of Dialog can be found here:

<https://linusakesson.net/dialog/>

The archive contains the full source code for the Dialog compiler and interactive debugger, as well as pre-built executable files for Linux (i386 and x86_64) and Windows. The compiler is called `dialogc`, and the debugger is called `dgdebug`.

Both the compiler and the interactive debugger have to be started from a commandline shell. Depending on what operating system you use, you may have to copy the executable files to a system-specific location, or simply navigate to the directory that contains them.

The Windows version of the debugger makes use of a third-party library, *Windows Glk* by David Kinder. Make sure to put the two included DLL files in the same directory as the executable file (`dgdebug.exe`).

To check that everything is in place, try the following two commands:

```
dialogc --version
dgdebug --version
```

Each of them should print the current version number of the Dialog software. The first two characters (a digit and a letter) identify the language version, which is currently **0m**. Following that is a slash, and then the software revision number.

The archive also contains the latest version of the Dialog standard library, `stdlib.dg`, and the standard debugging library extension, `stddebug.dg`. You should make local copies of these files for each of your projects, because if you ever wish to rebuild your story in the future, you'll want to have access to the exact version of the library you used. Also, you may wish to make story-specific modifications to the library.

A copy of this manual is also included in the archive, and a file called `license.txt` that details how you may use the software. In short, you have the right to use, modify, and redistribute the compiler and standard library (with or without source code), in whole or in part, as long as attribution is given.

The compiler

Dialog is a compiled language. Source code is stored in text files with the filename extension `.dg`. When international characters are used, they must be encoded in UTF-8. The compiler delivers its output in `.zblorb`, `.z8`, `.z5`, or `.aastory` format. When the `zblorb` format is chosen, the compiler can optionally include cover art with the story, but this feature is not regarded as a part of the Dialog language itself.



To compile a Dialog story into `zblorb` format, invoke the compiler with the source code filenames as arguments, and optionally use `-o` to specify an output filename. The default output filename is constructed from the name of the first source code file, combined with a filename extension that is appropriate for the current output format.

The order of source code filenames is important. The story must appear before the standard library, and any extensions (such as the standard debugging library) should appear in between. For instance:

```
dialogc story.dg stddebug.dg stdlib.dg
```

This will produce a file called `story.zblorb`, based on the name of the first source code file.

Producing raw Z-code images

The example games in this manual do not declare IFIDs, which is mandatory when the `zblorb` output format is used. To compile the examples, you must ask the compiler to emit raw `z8` (or `z5`) code by adding `-t z8` (or `-t z5`) to the command line:

```
dialogc -t z8 example.dg stdlib.dg
```

This will produce a file called `example.z8`.

The examples in Part I do not make use of the standard library at all, so they are compiled like this:

```
dialogc -t z8 example.dg
```

When you use the `z5` output format, the resulting file is limited to 256 KiB, compared to 512 KiB for `z8`. However, the `z5` format is more compact, and switching to this format can reduce the size of the story file by about 5%.

Producing stories for the Å-machine

The Å-machine (pronounced “awe machine”) is a compact, binary story format designed to improve the performance of Dialog stories on vintage hardware beyond what is possible using the Z-machine. It also allows Dialog stories to be published on the web with support for hyperlinks and CSS styles.

To compile for the Å-machine, add `-t aa` to the command line:

```
dialogc -t aa example.dg stdlib.dg
```

This will produce a file called `example.aastory`. The `.aastory` file may in turn be bundled with a javascript interpreter, or with an interpreter for the Commodore 64, using the tool `aambundle` from [the Å-machine release archive](#).

The following command:

```
aambundle -o my_game example.aastory
```

creates a directory called `my_game`, and populates it with all the necessary html, css, and js files. Simply point a web browser to the `play.html` file in this directory to run the game.

To create a Commodore 64 disk image, use the following command instead:

```
aambundle -t c64 -o my_game example.aastory
```

Other compiler flags

The `-v` flag makes the compiler more verbose. Give it once, and the compiler will print the wordcount for your story, as well as some technical stats. Give it twice (`-vv`) and the compiler will dump additional information that can be useful for debugging.

To see the full list of options supported by `dialogc`, type:

```
dialogc --help
```

The interactive debugger

This section uses terminology that may be confusing if you've only just started to learn about Dialog. Please skim through the section anyway, because having a back-of-the-mind awareness of these techniques can be very helpful later.

We've seen that the role of the [compiler](#) is to convert your Dialog program into a compact, widely-supported runtime format. In contrast, the *interactive debugger* runs your game directly from source code, and allows you to inspect and manipulate the program while it is running.



To debug your game, launch the debugger from the commandline with your source code filenames as arguments:

```
dgdebug story.dg stdlib.dg
```

The order of the filenames is significant, just like when compiling.

The debugger also accepts a couple of commandline options. For a complete list, type:

```
dgdebug --help
```

Running the game

At startup, the debugger analyzes your program, reporting any compilation errors. If everything looks fine, it launches the program. You can then play your game normally.

When the game is asking for input, you may instead choose to type an arbitrary [query](#), [multi-query](#), or [now-statement](#). This can be used to inspect or control the running program.

The debugger remains operational after your program terminates. To start over, type `(restart)`. To quit the debugger, either press Control-D (Linux), close the window (Windows), or type the special command `@quit`.

Modifying a running game

The interactive debugger watches your source code files for changes. New code is merged into the running program automatically, so you can test new functionality without restarting the game.

If there are compilation errors in the new version of the source code, the debugger will report them and put the execution on hold until you've fixed them.

Any [dynamic predicates](#) that have changed during gameplay retain their value, so that e.g. the player character remains

in the current room, with any picked-up objects still in their inventory. But unchanged dynamic predicates will reflect the initial value declarations as they're given in the new version of the source code.

The debugger also tries to match [select statements](#) in the old and new versions of the code, in order to make the transition as seamless as possible. But you may occasionally find that a select statement has been reset.

Be aware that by modifying the source code, you can introduce new [objects](#) and [dictionary words](#), but you can't remove them. This affects the operation of the built-in (object \$) predicate, as well as (get input \$), with particular consequences for games that use the [removable word endings](#) feature. To remove extraneous objects and dictionary words, use (restart) (or @replay; see below).

Debugging commands

In addition to arbitrary queries and now-statements, the debugger allows you to type *debugging commands* at the game prompt. These are recognized by a leading @ character.

The debugger maintains a list of all *accumulated input* that you've typed into the running program. Only proper input counts, not queries or debugging commands. The list is cleared on restart, and trimmed on undo, so in that sense it represents a straight path from the beginning of the game to the current game state. The following debugging commands make use of it:

@replay

Resets the game, and re-enters all accumulated input. This will normally reproduce the current game position from a clean start, but it may work differently if you've made changes to the source code, or if the game includes randomized behaviour. It can be useful for catching unintended non-local effects of a code change.

@again

Performs an undo operation, and then re-enters the most recent line of input. This command offers more fine-grained control than @replay, and generally works better for games with randomized behaviour. It lets you focus on the local effects of a code change.

@g

A synonym for @again.

@save

Saves the accumulated input as a simple text file. The debugger will ask you for a filename.

@restore

Resets the game, and reads input from a text file, thus recreating the saved game position. The debugger will ask you for a filename.

You can get a full list of debugging commands by typing @help at the prompt. These commands can be abbreviated as long as the abbreviation is unique; @h works for @help, for instance.

Suspending execution

The terminal version of the debugger (i.e. *not* the Windows Glk version) allows you to suspend a running computation by pressing Control-C at any time. This will immediately take you to a prompt where you can type queries and debugging commands.

To resume execution, type a blank line at this prompt.

Suspending is useful when you've enabled [tracing](#) and found that you got a bit more than you bargained for. At the [More] prompt, simply hit Control-C, type (trace off), and press return a second time.

It can also be used to escape from an accidental infinite loop, using (stop). This is rarely needed, however, because the debugger does not perform tail-call optimization, and there's a limit on the number of recursive calls.

Some useful debugging techniques

Use queries to inspect the state of the running program, e.g. type *(\$ has parent #box) into the game to get a list of every object that's currently a direct child of the #box object.

Insert (log) ... statements to print variables and other information while debugging.

The standard library provides (actions on) and (actions off), for controlling *action tracing*. When enabled, this feature makes the library print the names of [actions](#) as it tries them.

The library also provides (scope), for listing every object that's currently [in scope](#).

The following predicates are highly useful for manipulating the state of the running game:

(enter *Room*)

to teleport the player character to a given room,

(now) (*Object* is #heldby *Player*)

to purloin an object, and

(try *Action*)

to temporarily sidestep a parser problem, or to trace an action without also tracing the parser.

Query tracing can be enabled interactively with (trace on), and disabled with (trace off). Tracing a complete player command will produce a lot of output as the command is parsed. Sometimes it makes more sense to temporarily add (trace on) and (trace off) to the source code, surrounding the particular bit that you're interested in. Remember, the debugger lets you do this while the program is running.

Use the built-in predicate (breakpoint) to suspend the program from within the source code, in order to inspect the game state at arbitrary points during execution.

The command AGAIN (or G) lets the player repeat the last line of input. During development, you will often want to change something in the most recent response, and then retry the action to see what the new version looks like. AGAIN usually works for this, but there's a snag: If you've added new dictionary words, e.g. by changing a (dict \$) rule, then a regular AGAIN will fail to pick up the new words. Use the up-arrow to fetch the command from the input history instead. Another option is to use the debugging command @again (or @g), which will handle this corner-case transparently. That command also performs an implicit undo, which is particularly useful for debugging games with timed puzzles.

Finally, a word of warning: The interactive debugger does not try to emulate the Z-machine backend, nor the Å-machine backend. The Z-machine truncates dictionary words, but the debugger doesn't. The debugger allocates larger heap areas, but doesn't optimize tail calls. Always test the compiled version of your game thoroughly, preferably both in a Z-code interpreter and an Å-code interpreter, before releasing it.

Building from source code

To build the Dialog compiler and debugger from source, you need a working C compiler (such as gcc or clang) and some version of the make command (such as gmake). Unpack the archive, enter the src directory, and type “make”. If all went well, you should now have two executable files called dialogc and dgdebug in the current directory. These are the compiler and debugger, respectively.

If you are on a Unix-like system, and you wish to install the tools in a system-wide location, type “sudo make install”.

The Windows binaries can be cross-compiled on a Linux system using the Mingw32 toolchain.

Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 9: Non-player characters

([Movement](#) • [Other NPC actions](#) • [Taking orders](#) • [Ask and tell](#) • [Choice-based conversation](#))

Animated non-player characters (NPCs) can really make the game world come alive. In their simplest form, NPCs are just ordinary objects whose antics are narrated spontaneously from time to time:

```
#field
(name *) field
```

```
#field
(name *) field
(room *)
(look *)
    A scarecrow is here.
(on every tick in *)
    (par)
    (select)
    The scarecrow looks wistfully towards the horizon.
    (or)
    A growling noise emerges from the scarecrow's stomach.
    (or)
    The scarecrow suddenly sneezes.
    (or)
    (or)
    (or)
    (at random)
```

```
#scarecrow
(name *) scarecrow
(* is #in #field)
```

[\[Copy to clipboard\]](#)

A slightly more sophisticated NPC might interact with other objects in the room, such as the player's possessions:

```
#alley
(name *) back alley
```

```
#alley
(name *) back alley
(room *)
(look *)
    A suspicious-looking figure is lurking in the shadows.
(on every tick in *)
    (current player $Player)
    (collect $Obj)
    *($Obj is #heldby $Player)
    (into $List)
    %% The following rule fails (which is fine) if the list is empty.
    (randomly select $Target from $List)
    (par)
    The suspicious-looking figure eyes (the $Target) with interest.
    (random from 1 to 5 into 1)    %% Fails 4 times out of 5.
    (now) ($Target is #heldby #thief)
```

```
#thief
(name *) suspicious-looking figure
(dict *)  suspicious looking thief
(animate *)
(* is #in #alley)
(descr *)
    (if) ($ is #heldby *) (then)
    The figure seems to be carrying something.
    (else)
```

Very suspicious-looking.

(endif)

[\[Copy to clipboard\]](#)

The library predicate (randomly select \$Element from \$List) picks a random element from a given list, or fails if the list is empty.

Movement

The library provides a predicate called (let \$NPC go \$Direction), to allow NPCs to roam the game world.

Here's a character who moves around on a fixed schedule:

```
#workshop
(name *) workshop
```

```
#workshop
(name *) workshop
(room *)
(look *) You're in a noisy workshop.
(from * go #east to #backroom)
```

```
#backroom
(name *) back room
(room *)
(look *) Shelves line the walls in this dimly lit room.
(from * go #west to #workshop)
```

```
#mechanic
(name *) mechanic
(female *)
(appearance * $ $)
    A busy-looking mechanic is here, looking busy.
(* is #in #workshop)
```

```
(on every tick)
    (par)
        (select)
            (or)
            (or)
            (let * go #east)
            (or)
            (or)
            (let * go #west)
        (cycling)
```

[\[Copy to clipboard\]](#)

The caller of (let \$ go \$)—typically an (on every tick) rule, as above—is responsible for supplying a direction that corresponds to a valid exit.

It is instructive to look at the library code for (let \$ go \$):

```
(let $NPC go $Dir)
    ($NPC is in
```

```
(let $NPC go $Dir)
($NPC is in room $OldRoom)
(from $OldRoom go $Dir to room $NewRoom)
(if (player can see $NPC) (then)
(narrate $NPC leaving $OldRoom $Dir to $NewRoom)
(endif)
(now) ($NPC is #in $NewRoom)
(if (player can see $NPC) (then)
(narrate $NPC entering $NewRoom from $OldRoom)
(endif)
```

[\[Copy to clipboard\]](#)

The predicate (\$ is in room \$) traverses the object tree, via (\$ has parent \$) links, from a given object all the way to its enclosing room.

The predicate (from \$ go \$ to room \$) consults the story-supplied tables of obvious exits and doors, (from \$ go \$ to \$) and (from \$ through \$ to \$), to find out what room lies in a particular direction. It is also possible to query this predicate “backwards” with a given originating room and neighbouring room, to obtain the direction.

The library provides default implementations of (narrate \$ leaving \$ \$ to \$) and (narrate \$ entering \$ from \$), deliberately bland, which the story author may wish to override for flavour.

Note in particular the queries to (player can see \$). That predicate also comes in handy when writing story code that's dealing with moving NPCs, because we generally only want to describe their actions when they're in the same room as the player:

```
#mechanic
```

```
#mechanic
(on every tick)
(par)
(select)
(if) (player can see *) (then)
The mechanic
(select)
operates the hydraulic drill.
(or)
removes a couple of tiny screws.
(or)
makes a precise measurement.
(at random)
(endif)
(or)
(let * go #east)
(or)
(if) (player can see *) (then)
The mechanic
(select)
rifles through a box of parts.
(or)
looks something up in a binder.
(or)
wipes her hands on a dirty cloth.
(at random)
(endif)
(or)
(let * go #west)
(cycling)
```

[\[Copy to clipboard\]](#)

Random movement

Here is an example of an NPC moving randomly through the game world:

```
#rover
(name *) rover
```

```
#rover
(name *) rover
(singleton *)
(on every tick)
  (par)
    (* is in room $Room)
    (collect $Dir)
    *(from $Room go $Dir to room $)
    (into $Exits)
    (randomly select $Dir from $Exits)
    (let * go $Dir)
```

[\[Copy to clipboard\]](#)

But in the above example, the rover might easily go back and forth between neighbouring rooms several times in a row. To avoid that, we can refine the code with a global variable that keeps track of the last direction of movement—or rather its opposite:

```
#rover
(name *) rover
```

```
#rover
(name *) rover
(singleton *)
(global variable (rover avoids direction $))
(on every tick)
  (par)
  (* is in room $Room)
  (collect $Dir)
  *(from $Room go $Dir to room $)
  ~(rover avoids direction $Dir)
  (into $Exits)
  %% Clear the variable so we don't get stuck in dead-end rooms:
  (now) ~(rover avoids direction $)
  (randomly select $Dir from $Exits)  %% This can fail.
  (let * go $Dir)
  (opposite of $Dir is $OppDir)
  (now) (rover avoids direction $OppDir)
```

[\[Copy to clipboard\]](#)

To populate the game world with several identical rovers, we'd have to animate each one of them. In that case, it would make sense to use a per-object variable instead of a global variable:

```
#redrover
(name *) red rover
```

```
#redrover
(name *) red rover

#greenrover
(name *) green rover

#bluerover
(name *) blue rover

(on every tick)
  *($Rover is one of [#redrover #greenrover #bluerover])
  (par)
  ($Rover is in room $Room)
  (collect $Dir)
  *(from $Room go $Dir to room $)
  ~($Rover avoids direction $Dir)
  (into $Exits)
  %% Clear the variable so we don't get stuck in dead-end rooms:
  (now) ~($Rover avoids direction $)
  (randomly select $Dir from $Exits)
  (let $Rover go $Dir)
  (opposite of $Dir is $OppDir)
  (now) ($Rover avoids direction $OppDir)
```

[\[Copy to clipboard\]](#)

Moving towards an object

Using the path-finder from the standard library, it's straightforward to create an NPC that moves towards another object, such as the player character:

```
#duckling
(name *) duckling
```

```
#duckling
(name *) duckling
(animate *)
```

```
(on every tick)
  %% Only do this with 50% probability, to allow the player to get away.
  (random from 1 to 2 into 1)
  (par)
    (current player $Player)
    ($Player is in room $Target)
    (* is in room $Here)
    (if) (first step from $Here to $Target is $Dir) (then)
      (let * go $Dir)
    (elseif) (player can see *) (then)
      "Quack."
  (endif)
```

[\[Copy to clipboard\]](#)

Other NPC actions

In the previous section, we saw that (let \$ go \$) can be used to move non-player characters around, and that it makes queries to (narrate \$ leaving \$ \$ to \$) and (narrate \$ entering \$ from \$).

The following predicates are also provided:

Let-predicate	Narration predicate
(let \$NPC take \$Obj)	(narrate \$NPC taking \$Obj)
(let \$NPC drop \$Obj)	(narrate \$NPC dropping \$Obj)
(let \$NPC wear \$Obj)	(narrate \$NPC wearing \$Obj)
(let \$NPC remove \$Obj)	(narrate \$NPC removing \$Obj)
(let \$NPC put \$A \$Rel \$B)	(narrate \$NPC putting \$A \$Rel \$B)
(let \$NPC open \$Obj)	(narrate \$NPC opening \$Obj)
(let \$NPC close \$Obj)	(narrate \$NPC closing \$Obj)
(let \$NPC climb \$Obj)	(narrate \$NPC climbing \$Obj)
(let \$NPC enter \$Obj)	(narrate \$NPC entering \$Obj)
(let \$NPC leave \$Obj)	(narrate \$NPC leaving \$Obj)

All of these let-predicates have the same internal structure: First they check whether the player can see the action, and query the narration predicate if so. Then they update the world model, which often boils down to a single (now) statement.

In other words, these predicates are quite simplistic. They can work as a foundation for more sophisticated story-specific rules, or as a template for rapid prototyping of game ideas. But it is also possible to sidestep them entirely, and do everything from within the (on every tick) rules.

Note in particular that these predicates do not check whether the requested NPC action is possible; that is the responsibility of the story author. If you let an NPC pick up the moon, they will happily go ahead and do so, even if it's in a different room and not an item.

Taking orders

The parser understands e.g. TELL BOB TO GO EAST as well as BOB, E as the action [tell #bob to go #east]. That is, [tell \$NPC to | \$Action]—a list of the word tell, the NPC object, the word to, followed by whatever elements make up the requested action.

By default, NPCs refuse all such requests. Obedience is enabled on a case-by-case basis, by overriding action-handling predicates as usual. We can make use of the let-predicates of the previous section, but remember that we have to ensure that the action is possible. Thus:

#bob

(name *) Bob

```
#bob
(name *) Bob
(proper *)
(male *)

(perform [tell * to go $Dir])
  (* is in room $Room)
  (from $Room go $Dir to room $)

  (let * go $Dir)
```

[\[Copy to clipboard\]](#)

If there is no exit in the given direction, (from \$ go \$ to room \$) fails, and the perform rule falls back on the default rule provided by the library, wherein Bob refuses.

The above is sufficient in a game where Bob is free to roam the map using obvious exits. But if Bob is e.g. locked up in a cage at some point, the rules have to take that into account. Thus, for instance:

```
#bob
(name *) Bob

#bob
(name *) Bob
(proper *)
(male *)

(prevent [tell * to go $])
  (* is #in #cage)
  Bob scratches his head with the banana, and makes a sad gesture towards the cage door.

(perform [tell * to go $Dir])
  (* is in room $Room)
  (from $Room go $Dir to room $)
  "Oh oh. Ah ah!"
  (let * go $Dir)
```

[\[Copy to clipboard\]](#)

Ask and tell

The library provides [a set of standard actions](#) for communicating with NPCs. These include the classical “ask x about y” and “tell x about y” actions, which redirect by default via “talk to x about y” to a simple “talk to x”.

The story author deals with the [talk to \$] action as with any other; namely by overriding the default action-handling predicates. For instance:

```
#librarian
(name *) librarian

#librarian
(name *) librarian
(male *)

(perform [talk to *])
  "Shh! This is a library."
```

[\[Copy to clipboard\]](#)

The ASK and TELL verbs redirect to a common TALK TO action by default, but they can be overridden with more specific responses. However, modern players may expect games to treat ASK, TELL, and TALK TO as synonyms.

```
#librarian
(name *) librarian

#librarian
(name *) librarian
(male *)

(perform [talk to * about #book])
  "I'd like to borrow this book." (line)
  "Certainly. May I see your library card, please?"

(perform [talk to * about $])
  (The *) has nothing to say about that.
```

[\[Copy to clipboard\]](#)

In the above example, ASK LIBRARIAN ABOUT BOOK only works if both the librarian and the the book are in scope. Otherwise, the game will print a generic “I don't understand” message.

But in text adventures—and real life—people often want to talk about things that are outside the current room. The library provides a mechanism for this: Objects with the trait (topic \$) will always be recognized as valid topics, regardless of scope:

```
#fountain
(name *) oddly-

#fountain
(name *) oddly-shaped fountain
```


(an *)
(* is #in #townsquare)
(topic *)
(perform [talk to #librarian about *])
 “What's up with the oddly-shaped fountain in the town square?” (line)
 (select)
 “Ah, yes. Old Chancellor Fhtagn. May I recommend this excellent
introduction to the history of our town?” (line)
 The librarian hands you a dusty tome.
 (now) (#historybook is #heldby #player)
 (or)
 “It's Old Chancellor Fhtagn.”
 (stopping)

[\[Copy to clipboard\]](#)

Topic objects can also be used to refer to abstract concepts. Such intangible topics won't appear in any rooms, but they can have (name \$) and (dict \$) entries like any other objects. They are usually proper nouns. The game might refer to them during disambiguation: “Did you want to ask the librarian about life, the universe, and everything or the copy of Life magazine?”

#lifetopic
(name *) life, the

#lifetopic
(name *) life, the universe, and everything
(proper *)
(topic *)
(perform [talk to #librarian about *])
 “That would be under 823.9, Modern Period Fiction. A for Adams.”

[\[Copy to clipboard\]](#)

As a convenience, (proper topic \$) can be used to declare both traits in one go:

#lifetopic
(name *) life, the

#lifetopic
(name *) life, the universe, and everything
(proper topic *)

[\[Copy to clipboard\]](#)

Using dictionary words to represent topics

To save memory, it is also possible to represent conversational topics using dictionary words.

A couple of rules such as these:

(topic keyword
@childhood)

(topic keyword @childhood)
(topic keyword @youth implies @childhood)
(describe topic @childhood)
 your childhood

[\[Copy to clipboard\]](#)

will make the parser understand TELL DOCTOR ABOUT THE SHATTERED DREAMS OF CHILDHOOD as [tell #doctor about childhood] (recall that the @ character can be omitted inside list expressions). But if you also define:

(topic keyword
@dreams)

(topic keyword @dreams)
(describe topic @dreams)
 your dreams

[\[Copy to clipboard\]](#)

then THE SHATTERED DREAMS OF CHILDHOOD contains two valid keywords, and the game will ask, did you mean to tell the doctor about your childhood or your dreams?

Looking for keywords is a simple approach to topic parsing, but it is occasionally too crude. In the next chapter, we will see how to write arbitrarily complex parser rules for topics.

The unrecognized topic

The dictionary word @? (a single question mark) represents a topic that was unrecognized by the parser. It can be handled as a special case:

```
(perform [talk to
#librarian about ?])
```

```
(perform [talk to #librarian about ?])
```

```
"I really don't know about that."
```

[\[Copy to clipboard\]](#)

But bear in mind that the player might ask an NPC about any portable object, as well as any topic object. The parser will recognize those, and construct corresponding actions (with objects rather than @?), even if there are no explicit rule definitions that match those actions.

Most talking NPCs will therefore have some kind of catch-all rule, defined towards the end of the story source code, where they confess that they don't know much about the subject:

```
%% The following
must come after the
```

```
%% The following must come after the more specific [talk to #librarian about ...] rules.
```

```
(perform [talk to #librarian about $])
```

```
"I really don't know about that."
```

[\[Copy to clipboard\]](#)

The two techniques above can be combined:

```
(perform [talk to
#librarian about ?])
```

```
(perform [talk to #librarian about ?])
```

```
"I really don't know about that."
```

```
%% The following must come after the more specific [talk to #librarian about ...] rules.
```

```
(perform [talk to #librarian about $Obj])
```

```
"I really don't know much about (the $Obj)."
```

[\[Copy to clipboard\]](#)

Choice-based conversation

One possible approach to conversation in parser games is to temporarily switch to choice-based interaction. If you haven't already read the chapter on [choice mode](#), please do so before proceeding.

Typically, one would create a set of choice nodes where the labels represent lines spoken by the player character, and the display-texts contain responses from the NPC.

In the following playable example, the NPC object itself acts as a central hub node, offering an initial set of choices. Some choices cause the conversation to branch away from the hub in order to focus on a particular subject, represented by a separate hub-like structure. Some of the choices are gated, and appear only if certain other nodes of the conversation have been exposed.

```
[?]
#player
(current player *)

#player
(current player *)
(* is #in #repairshop)

#repairshop
(name *)      repair shop
(room *)

(look *)      You're in a noisy workshop.

#mechanic
(name *)      (if (#mech-name is exposed) (then)
               Lisa
             (else)
```

```

        busy-looking mechanic
    (endif)
(proper *)    (#mech-name is exposed)
(female *)
(* is #in #repairshop)
(appearance * $ $)
    (A *) is here, looking busy.

(on every tick)
    (if) (player can see *) (then)
    (The *)
    (select)
    rifles through a box of parts.
    (or)
    removes a couple of screws.
    (or)
    checks the oil pressure of a clunker.
    (at random)
    (endif)

(perform [talk to *])
    "Um, excuse me?"
    (par)
    "Can I help you?"
    (The #mechanic) wipes her hands on a dirty cloth and turns to face you.
    (activate node *)

(after disp (terminating $))
    (par)
    (try [look])
    (tick)

#mech-nice
(#mechanic offers *)
(label *)    "This looks like a nice establishment. Very authentic-looking."
(disp *)    "Happy to hear it."

#mech-wrong
(#mechanic offers *)
(sticky *)
(initial label *) "There's something wrong with my car."
(label *)    "About my car again..."
(disp *)    "Yee-es?"
(* flows to #mech-car)

#mech-car-nowork
(#mech-car offers *)
(label *)    "It doesn't start anymore, is the thing."
(disp *)    "I see. Have you checked the battery?"

#mech-battery
(#mech-car offers *)
    (#mech-car-nowork is exposed)
(label *)    "How do I check the battery?"
(disp *)    "Under the hood. Voltmeter on the plus and minus terminals."

#mech-voltmeter
(#mech-car offers *)
    (#mech-battery is exposed)
(label *)    "Uh, what's a voltmeter?"
(disp *)    "Or a multimeter. I could have a look at it, I guess. I have some time
    next Thursday."

#mech-gas
(#mech-car offers *)
    (#mech-car-nowork is exposed)
    (* is unexposed)    %% Only offer this node once, even if it's not a dead end.
(label *)    "The battery is brand new. Could it be something else, do you think?"
(disp *)    "And you're sure that there's gas in the tank?"

```

```

#mech-gasyes

(#mech-gas offers *)
(label *)      "Gas? Oh, gas! Yes, the man who sold me the car specifically said
               there was gas in the tank."
(dispatch *)   "I was afraid of that. This'll be expensive. I have a free timeslot next
               Thursday."

(* flows to #mech-car)
#mech-gasno
(#mech-gas offers *)
(label *)      "I should certainly think not! That sounds positively dangerous."
(dispatch *)   "I see. Well, I could have a look at it next Thursday."
(* flows to #mech-car)
#mech-deal
(#mech-car offers *)
               (#mech-voltmeter is exposed) (or) (#mech-gas is exposed)
(label *)      "Next Thursday is fine."
(dispatch *)   "All right then. You can leave the car out front."
               (The #mechanic) returns to her work.
               (par)
               "Right. Goodbye for now, then!"
               (par)
               (The #mechanic) nods, and you walk out into the rain.
               (game over { You have no car. })

#mech-nevermind
(#mech-car offers *)
(sticky *)
(label *)      "Never mind."
(* flows to #mechanic)
#mech-name
(#mechanic offers *)
(label *)      "What's your name?"
(dispatch *)   "I'm Lisa. Pleasure to meet you."

#mech-bye
(#mechanic offers *)
(label *)      "It was nice talking to you!"
(dispatch *)   "Any time!" (The #mechanic) returns to her work.
(terminating *)

```

[\[Copy to clipboard\]](#)

And this is what it looks like:

Repair shop
 You're in a noisy workshop.

A busy-looking mechanic is here, looking busy.

```
> talk to mechanic
"Um, excuse me?"
```

"Can I help you?" The busy-looking mechanic wipes her hands on a dirty cloth and turns to face you.

```
1. "This looks like a nice establishment. Very authentic-looking."
2. "There's something wrong with my car."
3. "What's your name?"
4. "It was nice talking to you!"
> 1
"This looks like a nice establishment. Very authentic-looking."
```

"Happy to hear it."

```
1. "There's something wrong with my car."
2. "What's your name?"
3. "It was nice talking to you!"
> undo
Undoing the last turn.
```

Repair shop

```
1. "This looks like a nice establishment. Very authentic-looking."
2. "There's something wrong with my car."
3. "What's your name?"
4. "It was nice talking to you!"
> 3
"What's your name?"
```

"I'm Lisa. Pleasure to meet you."

```
1. "This looks like a nice establishment. Very authentic-looking."
2. "There's something wrong with my car."
3. "It was nice talking to you!"
> 3
"It was nice talking to you!"
```

"Any time!" Lisa returns to her work.

Repair shop

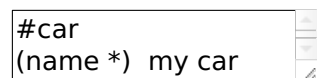
You're in a noisy workshop.

Lisa is here, looking busy.

Lisa checks the oil pressure of a clunker.

>

Ask-and-tell topic objects can serve as shortcuts to conversation nodes. Recall that (choose \$) prints the label, i.e. the player character's line, in a separate paragraph before activating the node:



```
#car
(name *) my car
```

```
#car
(name *) my car
(topic *)
(perform [ask #mechanic about *])
      (choose #mech-wrong)
```

```
#voltmeter
(name *) voltmeter
(topic *)
(perform [ask #mechanic about *])
      (choose #mech-voltmeter)
%% This node is a dead end, so we will remain in parser mode.
```

[\[Copy to clipboard\]](#)

Onwards to “[Chapter 10: Understanding player input](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 2: Objects and state

([Populating the game world](#) • [Descriptions, appearances, and synonyms](#) • [Defining new predicates](#) • [Object locations](#) • [Dynamic predicates](#) • [Hidden objects](#))

Populating the game world

Things in the game world are represented by [objects](#) (hashtags) in the Dialog language. Dialog objects are *thin*; they are identifiers without any code or data inside them. They exist, as names, simply by virtue of being mentioned somewhere in the source code. But they have no contents. Instead, the game world is modelled using [predicates](#), where a predicate is a collection of [rule definitions](#).

We will now extend our minimal example with two objects: A table (called #table) and a chair (#chair). We will give them names and descriptions, categorize them using traits (a kind of inheritable properties), and place them in their initial locations. All of this is achieved by defining rules:

```
%% Rules describing
the table:
```

```
%% Rules describing the table:
```

```
(name #table)      wooden table
```

```
(descr #table)     It's a sturdy wooden table.
```

```
(supporter #table)
```

```
(#table is #in #room)
```

```
%% Rules describing the chair:
```

```
(name #chair)      chair
```

```
(descr #chair)     It's a plastic chair, painted white.
```

```
(on-seat #chair)
```

```
(#chair is #in #room)
```

```
%% These are the rules from the minimal story in the previous chapter:
```

```
(current player #player)
```

```
(#player is #in #room)
```

```
(room #room)
```

[\[Copy to clipboard\]](#)

Every rule definition starts with a *rule head* (the part within parentheses), which must begin at the leftmost column of a line in the source code. The rule head is optionally followed by a *rule body* (that's the non-parenthesized text in the above example), which may continue onto subsequent lines, as long as those lines are indented by at least one space or tab character.

A double-percent in the source code begins a *comment* that lasts to the end of the line.

The rule body is a piece of code that can be executed. [Plain text is an instruction](#) to print that text. Some special characters (#, \$, @, ~, *, |, \, parentheses, brackets, and braces) need to be prefixed by a backslash (\). No special treatment is required for apostrophes or double quotes.

Other instructions may appear within parentheses in rule bodies: For instance, (line) inserts a line break, and (par) inserts a paragraph break. Text in the source code can be formatted freely, because it is broken down internally into a stream of words and punctuation characters, and then reassembled, with space characters automatically inserted in all the right places. This process eliminates the risk of duplicate or missing spaces, line breaks, and paragraph breaks.

```
(descr #table)
  It's a sturdy
```

```
(descr #table)
```

```
It's a sturdy wooden table.
```

```
%% The following instruction inserts a paragraph break (a blank line).
```

```
(par)
```

```
%% The following instruction would normally insert a line break, but it
```

```
%% has no effect here because of the preceding paragraph break.
```

```
(line)
```

```
You got this table from your late aunt Margareth.
```

[\[Copy to clipboard\]](#)

The resulting object description would look like this:

```
> EXAMINE TABLE
```

It's a sturdy wooden table.

You got this table from your late aunt Margareth.

To reduce the repetitiveness of coding by rules, Dialog provides a bit of syntactic sugar: It is possible to set a [current topic](#), which is an object name, by mentioning that object at the leftmost column of a line. Then, whenever an asterisk (*) appears in a position where an object name could go, it is understood as a reference to the current topic. This feature makes the code more concise, and easier to type out:

```
#table
(name *)
```

```
#table
(name *) wooden table
(descr *) It's a sturdy wooden table.
(supporter *)
(* is #in #room)
```

```
#chair
(name *) chair
(descr *) It's a plain chair, painted white.
(on-seat *)
(* is #in #room)
(current player #player)
(#player is #in #room)
(room #room)
```

[\[Copy to clipboard\]](#)

Thus, we have created two tangible objects in our room, given them source code names (#table, #chair), printed names (“wooden table”, “chair”), and descriptions to be revealed by the EXAMINE verb. We have also categorized the objects: The table is a *supporter*, which means that the player is allowed to put things on top of it. The chair is an *on-seat*, which is a special kind of supporter that the player may sit on. The standard library defines many other standard categories that may be used by stories, and we'll take a closer look at them in the upcoming chapter on [traits](#). The predefined categories include *containers* and *in-seats* (such as armchairs), as well as *rooms*.

We have also defined an *initial location* for each object. A location has two parts: A *relation* (#in) and a *parent object* (#room). In addition to #in, the standard library supports the relations #on, #under, #behind, #heldby, #wornby, and #partof.

Descriptions, appearances, and synonyms

The (descr \$) predicate prints the *external description* of an object; what it looks like when viewed from the outside. Another predicate, (look \$), prints the *internal description* of an object, i.e. what it looks like from the inside. This is used for room descriptions.

Let's add a room description to our example game:

```
#player
(current player *)
```

```
#player
(current player *)
(* is #in #room)

#room
(name *) tutorial room
(room *)
(look *) This is a very nondescript room, dominated by a wooden table.
        (notice #table)  %% Binds the word “it” to the table.
```

```
#table
(name *) wooden table
(descr *) It's a sturdy wooden table.
(supporter *)
(* is #in #room)
```

```
#chair
(name *) chair
(descr *) It's a plain chair, painted white.
(on-seat *)
(* is #in #room)
```

[\[Copy to clipboard\]](#)

Try this game! You can LOOK, and then X IT, and SIT (on what? Answer CHAIR).

Did you miss the chair in the room description? In Dialog, as a general design principle, game objects do not call attention to themselves. They are assumed to be part of the scenery, and it is up to the story author to mention (or subtly hint at) their existence in the prose, like we did with the table. Nevertheless, there is a way to furnish objects with *appearances*, which are displayed in separate paragraphs of their own after the room description:

```
#chair
(appearance *)
```

```
#chair
(appearance *) You notice a conspicuous chair here.
```

[\[Copy to clipboard\]](#)

The room description would then turn into:

```
> LOOK
Tutorial room
This is a very nondescript room, dominated by a wooden table.
```

You notice a conspicuous chair here.

Appearances can be very handy for objects that move around during gameplay. This includes objects that the player might pick up, and drop in another room. We will learn more about such objects—and appearances—when we get to the chapter about [Items](#).

How does the game know that CHAIR refers to the chair object? By default, the standard library assumes that every word that appears in the printed name of an object, i.e. the body of the (name \$) rule, can be used to refer to it. If the player types several words in succession, they must all refer to the same object, so WOODEN CHAIR does not match any object in this game. We can easily add extra synonyms to an object, using the (dict \$) predicate:

```
(dict #chair)
white plain
```

```
(dict #chair) white plain
```

[\[Copy to clipboard\]](#)

Now the game would understand SIT ON THE WHITE CHAIR, for instance. Add some synonyms to the game and try them out!

What happens if you add “wooden” as a synonym for the chair, and type EXAMINE WOODEN? What about SIT ON WOODEN?

Noun phrase heads

To assist with disambiguation, it is also possible to declare certain words to be potential *heads* of the noun phrase for a given object. The head of a noun phrase is the main noun, such as “cap” in “the bright red bottle cap of doom”.

Thus, we might define:

```
#bottle
(name *) red bottle
```

```
#bottle
(name *) red bottle
(dict *) crimson decanter
(heads *) bottle decanter
```

```
#cap
(name *) red bottle cap
(heads *) cap
```

[\[Copy to clipboard\]](#)

Now, if the player types EXAMINE BOTTLE, this is unambiguously interpreted as a request to examine the bottle, not the bottle cap, because one of the heads of #bottle was given. If the player types EXAMINE RED, the game will ask if they wanted to examine the red bottle or the red bottle cap. In response to that, the answer BOTTLE is unambiguously understood as the #bottle.

The list of noun heads is only consulted to resolve ambiguities. If the player attempts to TAKE BOTTLE while holding the bottle but not the cap, for instance, then that is interpreted as a request to take the bottle cap.

Typically, (heads \$) definitions are added as needed, on a case-by-case basis, when ambiguities turn up during playtesting.

Defining new predicates

It's easy to conjure up new predicates. We simply define one or more rules for them. For instance, we might want to put the primary construction material of our objects in a separate predicate that we call (material):

```
(material)
caramelized sugar
```

(material) caramelized sugar

[\[Copy to clipboard\]](#)

This predicate can then be [queried](#) from within rule bodies, like so:

```
#table
(name *)
```

```
#table
(name *) (material) table
(descr *) It's a sturdy table made of (material).
```

[\[Copy to clipboard\]](#)

We can use [variables](#) to pass parameters around. In the following example, a generic object description calls out to object-specific material and colour predicates. The standard library doesn't know about these predicates; we just created them by defining rules for them. The library queries (descr \$), and we take it from there:

```
#player
(current player *)
```

```
#player
(current player *)
(* is #in #room)
(descr *)    It's you.

#room
(name *)     tutorial room
(room *)
(look *)     This is a very nondescript room, dominated by a
              wooden table. (notice #table)
```

```
#table
(name *)     table
(material *)  caramelized sugar
(colour *)   dark brown
(supporter *)
(* is #in #room)
```

```
#chair
(name *)     chair
(material *)  plastic
(colour *)   white
(on-seat *)
(* is #in #room)
(descr $Obj) It's (colour $Obj) and made of (material $Obj).
```

[\[Copy to clipboard\]](#)

Note that the rule for (descr #player), on line four, supersedes the generic rule for (descr \$Obj), at the very last line. This is solely due to the order in which they appear in the source code. When coding in Dialog, make sure to always put specific rules before generic rules.

Variables are local to the rule definition in which they appear: In the example, \$Obj is only available from within the last rule. If we were to use a variable named \$Obj inside one of the other rules, that would be a completely unrelated variable.

Queries either [fail or succeed](#). When a predicate is queried, each rule definition is tried in program order, until a match is found. If there is no matching rule, the query fails. As a general rule of thumb, predicates that print text should be designed to always succeed. Therefore, we'll often want to put a catch-all rule at the end of the program, with a wildcard (\$) in the rule head:

```
(material $)
an unknown
```

(material \$) an unknown material

(colour \$) beige
[\[Copy to clipboard\]](#)

The standard library provides a catch-all rule for (descr \$), printing a stock message along the lines of “It seems to be harmless”. But for our material and colour predicates, we have to provide our own catch-all rules.

Object locations

At runtime, objects of the game world are organized into *object trees*. Every object has (at most) one parent, and a relation (in, on, under, behind, held by, or worn by) to that parent. The *root* of a tree has no parent. Sometimes you will run into the expression “the object tree of the game”, as though every object were part of a single, huge tree with only one root. Technically, it is more correct to say “the object forest of the game”, because there can be more than one root object. Rooms don't have parents, so every room is the root of a tree.

In the Dialog standard library, the object forest is encoded using two predicates: (\$Object has parent \$Parent) and (\$Object has relation \$Relation). For brevity, there's an [access predicate](#) that combines them into a single expression: (\$Object is \$Relation \$Parent). We have already seen that one defines the initial location of an object like this:

```
(#chair is #in #room)
```

(#chair is #in #room)
[\[Copy to clipboard\]](#)

and, due to the access predicate, the above is equivalent to the following pair of definitions:

```
(#chair has parent  
#room)
```

(#chair has parent #room)
(#chair has relation #in)
[\[Copy to clipboard\]](#)

From within a rule body, you may query the access predicate to determine the current location of an object:

```
(descr #apple)  
The apple
```

(descr #apple)
The apple
(if) (#apple is #in \$Parent) (then)
in (the \$Parent)
(endif)
looks yummy!
[\[Copy to clipboard\]](#)

The output of the above code might be:

The apple in the fruit basket looks yummy!

With a [multi-query](#), you can [backtrack](#) over every object that has a particular location:

```
(descr #basket)  
It's a plain fruit
```

(descr #basket)
It's a plain fruit basket.
(exhaust) {
*(\$Child is #in \$Obj)
There's (a \$Child) inside.
}
[\[Copy to clipboard\]](#)

Dynamic predicates

The location of an object is [dynamic](#), which means that it can be modified at runtime using the (now) keyword:

```
(descr #apple)  
(if) (#apple is
```

(descr #apple)
(if) (#apple is #in #basket) (then)

```
Yummy!
(else)
The apple seems to be very shy. As soon as you look at it, it
jumps of its own accord into the fruit basket.
(now) (#apple is #in #basket)
(endif)
```

[\[Copy to clipboard\]](#)

The standard library also uses dynamic predicates to track the *internal state* of game world objects. For instance, (\$ is closed) succeeds when a particular openable object (such as a container, or a door) is currently closed. The [access predicate](#) (\$ is open) is defined as its negation, ~(\$ is closed), allowing both forms to appear as queries, now-expressions, and initial value definitions.

A convenience predicate, (open or closed \$), prints the word “open” if the given object is open, and “closed” otherwise. The same thing can of course be coded explicitly with an [if statement](#).

```
#box
(openable *)
```

```
#box
(openable *)
(* is closed)
(descr *)
The box is (open or closed *).
(intro)
Pandora looks at her box. (descr #box)
(now) ~(#box is closed)
She looks away for five seconds, and then looks at it again,
just to check. (descr #box)
(game over { That's just life. })
```

[\[Copy to clipboard\]](#)

The following dynamic predicates are used by the library:

Dynamic predicate Negated version

(\$ is closed)	(\$ is open)	Changed by: OPEN, CLOSE.
(\$ is locked)	(\$ is unlocked)	Changed by: LOCK, UNLOCK.
(\$ is off)	(\$ is on)	Changed by: SWITCH ON, SWITCH OFF.
(\$ is broken)	(\$ is in order)	Never changed by the library.
(\$ is handled)	(\$ is pristine)	Object has been moved by the player.
(\$ is visited)	(\$ is unvisited)	The player has been inside this room.
(\$ is hidden)	(\$ is revealed)	Not suggested during disambiguation.

Hidden objects

Hidden objects, (\$ is hidden), can be in scope (meaning that the parser will recognize them as nouns), but the library is careful not to mention them. Thus, if the player carries a pink slip, and the current room contains a pink elephant that is hidden, then EXAMINE PINK will print the description of the slip, with no disambiguating questions asked. EXAMINE PINK ELEPHANT will examine the elephant, as would EXAMINE PINK if the slip weren't there. The idea is to improve the experience of replaying a game, while avoiding spoilers on the first playthrough.

Hidden objects can be revealed either by directly updating the flag, (now) (#elephant is revealed), or by querying (reveal \$) for the given object. A hidden object is also revealed implicitly when its name is printed, or when (notice \$) is invoked on it.

Onwards to “[Chapter 3: Traits](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 8: More built-in predicates

([Checking the type of a value](#) • [Numbers and arithmetic](#) • [List-related predicates](#) • [Manipulating dictionary words](#) • [System control](#))

Checking the type of a value

Dialog contains a set of built-in predicates for checking if a value is of a particular type. They are:

(number \$X)

Succeeds if and only if \$X is bound to a number.

(word \$X)

Succeeds if and only if \$X is bound to a dictionary word.

(unknown word \$X)

Succeeds if and only if \$X is bound to a word that wasn't found in the game dictionary.

(empty \$X)

Succeeds if and only if \$X is bound to an empty list.

(nonempty \$X)

Succeeds if and only if \$X is bound to a non-empty list.

(list \$X)

Succeeds if and only if \$X is bound to a list (empty or non-empty).

(bound \$X)

Succeeds if and only if \$X is bound to anything. That is, the predicate only fails if \$X is an unbound variable. It succeeds for numbers, dictionary words, and lists, including lists with one or more unbound variables inside.

(fully bound \$X)

Like (bound \$), but performs a full recursive check. It succeeds if and only if \$X is bound, and—in case of a list—contains only fully bound elements.

The last one comes with an extra feature:

(object \$X)

If \$X is bound to an object, the query succeeds. If it is bound to anything else, the query fails. But if \$X is unbound, * (object \$X) backtracks over every object in the game.

Numbers and arithmetic

The Dialog language is designed for symbolic manipulation, predicate logic, and storytelling. Arithmetic is possible, but the syntax is rather clunky.

(\$A plus \$B into \$C)

A and B must be bound to numbers; C is unified with their sum. If the result is outside the valid range of numbers, the query fails.

(\$A minus \$B into \$C)

A and B must be bound to numbers; C is unified with their difference. If the result is outside the valid range of numbers, the query fails.

(\$A times \$B into \$C)

A and B must be bound to numbers; C is unified with their product. If the product is outside the valid range of numbers, the query succeeds, but the numeric result is unpredictable (i.e. it depends on the interpreter).

(\$A divided by \$B into \$C)

A and B must be bound to numbers; C is unified with the (integer) quotient after dividing A by B. The query fails if B is zero.

(\$A modulo \$B into \$C)

A and B must be bound to numbers; C is unified with the remainder after dividing A by B. The query fails if B is zero.

(random from \$A to \$B into \$C)

A and B must be bound to numbers, such that B is greater than or equal to A. A random number in the range A to B (inclusive) is picked, and then unified with C.

(\$A < \$B)

This predicate succeeds if and only if A is numerically less than B.

(\$A > \$B)

This predicate succeeds if and only if A is numerically greater than B.

Common to all of the above predicates is that they fail if A or B is unbound, or bound to a non-number. C may be bound or unbound; it is unified with the result of the computation.

To check for numerical equality, use regular unification, i.e. (\$ = \$).

All numbers in Dialog are restricted to the range 0-16383 (inclusive). This range directly supports four-digit numbers such as years and PIN codes. Pocket money should be fairly straightforward to implement by counting in cents; story authors (or library developers) that require more sophisticated number crunching will have to get creative.

List-related predicates

The Dialog programming language provides the following built-in predicates for working with lists:

(\$Element is one of \$List)

Succeeds if \$Element appears in the \$List. If a multi-query is made, and \$Element is unbound, the predicate will backtrack over each member of the list.

(append \$A \$B \$AB)

Unifies \$AB with the concatenation of \$A and \$B. The first parameter (\$A) must be bound.

(split \$List by \$Keyword into \$Left and \$Right)

See below.

Splitting input by keywords

During parsing, it is often necessary to scan a list for certain keywords, and then split it into two sublists, representing the elements on either side of the matched keyword. It is straightforward to implement this using ordinary rules in Dialog. However, for performance reasons the language also provides a built-in predicate:

(split \$Input by
\$Keyword into \$Left

(split \$Input by \$Keyword into \$Left and \$Right)

[\[Copy to clipboard\]](#)

\$Input must be a list of simple values, i.e. it mustn't contain sublists. \$Keyword must be a simple value, or a list of simple values.

The \$Input list will be scanned, starting at its head, until the first element that is equal to (or appears in) \$Keyword is found. A list of the elements that came before the keyword is unified with \$Left, and a list of the elements that follow it is unified with \$Right. That is, neither \$Left nor \$Right includes the keyword itself.

When invoked as a multi-query, the predicate backtracks over each matching position. Thus:

*(split [the
good , the bad and

*(split [the good , the bad and the ugly]

by [and ,]

into \$Left and \$Right)

[\[Copy to clipboard\]](#)

will succeed twice: First, binding \$Left to [the good] and \$Right to [the bad and the ugly], and then a second time binding \$Left to [the good , the bad] and \$Right to [the ugly].

The split-by predicate can also be used to check whether a list contains one or more of a set of keywords. The standard library uses it that way in the following rule definition:

```
($X contains one of
$Y)
```

```
($X contains one of $Y)
(split $X by $Y into $ and $)
```

[\[Copy to clipboard\]](#)

Manipulating dictionary words

Dictionary words are usually treated as atomic units, but it is possible to extract their constituent characters using the built-in predicate (split word \$ into \$). The output is a list of single-character dictionary words and/or single-digit numbers. Thus:

```
(split word
@fission into $List)
```

```
(split word @fission into $List)
$List
```

[\[Copy to clipboard\]](#)

will print:

```
[f i s s i o n]
```

Conversely, it is possible to construct a dictionary word from a list of words (single-character or otherwise), using (join words \$ into \$):

```
(join words [f u
s i o n] into $Word)
```

```
(join words [f u s i o n] into $Word)
$Word
```

[\[Copy to clipboard\]](#)

will print:

```
fusion
```

The join-words predicate fails if any of the following is true:

- The input is anything other than a list of dictionary words and/or numbers.
- The input consists of more than a single character, and one of those characters is either a word representing a special keystroke (like backspace), or one of the word-separating characters (see [Input](#)): . , ; " * ()
- The resulting word would exceed a backend- or interpreter-imposed length limit. If there is a limit, it is guaranteed to be at least 64 characters.

It is possible to split and join numbers as though they were words:

```
> (get input
[$W])
```

```
> (get input [$W])
(split word $W into $Chars)
(split $Chars by 5 into $LeftChars and $RightChars)
$LeftChars, $RightChars. (line)
(join words $LeftChars into $Left)
(join words $RightChars into $Right)
($Left plus $Right into $Sum)
The sum is $Sum.
```

[\[Copy to clipboard\]](#)

This could result in the following interaction:

```
> 11522
[1 1], [2 2].
The sum is 33.
```

System control

The following built-in predicates offer low-level control over the interpreter and the Dialog runtime. This is decidedly in the domain of library code, so story authors rarely need to worry about these predicates.

(quit)

Immediately terminates the program. This predicate neither fails nor succeeds.

(restart)

Resets the program to its initial state. The only part of the game state that may survive a restart is the state of the output machinery (including the current style and on-screen contents, and whether the transcript feature is on or off). If the operation succeeds, execution resumes from the start of the program. If there is an error, or the interpreter doesn't support restarting, execution continues normally, i.e. the query succeeds.

(save \$ComingBack)

Attempts to save the current game state to a file. The interpreter takes care of asking the player for a filename. In the event of a save error, or if the operation was cancelled, the query fails. On success, the parameter is unified with 0 if we just saved the state, and with 1 if we just restored the state from a file saved by this query.

(restore)

Attempts to restore the current game state from a file. The interpreter takes care of asking the player for a filename. The only part of the game state that may survive a restore is the state of the output machinery (including the current style and on-screen contents, and whether the transcript feature is on or off). If the operation succeeds, execution resumes after the query from which the save file was written. Otherwise, in the event of a load error or if the operation was cancelled, execution continues normally, i.e. the query succeeds.

(save undo \$ComingBack)

Works just like (save \$), but stores the game state in a buffer in memory. This operation is typically invoked once per move.

(undo)

Works like (restore), but restores the game state from the undo buffer. If there is no saved undo state, the predicate fails. If there's some other problem—such as the interpreter imposing a limit on the number of undo states that are retained in memory—the predicate succeeds, and execution continues normally.

(interpreter supports quit)

Succeeds if and only if the current interpreter handles (quit) in a way that is meaningful to the player. For instance, it fails under the Å-machine web interpreter, because a web page cannot close itself.

(interpreter supports undo)

Succeeds if and only if the current interpreter declares that it supports undo functionality.

(transcript on)

Enables the transcript feature. The interpreter takes care of asking the player for a filename. If the operation succeeds, the query succeeds. In case of an error, or if the operation was cancelled, the query fails.

(transcript off)

Disables the transcript feature. This predicate always succeeds.

(display memory statistics)

Prints a line of information specific to the compiler backend, about the peak memory usage in the heap, auxiliary heap, and long-term heap areas. This only works for compiled code (Z-machine or Å-machine). The size of these areas can be adjusted by passing commandline options to the compiler. During debugging and testing, you may wish to invoke this predicate just before quitting, as it will tell you how close you are to the limits.

Onwards to “[Chapter 9: Beyond the program](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 3: Traits

([Custom traits](#) • [Linguistic predicates and traits](#) • [Full names](#) • [Standard traits for categorizing objects](#))

Traits are single-parameter [predicates](#) used by the standard library to categorize objects. For instance, the trait (supporter \$) [succeeds](#) for an object that the player is allowed to put things on top of. The rule bodies of traits never contain any side effects (such as printing text).

As far as the Dialog compiler is concerned, traits are just ordinary predicates.

We declare that the table is a supporter by ensuring that the query (supporter #table) succeeds. The easiest way to achieve this is with the rule definition:

```
(supporter #table)
```

(supporter #table)

[\[Copy to clipboard\]](#)

Trait predicates are supposed to fail for objects that aren't part of the category. For instance, if the player ever tries to put an object on top of themselves, the library will at some point make a query to (supporter #player) to check whether this is allowed. That query will fail (unless, of course, we also add (supporter #player) to our program).

In the standard library, as a convention, traits are named with a category noun or adjective preceding the object, e.g. (openable \$), whereas [dynamic per-object flags](#) have the object first, followed by “is” and an adjective, e.g. (\$ is open). Most per-object flags also have a negated form, whereas traits do not. You may of course ignore these conventions in your own code, if you wish. The Dialog programming language makes no syntactical difference between traits and dynamic flags, and (now) (openable #door) is under most circumstances a perfectly legal statement, transforming the trait into a dynamic predicate. The purpose of the naming convention is to make it easier to understand library code at a glance.

Custom traits

We can easily define our own traits. For instance, we can invent a trait called (fruit \$), to determine whether an object is a fruit. Let's define:

```
(fruit #apple)
(fruit #orange)
```

(fruit #apple)
(fruit #orange)

[\[Copy to clipboard\]](#)

Now, querying (fruit \$) for some object will succeed if that object is the apple or the orange, but fail otherwise. Then we could have a generic object description for fruit:

```
(descr $Obj)
(fruit $Obj) It looks
```

(descr \$Obj) (fruit \$Obj) It looks yummy!

[\[Copy to clipboard\]](#)

When the standard library wants to print the description of an object, it queries (descr \$) with the desired object as parameter. Dialog tries each (descr \$) rule definition in turn, in program order. When it gets to the fruit-rule above, the rule head matches, binding \$Obj to the object in question. Dialog starts to execute the rule body, which immediately makes a query to (fruit \$) for the given object. If that query succeeds, execution proceeds with the rest of the rule body, and “It looks yummy!” is printed. If it fails, the rule fails, and Dialog resumes to search through the remaining (descr \$) rules in the program.

When a rule body begins with a query, Dialog provides [syntactic sugar](#) that allows us to move the query into the rule head. The following line of code is equivalent to the one in the previous example:

```
(descr (fruit $))
It looks yummy!
```

(descr (fruit \$)) It looks yummy!

[\[Copy to clipboard\]](#)

It is possible to define *inheritance* relations between traits:

```
(berry #blueberry)
(berry #cherry)
```



```
(berry #blueberry)
(berry #cherry)
(fruit #apple)
(fruit *(berry $))
```

[\[Copy to clipboard\]](#)

That last rule is equivalent to:

```
(fruit $Obj)
*(berry $Obj)
```

```
(fruit $Obj) *(berry $Obj)
```

[\[Copy to clipboard\]](#)

What it means is: \$Obj is a fruit given that \$Obj is a berry. The asterisk (*) indicates a [multi-query](#). If you haven't gone down the rabbit hole of multi-queries yet, just memorize that inheritance definitions need to have an asterisk in front of the query in the rule body. The asterisk makes it possible to loop over every fruit in the game, for instance like this:

```
(intro)
  Welcome to the
```

```
(intro)
Welcome to the fruit game! Here you will meet \ (and possibly eat\):
```

```
(line)
(exhaust) {
*(fruit $Obj)
(The $Obj).
(line)
}
```

[\[Copy to clipboard\]](#)

We can easily add inheritance relations between our own traits and those of the standard library, in either direction. The following line of code adds a rule to the edibility-trait of the standard library, saying that all fruit are edible. This allows the player to EAT THE BLUEBERRY:

```
(edible *(fruit $))
```

```
(edible *(fruit $))
```

[\[Copy to clipboard\]](#)

Here is a complete example with edible fruit and berries:

```
#player
(current player *)
```

```
#player
(current player *)
(* is #in #room)

#room
(name *)      tutorial room
(room *)
(look *)      This is a very nondescript room, dominated by a
               wooden table. (notice #table)
```

```
#table
(name *)      wooden table
(descr *)     It's a sturdy wooden table.
               (if) ($ is #on *) (then)

               There seems to be something on it.
               (endif)
```

```
(supporter *)
(* is #in #room)

#blueberry
(name *)      blueberry
(berry *)

#cherry
(name *)      dark red cherry
```

```

(berry *)
#apple
(name *)      green apple
(fruit *)

%% All berries are fruit, and all fruit are edible.
(fruit *(berry $))
(edible *(fruit $))

%% The following are not trait inheritance definitions (descr, dict, and the
%% initial location are not traits), so no asterisk is required.

(descr (fruit $)) Yummy!
(dict (fruit $))  fruit
(dict (berry $))  berry
(*(fruit $) is #on #table)

(intro)
Welcome to the fruit game! Here you will meet \
(and possibly eat\):
(line)
(exhaust) {
*(fruit $Obj)
(The $Obj).
(line)
}

```

[\[Copy to clipboard\]](#)

Try this game! Try to examine the table, then SEARCH or LOOK ON it, then perhaps EAT BERRY or EAT CHERRY, and see if the description of the table really changes when it's empty.

Did you notice that it wasn't possible to pick up the fruit in this game? They were presumably eaten directly off the table. Objects that can be picked up are called *items*, and we will discuss this trait at length in [Chapter 4](#). But first, we will take a step back and see how the various traits provided by the standard library fit together.

Linguistic predicates and traits

To print the name of an object, most of the time you'll want to use a predicate called (the \$). This prints the correct determinate article for the given object, followed by its name. So, given the following object definition:

```
#apple
(name *)      green
```

```
#apple
(name *) green apple
```

[\[Copy to clipboard\]](#)

querying (the #apple) would result in the following text being printed:

```
the green apple
```

To print the name of an object together with an indeterminate article, use (a \$) instead. Querying (a #apple) results in:

```
a green apple
```

If you want the article to start with an uppercase letter, use (The \$) or (A \$), respectively.

The standard library offers a lot of flexibility when it comes to declaring object names. We have seen the (name \$) predicate, which provides the actual noun expression. But a number of *linguistic traits* affect how that name gets printed:

```
(an $)
```

Specifies that “an” is the correct indeterminate article for this object name.

```
(proper $)
```

Specifies that this is a proper noun, so that neither “a” nor “the” should appear before it.

```
(plural $)
```

Specifies that this is a plural noun, so that “some” should be used instead of “a” or “an”. This also changes the verb forms printed by certain predicates (see below).

```
(pair $)
```

Inherits all the properties of a plural noun, but also changes the indeterminate article “a” into “a pair of”.

(uncountable \$)

Specifies that the indeterminate article “some” should be used, but that the noun behaves like a singular in every other respect.

(singleton \$)

Specifies that “the” should be used, even in situations where “a” or “an” are usually called for.

(your \$)

Specifies that “your” should be used instead of “a” or “the”.

Some examples:

#orange
(an *)

#orange
(an *)
(name *) orange

#book
(proper *)
(name *) A Clockwork Orange

#bookshelves
(your *)
(plural *)
(name *) bookshelves

#boots
(pair *)
(name *) boots

#water
(uncountable *)
(name *) water

#sun
(singleton *)
(name *) sun

[\[Copy to clipboard\]](#)

To use an object name in a sentence, it is often necessary to select a matching verb form. Predicates are available for this, as well as for printing pronouns. To print the correct personal pronoun, for instance, use (it \$). This will print the word “it” by default, but if the object has the plural trait, it will print the word “they” instead. And if the object happens to be the current player character, the word “you” is printed. There are several such predicates, corresponding to the rows of the following table. The columns displayed here, corresponding to how the linguistic traits have been set up, are not exhaustive.

Predicate	Singular	Plural	Current player
(a \$)	a/an	some	yourself
(A \$)	A/An	Some	You
(the \$)	the	the	yourself
(The \$)	The	The	You
(it \$)	it	they	you
(It \$)	It	They	You
(its \$)	its	their	your
(Its \$)	Its	Their	Your
(itself \$)	itself	themselves	yourself
(them \$)	it	them	you
(that \$)	that	those	yourself
(That \$)	That	Those	You
(is \$)	is	are	are
(isn't \$)	isn't	aren't	aren't
(has \$)	has	have	have
(does \$)	does	do	do
(doesn't \$)	doesn't	don't	don't
(s \$)	(no space) s		

(es \$)	(no space)	es	
(it \$ is)	it is	they're	you're
(the \$ is)	(the \$) is	(the \$) are	you're
(The \$ is)	(The \$) is	(The \$) are	You're
(That's \$)	That's	Those are	You're

The predicates (s \$) and (es \$) are used for attaching verb endings, e.g. (The \$Obj) ponder(s \$Obj) (its \$Obj) existance.

For each of the objects in the previous example, the expression:

You see (a \$Obj). (The \$Obj is)

You see (a \$Obj). (The \$Obj is) drawing attention to (itself \$Obj).

[\[Copy to clipboard\]](#)

would produce:

- You see an orange. The orange is drawing attention to itself.
- You see A Clockwork Orange. A Clockwork Orange is drawing attention to itself.
- You see your bookshelves. Your bookshelves are drawing attention to themselves.
- You see a pair of boots. The boots are drawing attention to themselves.
- You see some water. The water is drawing attention to itself.
- You see the sun. The sun is drawing attention to itself.

And if \$Obj is the current player character, the output is:

- You see yourself. You're drawing attention to yourself.

When the parameter is a [list](#) of several objects, such as [#orange #boots #book], that's handled too:

You see an orange, a pair of boots, and A Clockwork Orange. The orange, the pair of boots, and A Clockwork Orange are drawing attention to themselves.

There are two additional traits, (male \$) and (female \$), that modify the pronouns accordingly.

Note: Don't confuse (a \$) with (an \$)! The former is a predicate for printing the indeterminate article (usually “a”) followed by the name of the object. The latter is a trait, specifying that “an” should be used instead. Thus, somewhere in the standard library, a rule definition for (a \$) contains a query to the predicate (an \$) in order to determine what article it needs to print.

Full names

Two additional predicates deserve to be mentioned here: Whenever the standard library describes an action (e.g. to narrate an automatic action such as opening a door before walking through it, or as part of a disambiguating question), it prints the names of the involved objects using (the full \$) or (a full \$). These predicates print the object name using (the \$) or (a \$), and then, if the query (clarify location of \$) succeeds for the object, some additional information pertaining to its location is printed. By default, this flag is enabled for all non-singleton doors, so that the game might ask the player: Did you want to open the door to the north or the door to the east?

Standard traits for categorizing objects

The standard library categorizes objects using a system of traits. Most of these traits model one of the following three different aspects of an object:

- Where it may appear in the object tree.
- Whether the object can be manipulated at all.
- What actions may be carried out on the object.

Traits that determine where an object may appear in the object tree:

Arrows indicate inheritance.



Container objects allow the player to put other objects #in them. *Supporter* objects allow the player to put other objects #on them.

Actor containers are containers that the player is allowed to enter. *Actor supporters* are supporters that the player is allowed to climb on top of.

Rooms are actor containers with no parents in the object tree. They are organized into a map using connections; this will be explained in the chapter on [moving around](#). Some of those connections involve *doors* (physical doors or other

kinds of gatekeepers). Doors are conceptually located in the liminal space between rooms, but for practical reasons they appear as children of rooms in the object tree: When the player enters a room, the library automatically moves adjacent door objects inside the room object.

A less commonly used trait is (seat \$), the category of objects that give the player a place to sit down. These can be divided into *on-seats* (that the player may sit on) and *in-seats* (that the player may sit in, such as armchairs). Some behaviour is common to all seats, for instance that when the player tries to go up, this is interpreted as a desire to leave the seat.

Note that e.g. an on-seat is a kind of seat, but an on-seat is also a kind of actor supporter. If you are familiar with object-oriented programming, you may recognize this as a case of *multiple inheritance*. In class-based programming languages, where objects have inherent types that determine what code to execute, multiple inheritance can be problematic. But in languages such as Dialog, where rules are always applied in source code order, this is not the case.

A *vehicle* is an object that moves with the player if the player attempts to go somewhere while the vehicle object is their parent. Usually, vehicles are either actor supporters or actor containers, but this is not enforced. For an example of a situation where a vehicle is neither a container or a supporter, the player might be #heldby some giant non-player character, and directing that character to move around.

The standard library defines twelve *directions* and seven *relations*. The directions are #north, #northwest, #west, #southwest, #south, #southeast, #east, #northeast, #up, #down, #in, and #out. The relations are #in, #on, #wornby, #under, #behind, #heldby, and #partof. Note that #in is both a relation and a direction.

Directions and relations are never part of the object tree. They only appear as predicate parameters, and inside action expressions. Both directions and relations have printed names, (name \$), but the relations also have several *name variants*:

Relation (name \$) (present-name \$) (towards-name \$) (reverse-name \$)				
#in	in	inside of	into	out of
#on	on	on top of	onto	off
#wornby	worn by	worn by	worn by	off
#under	under	under	under	out from under
#behind	behind	behind	behind	out from behind
#heldby	held by	held by	held by	away from
#partof	part of	part of	part of	away from

Traits that determine whether an object can be manipulated at all:



Dialog allows you to model objects that are understood by the parser, but do not really exist in the game world. For instance, a room description might call attention to a cockroach scuttling over the floor and disappearing into a hole in the wall. If the player then tries to do anything to the cockroach (such as EXAMINE it), a response message about the cockroach not being here is preferable to a generic parser error.

Actions involving (not here \$) objects generally fail with the message: “(The \$Obj) is not here.” Actions that involve the manipulation of (out of reach \$) objects fail with the message: “You can't reach (The \$Obj).” For (intangible \$) objects, the message is: “(The \$Obj) is intangible.”

For many actions, the player is allowed to refer to a collection of objects using the word ALL. Objects that are (excluded from all \$) are silently omitted from such collections. This also applies to objects marked as (not here \$) or (not reachable \$), via trait inheritance.

Most objects in the game world cannot be picked up—by default, only items can. The standard response when the player tries to pick up a non-item is: “You can't take (the \$Obj).” But if the object is (fine where it is \$), that error message is replaced by: “That's fine where it is.”

Topic objects are recognized by the parser in certain grammatical contexts (e.g. ASK BOB ABOUT ... or LOOK UP ... IN THE MANUAL), even if they are currently out of scope.

Traits that determine what actions may be carried out on an object:



Opaque objects hide their contents when closed. *Openable* objects can be opened (unless they are locked) and closed. *Lockable* objects can be locked and unlocked. Lockable objects are openable by inheritance, and openable objects are opaque (and start out closed). But such inheritance relations can be overridden on a per-object basis.

Items can be picked up by the player. Anything that is carried by the player (usually items) can be dropped, or put inside containers or on top of supporters. *Wearable* objects can be worn or removed. Wearable objects are items by inheritance.

Pushable objects can be pushed from room to room. *Switchable* objects can be turned on or off. *Sharp* objects can be used to cut other objects.

Edible objects can be eaten, which causes them to be removed from the object tree. The player may drink *potable* objects, but those are not removed (the player only takes a sip).

Consultable objects can be consulted about various subject matters. That is, objects and other topics can be looked up in them.

Animate objects can be instructed to do things, although they will refuse by default. They can also be talked to, given things, or shown things. Again, the default implementations of those actions merely print a stock response.

Male and *female* are better described as linguistic traits, as their main function is to replace the default pronouns. They are included in the diagram because of their inheritance relation to the animate trait.

Onwards to “[Chapter 4: Items](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 3: Choice points

([Disjunctions](#) • [Backtracking](#) • [Multi-queries](#) • [Visiting all solutions](#) • [Collecting values](#) • [Collecting words](#) • [Accumulating numbers](#) • [Just](#) • [Infinite loops](#))

Disjunctions

So far, we have seen that rule bodies can contain text and values to be printed, as well as queries to be made to predicates. As soon as a query fails, the entire rule fails. Hence, a rule body can be regarded as a *conjunction* (boolean “and”) of queries.

On the other hand, the rules that make up a predicate are tried one at a time until one succeeds—they form a *disjunction* (boolean “or”).

It is also possible, using special syntax, to put disjunctions inside rule bodies. The (or) keyword may look like a query, but is in fact an infix operator. Here is an example:

```
(program entry
point)
```

```
(program entry point)
Apple: (descr #apple) (line)
Steak: (descr #steak) (line)
Door: (descr #door) (line)

(descr $Obj)
(tasty $Obj) Yummy!
(descr $)
You see nothing unexpected about it.

(tasty $Obj)
(fruit $Obj)
(or)
($Obj = #steak) (player eats meat)
(fruit #apple)
(player eats meat)
\[Copy to clipboard\]
```

Note that this part of the code:

```
(tasty $Obj)
  (fruit $Obj)
```

```
(tasty $Obj)
(fruit $Obj)
(or)
($Obj = #steak) (player eats meat)
\[Copy to clipboard\]
```

is exactly equivalent to:

```
(tasty $Obj) (fruit
$Obj)
```

```
(tasty $Obj) (fruit $Obj)
(tasty $Obj) ($Obj = #steak) (player eats meat)
\[Copy to clipboard\]
```

The output from the program is:

```
Apple: Yummy!
Steak: Yummy!
Door: You see nothing unexpected about it.
```

There can be more than two subexpressions in a disjunction; just separate them all with (or) keywords, e.g. a (or) b (or) c. When Dialog reaches a disjunction, it will attempt to enter the first subexpression. If that subexpression succeeds, the entire disjunction succeeds. But if it fails, Dialog tries the second subexpression, and so on.

Note that the (or) operator has low precedence: It includes everything to the left and right of itself. Therefore, if we try to move the tasty condition verbatim into the Yummy rule itself, we run into trouble:

```
%% This will not
work as desired:
```

```
%% This will not work as desired:
(descr $Obj)
(fruit $Obj) (or) ($Obj = #steak) (player eats meat)
Yummy!
(descr $)
You see nothing unexpected about it.
(fruit #apple)
(player eats meat)
\[Copy to clipboard\]
```

Now if we try to describe the apple, only the first leg of the (or) expression will execute, and the query to (descr \$) will succeed without printing anything.

Another special syntax comes to the rescue: Curly braces { ... } can be used to organize Dialog code into *blocks*. The (or) operation extends as far as it can to the left and right, but it won't go beyond the limits of its containing block. Thus:

```
%% The following
version is correct:
```

```
%% The following version is correct:
(descr $Obj)
{ (fruit $Obj) (or) ($Obj = #steak) (player eats meat) }
Yummy!
(descr $)
You see nothing unexpected about it.
(fruit #apple)
(player eats meat)
\[Copy to clipboard\]
```

Backtracking

Now consider the following predicate, which determines who is considered royalty in some fictional world:

```
(royalty #king)
(royalty #queen)
```

```
(royalty #king)
(royalty #queen)
(royalty $Person)
{
  (the mother of $Person is $Parent)
  (or)
  (the father of $Person is $Parent)
}
(royalty $Parent)
\[Copy to clipboard\]
```

To check whether a person is royalty, unless they're the actual king or queen, we first consider their mother (in the first leg of the disjunction). Let's say the query succeeds, binding \$Parent to whoever is the mother of \$Person. Because the first leg succeeded, so does the entire disjunction. And if the parent turns out to be royalty, then the entire rule succeeds. But otherwise, we're in a position where \$Parent is bound, and the recursive call to (royalty \$) has failed. When this happens, Dialog needs to *unbind* \$Parent, before going back to explore the second leg of the disjunction. This is called *backtracking*.

Here is how backtracking works: When Dialog first encounters a disjunction, it creates a *choice point*. This is a bit like saving the state of a game: A snapshot is made of all relevant variables, and later on we can restore the snapshot and explore a different path forward. And we have already seen the condition that causes Dialog to revert back to the last choice point: It is failure (to satisfy a query).

On failure, all variable bindings are rolled back to their state at the time when the choice point was created, including any variable bindings made from within subroutine calls. But of course, side-effects cannot be undone: What's printed is printed. This can be used to narrate what's going on while the program is searching for a solution:

(program entry point)

```
(program entry point)
{
  ($X = #door)
  (or)
  ($X = #foot)
  (or)
  ($X = #apple)
  (or)
  ($X = #pencil)
}
Checking (the $X).
(fruit $X)  %% If the query fails, the most recent choice point is restored.
Yes, it's a fruit!
```

```
(fruit #apple)
(the #apple) the green apple
(the #door)  the oaken door
(the #foot)  my left foot
(the #pencil) the pencil
```

[\[Copy to clipboard\]](#)

The output is:

Checking the oaken door. Checking my left foot. Checking the green apple. Yes, it's a fruit!

Backtracking during rule matching

Recall that when a query is made, Dialog considers each rule definition in turn, in program order, unifying the parameters of the query with the parameters of the rule head. In case of a match, the rule body starts to execute. We've also seen, that if a failure occurs while executing the body, Dialog proceeds to check the next rule in the predicate, and it keeps doing this until one of the rules succeeds.

Now we are in a position to understand what is actually going on: Before attempting to unify the parameters with those of a rule head, Dialog creates a choice point. In case of failure, the query parameters (and all other variables that have been bound since) are restored to their earlier state, and the next rule is attempted.

However, there is one critical difference between queries and disjunctions: As soon as a rule succeeds, that query is considered over and done. So on success, Dialog simply discards any lingering choice points that were created as part of the query.

Multi-queries

Sometimes when we query a predicate, we want to be able to go back and reconsider every matching rule, even if we already found one that was successful. That is, we want to inhibit the default behaviour of discarding choice points as soon as a rule succeeds. This is done by putting an asterisk (*) before the query, which turns it into a *multi-query*.

No whitespace is allowed between the asterisk and the opening parenthesis.

Here is an example:

(program entry point)

```
(program entry point)
*(fruit $Obj)  %% This is a multi-query. Also, $Obj is unbound here.
$Obj is a fruit.
(colour $Obj)
```

We found a fruit that is also a colour!

```
(colour #blue)
(colour #orange)
(fruit #apple)
(fruit #orange)
(fruit #banana)
```

[\[Copy to clipboard\]](#)

The output of the program is:

#apple is a fruit. #orange is a fruit. We found a fruit that is also a colour!

A multi-query behaves like a disjunction, in that it installs a choice point for going back and trying something else (in this case, attempting to match the query with the next rule in the program). Unlike a normal query, it doesn't resemble a traditional subroutine call, because it can effectively "return" more than once.

The multi-query may set up choice points of its own, for instance using (or) expressions or by making nested multi-queries. All of these choice points remain in effect after the multi-query returns. This provides a very powerful mechanism for searching through a database of relations:

```
(#lisa is a child of
#marge)
```

(#lisa is a child of #marge)
(#lisa is a child of #homer)
(#bart is a child of #marge)
(#bart is a child of #homer)
(#homer is a child of #mona)
(#homer is a child of #abraham)
(#herb is a child of #mona)
(#herb is a child of #abraham)

(male #bart)
(male #homer)
(male #herb)
(male #abraham)

(\$X is the father of \$Y)
*(\$Y is a child of \$X)
(male \$X)

(\$X is a grandfather of \$Y)
*(\$Y is a child of \$Parent)
*(\$X is the father of \$Parent)

(program entry point)
(\$X is a grandfather of #lisa)

The answer is \$X.

[\[Copy to clipboard\]](#)

Output:

The answer is #abraham.

Multi-queries provide a clean mechanism for going back and trying various options. This gives the code a declarative flavour, which can often improve readability.

Although the declarative style makes it easy to see what problem we're trying to solve, it may not be obvious at first how the code in the previous example works. The following detailed description may help:

A query is first made to (\$ is a grandfather of \$), with the second parameter bound to #lisa. That rule in turn makes a multi-query, *(#lisa is a child of \$Parent), with the purpose of backtracking over Lisa's parents. At first, this matches the very first rule in the program, binding \$Parent to #marge. However, because this was a multi-query, a choice point remains in effect for coming back and looking for another parent of Lisa later.

Next, an attempt is made to bind \$X to the father of #marge. This invokes the father-rule, which in turn makes another multi-query, in this case *(#marge is a child of \$X). But Marge's parents aren't in the database, so this multi-query fails. The failure makes Dialog backtrack to the last choice point, unbinding \$Parent as it goes, and proceeding to look for another of Lisa's parents, starting with the second rule of the program. This succeeds, and \$Parent is now bound to #homer.

A query is now made to determine the father of #homer. We're in the father-rule again, making a multi-query: * (#homer is a child of \$X). This will first bind \$X to #mona, but (male #mona) fails. Thanks to the choice point created by the recent multi-query, Dialog goes back and binds \$X to #abraham instead. Now, starting with (male #abraham), everything succeeds, and we end up in the top-level rule again, with \$X bound to #abraham. Because the original query (\$X is a grandfather of #lisa) was a regular (non-multi) query, we also know that any choice points created inside it have now been discarded; a regular query is guaranteed to return at most once.

That example illustrates a general class of problems in the spirit of the pencil-and-paper game Sudoku: Dialog searches through the parameter space by tentatively binding variables to values and checking that all constraints are met, backtracking when they are not, until one or more solutions are found. The main difference between Sudoku and the family tree example is that in the former, all the variables are lined up on a grid from the start, whereas in the latter, the variables are parameters of recursive queries.

In interactive fiction, this technique can be useful for working with small relational databases, as in the example. But its

biggest strength is in parsing and disambiguation: It is possible to implement a parser in a clean, declarative style that takes input (such as NORTH, EAST), and backtracks over various possible interpretations (such as “Go north and east” and “Tell Mrs. North to go east”), which can then be weighed against each other based on their likelihood from a semantical point of view. This is indeed the approach taken by the Dialog standard library.

Now that we know about multi-queries, we are better equipped to deal with lists. There's a very handy built-in predicate called (`$ is one of $`) that takes two parameters—a value and a list—and attempts to unify the value with each member of the list in turn. If both parameters are bound, a regular query to (`$ is one of $`) can be used to check whether the value is in the list:

```
(descr $Obj) ($Obj
is one of [#apple
```

```
(descr $Obj) ($Obj is one of [#apple #orange #banana]) Yummy!
```

```
(descr $Obj) You see nothing unexpected about it.
```

[\[Copy to clipboard\]](#)

But with the help of a multi-query, the same predicate can be used as a list iterator. Here is a more elegant re-implementation of an earlier example:

```
(program entry
point)
```

```
(program entry point)
    *($X is one of [#door #foot #apple #pencil])
    Checking (the $X).
    (fruit $X)
    Yes, it's a fruit!
```

```
(fruit #apple)
```

```
(the #apple) the green apple
```

```
(the #door) the oaken door
```

```
(the #foot) my left foot
```

```
(the #pencil) the pencil
```

[\[Copy to clipboard\]](#)

The output is:

Checking the oaken door. Checking my left foot. Checking the green apple. Yes, it's a fruit!

Incidentally, (`$ is one of $`) is a built-in predicate for performance reasons only. We could have defined it ourselves like this:

```
($Element is one of
[$Element | $])
```

```
($Element is one of [$Element | $])
```

```
($Element is one of [$ | $Tail])
```

```
*( $Element is one of $Tail)
```

[\[Copy to clipboard\]](#)

Note the asterisk in front of the recursive call. By making this a multi-query, `$Element` iterates over the entire `$Tail`, and we simply propagate each of those successful returns up to our caller. The base case of the recursion is implicit, in that neither of the rules will match an empty list.

Visiting all solutions

A Dialog statement can be prefixed with the (`exhaust`) keyword. This will cause Dialog to consider every nook and cranny of the search tree, without remaining on the first successful branch. The statement—usually a block—executes, backtracking is performed on success as well as on failure, and in the end the entire (`exhaust`) construct succeeds. Here is an example:

```
(program entry
point)
```

```
(program entry point)
    (exhaust) {
    *($X is one of [#door #foot #apple #pencil])
    (line)
    Checking (the $X).
    (fruit $X)
    Yes, it's a fruit!
```

```
}  
(line)  
The program continues, but $X is unbound again.
```

```
(fruit #apple)  
(the #apple) the green apple  
(the #door) the oaken door  
(the #foot) my left foot  
(the #pencil) the pencil  
\[Copy to clipboard\]
```

The output is:

```
Checking the oaken door.  
Checking my left foot.  
Checking the green apple. Yes, it's a fruit!  
Checking the pencil.  
The program continues, but $ is unbound again.
```

Incidentally,

```
(exhaust)  
<i>statement</i>
```

(exhaust) *statement*

[\[Copy to clipboard\]](#)

is exactly equivalent to

```
{  
<i>statement</i>
```

```
{ statement (fail) (or) }
```

[\[Copy to clipboard\]](#)

where (fail) is a built-in predicate that always fails.

Collecting values

Computing our way through all the solutions of a query is useful, but sometimes we would like to collect the results of those computations into a list, and then perform some work on the list as a whole. This can be done with the following special syntax:

```
(collect  
$Element)
```

(collect \$Element)

...

(into \$List)

[\[Copy to clipboard\]](#)

The ellipsis represents some code that is expected to bind \$Element to a value. That value is remembered, and backtracking is performed until all possibilities have been exhausted. The collected values are placed into a list, in the order in which they were encountered, and that list is then unified with the output parameter, \$List.

Example:

```
(program entry  
point)
```

(program entry point)

(collect \$F)

*(fruit \$F)

(into \$FruitList)

Come and buy! \$FruitList!

(fruit #apple)

(fruit #orange)

(fruit #banana)

[\[Copy to clipboard\]](#)

The output is:

Come and buy! [#apple #orange #banana]!

Note that the query to (fruit \$) must be a multi-query, otherwise only the first fruit is returned. The first statement inside a collect-expression is nearly always a multi-query.

Collecting words

There is also a special variant of the collect-into syntax:

```
(collect words)
...
```

(collect words)
...
(into \$List)

[\[Copy to clipboard\]](#)

This makes Dialog execute the inner statements, while diverting all output into a list of dictionary words. That list is then unified with the output parameter. Typically, a game has a rule for printing the name of an object. This construct makes it possible to gather all the words that make up that name, in order to match them against player input.

It is assumed that words are collected for purposes of comparison. Thus, in the interest of performance, only the essential part of each dictionary word is reported. On the Z-machine, a long word such as north-west will come out as @north-we, but that will unify just fine with, say, a @north-west obtained from the player's input.

Example:

```
(name #apple)
green apple
```

```
(name #apple) green apple
(dict #apple)  yummy    %% Extra synonyms can be listed here.
(name #door)  mysterious door
(dict #door)   oaken oak

%% By default, include any words mentioned in the name rule:
(dict $Obj)    (name $Obj)

(program entry point)
  (exhaust) {
    *($Obj is one of [#apple #door])
    (collect words)
    *(dict $Obj)
    (into $List)
    The (name $Obj) can be referred to using the words $List.
    (line)
  }
```

[\[Copy to clipboard\]](#)

The output is:

The green apple can be referred to using the words [yummy green apple].
The mysterious door can be referred to using the words [oaken oak mysteriou door].

Printed values (numbers, lists, objects) also end up in the collection, as themselves.

Just as when parsing player input, certain punctuation characters are treated as separate words. They are: . , ; * " ()
When printed back, . , ; and) inhibit whitespace on their left, and (inhibits whitespace on its right:

```
(program entry
point)
```

```
(program entry point)
(collect words)
Hello, world!
(into $List)
The list is: $List (line)
Printing each word:
(exhaust) {
*($Word is one of $List)
$Word
```

```
}  
[Copy to clipboard]
```

produces the output:

The list is: [hello , world!]
Printing each word: hello, world!

Accumulating numbers

Instead of returning the collected values as a list, it is possible to add them together:

```
(accumulate  
$Element)  
  
(accumulate $Element)  
  
...  
(into $Sum)  
[Copy to clipboard]
```

The elements have to be numbers, and the resulting sum must be within the valid range of numbers (i.e. no more than 16383). If that is not the case, then the inner expression is still exhausted (including any side-effects), before the whole accumulate-statement fails.

The \$Element can be a constant. If it is 1, the resulting sum will be the number of ways the inner expression succeeded. Thus:

```
(program entry  
point)  
  
(program entry point)  
(accumulate 1)  
*(fruit $)  
(into $Num)  
I know of $Num pieces of fruit.  
  
(fruit #apple)  
(fruit #orange)  
(fruit #banana)  
[Copy to clipboard]
```

would produce the output:

I know of 3 pieces of fruit.

Just

We have seen that the rules of a predicate work together as a disjunction. Each rule is tried in turn, and failure causes Dialog to backtrack and resume with the next rule. When a rule succeeds, the backtracking may or may not stop: This depends on whether the caller was making a multi-query or a regular query. But sometimes, it makes sense to give the callee, i.e. the set of rules that make up the predicate being queried, influence over when to stop the backtracking.

This is achieved with the (just) statement. When a rule invokes (just)—and this can happen anywhere inside the rule body—Dialog discards any choice points created since the beginning of the present query.

Consider again the example in the previous section, where a list of synonyms is provided for each object by a (dict \$) rule. When a multi-query is made to that predicate, Dialog inevitably finds its way to the generic (dict \$) rule that also throws in all of the words making up the object's printed name. It is conceivable that a game has a few objects that the player should not be able to refer to using their printed names. A common example is the object that represents the player character inside the game world. Suppose (although this is not necessarily a good idea in practice) that you want this object to print as “you”, but parse only as “me”:

```
(name #player)  
you  
  
(name #player)    you  
(dict #player)    (just) me  
  
(name #door)      mysterious door  
(dict #door)      oaken oak    %% Extra synonyms.  
  
%% By default, include any words mentioned in the name rule:  
(dict $Obj)       (name $Obj)  
[Copy to clipboard]
```

Here, when a multi-query is made for `*(dict #player)`, Dialog sets up a choice point as usual, and enters the first matching rule. But this rule uses the `(just)` keyword, immediately discarding the choice point. The upshot of this is that the generic rule, `(dict $Obj)`, is never considered for the `#player`.

To recap, `(just)` discards any choice points that have been created so far while dealing with the current query. This includes choice points created by inner multi-queries, and it even extends a little bit outside the present rule, to the backtracking-over-matching-rules mechanism. But it doesn't go any further than that. Code such as `(just) (just)` is redundant; the second `(just)` has no effect.

Infinite loops

The built-in predicate `(repeat forever)` provides an unlimited supply of choice points. This can be used to create infinite loops (such as the main game loop or the game-over menu).

(program entry point)

`(program entry point)`
`*(repeat forever)`
This gets printed over and over. (line)
`(fail)`

[\[Copy to clipboard\]](#)

In the above example, the query to `(fail)` makes Dialog backtrack to the multi-query to `(repeat forever)`, which will keep on returning, successfully, over and over again.

It is possible to break out of the infinite loop by discarding the choice-point created by `(repeat forever)`. This can be done explicitly using `(just)`, or implicitly by successfully returning from a regular (non-multi) query.

Onwards to “[Chapter 4: More control structures](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 2: Manipulating data

([Local variables](#) • [Values](#) • [Unification](#) • [Partial lists and recursion](#))

Local variables

Recall that a dollar sign on its own is a wildcard. However, a dollar sign that is immediately followed by a word, with no whitespace in between, is a *local variable*. When incoming parameters have names, it becomes possible to pass them on to subqueries.

Variable names, like object names, may contain alphanumeric characters (including a limited range of international glyphs), plus (+), minus (-), and underscore (_) characters. They are case sensitive.

```
(program entry
point)
```

```
(program entry point)
      (descr #apple)
      (descr #orange)
      (descr #pear)

(descr $Thing)
      %% Here, $Thing is a variable that is passed to another query.
      (The $Thing) looks yummy. (line)

(The #apple) The green apple
(The #pear)  The juicy pear
(The $)      That    %% Here, $ is a wildcard. Its value is ignored.
```

[\[Copy to clipboard\]](#)

Since there is no explicit rule definition for (The #orange), the program will print:

```
The green apple looks yummy.
That looks yummy.
The juicy pear looks yummy.
```

It is also possible to print the value of a variable. In the case of objects, this will print the actual hashtag as it appears in the source code, so it is mainly useful for debugging:

```
(program entry
point)
```

```
(program entry point)
(descr #apple)

(descr $Tag)
No description for $Tag!
```

[\[Copy to clipboard\]](#)

This will print:

```
No description for #apple!
```

With the help of variables, we can refine the process of rule matching. Recall that when a query is made, Dialog considers every rule in program order, and attempts to match the parameters. If the match is successful, the rule body starts to execute. In case of a failure, Dialog proceeds to try the next rule in the program.

Ordinary parameter matching is a rather blunt instrument, so queries (sometimes called *guard conditions*) can be placed at the very beginning of a rule body to perform more sophisticated checks. If the guard condition succeeds, the rule applies, otherwise the search continues. Here is a very simplistic approach to world modelling:

```
%% A rule with a
blank body will
```

```
%% A rule with a blank body will succeed (assuming the parameters match).
%% The (fruit $) predicate will succeed for #apple and #orange, but fail for
%% anything else.

(fruit #apple)
(fruit #orange)

(descr #door)      The oaken door is oaken.
(descr $Obj)       (fruit $Obj) Yummy!
```


(descr \$) It seems harmless.
(program entry point)
Apple: (descr #apple) (line)
Door: (descr #door) (line)
Pencil: (descr #pencil) (line)

[\[Copy to clipboard\]](#)

The output is:

Apple: Yummy!
Door: The oaken door is oaken.
Pencil: It seems harmless.

The scope of a local variable is limited to the rule in which it appears.

Values

So far, we have seen one kind of value: the object. There are three more kinds of value in the Dialog programming language: *Number*, *dictionary word*, and *list*.

Number

A number is a non-negative integer in the range 0-16383 inclusive. The printed representation of a number is always in decimal form, with no unnecessary leading zeros. Numbers that appear in the source code must also adhere to this format: For instance, 007 in the source code is regarded as a word of text. This makes a difference inside rule heads and queries, where 007 would be considered part of the predicate name, and not a parameter.

Dictionary word

While objects represent elements of the game world, dictionary words represent input typed by the player. Dictionary words are prefixed with @ instead of #. Unlike object names, they are case-insensitive.

A dictionary word is internally separated into two parts: an *essential part* at the beginning of the word, and an *optional part* at the end. When two dictionary words are compared to each other, only the essential part is considered.

Usually, the optional part is blank. However, on the Z-machine, long dictionary words are split according to an internal, low-level dictionary format. This usually means that the first nine letters of the word are essential, while the rest are optional. For instance, @northeast and @NorthEastern are considered to be the same value. Non-alphabetical characters require multiple slots of storage in the Z-machine, so that e.g. @north-east is considered the same as @north-ea.

The printed representation of a dictionary word is the essential part followed by the optional part. No @ prefix is printed, and there is usually no visible seam between the two parts of the word. However, during [tracing](#), dictionary words are displayed with a plus sign (+) separating the essential part from the optional part.

Dictionary words may contain any characters supported by the underlying platform, e.g. the *ZSCII character set* in the case of the Z-machine.

A handful of characters are used to separate words when parsing input. These cannot appear together with other characters in a dictionary word—they have to stand on their own, as a single-character word. They are: . , ; * " ()

Characters that have special significance in Dialog source code, such as parentheses, can be escaped by a backslash:

@\(

@\(

[\[Copy to clipboard\]](#)

Dictionary words are case insensitive as a general rule, although the Z-machine compiler backend has limited support for case conversion of international characters.

List

A list is an ordered sequence of values. Lists are enclosed in square brackets in the source code:

(program entry point)

(program entry point)

This is a list containing three integers: [1 2 3]

This is an empty list: []

A list may contain values of different kinds: [@hello #world [2 3] 4]

[\[Copy to clipboard\]](#)

In the final expression above, the third element of the list is itself a list, containing the two values 2 and 3.

When a dictionary word appears inside a list, it is possible (and recommended) to omit the @ character.

Printing a list is possible, and highly useful during debugging. The program:

```
(program entry
point)
```

(program entry point)

Have a look at [#this inscrutable list]!

[\[Copy to clipboard\]](#)

will print out:

Have a look at [#this inscrutable list]!

Unbound variables

In addition to the four kinds of value we have seen (object, dictionary word, number, and list), there is a fifth kind, which is more of a pseudo-value: the *unbound variable*. If a local variable appears in a rule body without first being mentioned in the rule head, for instance, it will be unbound. Unbound variables are allowed wherever values are allowed. They can appear as parameters to queries, and even inside lists.

The wildcard that we saw earlier (\$) is in fact an unbound variable, although it also has the special property of being anonymous: Two instances of \$ do not refer to the same variable.

The printed representation of an unbound variable is always \$, regardless of its name.

```
(program entry
point)
```

(program entry point)

This list contains an unbound variable: [one \$Two three]

[\[Copy to clipboard\]](#)

The output is:

This list contains an unbound variable: [one \$ three]

Unification

At the heart of Dialog is a mechanism called *unification*. This is an operation that takes two values, and ensures that they are the same afterwards. If this cannot be done, the operation fails.

Unification is provided by a built-in predicate in Dialog, with the signature (\$ = \$). The equals sign has no special properties; it is treated as a regular word.

Two identical values, e.g. #apple and #apple, unify successfully. Two values that differ, e.g. #apple and #orange, do not unify, so the operation fails. Thus, unification can be used to check for equality:

```
(program entry
point)
```

(program entry point)

(#apple = #apple)

This text will be printed.

(#apple = #orange)

This will not, because the rule has failed by now.

[\[Copy to clipboard\]](#)

An unbound variable successfully unifies with any value, but this also has the effect of *binding* the variable to that value. Thus:

```
(program entry
point)
```

(program entry point)

(\$X = #apples)

(#oranges = \$Y)

I like \$X and \$Y.

[\[Copy to clipboard\]](#)

will print:

I like #apples and #oranges.

Observe that the same operation that was used to check for equality can be used for assignment. It is also symmetrical: The unbound variable can appear either to the left or to the right of the equals sign.

But once the variable is bound, it sticks to that value:

```
(program entry
point)
```

(program entry point)

(\$X = #apples)

I like \$X

(\$X = #oranges)

and \$X.

[\[Copy to clipboard\]](#)

will print:

I like #apples

and then the second unification fails, because \$X resolves to #apples, which is different from #oranges. As a result, the top-level predicate of this example fails, and the program terminates.

Two lists unify if their elements unify, at each and every position. This may have the side-effect of binding unbound variables inside the lists. Consider:

```
(program entry
point)
```

(program entry point)

(\$X = [#apples #pears \$])

(\$X = [\$ #pears #oranges])

I like \$X.

[\[Copy to clipboard\]](#)

The first unification operation will bind \$X to the list [#apples #pears \$], the last element of which is an anonymous unbound variable. The second unification operation will attempt to unify that list with another list, [\$ #pears #oranges]. This will succeed, and by now three bindings are in place: The first anonymous variable is bound to #oranges, the second anonymous variable is bound to #apples, and \$X is bound to [#apples #pears #oranges]. The output of the program is:

I like [#apples #pears #oranges].

Finally, it is possible to unify two unbound variables with each other. This creates a hidden link between them, entangling them, so that if one of them is later bound to a value, the other one will also become bound to the same value:

```
(program entry
point)
```

(program entry point)

(\$X = \$Y)

([spooky action at a distance] = \$X)

This is \$Y.

[\[Copy to clipboard\]](#)

The output of the program is:

This is [spooky action at a distance].

Parameters are passed by unification

Now, here's the kicker: In Dialog (and Prolog, for that matter), parameters to predicates are passed by unification. Remember that when a predicate is queried, the query is compared to each of the rule heads, in program order, until a match is found. That comparison is in fact carried out by attempting to unify each parameter of the query with the corresponding parameter of the rule head.

This has a very interesting and useful consequence, which is that parameters can be used interchangeably as inputs or outputs:

```
(#rock beats
#scissors)
```

```
(#rock beats #scissors)
(#scissors beats #paper)
(#paper beats #rock)
```

```
(program entry point)
(#rock beats $X)    %% Parameters are: Input, output.
When your opponent plays rock, you'd better not play $X.
($Y beats #rock)    %% Parameters are: Output, input.
When your opponent plays rock, you should play $Y.
```

[\[Copy to clipboard\]](#)

The first query (`#rock beats $X`) tells Dialog to search for a rule head with the signature (`$ beats $`), and attempt to unify `#rock` with the first parameter in the rule head, and the unbound variable `$X` with the second. This succeeds on the very first rule encountered, and as a side effect, `$X` is now bound to `#scissors`. That rule has no body, so it succeeds, and control returns to the top-level predicate.

For the second query (`$Y beats #rock`), Dialog searches for a rule head with the signature (`$ beats $`), and attempts to unify the unbound variable `$Y` with the first parameter, and `#rock` with the second. It gets to the first rule: `$Y` would unify successfully with `#rock` from the rule head, because `$Y` is unbound. But `#rock` does not equal `#scissors`, so this unification fails. Hence, the first rule was not a match. Now the second rule is considered: `$Y` is still unbound, and would unify perfectly fine with `#scissors`. But `#rock` from the query doesn't unify with `#paper` from the rule head, so the unification operation fails again. Finally, Dialog tries the third rule: `$Y` unifies successfully with `#paper`, and `#rock` unifies with `#rock`. This time the operation is successful! The rule body is empty, so it succeeds too, and control returns to the top-level predicate.

The output of the program is:

When your opponent plays rock, you'd better not play `#scissors`. When your opponent plays rock, you should play `#paper`.

Partial lists and recursion

Dialog has a special syntax for matching the head (first element) and tail (remaining elements) of a list. Recall that a list is usually a sequence of values in brackets:

```
[1 2 3 4]
```

```
[1 2 3 4]
\[Copy to clipboard\]
```

Such a sequence can be unified with a special expression, called a *partial list*:

```
[$Head | $Tail]
```

```
[$Head | $Tail]
\[Copy to clipboard\]
```

The unification succeeds if `$Head` unifies with the first element of the list, and `$Tail` unifies with a list containing the rest of the elements. Unification of a partial list and an empty list fails, because there is no head element to extract. Here is an example of extracting the head and tail of a list:

```
(program entry
point)
```

```
(program entry point)
([1 2 3 4] = [$A | $B])
A is $A.
B is $B.
```

[\[Copy to clipboard\]](#)

Here is the output:

A is 1. B is [2 3 4].

Unification works both ways, so the same syntax can be used to construct a list from a head and a tail:

```
(program entry
point)
```

```
(program entry point)
($A = 1)
($B = [2 3 4])
Tacking on a new head: [$A | $B]
```

[\[Copy to clipboard\]](#)

The output is:

```
Tacking on a new head: [1 2 3 4]
```

The syntax is not limited to just a single head: Any number of elements can be matched (or tacked on) from the beginning of the list. However, the tail is always represented by a single expression (usually a variable) after the vertical bar (|). So:

```
(program entry point)
([$First $Second | $Rest] = [a b c d e])
([$Second $First | $Rest] = $Result)
The result is $Result.
```

[\[Copy to clipboard\]](#)

produces the following output:

The result is [b a c d e].

Partial lists are very useful in combination with *recursive code*, i.e. predicates that query themselves. Here is an example that considers each element of a list in turn using recursive calls:

```
(program entry point)
      (observe objects [#banana #orange #apple #apple])
(observe objects [])
      You don't see any more fruit.
(observe objects [$Head | $Tail])
      You see (a $Head). (line)
      (observe objects $Tail)
(a #banana) a banana
(a #apple)  an apple
(a $)      an unknown fruit
```

[\[Copy to clipboard\]](#)

The output is:

```
You see a banana.
You see an unknown fruit.
You see an apple.
You see an apple.
You don't see any more fruit.
```

Onwards to “[Chapter 3: Choice points](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 7: The standard actions

([Core actions](#) • [Actions that reveal information](#) • [Actions that print a message](#) • [Diverting actions](#) • [Communication](#) • [Navigation](#) • [Miscellaneous actions](#) • [Debugging actions](#))

Core actions

The Dialog standard library contains many pre-implemented actions, but only eighteen of them are capable of modifying the game world. These are the so called *core actions*. The other (non-core) standard actions either reveal information about the world (that you supply via predicates such as (descr \$) and (from \$ go \$ to \$)), print stock responses, or divert to one of the core actions.

The core actions are:

[take \$Obj]

[drop \$Obj]

[wear \$Obj]

[remove \$Obj]

[put \$Obj \$Relation \$OtherObj]

[open \$Obj]

[close \$Obj]

[lock \$Obj]

[unlock \$Obj]

[switch on \$Obj]

[switch off \$Obj]

[eat \$Obj]

[climb \$Obj]

[enter \$Obj]

[leave \$Obj]

[leave \$Room \$Direction]

[leave \$Room \$Direction by \$Vehicle]

[leave \$Room \$Direction with \$PushedObj]

Common to all the core actions is that their default perform-rules call out to separate narration predicates to print messages to the player. This allows story authors to override only the narration. The narration predicates are queried before the game world is modified. This allows you to print for instance “you pick the apple” instead of “you take the apple” if the apple is still in its original location, e.g. part of the apple tree:

```
(narrate taking
#apple)
```

```
(narrate taking #apple)
```

```
You
```

```
(if) (#apple is pristine) (then) pick (else) take (endif)
```

```
the apple.
```

[\[Copy to clipboard\]](#)

Likewise, you might want to print a particular message when the player is pushing a cart around, without having to dive into the code for updating the position of the cart and player:

```
(narrate leaving $
$Dir with #cart)
```

```
(narrate leaving $ $Dir with #cart)
```

```
The wheels squeak as you push the cart (name $Dir).
```

[\[Copy to clipboard\]](#)

As a rule of thumb, narration predicates are only queried from the eighteen core actions, as part of their default perform-rules. Most other standard actions essentially just print messages; their performance **is** an act of narration. To add flavour to those actions, you must intercept (perform \$) directly.

An exception is the action [look \$Dir], for looking in a direction. It calls out to (narrate failing to look \$Dir) when there's no obvious exit in the given direction.

In order to present a robust and self-consistent world to the player, the story author will want to consider how each of the eighteen core actions might affect the game world. But these considerations only need to be made under certain conditions, as detailed in the following sections.

Items and clothing

If your game has any (item \$) or (wearable \$) objects, or if the player initially carries or wears anything, consider how those objects are affected by the following core actions:

[take \$Obj]

- Makes a query to (narrate taking \$Obj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.

[drop \$Obj]

- Makes a query to (narrate dropping \$Obj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.

Items also have a subtle effect on many of the standard actions, such as eating, drinking, and showing things to people: These actions have before-rules for automatically attempting to pick up objects that are required for the action, but only if those objects are items.

If your game has any (wearable \$) objects, or if the player initially wears anything, also consider the following core actions:

[wear \$Obj]

- Makes a query to (narrate wearing \$Obj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.

[remove \$Obj]

- Makes a query to (narrate removing \$Obj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.

Openable or lockable objects

If your game has any (openable \$) objects, consider how they are affected by:

[open \$Obj]

- Makes a query to (narrate opening \$Obj).
- Updates the (\$ is closed) flag of \$Obj.

[close \$Obj]

- Makes a query to (narrate closing \$Obj).
- Updates the (\$ is closed) flag of \$Obj.

If your game has any (lockable \$) objects, consider how they are affected by:

[lock \$Obj with \$Key]

- Makes a query to (narrate locking \$Obj with \$Key).
- Updates the (\$ is locked) flag of \$Obj.

[unlock \$Obj with \$Key]

- Makes a query to (narrate unlocking \$Obj with \$Key).
- Updates the (\$ is locked) flag of \$Obj.

Switchable or edible objects

If your game has any (switchable \$) objects, consider how those are affected by:

[switch on \$Obj]

- Makes a query to (narrate switching on \$Obj).
- Updates the (\$ is off) flag of \$Obj.

[switch off \$Obj]

- Makes a query to (narrate switching off \$Obj).
- Updates the (\$ is off) flag of \$Obj.

If your game has any (edible \$) objects, consider:

[eat \$Obj]

- Makes a query to (narrate eating \$Obj).
- Updates the location of \$Obj (to nowhere).

Rooms

If your game has any map connections at all, consider:

[leave \$Room \$Dir]

- Makes a query to (prevent entering \$).
- Possibly queries (narrate failing to leave \$Room \$Dir) and stops.
- Makes a query to (narrate leaving \$Room \$Dir).
- Updates the location of the player.
- Marks the new room as visited.
- Makes a query to (narrate entering \$), which usually diverts to [look].

If your game has any (vehicle \$) objects, consider:

[leave \$Room \$Dir by \$Vehicle]

- Makes a query to (prevent entering \$).
- Possibly queries (narrate failing to leave \$Room \$Dir) and stops.
- Makes a query to (narrate leaving \$Room \$Dir by \$Vehicle).
- Updates the location of the vehicle object.
- Marks the new room as visited.
- Makes a query to (narrate entering \$), which usually diverts to [look].

If your game has any (pushable \$) objects, consider:

[leave \$Room \$Dir with \$Obj]

- Makes a query to (prevent entering \$).
- Possibly queries (narrate failing to leave \$Room \$Dir) and stops.
- Makes a query to (narrate leaving \$Room \$Dir with \$Obj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.
- Updates the location of the player.
- Marks the new room as visited.
- Makes a query to (narrate entering \$), which usually diverts to [look].

Containers and supporters

If your game has any (container \$) or (supporter \$) objects, consider how they are affected by the following core action:

[put \$Obj \$Rel \$OtherObj]

- Makes a query to (narrate putting \$Obj \$Rel \$OtherObj).
- Updates the location of \$Obj.
- Marks \$Obj as handled.

If your game has any (actor supporter \$) objects, consider:

[climb \$Obj]

- Makes a query to (narrate climbing \$Obj).
- Updates the location of the player.

If your game has any (actor container \$) objects, (door \$) objects, or map connections, consider:

[enter \$Obj]

- Makes a query to (prevent entering \$Obj).
- Makes a query to (narrate entering \$Obj).
- Updates the location of the player.

If your game has any (actor supporter \$) objects or (actor container \$) objects, or if the player is initially on top of an object, or inside a non-room object, consider:

[leave \$Obj]

- Makes a query to (narrate leaving \$Obj).
- Updates the location of the player.

Actions that reveal information

The following standard actions do not modify the game world, except to reveal hidden objects. Thus, if (#key is hidden) and (#key is #under #rug), then LOOK UNDER RUG will clear the hidden-flag, so that a subsequent GET KEY is allowed to ask: “Did you mean the key under the rug, or the key on top of the table?”

The act of revealing hidden objects isn't listed explicitly in the following table, because it is carried out automatically by the predicates for printing object names, e.g. (a \$) and (the \$).

[examine \$Obj]

- Makes a query to (descr \$Obj).
- Makes queries to (appearance \$ \$ \$Obj).

[look]

- Makes a query to (look \$).
- Makes queries to (appearance \$ \$ \$).

[exits]

- Displays information from (from \$ go \$ to \$).

[look \$Dir]

- Displays information from (from \$ go \$ to \$).
- If there's no exit in that direction, makes a query to (narrate failing to look \$Dir).

[look \$Rel \$Obj]

- Lists children of \$Obj having relation \$Rel.

[search \$Obj]

- Lists children of \$Obj (with relations #in, #on, #under, and #behind).

[feel \$Obj]

- Makes a query to (feel \$Obj).

[inventory]

- Lists any objects held or worn by the current player.

Actions that print a message

The following actions are part of the standard library, but all they do is print stock messages. Those can be error responses or bland statements about how the action had no effect.

[read \$Obj]

[listen to \$Obj]

[taste \$Obj]

[smell \$Obj]

[smell]

[kiss \$Obj]

[attack \$Obj]

[squeeze \$Obj]

[fix \$Obj]

[clean \$Obj]

[cut \$Obj with \$OtherObj]

[pull \$Obj]
[turn \$Obj]
[flush \$Obj]
[swim in \$Obj]
[tie \$Obj to \$OtherObj]
[talk to \$Obj]
[consult \$Obj about \$Topic]
[greet]
[wait]
[jump]
[dance]
[wave]
[shrug]
[exist]
[sing]
[fly]
[think]
[sleep]
[pray]
[curse]
[wake up]

The following actions require something to be held, so they first attempt to [take \$] that object (if it is an item). Then they print a stock message.

[throw \$Obj at \$OtherObj]
[give \$Obj to \$OtherObj]
[show \$Obj to \$OtherObj]
[attack \$Obj with \$Weapon]
[wave \$Obj]
[drink \$Obj]

Diverting actions

A number of actions simply divert to other actions, possibly after asking the player for clarification. For instance, [give #money] prints “To whom?”, and sets up an *implicit action*, [give #money to []], with a blank for the missing noun. When the player responds e.g. CLERK, that is understood as the complete action, [give #money to #clerk]. This mechanism will be described in more detail in the chapter on [understanding player input](#).

In some situations, it will be clear from context who is the intended recipient of a [give \$] action. Story authors can add rules like the following:

(instead of [give \$Obj])

(current room #store)

(try [give \$Obj to #clerk])

[Copy to clipboard]

Most diverting actions are similar to the above example. They ask for a second noun, and set up an implicit action that will receive it:

[give \$Obj]

- Diverts to [give \$ to \$Obj], after asking for clarification.

[show \$Obj]

- Diverts to [show \$Obj to \$], after asking for clarification.

[tie \$Obj]

- Diverts to [tie \$Obj to \$], after asking for clarification.

[cut \$Obj]

- Diverts to [cut \$Obj with \$], after asking for clarification.

[flush]

- Diverts to [flush \$], after asking for clarification.

[swim]

- Diverts to [swim in \$], after asking for clarification.

[throw \$Obj]

- Diverts to [throw \$Obj at \$], after asking for clarification.

[throw \$Obj \$Dir]

- Diverts to [throw \$Obj at \$] or [throw \$Obj], after consulting the (from \$ go \$ to \$) predicate.

[hug \$Obj]

- Diverts to [kiss \$Obj].

[bite \$Obj]

- Diverts to [attack \$Obj] if \$Obj is animate.
- Otherwise diverts to [eat \$Obj].

[switch \$Obj]

- Diverts to [switch on \$Obj] or [switch off \$Obj] if \$Obj is switchable.
- Otherwise prints a stock message.

[lock \$Obj]

- Asks for clarification, unless the correct key is already held.
- Diverts to [lock \$ with \$].

[unlock \$Obj]

- Asks for clarification, unless the correct key is already held.
- Diverts to [unlock \$ with \$].

The opposite is true for [take \$ from \$], where the default implementation diverts to the core action [take \$] (or [remove \$], for worn objects) by removing the extra noun:

[take \$Obj from \$OtherObj]

- Diverts to [take \$Obj] or [remove \$Obj] if \$Obj is a child of \$OtherObj.
- Otherwise prints a stock message.

LISTEN (without a noun) diverts to an action for listening to the current room, although story authors may wish to override it if a noisy object is nearby:

[listen]

- Diverts to [listen to \$CurrentRoom].

Communication

A number of standard actions are related to communication. They divert according to a tree-like structure, eventually rooted in [talk to \$] by default. But story authors may jack into this structure at any point.



[talk]

- Diverts to [talk to \$], after asking for clarification.

[shout to \$Obj]

- Diverts to [talk to \$].

[shout]

- Diverts to [shout to \$], after asking for clarification.

[call \$Obj]

- Diverts to [shout to \$].

[call]

- Diverts to [shout].

[ask \$Obj about \$Topic]

- Diverts to [talk to \$ about \$].

[tell \$Obj about \$Topic]

- Diverts to [talk to \$ about \$].

[talk to \$Obj about \$Topic]

- Diverts to [talk to \$].

[ask \$Obj]

- Diverts to [talk to \$].

[tell \$Obj]

- Diverts to [talk to \$].

[greet \$Obj]

- Diverts to [talk to \$].

The input JEEVES, CLEAN MY SHOES is represented by the action [tell #jeeves to clean #shoes]. More generally:

[tell \$Obj to | \$Action]

- May divert to [greet \$Obj] (if the inner action was [greet]).
- Otherwise, prints a stock message.

The ask and tell actions have a *topic parameter*. Topics can be ordinary objects, subject to normal scope rules. In addition, objects with the (topic \$) trait are always recognized where the grammar expects a topic, even if they are out of scope—such objects can be used to represent abstract topics that do not correspond to tangible objects in the game world. Topics are described in more detail in the upcoming chapter on [non-player characters](#).

Navigation

A number of standard actions are related to movement of the player character. These also form a tree of diversions, allowing actions to be intercepted at any level. However, when adjusting the behaviour of these actions, it is generally best to override one of the core actions, e.g. [leave \$ \$], [leave \$ \$ by \$], or [leave \$ \$ with \$].



[leave]

- Diverts to [leave \$] or [go #out].

[stand]

- May divert to [leave \$].
- Otherwise, prints a stock message.

[sit]

- Diverts to [climb \$], after asking for clarification.

[push \$Obj]

- Diverts to [push \$Obj \$], after asking for clarification.

[go \$Dir]

- May divert to [leave \$].
- Diverts to [leave \$ \$] or [leave \$ \$ by \$].

[push \$Obj \$Dir]

- Diverts to [leave \$ \$ with \$].

[go to \$Room]

- Diverts to [go \$] (multiple).

[find \$Obj]

- Diverts to [go to \$].

Miscellaneous actions

The verb USE is rarely used in interactive fiction, but it makes sense to treat it as a valid action for set phrases, such as USE TOILET, or LUKE, USE THE FORCE. The standard library accepts USE DOOR for entering doors. Furthermore, when the player has typed an incomplete command, such as CUT ROPE, the game might ask "With what?". In response to such a question, the player should be allowed to answer e.g. USE THE HERRING.

[use \$Obj]

- May divert to any action, if the game recently asked for clarification.
- May divert to [enter \$] (for doors).
- Otherwise, prints a stock message.

The following actions are so called *commands*. Their effects aren't part of the story, and no time passes in the game world when they are issued. Please refer to the library source code for details about what these actions do.

[notify off]

[notify on]

[pronouns]

[quit]

[restart]

[restore]

[save]

[score]

[transcript off]

[transcript on]

[undo]

[verbose]

[version]

AGAIN (G) and OOPS are treated as special cases by the parser. They are not actions.

Debugging actions

In addition to the standard library, the official Dialog distribution archive includes a *debugging extension*, which can be added to the compiler commandline before the standard library (but after the story). This extension adds a number of useful actions that generally should not be part of a released game. The in-game syntax should be evident from the action names:

[actions on]

Queries (actions on) to enable action logging. Whenever an action is about to be tried, its internal representation (data structure) and external representation (description) are printed.

[actions off]

Queries (actions off) to disable action logging.

[scope]

Queries (scope) to display the current scope.

[allrooms]

Prints a list of every room in the game, including unvisited ones.

[teleport \$Room]

Moves the current player character into the named room.

[purloin \$Object]

Moves the named object into the current player character's inventory.

[meminfo]

Prints a line of backend-specific memory statistics.

Onwards to “[Chapter 8: Ticks, scenes, and progress](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 6: Dynamic predicates

([Global flags](#) • [Per-object flags](#) • [Global variables](#) • [Per-object variables](#) • [Has parent](#))

So far, every example of world modelling that we have seen has been eerily static. An apple is yummy, and will remain so forever. The description of a steak might depend on whether the player eats meat or not, but we haven't seen any language facility that would allow us to adjust the vegetarianism of the player during gameplay, or even to change who the current player is.

Dialog supports four kinds of *dynamic predicates*. They are called *global flags*, *per-object flags*, *global variables*, and *per-object variables*.

Global flags

A *global flag* is a predicate with no parameters, and no side effects. It can either succeed (once) or fail. To set a global flag, use the (now) keyword:

```
(now) (player
eats meat)
```

(now) (player eats meat)

[\[Copy to clipboard\]](#)

To clear it, use (now) together with a negation:

```
(now) ~(player
eats meat)
```

(now) ~(player eats meat)

[\[Copy to clipboard\]](#)

We are allowed to define static rules for the global flag, but they won't be accessible at runtime, because queries will merely check the current value of the flag.

However, the compiler will look at the static rules to determine the initial value of the global flag. If there are no rules defined, the flag will be off at the start of the program.

```
(player eats meat)
%% Flag is initially
```

(player eats meat) %% Flag is initially set.

[\[Copy to clipboard\]](#)

Note that there is no need to declare anywhere that (player eats meat) is a dynamic predicate. That follows implicitly from the fact that we try to use (now) on it. However, if—due to the use of (now)—a predicate is dynamic, then the compiler will enforce certain restrictions on the static rules that define its initial value, for instance that they have no side-effects, and that they don't depend on the value of other dynamic predicates.

Per-object flags

A *per-object flag* is a predicate with one parameter, and no side effects. When queried with a particular object as parameter, it will either succeed (once) or fail. To set the flag for a particular object, use the (now) keyword:

```
(now) (#door is
open)
```

(now) (#door is open)

[\[Copy to clipboard\]](#)

Per-object flags may only be set for objects (not e.g. lists or numbers). If an attempt is made to set the flag when the parameter isn't bound to an object, a fatal runtime error occurs.

To clear the flag, use (now) together with a negated query:

```
(now) ~(#door
is open)
```

(now) ~(#door is open)

[\[Copy to clipboard\]](#)

It is safe to attempt to clear the flag for a non-object: Nothing happens in that case.

It is also possible to clear the flag for every object in the game, by supplying an anonymous variable as parameter. This

is faster than iterating over each object in turn:

```
(now) ~($ is open)
```

(now) ~(\$ is open)

[\[Copy to clipboard\]](#)

But it is not possible to set the flag for every object in the game in this way; that triggers a runtime error. The rationale for this is that an unbound variable (such as \$) may take on any value, e.g. a number or list, and per-object flags may only be set for objects.

To check if the flag is set for a particular object, simply query the predicate:

```
The door is (if)
(#door is open)
```

The door is (if) (#door is open) (then) open (else) closed (endif).

[\[Copy to clipboard\]](#)

If the predicate is queried with a parameter that's bound to something other than an object, it fails without generating a fatal error: A per-object flag may only be set for objects, but it can be queried for any value.

If the parameter is unbound, the query binds it to an object for which the flag is set (failing if there are none). With a multi-query, it is possible to backtrack over all objects that have the flag set:

```
(#reddoor is open)
(#bluedoor is open)
```

(#reddoor is open)
(#bluedoor is open)
(#greendoor is open)
(program entry point)
(now) ~(#bluedoor is open)
(collect \$Thing)
*(\$Thing is open)
(into \$List)
The open things are: \$List

[\[Copy to clipboard\]](#)

This produces the output:

The open things are: [#reddoor #greendoor]

Let's pause for a while and consider a matter of design philosophy. I have claimed that in Dialog, objects are nothing but names, and the world is modelled using relations, represented by predicates. When it comes to per-object flags (and per-object variables, to be introduced below), that standpoint is starting to look tenuous. To be sure, dynamic predicates are designed to look and behave just like ordinary predicates when you query them. And in order to change the state of the game world, you issue now-commands that appear to modify predicates rather than objects. But at the same time, there are restrictions on how you can update those predicates, so that, in effect, what you can do is more or less exactly what you could do by storing flags (and properties) inside objects.

In the end, whether you choose to regard a dynamic per-object flag as something that resides inside the object, or in a separate data structure that represents the dynamic predicate, is entirely up to you. The actual runtime representation is irrelevant, and the compiler will in fact choose among different representations based on how the predicate is accessed from various parts of the program.

Global variables

Global variables are dynamic predicates with a single parameter. To distinguish them from per-object flags, global variables have to be *declared*, using special syntax. This is what it looks like:

```
(global variable <i>
(signature)</i>)
```

(global variable (*signature*))

[\[Copy to clipboard\]](#)

starting at the leftmost column of a line. For instance:

```
(global variable
(current player $))
```

(global variable (current player \$))

[\[Copy to clipboard\]](#)

The initial value of the global variable can be specified with an ordinary rule definition:

```
(current player #bob)
```

(current player #bob)

[\[Copy to clipboard\]](#)

But it is also possible to provide the initial value inside the global variable declaration itself:

```
(global variable  
(current player
```

(global variable (current player #bob))

[\[Copy to clipboard\]](#)

To change the value of a global variable, use the (now) keyword:

```
(now) (current  
player #alice)
```

(now) (current player #alice)

[\[Copy to clipboard\]](#)

Note that the above is a low-level operation. When working with the standard library, the predicate (select player \$) should be used to change the current player. But we'll get to that in Part II.

It is also possible to *unset* the global variable, using ~, so that subsequent queries to the predicate will fail. The parameter must be an anonymous variable. The following line of code could be pronounced “Now, the current player is nobody”, or, in case you're a logician, “Now, it is not the case that the current player is anybody”:

```
(now) ~(current  
player $)
```

(now) ~(current player \$)

[\[Copy to clipboard\]](#)

This is also the default state of a global variable, if no initial value is defined.

As usual, we can query the predicate with a bound parameter (to check if the global variable has that particular value), or with an unbound variable (to obtain the current value). Either query will fail if the global variable is unset.

The value that is stored in a global variable must be bound. Attempts to set a global variable to an unbound variable, or a list with an unbound variable inside, will result in a fatal runtime error.

In the following example, a complex global variable is used to implement an AGAIN command:

```
(global variable (last  
input $))
```

(global variable (last input \$))

(ask for command \$Result)

> (get input \$Words)

(if) (\$Words = [again]) (or) (\$Words = [g]) (then)

(last input \$Result)

(else)

(\$Result = \$Words)

(now) (last input \$Result)

(endif)

[\[Copy to clipboard\]](#)

Per-object variables

A *per-object variable* (sometimes referred to as an object property) is a predicate with two parameters, and no side effects. The first parameter is always an object, and the second parameter can be any bound value. A per-object variable can also be unset for a given object.

When a per-object variable is queried with the first parameter bound to an object, the second parameter will be unified with the current value of the variable. As usual, this can be used to check for a particular value:

```
(if) (#troll
wields #axe) (then)
```

(if) (#troll wields #axe) (then) ... (endif)

[\[Copy to clipboard\]](#)

or to read the current value:

```
(narrate fight with
$Enemy) %%
```

(narrate fight with \$Enemy) %% Assume \$Enemy is bound
(\$Enemy wields \$Weapon)
(The \$Enemy) swings (the \$Weapon) at you.

[\[Copy to clipboard\]](#)

The query fails if the variable is unset.

When a per-object variable is queried with an unbound first parameter, Dialog backtracks over every object in the game, and attempts to unify the second parameter with the current value of the corresponding per-object variable. This is potentially a very slow operation, at least on vintage hardware, and the compiler will print a warning if you attempt to do it. To get rid of the warning, you can explicitly backtrack over all objects yourself, by querying `*(object $)`, and then checking the property.

```
($X wields
#axe) The axe is
```

(\$X wields #axe) The axe is currently in the hands of (the \$X).

[\[Copy to clipboard\]](#)

To change the value of a per-object variable, use the (now) keyword:

```
(now) (#troll
wields #club)
```

(now) (#troll wields #club)

[\[Copy to clipboard\]](#)

Again, the first parameter must be an object, and the second parameter must be bound. If this is not the case, a fatal runtime error occurs.

To unset the per-object variable, use the following syntax, where the second parameter must be an anonymous variable:

```
(now) ~(#troll
wields $)
```

(now) ~(#troll wields \$)

[\[Copy to clipboard\]](#)

The following is also allowed, and faster than doing it explicitly for each object:

```
(now) ~($
wields $) %%
```

(now) ~(\$ wields \$) %% Nobody wields anything anymore.

[\[Copy to clipboard\]](#)

As with the other dynamic predicates, the initial value of a per-object variable is defined with ordinary rules:

```
(#troll wields #club)
```

(#troll wields #club)

[\[Copy to clipboard\]](#)

Has parent

There is one built-in per-object variable with special properties. This is the `($ has parent $)` predicate. It is used to track where in the game world objects are located. In other words, it is an abstraction of the low-level Z-machine object tree operations.

In many ways, `($ has parent $)` works just like any other per-object variable: It is queried and modified in the same way, and when it is modified, the first parameter must always be an object. But it has the additional restriction that the second parameter must also be an object. The benefit of this is that reverse lookup operations—backtracking over every child of a particular object—can be implemented very efficiently.

The (\$ has parent \$) property can be unset in order to detach an object from the object tree.

Here are some examples:

```
(#troll has parent $Room)
```

Determine where the troll is.

```
*($Obj has parent #library)
```

Backtrack over every object in the library.

```
(now) (#troll has parent #cave)
```

Set the parent object of the troll. Under the hood, this will also update the linked structures representing the children of the cave, and the children of the previous parent of the troll.

```
(now) ~(#axe has parent $)
```

Remove the axe from play (i.e. detach it from the object tree).

You are responsible for maintaining a well-formed object tree. This means that you're not allowed to create cycles, such as a pair of boxes inside each other. Compiled Dialog code cannot detect violations of this rule at runtime, but the interactive debugger does.

Be wary of updating the object tree while there is an ongoing iteration: An untimely change of a sibling pointer could easily divert the iterating code into a different part of the object tree.

That being said, Dialog guarantees that you can safely iterate over all objects with a particular parent, and move them (or a subset of them) to a different part of the object tree:

```
(exhaust) {  
  *($Obj has
```

```
  (exhaust) {  
    *($Obj has parent #safe)  
    (now) ($Obj has parent #knapsack)  
  }  
}
```

[\[Copy to clipboard\]](#)

Initial object locations

The initial value of (\$ has parent \$) is computed by making a multi-query to the predicate at compile-time, and noting down the first parent encountered for each object.

The following definitions:

```
(edible #apple)  
(edible #lettuce)
```

```
(edible #apple)  
(edible #lettuce)  
(#apple has parent #bowl)  
(* (edible $) has parent #fridge)
```

[\[Copy to clipboard\]](#)

cause the multi-query to succeed three times, first with (#apple has parent #bowl), then with (#apple has parent #fridge), and finally with (#lettuce has parent #fridge). In the initial object tree, the apple will be located in the bowl, and the lettuce in the fridge.

Onwards to “[Chapter 7: Syntactic sugar](#)” • [Back to the Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 1: Flow of execution

([Predicates and rules](#) • [Printing text](#) • [Parameters, objects, and wildcards](#) • [Success and failure](#))

Dialog is heavily inspired by *Prolog*. Unless you've programmed in Prolog before, you may find the material in the first three or so chapters to be increasingly counter-intuitive, odd, or even downright crazy. But don't worry, we'll take it in small steps, and hopefully the concepts will click together in the end. If at any point you feel that the discussion is too abstract, don't hesitate to head over to [Part II](#) for a while, and then come back.

If you already know Prolog, you're still going to have to pay attention to the details, as there are both obvious and subtle differences between the two languages.

Predicates and rules

This is what “hello world” looks like in Dialog:

```
(program entry
point)
```

```
(program entry point)
Hello, world!
```

[\[Copy to clipboard\]](#)

A Dialog program is a list of *rule definitions*, or *rules* for short. A rule definition always begins at the very first column of a line, but it may span several lines. Subsequent lines of code that belong to the same rule have to be indented by at least one space or tab character.

There can be multiple rules with the same name. If so, they are said to belong to the same *predicate*. The program above contains one predicate, built from a single rule definition. The head of the rule is (program entry point), and this is also the name of the predicate. The body of the rule is some text to be printed verbatim, in this case “Hello, world!” Program execution always begins with the predicate called (program entry point). Sometimes we will refer to this as the top-level predicate.

A predicate may call upon another predicate, similar to subroutine calls in other programming languages. This is referred to as making a *query*. The following program produces exactly the same output as the previous one:

```
(program entry
point)
```

```
(program entry point)
Hello (my own rule)!

(my own rule)
, world
```

[\[Copy to clipboard\]](#)

The parentheses indicate that a query is to be made. When a query is made to a predicate, all rules defined for that predicate are consulted in program order, until one of them *succeeds*; we will return to the concept of success at the end of the present chapter. If the program comprises several source code files, the order of the rules is determined by the order in which the filenames appeared on the command line.

Predicate names are case-sensitive.

Printing text

Rule bodies contain statements to be executed. Plain text in a rule body is an instruction to print that text. Some special characters (#, \$, @, ~, *, |, \, parentheses, brackets, and braces) need to be prefixed by a backslash (\). No special treatment is required for apostrophes or double quotes.

Comments in the source code are prefixed with %% and last to the end of the line.

Thus:

```
(program entry
point)
```

```
(program entry point)
Hello y'all \( and "welcome" \) !    %% This is a comment.
```

[\[Copy to clipboard\]](#)

produces the output:

Hello y'all (and "welcome")!

Dialog is smart about punctuation and whitespace: It interprets the source code text as a stream of words and punctuation characters, and rebuilds the output from that stream, inserting whitespace as appropriate. It knows that no space should precede a comma, that a space should go before an opening parenthesis but not after it, and so on.

```
(program entry
point)
```

(program entry point)
For instance
:This
text

\(which,to all intents and purposes,is silly\(
indeed
\) \)
, prints properly.
[\[Copy to clipboard\]](#)

The output of that is:

For instance: This text (which, to all intents and purposes, is silly (indeed)), prints properly.

It is possible to override the automatic whitespace decisions on a case-by-case basis using the *built-in predicates* (space) and (no space):

```
(program entry
point)
```

(program entry point)
To (no space) gether (space) , apart.
[\[Copy to clipboard\]](#)

The output of that is:

Together , apart.

Line breaks are inserted with the (line) built-in predicate; paragraph breaks with (par). On the Z-machine, a paragraph break is implemented as a blank line of output.

Several adjacent line breaks are merged into one, preventing accidental paragraph breaks. Likewise, several adjacent paragraph breaks are merged into one, along with any adjacent line breaks, preventing accidental runs of more than one blank line.

The following program:

```
(note)
  (line) This goes
```

(note)
(line) This goes on a line of its own. (line)
(program entry point)
(note)
(note)
(par) This goes in a paragraph of its own. (par)
This
is
not broken up.
(note)
[\[Copy to clipboard\]](#)

produces the following output:

This goes on a line of its own.
This goes on a line of its own.

This goes in a paragraph of its own.

This is not broken up.
This goes on a line of its own.

Parameters, objects, and wildcards

Predicates can have *parameters*. The name of a predicate, called its *signature*, is written with dollar signs as placeholders for the parameters. These may appear anywhere in the predicate name. For instance, (descr \$) is the signature of a predicate with one parameter, and in this case the parameter is at the end.

In rule heads and queries, *values* may appear in place of these dollar signs. A common kind of value is the *object*. Objects in Dialog are short, programmer-friendly names that start with a # character and refer to entities in the game world.

Consider the following three rule definitions for the (descr \$) predicate:

```
(descr #apple) The
apple looks yummy.
```

(descr #apple) The apple looks yummy.
(descr #door) The oaken door is oaken.
(descr \$) It looks pretty harmless.

[Copy to clipboard]

When a value appears inside a rule head, the given parameter must have that particular value in order for the rule to succeed. It is also possible to use dollar signs as wildcards in rule heads.

Let's take a look at how the three rule definitions above might be used in a program. To print the description of an object, let's say the #door, one would make the query (descr #door). Dialog would consider each of the three rule definitions in program order. The first rule head doesn't match the query, but the second does. Thus, the text "The oaken door is oaken." is printed. The query (descr #orange) would cause the text "It looks pretty harmless." to be printed.

Note that the general rule, the one with the wildcard, appears last. This is crucial: If the general rule were to appear before e.g. the #door rule in the source code, it would supersede that rule every time, and the door would be described as harmless.

Signatures (predicate names) rarely appear explicitly in the source code. They are implied by rule heads and queries, where parameter values are typically used instead of dollar signs.

Objects are thin

Dialog objects are *thin*, in the sense that each hashtag is a mere identifier, without any inherent behaviour or properties. This is in contrast with object-oriented programming languages, where code and data are organized inside objects and classes. In Dialog, the world is modelled using predicates that specify relations between objects, but the objects themselves are just names.

Object names may contain alphanumeric characters (including a limited range of international glyphs), plus (+), minus (-), and underscore (_) characters. They are case sensitive.

Success and failure

If a query is made to a predicate, but there is no matching rule in the program, the query *fails*. When a rule makes a query, and that query fails, the rule also fails and is immediately abandoned. In this way, the failure condition might propagate to the calling rule, to its calling rule in turn, and so on, all the way to the top-level predicate. Here is a simple program that fails:

```
(program entry
point)
```

(program entry point)
You see an orange. (descr #orange) Now what do you do?
(descr #apple) The apple looks yummy.
(descr #door) The oaken door is oaken.

[Copy to clipboard]

This program will print "You see an orange". Then, because there is neither a rule for (descr #orange) nor a rule for (descr \$), the query (descr #orange) fails. This causes the top rule, i.e. the program entry point, to fail, at which point the entire program terminates. Hence, "Now what do you do?" is never printed.

If failure would always propagate all the way to the top and terminate the program, it would be of little use. So of course, there's more to the story: Recall that a query to a predicate causes each of its rule definitions to be tried, in source code order, until a match is found. What happens when a rule fails, is that this search continues where it left off. Consider the following example:

```
(program entry
point)
```

(program entry point)
(descr #apple)

```
Over and out.  
(descr #apple)  
(the player dislikes #apple)  
Yuck!  
(descr $)  
It looks yummy!  
(the player dislikes #orange)  
\[Copy to clipboard\]
```

A query is made: (descr #apple). There's a matching rule, and this rule makes a query in turn, to the predicate (the player dislikes \$), with the parameter #apple. But this time, there is no matching rule definition, so the query fails. This aborts the execution of the (descr #apple) rule, and the quest to satisfy the original query, (descr #apple), resumes. And indeed there's another match: (descr \$) prints “It looks yummy!” and succeeds. Thus, the program entry point rule will proceed to print “Over and out”.

The complete output is:

It looks yummy! Over and out.

Onwards to “[Chapter 2: Manipulating data](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Appendix: Quick reference

The bare bones

Comments:

From %% to end of line.

Special characters:

#, \$, @, ~, *, |, \, parentheses, brackets, and braces. Prefix by \ to print.

Rule definitions:

- The rule head is in the leftmost column.
- The rule body is a sequence of statements, not in the leftmost column. They form a conjunction.
- Multiple rule definitions for the same predicate are tried in order. They form a disjunction.
- Queries succeed or fail. Failure causes backtracking.

Statements:

- Queries (normal, negated, or multi-) to predicates,
- text to be printed, or
- special syntax.

Value syntax:

#abc

Object name.

@abc

Dictionary word. The @ is optional inside list expressions. Special words to represent non-printable keys: @\n, @\b, @\s, @\u, @\d, @\l, @\r.

\$abc

Variable.

123

Number (the valid range is 0-16383).

[]

Empty list.

[*element1 element2 element3 ...*]

Complete list.

[*element1 element2 element3 ... | tail*]

Partial list.

{ ... }

Closure.

*

Current topic.

The current topic is set with an object name in the leftmost column.

Unification:

- Arguments are passed by unification.
- Simple values unify with themselves.
- Lists unify when each element unifies.
- Unbound variables unify (with values or unbound variables) by binding.

Special syntax

{ ... }

Conjunction.

`~{ ... }`

Negated conjunction.

`... (or) ... (or) ...`

Disjunction.

`(if) ... (then) ... (elseif) ... (then) ... (else) ... (endif)`

Conditions are evaluated at most once. Missing else-clause is assumed to be empty, i.e. succeeds.

`(select) ... (or) ... (or) ... (stopping)`

Branches entered one at a time, then the final branch repeats.

`(select) ... (or) ... (or) ... (cycling)`

Branches entered one at a time, then the cycle repeats.

`(select) ... (or) ... (or) ... (at random)`

Branches entered at random, avoiding repetition.

`(select) ... (or) ... (or) ... (purely at random)`

Branches entered at random, allowing repetition.

`(select) ... (or) ... (or) ... (then at random)`

First like `(select) ... (stopping)`, then like `(select) ... (at random)`.

`(select) ... (or) ... (or) ... (then purely at random)`

First like `(select) ... (stopping)`, then like `(select) ... (purely at random)`.

`(exhaust) statement`

Backtracks over all solutions to the statement (typically a block).

`(collect $Element) ... (into $List)`

Backtracks over all solutions to the inner expression. Values bound to `$Element` are collected in order and unified with `$List`.

`(collect words) ... (into $List)`

Backtracks over all solutions to the inner expression, grabbing all output. Printed words are diverted into `$List`, possibly out of order.

`(accumulate $Element) ... (into $Sum)`

Backtracks over all solutions to the inner expression. Values bound to `$Element` are added and their sum is unified with `$Sum`.

`(determine object $Obj) ... (from words) ... (matching all of $List)`

Backtracks over every object `$Obj` that makes the first inner expression succeed, and for which the second expression (when exhausted) emits at least every word in `$List`.

`(stoppable) statement`

The inner statement executes, succeeding at most once. The `(stop)` built-in breaks out of the innermost stoppable environment. The stoppable environment itself always succeeds.

`(span $Class) statement`

The inner statement executes, succeeding at most once. All output is rendered according to the given style class.

`(div $Class) statement`

The inner statement executes, succeeding at most once. All output is rendered into a rectangular area according to the given style class.

`(status bar $Class) statement`

Like `(div $)`, but the output is rendered into the top status area, which is created if necessary.

(inline status bar \$Class) *statement*

Like (div \$), but the output is rendered into an inline status area. The previous inline status area, if any, is removed from display.

(link) *statement*

The inner statement executes, succeeding at most once. The interpreter displays the output, optionally in the form of a hyperlink. If the hyperlink is selected by the player, the output from the inner statement is appended to the current input buffer, which is submitted.

(link \$Words) *statement*

The inner statement executes, succeeding at most once. The interpreter displays the output, optionally in the form of a hyperlink. If the hyperlink is selected by the player, the given \$Words are appended to the current input buffer, which is submitted.

(link resource \$Id) *statement*

The inner statement executes, succeeding at most once. The interpreter displays the output, optionally in the form of a hyperlink. The hyperlink leads to the resource identified by \$Id.

(log) *statement*

If running in the debugger, execute the inner statement in a stoppable environment. The output appears between line breaks, in a distinct style.

(now) *pseudo-query*

Updates a dynamic predicate.

(just)

Removes choice-points created since the current predicate was queried.

(global variable (name of predicate \$))

Declares a global variable.

(interface (name of predicate))

Declares an [interface](#), i.e. the intended use of a predicate.

(generate \$N (name of predicate \$))

Creates \$N anonymous objects, for which the predicate will succeed.

@(...) ...

Defines an access predicate. Queries or definitions matching the rule head are transformed into the rule body at compile-time.

Built-in predicates

The list is sorted alphabetically, considering just the non-parameter words.

(\$X = \$Y)

Unifies \$X with \$Y.

(\$X < \$Y)

Succeeds if \$X and \$Y are numbers, and \$X is strictly less than \$Y.

(\$X > \$Y)

Succeeds if \$X and \$Y are numbers, and \$X is strictly greater than \$Y.

(append \$A \$B \$AB)

Unifies \$AB with the concatenation of \$A (which must be bound) and \$B.

(bold)

Enables bold text.

(bound \$X)

Succeeds if \$X is bound to a value.

(breakpoint)

Suspends execution (if running in the debugger).

(clear)

Clears the main window, but not the top status area.

(clear all)

Clears the screen and disables the top status area.

(clear div)

Clears or hides the current div.

(clear links)

Transforms all hyperlinks into plain text, except in the status areas.

(clear old)

Clears the screen from all text that the player has had a chance to read.

(compiler version)

Prints the name and version of the compiler.

(display memory statistics)

Displays a backend-specific line of memory usage statistics.

(\$X divided by \$Y into \$Z)

Unifies \$Z with the quotient after dividing \$X by \$Y.

(embed resource \$Id)

Displays the resource identified by \$Id, embedded in the story text.

(empty \$X)

Succeeds if \$X is bound to an empty list.

(fail)

Fails. Equivalent in functionality to e.g. $(1 = 2)$.

(fixed pitch)

Enables fixed-pitch text.

(fully bound \$X)

Succeeds if \$X is bound to a value, and—in case of a list—contains only fully bound elements.

(get input \$)

Reads a line of input from the player. Returns a list of words.

(get key \$)

Waits for the player to press a key. Returns a single-character word.

(\$X has parent \$Y)

Dynamic predicate that succeeds when \$X is a direct child of \$Y in the object tree. Either parameter can be unbound.

(interpreter can embed \$Id)

Succeeds if the current interpreter supports the resource identified by \$Id, and is able to display it using (embed resource \$Id) without falling back on just printing the alt-text.

(interpreter supports inline status bar)

Succeeds if the current interpreter supports inline status areas.

(interpreter supports links)

Succeeds if the current interpreter claims to support hyperlinks, and they are currently enabled. This can change at runtime, for instance if a game is saved and subsequently restored on a different interpreter.

(interpreter supports quit)

Succeeds if the current interpreter supports quit in a way that is meaningful to the player. This can change at runtime, for instance if a game is saved and subsequently restored on a different interpreter.

(interpreter supports status bar)

Succeeds if the current interpreter supports the top status area.

(interpreter supports undo)

Succeeds if the current interpreter claims to support undo. This can change at runtime, for instance if a game is saved and subsequently restored on a different interpreter.

(\$X is one of \$Y)

Unifies \$X with each element of \$Y in turn.

(italic)

Enables italic text.

(join words \$List into \$Word)

Concatenates the dictionary words (or numbers) in \$List into a new dictionary word (or number), and unifies the result with \$Word.

(line)

Requests a line break.

(list \$X)

Succeeds if \$X is bound to a list (empty or non-empty).

(\$X minus \$Y into \$Z)

Unifies \$Z with the difference between \$X and \$Y.

(\$X modulo \$Y into \$Z)

Unifies \$Z with the remainder after dividing \$X by \$Y.

(nonempty \$X)

Succeeds if \$X is bound to an non-empty list.

(no space)

Inhibits automatic whitespace before the next word or punctuation mark.

(number \$X)

Succeeds if \$X is bound to a number.

(object \$X)

Checks if \$X is an object, or—in a multi-query—backtracks over every object.

(par)

Requests a paragraph break.

(progress bar \$ of \$)

Draws a progress bar scaled to fit the width of the current div.

(\$X plus \$Y into \$Z)

Unifies \$Z with the sum of \$X and \$Y.

(quit)

Immediately terminates the interpreter.

(random from \$X to \$Y into \$Z)

Unifies \$Z with a random number in the range \$X to \$Y inclusive.

(repeat forever)

Provides an infinite supply of choice points. Generally invoked with a multi-query.

(restart)

Restarts the program.

(restore)

Restores a saved game (the interpreter asks for a filename).

(reverse)

Enables reverse-video text.

(roman)

Disables all text styles (bold, italic, reverse, and fixed pitch).

(save \$ComingBack)

Saves the current game (the interpreter asks for a filename). Unifies \$ComingBack with 0 after a successful save, 1 after a successful restore.

(save undo \$ComingBack)

Saves the current program state in memory. Unifies \$ComingBack with 0 after a successful save, 1 after a successful restore.

(script off)

Disables transcription.

(script on)

Enables transcription (the interpreter asks for a filename).

(serial number)

Prints the serial number (compilation date) of the current program.

(space)

Forces whitespace before the next word or punctuation mark.

(space \$N)

Prints \$N space characters.

(split \$X by \$Y into \$Left and \$Right)

Splits \$X into two halves around each occurrence of \$Y or any member of \$Y.

(split word \$Word into \$List)

Converts the dictionary word (or number) \$Word into a list of its constituent characters, and unifies the result with \$List.

(stop)

Breaks out of the innermost (stoppable) environment.

(\$X times \$Y into \$Z)

Unifies \$Z with the product of \$X and \$Y.

(trace off)

Disables query tracing.

(trace on)

Enables query tracing.

(undo)

Restores the program state at the time of the latest (save undo 0).

(unknown word \$X)

Succeeds if \$X is bound to a word that wasn't found in the game dictionary.

(unstyle)

Select the default text style for the current division.

(uppercase)

Convert the next printed character to uppercase.

(word \$X)

Succeeds if \$X is bound to a dictionary word.

Entry points and metadata predicates

(error \$ErrorCode entry point)

Execution restarts here when a fatal error has occurred.

(program entry point)

Normal execution starts here.

(story author)

Metadata: Defines the author of the story.

(story blurb)

Metadata: Defines the blurb for the story.

(story ifid)

Metadata: Defines the IFID of the story.

(story noun)

Metadata: Defines the noun (also known as the headline) of the story.

(story release \$N)

Metadata: Defines the release number of the story.

(story title)

Metadata: Defines the title of the story.

(library version)

Defines the library version and is used to identify the library source code file.

(removable word endings)

Defines one or more word endings that can be removed when parsing user input.

(style class \$Name)

Associates one or more style attributes with the given class name.

(define resource \$Id)

Defines the location (local filename or URL) and alt-text of a resource.

Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 7: Syntactic sugar

[\(Access predicates\)](#) • [The current topic](#) • [Nested queries in rule heads](#) • [Alternatives in rule heads](#) • [Automated object generation](#)

Dialog provides a small amount of syntactic sugar, i.e. optional syntax variations that can help make certain code more readable and succinct.

Access predicates

Whereas normal predicates describe what will happen when the program is running, *access predicates* transform the code at compile-time. As a general rule of thumb, they should be used sparingly, and **only for the purpose of increasing source code readability**. This is of course a subjective call.

The rule head of an access predicate definition is prefixed by an @ character. Here is an example from the standard library:

```
@($Obj is open)
~($Obj is closed)
```

@(\$Obj is open) ~(\$Obj is closed)

[\[Copy to clipboard\]](#)

The rule body must be a straightforward conjunction of queries: There can be more than one query, and they can be regular, negated, or multi-queries, but no special syntax such as if-statements is allowed.

Access predicates transform queries. Thus, for instance:

```
(if) (#door is
open) (then) ...
```

(if) (#door is open) (then) ... (endif)

[\[Copy to clipboard\]](#)

is reinterpreted, at compile-time, as:

```
(if) ~(#door is
closed) (then) ...
```

(if) ~(#door is closed) (then) ... (endif)

[\[Copy to clipboard\]](#)

When an access predicate is part of a (now) statement, the (now) operation is applied to each query appearing inside the rule body of the access predicate. In other words,

```
(now) (#door is
open)
```

(now) (#door is open)

[\[Copy to clipboard\]](#)

behaves exactly like:

```
(now) ~(#door
is closed)
```

(now) ~(#door is closed)

[\[Copy to clipboard\]](#)

The standard library uses the following access predicate extensively:

```
@($Obj is $Rel
$Parent)
```

@(\$Obj is \$Rel \$Parent)

*(\$Obj has parent \$Parent)

*(\$Obj has relation \$Rel)

[\[Copy to clipboard\]](#)

Relations in the standard library are #in, #on, #heldby, etc. The predicate (\$ has relation \$) is an ordinary per-object variable, and (\$ has parent \$) is the special built-in predicate that abstracts the Z-machine object tree operations. Thus, a statement such as:

```
(now) (#pizza
is #in #oven)
```

(now) (#pizza is #in #oven)

[\[Copy to clipboard\]](#)

behaves like the following block of statements:

```
{
    (now)
```

```
{
(now) (#pizza has parent #oven)
(now) (#pizza has relation #in)
}
```

[\[Copy to clipboard\]](#)

which makes the pizza a child of the oven in the object tree, and sets the “has relation” property to #in.

Negative now-statements are allowed with access predicates, but only if the body of the access predicate definition is a single query. This inverts the sense of that query, so that:

```
(now) ~(#door
is open)
```

(now) ~(#door is open)

[\[Copy to clipboard\]](#)

is equivalent to:

```
(now) (#door is
closed)
```

(now) (#door is closed)

[\[Copy to clipboard\]](#)

Negative now-statements are not allowed for access predicates with more than one query in the body, because a negated conjunction is under-specified: In the statement (now) ~(#pizza is \$ \$), do we unset the parent, the relation, or both? Incidentally, the standard library sidesteps this thorny philosophical issue by providing a separate access predicate:

```
(now) (#pizza
is nowhere)
```

(now) (#pizza is nowhere)

[\[Copy to clipboard\]](#)

When the name of an access predicate appears in rule-head position, it behaves like a collection of rule definitions, lined up vertically. Any body statements in the original clause affect each of the expanded rule definitions. Consider the following definition:

```
($Obj is #in #oven)
*(edible $Obj)
```

```
($Obj is #in #oven)
*(edible $Obj)
```

[\[Copy to clipboard\]](#)

Because that rule-head matches an access predicate (defined in the standard library), the code above is equivalent to the following pair of rule definitions:

```
($Obj has parent
#oven)
```

```
($Obj has parent #oven)
*(edible $Obj)
```

```
($Obj has relation #in)
*(edible $Obj)
```

[\[Copy to clipboard\]](#)

These rule definitions then contribute to the compile-time computation of initial values for the (\$ has parent \$) and (\$ has relation \$) dynamic predicates.

When you are modelling your game world, and you wish to create a flag that can be accessed with a pair of antonyms as

in the open/closed example, you have to decide which one of them is the real dynamic predicate (closed in the example), and which one is the access predicate (open). A good guiding principle is to choose a representation where the actual per-object flag is initially unset for most objects. That's because you're not allowed to have a catch-all rule (for the initial value) such as:

```
($ is closed)  %%
Error! This is not
```

(\$ is closed) %% Error! This is not allowed for dynamic predicates.

[\[Copy to clipboard\]](#)

The reason is that dynamic per-object flags can only be set for objects. However, it's perfectly all right to have a rule that says:

```
($Obj is closed)
(door $Obj)
```

(\$Obj is closed) (door \$Obj)

[\[Copy to clipboard\]](#)

as long as (door \$) only succeeds for objects.

An access predicate can have multiple rules. Each rule is tried in order, and the formal parameters are matched against program source code. This is not unification, but a simple form of structural matching:

- A variable (in the access predicate rule head) matches any source-code expression.
- A simple constant, like an object name or the empty list, matches only that constant.
- A list (or partial list) matches a list (or partial list) if the heads match, and the tails match.

Note in particular that this behaviour is assymetric: A variable in the program source code does not match a constant in the access predicate rule.

The result of the transformation is again subjected to access predicate matching, so with recursion it is possible to transform complex source code expressions. The standard library uses this technique to deal with grammar declarations.

Like normal rule definitions, access predicate definitions can appear anywhere in the source code, i.e. before or after the rules in which they are used.

The current topic

Rules that belong to the same predicate form a kind of disjunction, but unlike a single, big (or) statement, the rule definitions can be scattered all over the source code. This allows a kind of aspect-oriented programming, where rules are organized according to their high-level purpose.

In object-oriented languages, source code that is specific to a particular object or class tends to be nested inside a common lexical structure, such as a class definition. Since Dialog objects are just names, we can't organize our code in that particular way. But we may still want to put rules pertaining to a particular object close together:

```
(name #apple)
green apple
```

(name #apple) green apple
(dict #apple) yummy
(fruit #apple)

[\[Copy to clipboard\]](#)

To make such code a little less repetitive, Dialog maintains a *current topic*. The current topic is always an object, and we select it by placing the desired object name on a line of its own, beginning in the very first column (as if it were a rule head):

```
#apple
```

#apple

[\[Copy to clipboard\]](#)

Then, when we want to use that object in a place where a value is expected, we simply type an asterisk (*) instead:

```
#apple
(name *) green
```

#apple
(name *) green apple
(dict *) yummy

(fruit *)

[\[Copy to clipboard\]](#)

Use of the current topic is not restricted to rule heads. It works equally well inside queries and list expressions in the rule bodies. Thus, something like this is allowed:

```
#apple
(descr *) Your eyes
```

```
#apple
(descr *) Your eyes feast upon the (name *).
```

[\[Copy to clipboard\]](#)

It is possible to change the topic at any time, and even to return to an earlier topic in a different part of the source code.

Nested queries in rule heads

As we have seen in many of the examples, predicates are often used to categorize objects. For instance, if (fruit \$) is defined for some of the objects in the game, then it's straightforward to query that predicate in order to check whether a particular object is a fruit or not. In addition, a multi-query such as *(fruit \$F) can be used to backtrack over every fruit in the game.

We have also seen several examples of rules that employ such a category check as a guard condition:

```
(descr #door) The
oaken door is oaken.
```

```
(descr #door) The oaken door is oaken.
(descr $Obj) (fruit $Obj) Yummy!
(descr $) It seems harmless.
```

[\[Copy to clipboard\]](#)

Dialog provides syntactic sugar to make this look even cleaner: Nested query-expressions in rule heads. These queries are automatically inserted at the beginning of the rule body, in left-to-right order as they appear in the rule head. The nested rules must have at least one parameter, and that (first) parameter is copied into the rule head, replacing the nested query.

Thus,

```
(descr (fruit $Obj))
Yummy!
```

```
(descr (fruit $Obj))
Yummy!
```

[\[Copy to clipboard\]](#)

is exactly equivalent to:

```
(descr $Obj)
(fruit $Obj)
```

```
(descr $Obj)
(fruit $Obj) Yummy!
```

[\[Copy to clipboard\]](#)

Nested queries can appear anywhere in rule heads, and both negative rules and multi-queries are allowed. The following:

```
(prevent [give
(edible $Obj) to ~
```

```
(prevent [give (edible $Obj) to ~(animate $Target)])
You can't feed something inanimate.
```

[\[Copy to clipboard\]](#)

is exactly equivalent to:

```
(prevent [give $Obj
to $Target])
```

```
(prevent [give $Obj to $Target])
(edible $Obj)
```

~(animate \$Target)

You can't feed something inanimate.

[\[Copy to clipboard\]](#)

If a non-anonymous variable appears only once in a rule, the compiler prints a warning about it, because it is likely a typo. Thus, to avoid this warning, it is recommended to simplify:

```
(descr (fruit $Obj))
  Yummy!
```

```
(descr (fruit $Obj))
  Yummy!
```

[\[Copy to clipboard\]](#)

into:

```
(descr (fruit $))
  Yummy!
```

```
(descr (fruit $))
  Yummy!
```

[\[Copy to clipboard\]](#)

It will still be treated as:

```
(descr $Obj)
  (fruit $Obj)
```

```
(descr $Obj)
(fruit $Obj) Yummy!
```

[\[Copy to clipboard\]](#)

but with some unique, internally-generated variable name instead of “Obj”.

Nested rule-expressions may only appear in rule heads, never inside rule bodies.

Alternatives in rule heads

Dialog provides a shorthand syntax for specifying alternatives in rule heads. A set of simple values (dictionary words, objects, numbers, or the empty list) separated by forward slashes is called a *slash expression*. It is transformed into a nested multi-query to the built-in predicate (\$ is one of \$):

```
(descr
#apple/#banana/#o
```

```
(descr #apple/#banana/#orange)
  Yummy!
```

[\[Copy to clipboard\]](#)

is equivalent to

```
(descr *($ is one of
[#apple #banana
```

```
(descr *($ is one of [#apple #banana #orange]))
  Yummy!
```

[\[Copy to clipboard\]](#)

which in turn is equivalent to

```
(descr $X)
  *($X is one of
```

```
(descr $X)
*($X is one of [#apple #banana #orange])
  Yummy!
```

[\[Copy to clipboard\]](#)

where X represents some internally generated name.

Slash expressions are very useful when dealing with user input and synonyms. Here is an example from the standard library:

```
(grammar
[pull/yank/drag/tug/t
```

```
(grammar [pull/yank/drag/tug/tow [object]] for [pull $])
```

[\[Copy to clipboard\]](#)

Because these expressions expand into multi-queries, they can also function as output parameters:

```
(bird
#blackbird/#duck/#
```

```
(bird #blackbird/#duck/#penguin)
```

```
(program entry point)
```

```
(exhaust) {
*(bird $B)
$B is a bird.
}
```

[\[Copy to clipboard\]](#)

Slash-expressions may only appear in rule heads, never inside rule bodies.

Automated object generation

Sometimes it is desirable to instantiate several identical objects in a game. It is possible to create each object manually, like this:

```
(green grape
#ggrape1)
```

```
(green grape #ggrape1)
```

```
(green grape #ggrape2)
```

```
(green grape #ggrape3)
```

```
(blue grape #bgrape1)
```

```
(blue grape #bgrape2)
```

```
(blue grape #bgrape3)
```

```
(blue grape #bgrape4)
```

```
(blue grape #bgrape5)
```

```
(fruit *(green grape $))
```

```
(fruit *(blue grape $))
```

```
(program entry point)
```

```
(exhaust) {
*(fruit $F)
$F is a fruit. (line)
}
```

[\[Copy to clipboard\]](#)

However, Dialog provides a convenient mechanism for automating the process. The following is functionally equivalent to the above example, although the printed representations of these objects will be different:

```
(generate 3 (green
grape $))
```

```
(generate 3 (green grape $))
```

```
(generate 5 (blue grape $))
```

```
(fruit *(green grape $))
```

```
(fruit *(blue grape $))
```

```
(program entry point)
```

```
(exhaust) {
*(fruit $F)
$F is a fruit. (line)
}
```

[\[Copy to clipboard\]](#)

The printed representation of a generated object is a hash character followed by some unique number, since these objects have no source-code names.

Onwards to “[Chapter 8: More built-in predicates](#)” • Back to the [Table of Contents](#)

Chapter 9: Beyond the program

([Story metadata](#) • [Interfaces](#) • [Runtime errors](#) • [Some notes on performance](#) • [Limitations and the future of Dialog](#))

Story metadata

When an interactive story is released into the wild, the community takes over, cataloguing it, preserving it for posterity, and making it available via archives and databases. To help streamline this work, authors are strongly encouraged to provide *metadata*, such as the title of the work and the name of the author, in a machine-readable format inside the story file.

Dialog strives to comply with the *Treaty of Babel*, which is an initiative to establish a common standard for interactive-fiction metadata. The information is supplied via a special set of predicates, that are queried at compile-time.

Each published story should at the very least include a unique identifier—called the *IFID*—to distinguish it from other works. Different versions of the same story (e.g. bug-fix releases) should be marked with the same IFID. To declare the IFID, use the following predicate:

```
(story ifid)
XXXXXXXX-XXXX-
```

```
(story ifid) XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXXX
```

[\[Copy to clipboard\]](#)

where each X is an uppercase hexadecimal digit. There's an online tool on the [Dialog website](#) that lets you generate such declarations with fresh IFID numbers. You may also use any program capable of generating a universally unique identifier (UUID), such as the `uuidgen` command under Linux (available in the Debian package `uuid-runtime`).

The Dialog compiler will complain if the story contains more than one hundred lines of source code and no IFID is provided. The standard library is not included in the line count.

A story author may also want to specify one or more of the following:

(story title)

Put the title of the story in the body of this rule definition.

(story author)

The full name of the author, or a pseudonym, should go here.

(story blurb)

Put a brief, selling description of the story here.

(story noun)

The default story noun is “An interactive fiction”.

(story release \$)

Specify the release number in the parameter; the rule body should be empty.

The story blurb is allowed to contain paragraph breaks, (`par`), but otherwise these rule bodies should consist entirely of plain, static text. The predicates are queried at compile-time, and the text is extracted and stored in the relevant locations in the output files. You may also query these predicates from your own code; the standard library invokes several of them while printing the banner, for instance.

When compiling to the `zblorb` file format, it is possible to include cover art with your story. This is not part of the Dialog language itself, but is an extra service provided by the Dialog compiler. Specify the name of a PNG file and, optionally, a short textual description of the image, using the commandline options `-c` and `-a` respectively. The cover image should be no larger than 1200 x 1200 pixels in size, and preferably square.

The raw `z8` format has limited support for metadata, so if you select this output format, only the IFID and release number will be included in standard locations in the file. The other predicates are of course still accessible from within the program.

Two elements of metadata are supplied automatically by the compiler: The *compiler version string* and the *serial number*. These can be printed from within the program (and you are strongly encouraged to do so as part of the story banner), by querying the following built-in predicates:

(serial number)

(compiler version)

The serial number is the compilation date in YYMMDD format.

The standard library defines a rule for (library version), which prints the library version string. The compiler looks for this definition to check that a library appears as the last filename on the commandline, and prints a warning otherwise.

Interfaces

Predicates are versatile building blocks with many potential usage patterns. Their parameters can be inputs, outputs, or both, and they can be bound, unbound, or partially bound (i.e. lists with unbound variables inside).

Furthermore, the behaviour of a predicate can depend on multiple rule definitions, and some of the rules that deal with special cases might rely on pattern matching in the rule heads. Thus, even with access to source code, it may not be obvious how a predicate is supposed to be used. To alleviate this problem, Dialog lets you document the intended use of a predicate with an *interface declaration*:

```
(interface
(<i>predicate
```

```
(interface (predicate name))
```

[\[Copy to clipboard\]](#)

In an interface declaration, all parameters should be variables with sensible, self-explaining names. Furthermore, the first character of the variable name has special significance:

<

This parameter is supposed to be fully bound **before** querying the predicate.

>

This parameter is supposed to be fully bound **after** the query succeeds.

Here is an example:

```
(interface
(understand
```

```
(interface (understand $<Words as $>Action))
```

[\[Copy to clipboard\]](#)

From the above declaration, we learn that the predicate is supposed to be queried with a bound value—some words, presumably in a list—as the first parameter. When it succeeds, it will have unified the second parameter with a bound value, representing an action.

Note that < and > have no special significance in variable names in ordinary rule definitions, only in interface declarations.

Being bound before or after a query is subtly different from being an input parameter or an output parameter. For instance, the library provides a predicated called (length of \$ into \$) that counts the elements in a list. The list is allowed to contain unbound values, so the first parameter of the interface declaration cannot begin with a <, even though it is an input. The actual declaration is:

```
(interface (length of
$List into
```

```
(interface (length of $List into $>Number))
```

[\[Copy to clipboard\]](#)

Interface declarations are machine-readable documentation. They do not affect the generated story file in any way. However, the compiler will check the program for interface violations—unbound values that could end up in the wrong place at the wrong time—and warn you about them at compile-time.

Runtime errors

Attempts to violate the constraints imposed on [dynamic predicates](#) will result in fatal runtime errors.

Such errors can also occur if one of the heaps (main, auxiliary, and long-term) is ever exhausted. The heaps are three relatively large memory arrays that the Dialog runtime system uses to keep track of local variables, lists, activation records, choice points and other transient data structures. When one of these arrays fills up—which could happen at any time, in the middle of any operation—there's not much Dialog can do, except abandon all hope and re-initialize itself.

For compiled code, the main heap, auxiliary heap, and long-term heap occupy 1000, 500, and 500 words respectively by default, and this should be more than enough for most interactive stories. The size of each area can be adjusted by passing commandline options to the compiler. A built-in predicate, (display memory statistics), prints the peak usage of

each memory area so far. During debugging and testing, you may wish to invoke this predicate just before quitting, as it will tell you how close you are to the limits.

The main rationale for throwing runtime errors when an invalid (now) operation is attempted, instead of merely failing, is that the compiler can do a better job of optimizing the code if it can assume that now-expressions always succeed.

A fatal runtime error will reset the Dialog evaluation state, clear the heaps, and restart the program from the (error \$ entry point) predicate. The parameter is a small integer, representing the nature of the error:

- 1: Heap space exhausted.
- 2: Auxiliary heap space exhausted.
- 3: Type error: Expected object.
- 4: Type error: Expected bound value.
- 5: Invalid dynamic operation.
- 6: Long-term heap space exhausted.
- 7: Invalid output state.

After a fatal error, the game world could be in an inconsistent state, and there's not much one can do except print an error message and quit. Or is there? The standard library attempts to bring the game back to a known state via the undo facility of the Z-machine.

Some notes on performance

Both the Dialog language and its compiler have been designed with runtime performance in mind. In particular, the language lends itself well to static analysis, and the compiler performs a thorough global analysis of the program in order to choose the ideal way to represent each predicate. Often, what looks like a lengthy search through a series of rule heads can compile down to a simple property lookup or a bunch of comparison instructions.

Execution speed

The programmer can lay the foundation for a well-optimized program, by following two important design principles.

The first principle is: Always dispatch on the first parameter. Whenever a bound value is used to select among many different rule definitions, try to phrase the name of the predicate so that the bound value is the first parameter. That's because the Dialog compiler considers the parameters in left-to-right order when generating accelerated code for rule lookups. An example of a carefully named predicate is (from \$Room go \$Direction to \$Target) from the standard library, which is always queried with a known \$Room.

The second principle is: Make tail calls. The last statement of a rule body is said to be in tail position. Making a query in tail position is cheaper than making it elsewhere. Thus, the following implementation:

```
(say it with $Obj)
My hovercraft is full
```

(say it with \$Obj) My hovercraft is full of (name \$Obj).
(stash \$Obj) (say it with \$Obj) (now) (\$Obj is #in #hovercraft)

[\[Copy to clipboard\]](#)

is not as efficient as this one:

```
(say it with $Obj)
My hovercraft is full
```

(say it with \$Obj) My hovercraft is full of (name \$Obj).
(stash \$Obj) (now) (\$Obj is #in #hovercraft) (say it with \$Obj)

[\[Copy to clipboard\]](#)

In the first version, the code for (stash \$) needs to create an activation record, and set things up so that execution can resume at the (now) statement, with \$Obj still bound to the correct value, after the inner query succeeds. In the second version, there is no need for an activation record: The (now) statement is handled locally, and then control is simply passed to (say it with \$Obj).

If the final statement of a rule is a disjunction or an if-statement, then the last statement inside every branch is also in tail position.

It is especially fruitful to place recursive calls in tail position, as we can then avoid creating an activation record for every step in the recursion.

Memory footprint

Per-object variables have a relatively large memory footprint. If you would like your game to be playable on vintage hardware, try to minimize the number of per-object variables used by your design, especially if they will remain unset for most objects.

For instance, looking back at the (`#troll wields #axe`) example from the chapter on [dynamic predicates](#), if the troll and the player are the only characters that wield weapons, it would be much better to use a pair of global variables:

```
(global variable (troll
wields $))
```

```
(global variable (troll wields $))
```

```
(global variable (player wields $))
```

[\[Copy to clipboard\]](#)

If desired, wrapper predicates could be defined for querying or updating those variables as though they were per-object variables:

```
(#troll wields
$Weapon)
```

```
(#troll wields $Weapon)
```

```
(troll wields $Weapon)
```

```
(#player wields $Weapon)
```

```
(player wields $Weapon)
```

```
(now #troll wields $Weapon)
```

```
(now) (troll wields $Weapon)
```

```
(now #player wields $Weapon)
```

```
(now) (player wields $Weapon)
```

[\[Copy to clipboard\]](#)

To conserve heap memory, use backtracking whenever possible. Identify places in the code where you can implement loops with backtracking instead of recursion. For instance, to iterate over a list and print something for each element, use `exhaust`:

```
(exhaust) {
  *($X is one
```

```
(exhaust) {
```

```
*($X is one of $List)
```

```
(report on $X)
```

```
}
```

[\[Copy to clipboard\]](#)

Tail-call optimization is backend-dependent. This means that infinite loops must be implemented using `(repeat forever)`. If they are implemented using recursion, some memory might leak with every iteration, and the program will eventually crash.

Asymptotic complexity of per-object flags

Per-object flags are implemented in one of two ways. Most flags are stored in some array-like structure indexed by object number, which means that checking or updating the flag is a constant-time operation. Some flags are also tracked by a separate data structure on the side: A single-linked list of all objects for which the flag is currently set.

The linked list is enabled for any flag predicate that the program might query with an unbound parameter, as this allows Dialog to loop efficiently over every object for which the flag is set. Such per-object flags can also be checked or set in constant time, but **clearing them is linear** in the number of objects having the flag set, because it is necessary to traverse the list at runtime in order to unlink the object in question.

The linked list is also enabled for flags that the program might clear for every object, e.g. `(now) ~($ is marked)`. That operation is always linear in the number of objects that have the flag set.

Limitations and the future of Dialog

The support for numerical computation in Dialog is quite minimal at the moment. A future version of Dialog might include more built-in predicates, and an extended numerical range. However, this must be balanced against the stated design goal of a small, elegant language.

There is currently no way to provide a fallback if the interpreter fails to print a particular unicode character, although the Z-machine backend has built-in fallbacks for certain common characters. Unsupported characters come out as question marks, which is decidedly ugly. A future version of the language may introduce functionality for dealing with unsupported characters.

The (uppercase) built-in predicate is currently limited to English letters (A-Z) for the Z-machine backend. It should at least be upgraded to support the so called default extra characters in the ZSCII character set, which cover the needs of several European languages. The Å-machine backend and the debugger have full support.

Onwards to “[Appendix: Quick reference](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Introduction

([Overview](#) • [Structure of the manual](#) • [Acknowledgements](#))

Overview

Dialog is a domain-specific language for creating interactive fiction. It is heavily inspired by Inform 7 (Graham Nelson et al. 2006) and Prolog (Alain Colmerauer et al. 1972), and substantially different from both.

An optimizing compiler translates high-level Dialog code into Z-code (versions 8 and 5), a platform-independent runtime format supported by a wide range of interpreters and tools. A custom virtual machine called the Å-machine is also supported.

Dialog has been designed with three guiding principles in mind:

Lucidity: Story authors must be able to read, understand, and modify the standard library. The same programming language constructs should be used to express everything from world modelling and narrative structure to parsing and low-level utility functions.

Minimalism: The language should be elegant and small, and rest on a few powerful concepts that work well together. Being a high-level language, it should provide an abstraction of the underlying platform, and the details of the Z-machine should not shine through except where strictly necessary for performance reasons.

Performance: The interactive fiction community settled on the Z-machine as a common platform partly for nostalgic reasons. Yet the majority of recent Z-code games are unplayable on vintage hardware. The new language must be designed from the ground up with efficient object code in mind, both in terms of execution speed and memory footprint.

At its core, Dialog is a general-purpose, rule-based language in the *logic programming* family of languages. Most of the functionality pertaining to interactive storytelling, such as the world model and the parser, are implemented on top of this general-purpose language, in the form of a standard library. In other words, the library is implemented in terms of the same high-level constructs that are used to craft the story. Hopefully, this makes the standard library transparent to story authors, who are encouraged to extend and adapt the library to each particular story, as they see fit.

In addition to adhering to the principles listed above, Dialog features:

- A **concise, formal syntax**. Apart from a few built-in keywords and function names, Dialog source code is not confined to any particular natural language.
- A **comprehensive standard library** providing (among other features) an extensible parser that's capable of supporting arbitrarily complex actions. The standard library is currently limited to English stories, but it could be translated and adapted to other languages.
- A compiler that uses **advanced optimization** techniques to convert high-level rule definitions into compact, efficient Z-code that responds quickly to player input, even on resource-constrained platforms—including many vintage systems. Code compiled for the Å-machine can also take advantage of certain features of modern web browsers, such as CSS style declarations.
- A **fully open-source** implementation, provided under a 2-clause BSD license. Permission is granted to use, modify, and redistribute the compiler and standard library (with or without source code), in whole or in part, as long as attribution is given. Story authors are kindly asked, but not formally required, to include a traditionally-formatted banner in their works.

The name Dialog is suggestive of interactive storytelling, but it is also a nod to Prolog, which is a portmanteau of *programmation en logique* (programming in logic). That's why the name of the language appears to follow a North American spelling convention, even though the rest of this manual does not.

The official homepage of Dialog is: <https://linusakesson.net/dialog/>

Structure of the manual

The bulk of this manual is divided into two parts: **Part I** describes the Dialog programming language, starting with the basics and working its way through every language feature. This also serves as the official language specification. **Part II** describes the standard library, and starts with a hands-on tutorial, where a small game is built from scratch.

Readers who prefer a coherent step-by-step exposition, where every concept is explained before it is used, are encouraged to start with **Part I** and read all chapters in order.

Readers who are itching to get started should begin with **Part II**, play with the various examples, and optionally follow the hyperlinks back to **Part I** in order to dig deeper into the underlying mechanisms.

Acknowledgements

Dialog has been my on-and-off spare time project for many years now, and I'm very happy to have reached a point where it can be released to the public. Working on Dialog has been an extraordinary journey for me personally, but I wouldn't have gotten very far without the maps and lanterns strewn about by the brave adventurers who navigated

these twisty passages before me.

The *Z-machine Standards Document* has been an invaluable source of information and insight. I stand in awe of the dozens of people who reverse-engineered and documented the Z-machine back in the 1990s, and described their findings in such a well-written and concise reference work. Thank you for that **David Beazley, Kevin Bracey, Paul David Doherty, David Fillmore, Russell Hoare, Mark Howell, George Janczuk, Stefan Jokisch, Marnix Klooster, Mark Knibbs, Peter Lisle, Graham Nelson, Jason C. Penney, Matthias Pfaller, Andrew Plotkin, Matthew Russotto, Chris Tham, Mike Threepoint**, and **Dannii Willis**.

Writing and debugging a compiler involves hours of staring at disassembled code, and I thank **Mark Howell** and **Matthew Russotto** for creating and maintaining the ztools package, in particular infodump and txd.

I've also used and abused the highly hackable Z-code interpreter dumbfrotz, part of the frotz family, for which **Stefan Jokisch, Galen Hazelwood, David Griffith**, and **Alembic Petrofsky** deserve credit.

Last but not least, I extend my deepest gratitude to **Graham Nelson** and (where applicable) **Emily Short** for developing and releasing Inform 6 and 7, for writing the *Inform Designer's Manual* and *Writing with Inform*, and for distilling so many years of community experience into the default Inform world model and Standard Rules. These treasure troves are overflowing with so many good ideas that it would have been irresponsible not to steal a substantial part of them, and incorporate them into my own standard library. Any shortcomings in the Dialog world model are of course my own fault.

Lund, November 2018

Linus Åkesson

Onwards to “[Software](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 1: Getting started

([Architecture of the library](#) • [Running the code examples](#) • [A minimal story](#))

The Dialog standard library is a framework for creating interactive stories in English, coded in the Dialog programming language. The library provides the basic implementation layer of an interactive game world—classification of objects, standard actions, parser—and encodes the knowledge required to deal with a multitude of special cases and gotchas.

At roughly 6000 lines of high-level code, the standard library is intended to be substantial yet accessible. Both casual readers and determined explorers are invited to dive into its inner workings, and see how everything fits together.

The world provided by the library is deliberately bland, serving as a neutral backdrop for the stories that you will write. Dialog makes it easy to sketch the outlines of an interactive story or old-school text adventure by merely defining a couple of objects and extending the action-handling rules of the standard library. For more ambitious works, you are encouraged to read and modify the library code, adapting and reshaping the functionality, and wrapping it tightly around the characteristics of your particular story.

Architecture of the library

The chart below illustrates the overall architecture of the standard library, and the ways in which it interfaces with story code. In the chart, blocks generally only make queries to blocks that are directly below them; that's a simplification, but it will do for now.



Running the code examples

This part of the manual contains numerous examples, and readers are encouraged to try them out. Copy the source code, put it in a text file, compile it together with the standard library, and run the resulting story file using your favourite Z-code interpreter. A quick reminder:

```
dialogc -t z8 story.dg stdlib.dg
```

where story.dg is your source code file, and stdlib.dg is the standard library source code file. The order of the filenames is significant; the story must appear before the library. In the above example, the output filename will be story.z8 (based on the output format and the name of the first source code file), but this can be changed with the -o option.

You are encouraged to put the command in a makefile or batch file, to save typing.

As you compile the example programs, you will get warnings about the lack of story title, author, and IFID. Since the example mini-stories aren't meant to be published, you may ignore these warnings. Before you release a work of IF written in Dialog, please add the required [story metadata](#), including a freshly-generated IFID.

A minimal story

The following is a perfectly functional, albeit simple, Dialog game:

```
(current player
#player)
```

```
(current player #player)
(#player is #in #room)
(room #room)
```

[\[Copy to clipboard\]](#)

This game has two objects (#player and #room) in addition to a handful of default objects provided by the standard library (such as #in).

The first line says that when the game starts, the player character is represented by the #player object.

The second line says that the #player object is initially located inside another object, called #room.

The third line says that the object called #room is, in fact, a room: This tells the library that the object can be entered, that it may have connections to other rooms, and so on.

You should try this game now! There's not much of a plot, but you can poke around at the two objects (they can be referred to as ME and HERE), and try a few standard verbs such as LOOK (L), INVENTORY (I), JUMP, and QUIT (Q).

Onwards to [“Chapter 2: Objects and state”](#) • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 4: Items

([Pristine and handled objects](#) • [Plural forms](#) • [All about appearances](#) • [Pristineness of nested objects](#) • [Clothing](#))

Items are objects that can be picked up by the player:

```
#bowl
(name *)      small
```

```
#bowl
(name *) small bowl
(descr *) It's a small bowl.
(item *)  %% This is what allows the player to pick up the bowl.
(* is #on #table)
```

[\[Copy to clipboard\]](#)

Pristine and handled objects

As a general rule, the standard library doesn't call attention to game objects. The story author is expected to mention them as part of the description of their parent object (such as the room).

But there is an important exception: As soon as the player picks up an object (or otherwise moves it from its original location), the responsibility for mentioning that object is transferred to the library. We say that the object has become *handled* or, equivalently, that it is no longer *pristine*.

There's a compelling pragmatic reason for this: When players are able to move objects around, those objects will eventually end up in strange locations, unanticipated by the story author. Players will put the flower pot on top of the bed, or take off their clothes and put them inside the ashtray, and the story author cannot be expected to provide custom text for every combination the player can think of. So, once objects start moving around, the library takes over the initiative in calling attention to them, using bland, default messages. The story author may then choose to override those messages on a case-by-case basis, and we will soon see how that's done in practice.

For now, the important thing to remember is that items (and wearable objects) can move around, and therefore we should only call them out in room descriptions—and in the descriptions of containers and supporters—when they are in their pristine state. Whenever we include a movable object in our story, we have a responsibility to check this. It can be done with an if-statement as in the following example:

```
#bowl
(name *)      small
```

```
#bowl
(name *) small bowl
(descr *) It's a small bowl.
(item *)
(* is #on #table)
(descr #table)
    It's wooden; possibly mahogany.
    (if) (#bowl is pristine) (then)
        A small bowl is placed exactly in its centre.
    (endif)
```

[\[Copy to clipboard\]](#)

Non-movable objects will remain pristine forever, so they can be described without a check.

Here is a complete example game with movable objects:

```
(current player
#player)
```

```
(current player #player)

#room
(name *) tutorial room
(room *)
(#player is #in *)
(look *)  This is a very nondescript room, dominated by a
          wooden table. (notice #table)

#table
(name *) wooden table
```

```
(dict *) mahogany %% Add a synonym.
(supporter *)
(* is #in #room)
(descr *) It's wooden; possibly mahogany.
    (if) (#bowl is pristine) (then)
        A small bowl is placed exactly in its centre.
    (endif)
```

```
#sapphire
(name *) sapphire
(stone *) %% This is a custom, story-specific trait.
```

```
#amethyst
(an *) %% The indefinite article should be 'an'.
(name *) amethyst
(stone *)
```

```
#bowl
(name *) small bowl
(item *)
(container *)
(* is #on #table)
(descr *) It's a small bowl.
```

%% Some generic properties of stones:

```
(item (stone $))
(*(stone $) is #in #bowl)
(descr (stone $Obj))
    (The $Obj) looks very pretty.
(dict (stone $))
    precious stone gem
(plural dict (stone $))
    stones
```

[\[Copy to clipboard\]](#)

Try it! You might want to LOOK, X TABLE, LOOK IN BOWL, GET ALL, PUT STONE IN BOWL, PUT STONE ON TABLE, DROP ALL...

Plural forms

(dict \$) and (plural dict \$) can be used to add synonyms to objects. In the example above, we added both singular and plural synonyms to all objects belonging to the (stone \$) category. A command such as GET STONES will result in every stone being picked up, due to the plural form. In contrast, GET STONE triggers a disambiguating question, where the game asks the player whether they meant to pick up the amethyst or the sapphire.

Note that (dict \$) may contain adjectives, but (plural dict \$) should only contain nouns (in plural form).

All about appearances

You may have noticed a problem with the last example: When the player examines the bowl, there is no mention of the stones within. For an oldschool game, it may be acceptable to expect the player to SEARCH or LOOK IN the bowl in order to find them. But for modern, narrative-driven games, that approach is generally frowned upon. We could mention the stones in the description of the bowl. But there are two stones, so how do we do that? Do we check whether they are both pristine? I.e.:

(descr #bowl)	
It's a small	

```
(descr #bowl)
It's a small bowl.
(if)
    (#sapphire is pristine)
    (#amethyst is pristine)
(then)
    There are two precious stones in it.
(endif)
```

[\[Copy to clipboard\]](#)

But what if the player only picks up the amethyst, and then puts it back? The sapphire is still in the bowl, but the story

doesn't mention it, and the library only prints a stock message about the amethyst (because it is no longer pristine). Another option is to add lots of special cases:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
  It's a small bowl.
(if)
(#sapphire is pristine)
(#amethyst is pristine)
(then)
  There are two precious stones in it.
(elseif)
*($Stone is one of [#sapphire #amethyst])
($Stone is pristine)
(then)
  There's (a $Stone) in it.
(endif)
```

[\[Copy to clipboard\]](#)

But this doesn't scale well, if there were more than two precious stones in the bowl to begin with. We also have the option to cop out entirely, and tell the library to narrate these objects already from the start:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
  It's a small bowl.
(#sapphire is handled)
(#amethyst is handled)
```

[\[Copy to clipboard\]](#)

Remember, handled is the opposite of pristine in this context. Now, when the player first examines the bowl, the game responds:

It's a small bowl.

An amethyst is in the small bowl.

A sapphire is in the small bowl.

But that's decidedly clunky. A somewhat better approach, although still a cop-out, is to print a vague message that encourages the player to look inside the bowl, without mentioning any details about what's inside:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
  It's a small bowl.
(if) ($ is #in #bowl) (then)
  There appears to be something inside.
(endif)
```

[\[Copy to clipboard\]](#)

But this will backfire, in a sense, if the player takes the amethyst and then puts it back. Examining the bowl would then result in the following output:

It's a small bowl. There appears to be something inside.

An amethyst is in the small bowl.

Here, the library called attention to the amethyst (handled), but not to the sapphire (pristine). The printed text is technically correct, but while the first paragraph encourages the player to look inside the bowl, the second paragraph takes that incentive away, and the player is mislead to believe that there's nothing in the bowl apart from the amethyst.

A better way to handle this situation is to selectively override the *appearance* message that's printed for handled objects by the library. The text “An amethyst is in the small bowl” originates from a predicate called (appearance \$Object \$Relation \$Parent). The first step is to tell the library to refrain from printing such a message about any object that's currently in the bowl:


```
~(appearance $ #in
#bowl) %% Objects
```

~(appearance \$ #in #bowl) %% Objects in the bowl have no appearance.

[\[Copy to clipboard\]](#)

Now that we have silenced those particular messages from the standard library, we must provide our own variant in the description of the bowl. But we have to be careful: With the rule above, we turned off automatic descriptions for any object in the bowl, not just the amethyst and the sapphire. So we have to take care of any foreign objects that might end up there too. In some situations, it might be sufficient to drop a vague hint:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
```

```
It's a small bowl.
```

```
(if) ($ is #in #bowl) (then)
```

```
There appears to be something inside.
```

```
(endif)
```

```
~(appearance $ #in #bowl)
```

[\[Copy to clipboard\]](#)

The library provides a predicate, (list objects \$Rel \$Obj), that prints a neutral sentence along the lines of “In the small bowl are an amethyst and a sapphire”, or nothing at all if there is no object in the specified location. Thus:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
```

```
It's a small bowl.
```

```
(par)
```

```
(list objects #in *)
```

```
~(appearance $ #in #bowl)
```

[\[Copy to clipboard\]](#)

A more advanced technique is to use a [multi-query](#) and a [collect statement](#) to print a list of all objects currently inside the bowl:

```
(descr #bowl)
  It's a small
```

```
(descr #bowl)
```

```
It's a small bowl.
```

```
(collect $Obj)
```

```
*($Obj is #in #bowl)
```

```
(into $List)
```

```
You can see (a $List) in it.
```

```
~(appearance $ #in #bowl)
```

[\[Copy to clipboard\]](#)

An entirely different approach is to allow objects to call attention to themselves, but to replace the stock message with a custom one. This is done by overriding (appearance \$ \$ \$) with a rule that prints text:

```
#sapphire
(appearance * $ $)
```

```
#sapphire
```

```
(appearance * $ $)
```

```
(* is handled)
```

```
A gleaming sapphire catches your eye.
```

[\[Copy to clipboard\]](#)

That rule did not check the location of the sapphire, so it would override the default message also when the sapphire makes an appearance as part of a room description, or in any other place. Without the line (* is handled), the message would also be printed while the object is still pristine.

When the last two parameters are wildcards (as above), they can be omitted:

```
#sapphire
(appearance *)
```

```
#sapphire
(appearance *)
(* is handled)
A gleaming sapphire catches your eye.
```

[\[Copy to clipboard\]](#)

In story code, (appearance \$ \$ \$) rules always take precedence over (appearance \$) rules. The way this works internally, is that the library queries (appearance \$ \$ \$), but there is only a single rule definition for that predicate in the library: It queries (appearance \$), which in turn contains the default code for calling attention to handled objects.

There's one more subtlety to be aware of: Whenever an appearance-rule succeeds, the object in question gets *noticed* by the library. This binds the appropriate pronoun (usually “it”) to the object. Therefore, if the appearance-rule doesn't print a sentence about the object, it should *fail* in order to prevent the noticing. That is why there is a tilde character in front of the rule head in some of the examples above.

Pristineness of nested objects

By definition, objects are pristine until they are moved from their initial location. That initial location could be e.g. a portable container or the player character. Be aware that if the containing object is moved from its initial location, its contents nevertheless remain pristine.

So, for instance, if the player starts out with a wallet containing a receipt, then both the wallet and the receipt are initially pristine, even though they are part of the player's inventory. If the player drops the wallet, it becomes handled, but the receipt inside remains in its original location—the wallet—and is still considered pristine.

It is therefore the story author who should call attention to the receipt, as part of the description of the wallet, until the receipt is no longer pristine:

```
(descr #wallet)
  Imitation
```

```
(descr #wallet)
Imitation leather. Jammed zipper.
(if) (#receipt is pristine) (then)
One pitiful receipt inside.
(endif)
```

[\[Copy to clipboard\]](#)

To summarize, movable items are more complicated than other objects, because there is a transfer of responsibility for calling attention to them. At first, while they are pristine, the story author should mention them as a natural part of the prose describing nearby objects (e.g. the room). As soon as they are handled, the library takes over, unless the story author explicitly reclaims control over their appearance.

Clothing

Objects—typically animate ones—can wear clothes. Clothes are objects that have the (wearable \$) trait, and therefore the (item \$) trait by inheritance.

The *outermost* layer of clothing is modelled by the #wornby relation:

```
(#trenchcoat is
#wornby #bob)
```

```
(#trenchcoat is #wornby #bob)
(#shoes is #wornby #bob)
```

[\[Copy to clipboard\]](#)

Clothes may also be worn #under other garments:

```
(#shirt is #under
#trenchcoat)
```

```
(#shirt is #under #trenchcoat)
(#pants is #under #trenchcoat)
(#socks is #under #shoes)
```

[\[Copy to clipboard\]](#)

Use (\$ is worn by \$) to check whether an object is currently worn by somebody, at any level of nesting.

By default, clothes are see-through, so Bob's socks are visible despite being located #under his shoes. The outer garment can be made opaque to prevent this:

```
(opaque #trenchcoat)
```

```
(opaque #trenchcoat)
```

[\[Copy to clipboard\]](#)

Should the player try to remove a piece of clothing that's underneath another, an attempt is made to remove the outer item first. If this fails, the entire action is stopped.

It's possible to indicate that some garments can't be worn together with others. This is done by adding rules to the (wearing \$ removes \$) predicate:

```
(wearing #glasses  
removes
```

```
(wearing #glasses removes #sunglasses)
```

```
(wearing #sunglasses removes #glasses)
```

[\[Copy to clipboard\]](#)

This might lead to the following exchange:

> wear sunglasses

(first attempting to remove the glasses)

You take off the glasses.

You put on the sunglasses.

For a larger number of mutually exclusive items, it is more convenient to define a trait:

```
(glasses #glasses)  
(glasses
```

```
(glasses #glasses)
```

```
(glasses #sunglasses)
```

```
(glasses #monocle)
```

```
(wearing (glasses $) removes (glasses $))
```

[\[Copy to clipboard\]](#)

Other articles of clothing would typically be worn *over* others; this is indicated with the (wearing \$ covers \$) predicate:

```
(wearing  
#trenchcoat covers
```

```
(wearing #trenchcoat covers #shirt/#pants)
```

```
(wearing #shoes covers #socks)
```

[\[Copy to clipboard\]](#)

In the above example, if the player tries to wear the shoes while already wearing the socks, the socks will end up *#under* the shoes. Later, if the player tries to remove the socks, an attempt is first made to remove the shoes. But we didn't say anything about putting on socks while wearing shoes, so this is allowed. To properly model the socks-shoes relationship, we would also have to define:

```
(wearing #socks  
removes #shoes)
```

```
(wearing #socks removes #shoes)
```

[\[Copy to clipboard\]](#)

But this combination of constraints—(wearing \$A covers \$B) and (wearing \$B removes \$A)—is so common that the library gives us the option to specify both relations in one go:

```
(#socks goes  
underneath #shoes)
```

```
(#socks goes underneath #shoes)
```

[\[Copy to clipboard\]](#)

Actually, (\$ goes underneath \$) does more: It is treated as a transitive relation, meaning that if the shirt goes underneath the jacket and the jacket goes underneath the trenchcoat, then the library can figure out that the shirt must go underneath the trenchcoat. Thus the trenchcoat would automatically cover the shirt, and putting on the shirt would involve removing the trenchcoat first.

But for this to work, the library must be able to invoke (\$ goes underneath \$) in a [multi-query](#), with the second parameter unbound. Therefore, be aware that if the second parameter is a trait, it needs to be prefixed by an asterisk:

(#underpants goes underneath *(pants

(#underpants goes underneath *(pants \$))
((pants \$) goes underneath #trenchcoat)

[\[Copy to clipboard\]](#)

Advanced technique: Multiple covers

So far in this chapter, we've tacitly assumed that an article of clothing can only ever be worn underneath a single parent. This is inherent in the object-tree model, but it rules out situations such as a spandex one-piece worn underneath a shirt and a pair of trousers at the same time. The library doesn't support such a use case directly, but it can be implemented with the help of [before- and after-rules](#):

#onepiece

#onepiece
(* goes underneath #shirt/#trousers)
(before [remove *])
(* is worn by #player)
*(\$Outer is one of [#shirt #trousers])
(\$Outer is worn by #player)
(first try [remove \$Outer])
(after [remove #shirt/#trousers])
(* is worn by #player)
*(\$Outer is one of [#shirt #trousers])
(\$Outer is worn by #player)
(now) (* is #under \$Outer)

[\[Copy to clipboard\]](#)

Onwards to “[Chapter 5: Moving around](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)

Chapter 8: Ticks, scenes, and progress

([Timed code](#) • [Cutscenes](#) • [The intro](#) • [Keeping score](#) • [The status bar](#) • [Game over](#) • [Choice mode](#))

Timed code

As we've seen earlier ([Stopping and ticking](#)), the standard library measures time in *ticks*. One tick corresponds to one action. The library makes a [multi-query](#) to the predicate (on every tick) on every tick. By default, this predicate contains a rule that in turn makes a multi-query to (on every tick in \$), with the current room as parameter.

To add flavour text to a location, you can combine this mechanism with a [select statement](#):

```
(on every tick in
#library)
```

```
(on every tick in #library)
```

```
(select)
```

```
Somebody tries to hold back a sneeze.
```

```
(or)
```

```
You hear the rustle of pages turned.
```

```
(or)
```

```
The librarian gives you a stern look.
```

```
(or)
```

```
(or)  %% Don't print on every single turn.
```

```
(or)
```

```
(or)
```

```
(at random)
```

[\[Copy to clipboard\]](#)

For fine-grained control, you can use a [global variable](#) to implement *timed events*:

```
(global variable
dragon's anger)
```

```
(global variable (dragon's anger level 0))
```

```
(narrate entering #lair)
```

```
(now) (dragon's anger level 0)
```

```
(fail)  %% Proceed with the default '(narrate entering $)' rule.
```

```
(on every tick in #lair)
```

```
(#dragon is #in #lair)  %% Only proceed if the dragon is here.
```

```
(dragon's anger level $Anger)
```

```
(narrate dragon's anger $Anger)
```

```
($Anger plus 1 into $NewAnger)
```

```
(now) (dragon's anger level $NewAnger)
```

```
(narrate dragon's anger 0)
```

```
The dragon gives you a skeptical look.
```

```
(narrate dragon's anger 1)
```

```
The dragon huffs and puffs.
```

```
(narrate dragon's anger 2)
```

```
The dragon looks at you with narrowed eyes.
```

```
(narrate dragon's anger 3)
```

```
The dragon roars! You'd better get out.
```

```
(narrate dragon's anger 4)
```

```
The dragon almost hits you with a burst of flame. You flee.
```

```
(enter #outsideLair)
```

[\[Copy to clipboard\]](#)

To model a scene that plays out in the background for several moves, use a global flag and a tick handler:

```
(perform [read
#notice])
```

```
(perform [read #notice])
```

```
Auction! Today! In the marketplace!
```

```
(now) (auction scene is running)
```

```
(on every tick in #marketplace)
(auction scene is running)
"Who can give me five dollars for this lovely
(select) spatula (or) glass bead (or) stuffed wombat (at random)?"
shouts the auctioneer at the top of his lungs.

(perform [wave])
(current room #marketplace)
(auction scene is running)
Just as you are about to place a bid, an unexpected thunderstorm
emerges from a gaping plot hole. "Auction suspended", says the
auctioneer.

(now) ~(auction scene is running)
```

[\[Copy to clipboard\]](#)

Cutscenes

In their simplest form, cutscenes are just large blocks of text and perhaps a couple of modifications to the object tree. As such, they can appear anywhere in the program. If a cutscene is triggered by an action handler or a tick callback, it is customary to end the scene with (stop) or (tick) (stop), to inhibit any subsequent actions mentioned in the player's input. For instance:

```
(perform [pull
#handle])

(perform [pull #handle])
You grab the handle of the machine, and hesitate for a moment. Is
this really safe?

(par)
But you have no choice. You pull the handle. Sparks hum in the air
as you are sucked into the vortex of the machine.

(par)
You find yourself in a barn.

(move player to #on #haystack)
(try [look])
(tick) (stop)
```

[\[Copy to clipboard\]](#)

If the cutscene can be triggered in multiple ways, put it [in a separate predicate](#) and query that from as many places in the code as you wish.

To prevent a cutscene from occurring twice, use a [global flag](#):

```
(perform [pull
#handle])

(perform [pull #handle])
(if) (have teleported to barn) (then)
Nothing happens when you pull the handle.
(else)
(teleport to barn cutscene)
(endif)

(teleport to barn cutscene)
...
(now) (have teleported to barn)

(stop)
```

[\[Copy to clipboard\]](#)

The intro

When a story begins, the standard library queries the (intro) predicate, which story authors are encouraged to override.

The default implementation of (intro) just prints the *story banner* by querying (banner). The banner includes version information for the compiler and the standard library. By convention, stories should print the banner at some point during play. With Dialog, there is no formal requirement to print the banner at all, but it is helpful to the community and to your future self, and it looks professional.

```
(intro)
  "In medias
```

```
(intro)
  "In medias res?" exclaimed the broad-shouldered Baron furiously.
  "We find it preposterously cliché!"
```

```
(banner)
```

```
(try [look])
```

[\[Copy to clipboard\]](#)

The (banner) predicate calls out to (additional banner text), which is empty by default but can be overridden to include e.g. a subtitle, co-credit, or dedication.

Keeping score

The player's progress can be tracked by a global score variable. This feature needs to be enabled, by including the following rule definition somewhere in the story:

```
(scoring enabled)
```

```
(scoring enabled)
```

[\[Copy to clipboard\]](#)

For scored games, the current score is displayed in the status bar.

The global variable is called (current score \$).

Points can be added to the score with (increase score by \$), and subtracted with (decrease score by \$). These predicates fail if the score would end up outside the valid range of integers in Dialog, which is 0-16383 inclusive.

After every move, the standard library will mention if the score has gone up or down, and by how much, unless the player has disabled this feature using NOTIFY OFF.

If you know what the maximum score is, you can declare it:

```
(maximum score 30)
```

```
(maximum score 30)
```

[\[Copy to clipboard\]](#)

When declared, the maximum score is mentioned by the default implementation of the SCORE command, in the status bar, as well as by the (game over \$) predicate. It does not affect the operation of (increase score by \$).

The status bar

It is straightforward to supply your own, custom status bar. Define a rule for the predicate (redraw status bar), and make use of [the status area functionality](#) built into the Dialog programming language.

The standard library defines (status headline), which can be used to print the location of the player character in the usual way. That would normally be the current room header, followed by something like “(on the chair)” if the player character is the child of a non-room object. But if the player character is in a dark location, control is instead passed to (darkness headline), which usually prints “In the dark”.

```
%% A thicker status
bar with the name of
```

```
%% A thicker status bar with the name of the current player in the upper right corner.
```

```
(style class @status)
```

```
  height: 3em;
```

```
(style class @playername)
```

```
  float: right;
```

```
  width: 20ch;
```

```
  margin-top: 1em;
```

```
(style class @roomname)
```

```
  margin-top: 1em;
```

```
(redraw status bar)
```

```
(status bar @status) {
```

```
(div @playername) {
```

```
(current player $Player)
(name $Player)
}
(div @roomname) {
(space 1) (status headline)
}
}
```

[\[Copy to clipboard\]](#)

Game over

The library provides a predicate called (game over \$). Its parameter is a [closure](#) containing a final message, which the library will print in bold text, enclosed by asterisks. Then it will:

- Invoke (game over status bar) which sets the status bar to “Game over”, unless you override it.
- Report the final score (if scoring is enabled), and the maximum score (if one has been declared).
- Enter an infinite loop where the player is asked if they wish to RESTART, RESTORE, UNDO the last move, or QUIT.

Here is an example of a (very small) cutscene that ends the game:

```
(perform [eat
#apple])
```

```
(perform [eat #apple])
```

The apple is yummy. You feel that your mission has come to an end.

```
(game over { You are no longer hungry. })
```

[\[Copy to clipboard\]](#)

A fifth option can be added to the game-over menu: AMUSING. First, add the option to the menu with the following rule definition:

```
(amusing enabled)
```

```
(amusing enabled)
```

[\[Copy to clipboard\]](#)

Then, implement a predicate called (amusing) that prints a list of amusing things the player might want to try:

```
(amusing)
(par)
```

```
(amusing)
```

```
(par)
```

Have you tried...

```
(par)
```

```
(space 10) ...eating the transmogrifier? (line)
```

```
(space 10) ...xyzzzy? (line)
```

[\[Copy to clipboard\]](#)

Custom options can be added to the menu by defining rules for (game over option). A multi-query will be made to this predicate, and the output is supposed to end with a comma. For instance:

```
(game over option)
read a NOTE by
```

```
(game over option)
```

```
read a NOTE by the author,
```

[\[Copy to clipboard\]](#)

And here is how to specify what happens when the user types the given word:

```
(parse game over
[note])
```

```
(parse game over [note])
```

```
(par)
```

Thanks for playing!

```
(line)
```

```
-- (space) The Author
```


(par)

[\[Copy to clipboard\]](#)

Choice mode

As a complement to the parser, the Dialog standard library offers a *choice mode*, where the player navigates a set of *nodes* (text passages) by choosing from explicit lists of options. Choice mode can be used for interactive cutscenes, conversations, mini-games, or even as the primary mode of interaction of a game. The author is free to switch between parser-based and choice-based interaction at any time, as behoves the story.

Nodes are represented by ordinary Dialog objects. In the simplest mode of operation, each node has a label and some display-text, and offers a set of links to other nodes:



(intro)	
(activate node	

(intro) (activate node #start)

#start

(disp *) You extend your wings. A warm, buzzy feeling spreads through your body
as you leave the hive.

(* offers #rosebush)

(* offers #poppies)

#rosebush

(label *) Follow a scent of roses.

(disp *) You hover for a while near the pink rosebush.

(* offers #poppies)

#poppies

(label *) Follow a scent of poppies.

(disp *) You circle a patch of poppies by the pondside.

(* offers #rosebush/#pond)

#pond

(label *) Approach the pond.

(disp *) You flutter across the water, enjoying the sweet bouquet of water lilies.

(* offers #poppies)

#land

(label *) Return back home.

(disp *) After an impeccable landing, you find yourself back at the hive.

(game over { A day well spent! })

(#rosebush/#poppies/#pond offers *)

[\[Copy to clipboard\]](#)

To select choice mode—or remain in choice mode but force a transition to a different node—make a query to (activate node \$) with the desired node object as parameter. To select parser mode, make a query to (activate parser). Be aware that both of these predicates invoke (stop), thereby effecting an immediate return to the main loop.

In the main loop, if choice mode is on, the library determines what nodes are reachable from the currently active node, and prints a numbered list of their labels. If the player types one of the numbers (or clicks one of the labels, if library links are enabled and the interpreter supports them) then the corresponding node is activated. Otherwise, the input is parsed in the usual way. By default, all in-world actions are disabled in choice mode; only commands (e.g. UNDO, SAVE) work. Here is an example session:

You extend your wings. A warm, buzzy feeling spreads through your body as you leave the hive.

1. Follow a scent of roses.
2. Follow a scent of poppies.
> 2
Follow a scent of poppies.

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
2. Approach the pond.
3. Return back home.
> undo
Undoing the last turn.

1. Follow a scent of roses.

2. Follow a scent of poppies.
> look
(That action is currently disabled.)

1. Follow a scent of roses.
2. Follow a scent of poppies.
>

Exposed and unexposed nodes

When a node has been activated at least once, it is considered *exposed*, and the flag (\$ is exposed) is set. This is handy for putting conditions on the links between nodes:

...

...
#land
(label *) Return back home.
(disp *) After an impeccable landing, you find yourself back at the hive.
 (game over { A day well spent! })
(#rosebush/#poppies/#pond offers *)
 (#rosebush is exposed)
 (#poppies is exposed)

[Copy to clipboard]

In the above example, the “Return back home” option will only show up after the player has visited both the poppies and the rosebush.

The access predicate (\$ is unexposed) is the negation of (\$ is exposed).

A node can have an *initial label* which is shown instead of the regular label while the node is unexposed:

#poppies
(initial label *)

#poppies
(initial label *) Follow a scent of poppies.
(label *) Return to the patch of poppies.
(disp *) You circle a patch of poppies by the pondside.
(* offers #rosebush/#pond)

[Copy to clipboard]

The default implementation of (initial label \$) simply passes control to (label \$).

Conditional labels

The currently active node is indicated by the global variable (current node \$). In parser mode, (current node \$) is unset. You shouldn't update this variable directly—use (activate node \$) and (activate parser)—but you may query it.

Labels can have conditions. In particular, they can depend on the current node:

(label #poppies)
(current node

(label #poppies) (current node #rosebush)
 Leave the rosebush and follow a scent of poppies.
(label #poppies) (current node #pond)
 Leave the pond and return to the poppies.
(label #poppies) Follow a scent of poppies.

[Copy to clipboard]

Dead ends and sticky nodes

A *dead end* is a node that doesn't offer any further choices. When the current node is a dead end, control flows back to the most recent node by default. In addition, the dead-end node becomes *unavailable*. Unavailable nodes do not show up in option lists, even if they are declared using (\$ offers \$).

#poppies-collect
(#poppies offers *)

#poppies-collect
(#poppies offers *)
(label *) Collect nectar from the poppies.
(disp *) Yum!
[\[Copy to clipboard\]](#)



You extend your wings. A warm, buzzy feeling spreads through your body as you leave the hive.

1. Follow a scent of roses.
 2. Follow a scent of poppies.
- > 2
Follow a scent of poppies.

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
 2. Approach the pond.
 3. Collect nectar from the poppies.
- > 3
Collect nectar from the poppies.

Yum!

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
 2. Approach the pond.
- >

Sometimes you'll want a dead-end node that remains available in choice-listings even after it has been exposed. Just mark the node as *sticky*:

#poppies-collect
(sticky *)

#poppies-collect
(sticky *)

[\[Copy to clipboard\]](#)

Flow and converging paths

It is possible to specify a different target for a dead-end node using (\$ flows to \$):

#poppies-collect
(#poppies offers *)

#poppies-collect
(#poppies offers *)
(label *) Collect nectar from the poppies.
(disp *) Yum!
(par)

A sudden gust of wind throws you in the direction of the pond.

(* flows to #pond)

[\[Copy to clipboard\]](#)



This mechanism can be used to implement *converging paths*, where a single node can be reached in several ways, with different display-text for every path. The common node itself can have a blank display-text, and only serve as an anonymous bag of subsequent choices:



Note: In this example, #poppies-collect and #rosebush-collect are declared sticky. This prevents the game from becoming unwinnable if the player revisits a plant after collecting both kinds of nectar.

(intro)
(activate node

```

(intro)          (activate node #start)

#start
(dispatch *)      You extend your wings. A warm, buzzy feeling spreads through your body
                  as you leave the hive.

(* offers #rosebush)
(* offers #poppies)

#rosebush
(initial label *) Follow a scent of roses.
(label *)         Return to the rosebush.
(dispatch *)      You hover for a while near the pink rosebush.
(* offers #poppies)

#poppies
(initial label *) Follow a scent of poppies.
(label *)         Return to the patch of poppies.
(dispatch *)      You circle a fragrant patch of poppies.
(* offers #rosebush)

#rosebush-collect
(#rosebush offers *)
(sticky *)
(label *)         Collect nectar from the rosebush.
(dispatch *)      Yum! Rose nectar!
(* flows to #collect-done)

#poppies-collect
(#poppies offers *)
(sticky *)
(label *)         Collect nectar from the poppies.
(dispatch *)      Yum! Poppy nectar!
(* flows to #collect-done)

#collect-done
(* offers #rosebush/#poppies/#land)

#land
(label *)         Return back home.
(dispatch *)      After an impeccable landing, you find yourself back at the hive.
                  (game over { A day well spent! })

```

[\[Copy to clipboard\]](#)


The default behaviour of automatically returning to the previous node is actually implemented as a fallback rule for the (\$ flows to \$) predicate.

Only a single level of node history is recorded, so automatic backtracking only works once. If the previous node also offers no choices, then the library dumps the player back into parser mode. If this is undesirable, always provide explicit (\$ flows to \$) links for nodes that might *turn into* dead ends. That is, nodes that offer choices initially, but where all of those choices may eventually go away.

The hub pattern

Another way to organize a choice-based sequence is to have a central hub node, and to selectively offer links based on which peripheral nodes have been exposed so far.

In the following example, the player can collect nectar once from each plant, and needs at least some nectar in order to proceed with the landing:



(intro)

You

extend your wings. A

(intro)

You extend your wings. A warm, buzzy feeling spreads through your body as you leave the hive.

(activate node #hub)

#hub

(* offers #rosebush)

(* offers #poppies)

#rosebush

(sticky *)

```

(initial label *) Follow a scent of roses.
(label *)      Return to the rosebush.
(dispatch *)   You (select) discover a (or) revisit the (stopping) pink rosebush.

#poppies
(sticky *)
(initial label *) Follow a scent of poppies.
(label *)      Return to the patch of poppies.
(dispatch *)   You (select) find a (or) circle the (stopping) fragrant patch of poppies.

#rosebush-collect
(#hub offers *) (#rosebush is exposed)
(label *)      Collect nectar from the rosebush.
(dispatch *)   Yum! Rose nectar!

#poppies-collect
(#hub offers *) (#poppies is exposed)
(label *)      Collect nectar from the poppies.
(dispatch *)   Yum! Poppy nectar!

#land
(#hub offers *) (#rosebush-collect is exposed) (or) (#poppies-collect is exposed)
(label *)      Return back home.
(dispatch *)   After an impeccable landing, you find yourself back at the hive.
               (game over { A day well spent! })


```

[\[Copy to clipboard\]](#)

Choice/parser integration

A *terminating* dead-end node has the side-effect of leaving choice mode. This is handy when integrating choice-based sequences into a larger game:

```



(current player  
#player)

(current player #player)
(#player is #in #beehive)

#beehive
(room *)
(look *) Honeycombs line every wall. The exit is due east.
(instead of [leave * #east])
      (activate node #start)

#start
(dispatch *) You extend your wings. A warm, buzzy feeling spreads through your body
            as you leave the hive.
(* offers #rosebush)
(* offers #poppies)

#rosebush
(label *) Follow a scent of roses.
(dispatch *) You hover for a while near the pink rosebush.

(* offers #poppies/#land)

#poppies
(label *) Follow a scent of poppies.
(dispatch *) You circle a patch of poppies by the pondside.
(* offers #rosebush/#land)

#land
(label *) Return back home.
(dispatch *) After an impeccable landing, you find yourself back at the hive.
(terminating *)

```

[\[Copy to clipboard\]](#)

Terminating nodes are implicitly sticky.

Nothing prevents you from letting arbitrary game objects double as nodes in choice mode. For instance, in the above example we could have used `#beehive` as the starting node, instead of introducing a separate `#start` object. This is

particularly handy when a game contains multiple choice-based sequences that are triggered in a similar way. For instance, conversations with non-player characters could be launched with a generic rule, such as:

```
(perform [talk to  
(animate $NPC)])
```

```
(perform [talk to (animate $NPC)])  
(activate node $NPC)
```

[\[Copy to clipboard\]](#)

When the player types a number during choice mode, a query is made to (choose \$). The default behaviour of this predicate—feel free to override it!—is to print the label of the object followed by a paragraph break, and then make a query to (activate node \$). This predicate, in turn, performs some internal housekeeping, and *displays* the node using a predicate called (display \$):

```
(display $Obj)  
(exhaust) { *
```

```
(display $Obj)  
(exhaust) { *(before disp $Obj) }  
(disp $Obj)  
(exhaust) { *(after disp $Obj) }
```

[\[Copy to clipboard\]](#)

So as a complement to the normal (disp \$) rules, story authors can put header and footer material in (before disp \$) and (after disp \$), respectively.

After querying (display \$), (activate node \$) marks the node as exposed, and invokes (stop). But no query is made to (tick). Hence, by default, no in-game time passes in choice mode.

To change this, just add an (after disp \$) rule with an explicit query to (tick):

```
(after disp $) (tick)
```

```
(after disp $) (tick)
```

[\[Copy to clipboard\]](#)

Sometimes, it is more natural to advance time only at terminating nodes, i.e. just before the game transitions from choice mode to parser mode:

```
(after disp  
(terminating $))
```

```
(after disp (terminating $))  
(tick)
```

[\[Copy to clipboard\]](#)

And here is a variant that also prints the current room description, sending a signal to the player that the game is now in parser mode:

```
(after disp  
(terminating $))
```

```
(after disp (terminating $))  
(par)  
(try [look])  
(tick)
```

[\[Copy to clipboard\]](#)

Hybrid modes

The library also supports parser/choice hybrid modes, where certain actions are available in addition to the numbered choices. For instance, it could be useful to allow SHOW X TO Y from within a choice-based conversation.

By default, all actions are allowed in parser mode, while only commands are allowed in choice mode. Change this by adding rules to (allowed action \$), e.g.:

```
(allowed action [look])
```

```
(allowed action [look])
```

[\[Copy to clipboard\]](#)

As an arbitrary example, you could allow LOOK from a subset of the nodes:

(allowed action
[look])

(allowed action [look])
(current node \$Node)
(\$Node is one of [#rosebush #poppies])

[\[Copy to clipboard\]](#)

Onwards to “[Chapter 9: Non-player characters](#)” • Back to the [Table of Contents](#)

The Dialog Manual, Revision 31, by [Linus Åkesson](#)