CUED Standard Dialogue Acts

Steve Young

June 19, 2009

Abstract

Within all CUED spoken dialogue systems, interactions at the intention level are represented by a core set of dialogue acts. A key feature of the CUED scheme is the provision for representing a distribution of dialogue act hypotheses. To obviate the need for combining multiple acts and the consequent normalisation issues that this would raise, CUED dialogue acts are high level and a single act can encapsulate a variety of intentions. This document describes the syntax and semantics of the core set of CUED dialogue acts.

1 Attributes, values and the application domain

Dialogue acts refer to entities in the application domain. Each entity has a number of attributes and understanding how the attributes of an entity are structured is an essential pre-requisite to understanding the way that dialogue acts are defined.

All current CUED spoken dialogue systems (SDS) are designed to implement information seeking applications. The universe of discussion is defined by a set of simple ontology rules which define a tree structure such that the leaves of the trees are attribute values and the hierarchy defines the relationships between attributes. The root of the tree corresponds to a specific entity under discussion, and the nodes of the tree define the features which characterise that entity. This entity typically represents the user's information-seeking goal and hence it is often referred to as the *user goal tree*.

Figure 1 shows an example (incomplete) ontology for a simple tourist information system.

```
entity     -> venue(type,name,area,addr);
entity     -> landmark(name,area,addr);
type       -> restaurant(food,music,decor);
type       -> hotel(pricerange, stars);
venue.name =  ("Toni's" | "Quick Bite" | ....);
landmark.name =  ("Water Tower" | "Museum" | ....);
food       =  ("Italian" | "Chinese" | "Russian" | ...);
music      =  ("Jazz" | "Pop" | "Folk" | ...);
decor      =  ("Traditional" | "Roman" | "Art Deco" | ...
area       =  ("central" | "east" | "north" | "south" | ...);
addr       =  ("Main Street" | "Market Square" |  ...);
```

Figure 1: Example Ontology Rules

The ontology rules operate like context-free rewrite rules such that the node on the left derives the daughter nodes written as a comma-separated list to the right. All possible expansions of these rules would enumerate all possible entities with distinguishable characteristics. Fig 2 shows an example derivation.

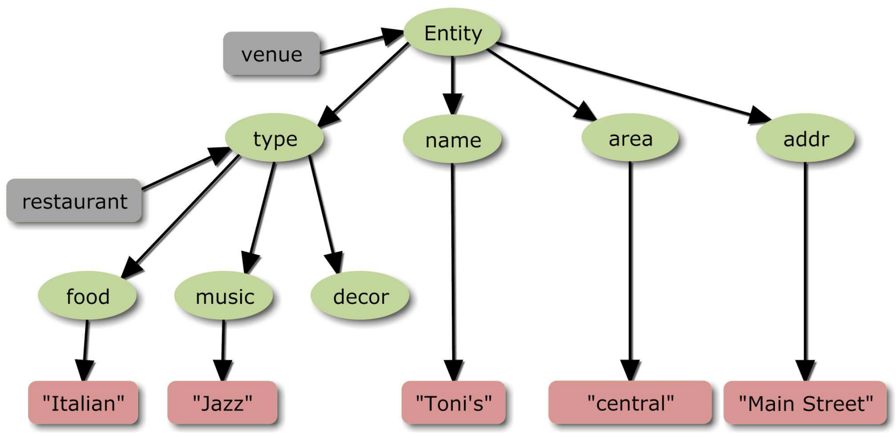

Figure 2: Example Tree for TownInfo Application

However, unlike simple rewrite rules, each node expansion can also be tagged as some specific *subtype*. For example, an `entity` in the above can either be of subtype `venue` or subtype `landmark`. The subtype tag is indicated by a reverse arrow in Fig 2. Syntactically, subtype tags appear as functors on the rhs of the rules with the functor's arguments being the daughter nodes.

The ontology tree structure defines the way that attributes (ie nodes) are referenced in dialogue acts. Each node corresponds to an attribute, and subtypes and atomic leaf nodes are values. Thus, in the example shown in Fig 2, valid nodes are: `entity`, `area`, `addr`, etc and valid values are `"Italian"`, `"restaurant"`, `"Main Street"`.

Since some nodes may re-occur in different contexts, node names can be qualified. In the example rules, `name` is ambiguous, hence it is qualified in the rules as either `landmark.name` or `venue.name`. Furthermore, when used in a qualifying position, node names and their subtypes are interchangeable. Thus, `restaurant.food` and `type.food` are both valid references to the food node.

Node references and their values are used in a dialogue to query and supply information. Information is supplied via attribute-value pairs, e.g. `music="Jazz"`, `type="restaurant"`, `restaurant.food="Italian"`, etc. Queries, however, are constructed with simple node-names since the point of the query is to find a value, e.g. `music`, `decor`, `restaurant.food`, etc.

Having explained how attributes and values are specified, the next section describes the structure of dialogue acts. The remaining sections then describe the dialogue acts themselves.

2 Structure of a Dialogue Act

The full syntax for depicting a set of dialogue acts is shown in Fig. 3 where vertical bars denote alternatives, brackets denote options and curly braces denote zero or more repetitions.

A `name` consists of any alphanumeric sequence starting with a letter, a `string` is an arbitrary character sequence enclosed in quotes and a `float` is any floating point number.[1]

A set of dialogue acts corresponds to one turn of a dialogue. Each member of the set represents one possible hypothesis about the speaker's intention. The probability of each hypothesis is given by the value of `prob`. If `prob` is omitted, then all dialogue acts are deemed to be equally likely. For example, the dialogue set

```
inform(food="Italian"){0.8}, inform(food="Russian"){0.2}
```

[1]In fact, the quotes for strings are only strictly necessary if the value contains a non-alphanumeric character.

```
actset     =  act { "," act }
act        =  acttype "(" [item { "," item }] ")"  ["{" prob "}"]
item       =  bareattr | attrvalue | barevalue ;
bareattr   =  attr
attrvalue  =  attr eq value
barevalue  =  eq value
eq         =  "=" | "!="
attr       =  { qual "." } name
qual       =  name
value      =  string | subtype_name
prob       =  float
```

Figure 3: Syntax of CUED Dialogue Acts

conveys the information that `food="Italian"` with probability 0.8 and `food="Russian"` with probability 0.2.

Each `attribute=value` pair is called an *item* and each item refers to a specific attribute or entity within the application domain. As explained in section 1, in CUED systems, each attribute corresponds to a node in a user goal tree.

In the linguistics literature, dialogue turns are commonly analysed as a combination of several primitive dialogue acts. However, in the CUED standard, every dialogue turn must be represented by a single act to ensure that the probability of alternative hypotheses always sums to one in a simple and consistent manner. To compensate for the inability to combine dialog acts, the CUED standard therefore allows a single dialogue act to contain multiple items. For example, the utterance "I want to eat some Italian food and listen to some Jazz." would be rendered as

```
inform(food="Italian", music="Jazz")
```

This is quite different to

```
inform(food="Italian"), inform(music="Jazz")
```

which would indicate that the speaker intended to either convey the information that `food="Italian"` or that `music="Jazz"` but not both.

Consistent with the conventions for node references described in section 1, an attribute name can be either a simple name or a qualified name. For example, `name` is a simple name whereas `venue.name` is a qualified name. Qualifiers can be concatenated to form a path in the tree and subtype names can be substituted for qualifiers. Hence, for example, `entity.type.music` could also be written as `venue.type.music` or `venue.restaurant.music`. In general qualifiers are used to resolve ambiguities in cases where there are multiple attributes or entities with the same name.

The value assigned to an attribute can be either a subtype name or an atom. In the former case, the information being conveyed is structural. For example, the act `inform(type=restaurant)` indicates that the node `type` is expanded as the subtype `restaurant` whereas `inform(food="Italian")` simply asserts that the value of the lexical node `food` is `"Italian"`. The value `"dontcare"` can be specified for any attribute to specify that the attribute is not important and any value will do. The value part of an item is optional since in some acts, the purpose of the act is to elicit a value. For example, `request(name,type=hotel)` is a request for the name of a hotel. A dialogue act can also include bare values as in `inform(="dontcare")`. In this case, the user has simply said "I dont care", and there is no context from which the associated node can be identified.

3 Semantic Decoding and Ambiguity

For a variety of reasons (e.g. user imprecision, asr errors, ...) there will often be multiple ways of interpreting a user input. For example,

```
<garbage> central <garbage>
```

might be `inform(area=central)` or `confirm(area=central)`. In such cases, the semantic decoder may output multiple interpretations as alternative hypotheses. [2]

Since the CUED standard does not allow arbitrary combinations of primitive dialogue acts, a similar situation will arise when the user issues multiple conflicting utterances. For example, if the user says:

```
What's the price? Is it expensive?
```

then there are two separate translations: `request(price)` and `confirm(price=expensive)`, each demanding a different answer. In cases such as these, a semantic decoder can output either interpretation or output both as alternatives

```
request(price), confirm(price=expensive)
```

The dialogue manager will then see these as alternative interpretations of the input and act accordingly.[3]

Finally, note that semantic decoders should only provide information that is actually in the sentence. For example, consider the following:

```
I want an Italian restaurant   <=> inform(type=restaurant,food="Italian")
I would like some Italian food <=> inform(food="Italian")
```

In the first case, the sentence refers both to a restaurant and Italian food. The second refers only to food with no mention of a restaurant. Thus, although both utterances are superficially similar, they translate to different dialogue acts.

4 Dialogue Act Definitions

This section describes the dialogue acts defined by the CUED standard. For convenience, they are divided into 4 groups: information providing; query; confirmation; and housekeeping.

The full set of dialogue acts is summarised in Appendix A. Note that the standard currently distinguishes between dialogue acts generated by a human user and acts generated automatically by a system. Some acts are specific to each source and others are common. In future, this distinction may be abandoned. Some of the special cases are specific to information tasks and these may require further generalisation in the future.

4.1 Information Providing

Act	System	User	Description
inform(a=x,b=y,...)	√	√	give information a=x, b=y, ...
inform(name=none)	√	×	inform that no suitable entity can be found
inform(a!=x,...)	×	√	inform that a is not equal to x
inform(a=dontcare,...)	×	√	inform that a is a "don't care" value
inform(name=none,a!=x,b=y,...)	√	×	inform that all entities with b=y satisfy a=x
inform(name=none,name!=a,b=y,...)	√	×	inform that a is the only entity with b=y, ...

[2]The recogniser might output its N-best hypotheses, and the semantic decoder might then output M alternatives for each hypothesis giving upto $M \times N$ alternative dialogue acts.

[3]Note that a sensible response to either will probably satisfy the user.

The inform act is used by the speaker to convey one or more items of information. It does not invite any specific response from the hearer. Some examples are:

```
I would like a Italian restaurant. <=>  inform(type=restaurant,food="Italian")
In the centre of town.             <=>  inform(area="central")
```

There are several special cases associated with inform acts. Firstly, the name attribute can be assigned the reserved value none. This indicates that there is no entity in the database whose characteristics match the provided attribute values. In effect, name=none indicates a null database match.

Secondly, the generic value dontcare can be used to specify a ”wildcard” i.e. a value which will match anything. Thirdly, the special form inform(food!="Italian") indicates that the food can be any value except Italian[4]. Some examples of the use of these special cases are

```
I'll eat anything except Russian. <=> inform(food!="Russian")
Any type of music is fine.        <=> inform(music=dontcare)
I dont care.                      <=> inform(=dontcare)
```

The reserved item name=none and negated attributes (e.g, inform(food!="Italian")) can be combined to express the fact that there is no match in the database for entities *without* a specific attribute value. In other words, this can be used to indicate that *all* entities in the database have a specific attribute value.

Analogously, the combination of the reserved item name=none with a negated name attribute (e.g, inform(name=none,name!="Char Sue",food=Chinese)) indicates that the negated entity is the *only* one in the database matching the remaining attributes, e.g. there is no match in the database for a venue that serves Chinese food and that is not Char Sue. Some examples of the use of these special cases are

```
All bars in the centre are expensive. <=> inform(name=none,pricerange!=expensive,area="centre
Botchka is the only restaurant in the south. <=> inform(name=none,name!="Botchka",area="South
```

4.2 Query

Act	System	User	Description
request(a)	√	√	request value of a
request(a,b=x,...)	√	√	request value for a given b=x ...
reqalts()	×	√	request alternative solution
reqalts(a=x,..)	×	√	request alternative consistent with a=x,...
reqalts(a=dontcare,..)	×	√	request alternative relaxing constraint a
reqmore()	√	×	inquire if user wants anything more
reqmore(a=dontcare)	√	×	inquire if user would like to relax a
reqmore()	×	√	request more information about current solution
reqmore(a=x,b=y,...)	×	√	request more info given a=x, b=y ...

A query dialogue act invites an answer to a specific question. The basic query dialogue act is the request act which takes a single item denoting an attribute as its first argument. The normal expectation of the speaker is that the hearer will respond by providing information about the queried attribute. A request act can also include an optional number of attribute/value pairs which provide conditional information to constrain the request. Examples of the use of request acts are

[4]Note that inform(food!=Italian) is not the same as deny(food=Italian) since the former asserts a constraint on the value of food whereas the latter is correcting a misunderstanding.

```
What is the address?                 <=> request(addr)
What’s the address of Toni’s place? <=> request(addr,name="Toni’s")
What kind of music do they play?    <=> request(music)
Where is the Art Deco restaurant?   <=> request(area,type=restaurant,
                                                decor="ArtDeco")
```

In addition to the basic `request` act, there are two more specialised forms of query. Firstly, the `reqalts` act indicates that the user wants to pursue a different goal. For example, if the user is given information about a specific restaurant, he or she might respond with

```
Are there any more?             <=>  reqalts()
```

Alternatively, if the user has something more specific in mind, he or she might provide some extra information, as in

```
Is there anything more central?  <=>  reqalts(area="central")
```

or relax the user’s constraints as in

```
Is there a chinese anywhere?     <=>  reqalts(food="Chinese", area=dontcare)
```

Secondly, the `reqmore` act is provided to prompt for more information about either the current topic or some specific attribute. Extra attribute/value pairs can be included to identify a specific entity that the user might have in mind. Examples are

```
Tell me more.  <=>  reqmore()
Tell me more about the hotel in the centre of town.
                                <=> reqmore(type=hotel,area="Central")
```

4.3 Confirmation

Act	System	User	Description
confirm(a=x,b=y,..)	√	√	confirm a=x,b=y,..
confirm(a!=x,..)	√	√	confirm a != x etc
confirm(name=none)	×	√	confirm that no suitable entity can be found
confirm(a=dontcare,...)	√	√	confirm that a is a ”don’t care” value
confreq(a=x,..,c=z, d)	√	×	confirm a=x,..,c=z and request value of d
select(a=x,a=y)	√	×	select either a=x or a=y
affirm()	√	√	simple yes response
affirm(a=x,b=y,...)	√	√	affirm and give further info a=x, b=y, ...
negate()	√	√	simple no
negate(a=x)	√	√	negate and give corrected value for a
negate(a=x,b=y,...)	√	√	negate(a=x) and give further info b=y, ...
deny(a=x,b=y)	×	√	no, a!=x and give further info b=y, ...

Confirm acts invite ”yes”/”no” answers, either explicitly or implicitly. They are used primarily by the system to guard against misunderstandings caused by speech errors. However, the user can also issue confirm requests to check that information supplied really does match their needs.

There are two types of confirmation. The `confirm` act itself represents an explicit confirmation request requiring an answer of either ”yes” or ”no”. The `confreq` act represents an implicit confirmation request. It combines one or more attribute/value pairs to confirm plus a query item. If the attribute/value pairs are correct, the user can ignore them and simply respond to the request. If they are not correct, the user would be expected to respond with a ”No” and ignore the request for further information. Some examples are

```
You want a restaurant playing Jazz music? <=> confirm(type=restaurant,music="Jazz")
Is that in the centre of town?            <=> confirm(area="central")
What part of town do you want to dine in? <=> confreq(area,type=restaurant)
```

An explicit positive response to a confirmation is indicated by an `affirm` act. An `affirm` act can also include additional information. In this form it is identical to an `affirm` act followed by an `inform` act.[5] Negative responses are provided by `negate` and `deny` acts. The `negate` act without arguments simply means ”No”. With arguments, there are two ways of interpreting the first argument. If the first argument provides a corrected value, then the `negate` act is used. Alternatively, if the first argument simply confirms the error, then the `deny` act is used. In both cases, any further arguments are taken to be further information as in the `affirm` act. Examples are as follows

```
Yes.                             <=> affirm()
Yes, with a nice Roman decor.    <=> affirm(decor="Roman")
No.                              <=> negate()
No, I want Chinese food.         <=> negate(food="Chinese")
No, not Russian Food.            <=> deny(food="Russian")
No, I want Chinese food in the   <=> negate(food="Chinese",area="Central")
    centre of town.
```

Finally, the `select` act provides a forced choice response

```
Do you want Chinese or Russian?  <=> select(food="Chinese",food="Russian")
```

4.4 Housekeeping

Act	System	User	Description
hello()	√	√	start dialogue
hello(a=x,b=y,...)	×	√	start dialogue and give information a=x, b=y, ...
silence()	×	√	the user was silent
thankyou()	×	√	non-specific positive response from the user
ack()	×	√	back-channel eg ”uh uh”, ”ok”, etc
bye()	√	√	verbally end dialogue
hangup()	×	√	user hangs-up
repeat()	√	√	request to repeat last act
help()	×	√	request for help
restart()	×	√	request to restart
null()	√	√	null act - does nothing

The house-keeping dialogue acts are mostly for maintaining turn taking and their meanings are straightforward. The `hello` act with arguments is essentially equivalent to the `inform` act. Some examples are,

```
Hello, I want to find a hotel.   <=> hello(type=hotel)
Can we start again?              <=> restart()
Ok.                              <=> ack()
Ok, thank you.                   <=> thankyou()
What can I say?                  <=> help()
```

The `null` act indicates a response from the user which could not be identified. It is effectively the default when all else fails. It is also used implicitly to indicate uncertainty. For example, if the user’s utterance was very uncertain, it might be represented as

```
    Mumble food mumble.  <=>  inform(type=restaurant) {0.2}, null() {0.8}
```

[5] But of course, dialog acts cannot be combined in the CUED scheme hence the `affirm` act is extended to provide the same functionality.

5 Validation and Evaluation

The mapping of an utterance into a CUED dialogue act is performed by a semantic decoder. All validation and testing of semantic decoders makes use of *semantic map* files. Each semantic map file contains a list of

```
utterance <=> act(x=y, ....)
```

pairs in the same format as used for the examples above. To provide a "gold standard" for basic compliance and regression testing, the file `GoldSemRef` contains 500 utterance to act mappings which cover all of the definitions and cases described above *for the user side* of the conversation (see Appendix B for a partial listing).

In order to validate and test a semantic decoder for compliance with the above specification, a standard semantic decoder called `SemDecode` and two Perl scripts are supplied:

1. `SemDecode -C config [-v -d DecoderType] inputfile outputfile`

   All semantic decoders for use in CUED systems are implemented as classes within the `SemIO` library. For off-line testing, a single front end called `SemDecode` is provided. `SemDecode` reads an input semantic map file and strips off any existing dialogue acts to retain only the utterances. Each utterance is then decoded and a dialogue act attached. The output file contains the decoded results in the same standard format as the input file.

   The specific decoder to use is selected by the `-d` flag. The currently supported types are:

`YTag`	- the SEMIBASIC YTag-based parser used in the prototype HIS system
`Phoenix`	- an implementation based on the CMU/Colorado Phoenix Parser
`HVS`	- a statistical parser based on the hidden vector state model

   The config file contains the various resource files needed to run the decoder. For example, to run the `YTag` parser, the config file must contain definitions for

   ```
   SEMIBASIC:      RULEFILE1       = "pre.rules"
   SEMIBASIC:      RULEFILE2       = "sort.rules"
   SEMIBASIC:      RULEFILE3       = "da.rules"
   ```

   Setting the `-v` flag enables verbose mode in which each decoded utterance is output to the terminal. Further debugging can be enabled by setting the trace flags of the relevant `SemIO` modules.

2. `ChkSemRef.pl dictfile rulesfile semfile`

   This script implements a simple validation check on the integrity of a semantic map file. It reads a HTK-format dictionary file, a HIS model rules file and the semantic map file to be validated. The script checks that

   - all words in every sentence are in the dictionary
   - all dialogue act types are valid
   - all attribute names are valid node names
   - all values and subtypes are defined in the rules

3. `SemScore.pl semtst semref`

   This script reads a test semantic map file and a reference semantic map file. Each file must have the same number of entries in the same order. `SemScore.pl` compares each corresponding line of each file and does the following:

   - check that both utterances are identical

- check that the dialogue types are the same
- check that all items in the ref appear in the test

When comparing the items in each act, `SemScore.pl` ignores the order except where it makes a difference to the interpretation of the dialogue act. Typically this means that if the act is order sensitive (e.g. `negate`), the first item must be first but the remaining items can be in any order. `SemScore.pl` computes the following statistics

H_a	- the number of correctly recognised dialog acts
N_a	- the total number of dialog acts
H_i	- the number of items correctly recognised
N_i	- the total number of items in the reference
R_i	- the total number of items in the test

Then

Act Type Accuracy	Item Precision	Item Recall	Item F-measure
$A = \frac{H_a}{N_a}$	$P = \frac{H_i}{R_i}$	$R = \frac{H_i}{N_i}$	$F = \frac{2PR}{P+R}$

History of Changes

- 19/06/2009 by François Mairesse

  Added support for two new system acts:

  - inform(name=none,a!=x,b=y,...)
  - inform(name=none,name!=a,b=y,...)

Appendix A: Summary of dialogue acts

Act	System	User	Description
hello()	√	√	start dialogue
hello(a=x,b=y,...)	×	√	start dialogue and give information a=x, b=y, ...
silence()	×	√	the user was silent
thankyou()	×	√	non-specific positive response from the user
ack()	×	√	back-channel eg ”uh uh”, ”ok”, etc
bye()	√	√	end dialogue
hangup()	×	√	user hangs-up
inform(a=x,b=y,...)	√	√	give information a=x, b=y, ...
inform(name=none)	√	×	inform that no suitable entity can be found
inform(a!=x,...)	×	√	inform that a is not equal to x
inform(a=dontcare,...)	×	√	inform that a is a ”don’t care” value
inform(name=none,a!=x,...)	√	×	inform that all entities satisfy a=x
inform(name=none,name!=a,...)	√	×	inform that a is the only entity with ...
request(a)	√	√	request value of a
request(a,b=x,...)	√	√	request value for a given b=x ...
reqalts()	×	√	request alternative solution
reqalts(a=x,..)	×	√	request alternative consistent with a=x,...
reqalts(a=dontcare,..)	×	√	request alternative relaxing constraint a
reqmore()	√	×	inquire if user wants anything more
reqmore(a=dontcare)	√	×	inquire if user would like to relax a
reqmore()	×	√	request more information about current solution
reqmore(a=x,b=y,...)	×	√	request more info given a=x, b=y ...
confirm(a=x,b=y,..)	√	√	confirm a=x,b=y,..
confirm(a!=x,..)	√	√	confirm a != x etc
confirm(name=none)	×	√	confirm that no suitable entity can be found
confreq(a=x,..,c=z, d)	√	×	confirm a=x,..,c=z and request value of d
select(a=x,a=y)	√	×	select either a=x or a=y
affirm()	√	√	simple yes response
affirm(a=x,b=y,...)	√	√	affirm and give further info a=x, b=y, ...
negate()	√	√	simple no
negate(a=x)	√	√	negate and give corrected value for a
negate(a=x,b=y,...)	√	√	negate(a=x) and give further info b=y, ...
deny(a=x,b=y)	×	√	no, a!=x and give further info b=y, ...
repeat()	√	√	request to repeat last act
help()	×	√	request for help
restart()	×	√	request to restart
null()	√	√	null act - does nothing

Appendix B: Example "Gold Standard" Dialogue Act Mappings

ok	ack()
that is correct	affirm()
yeah but i need a five star hotel	affirm(type=hotel, stars="5")
yes i'm looking for a bar	affirm(task=find, type=bar)
yes something quite basic	affirm(pricerange="cheap")
yes somewhere near the tourist information office	affirm(near="Tourist Information")
yes that sounds fine	affirm()
cool thank you goodbye	bye()
but is it near the park	confirm(near="Park")
is it reasonably priced	confirm(pricerange="moderate")
is that a two star hotel	confirm(type=hotel, stars="2")
ok so it's called number one	confirm(venue.name="Number One")
so there is no hotel near the tower	confirm(name=none, type=hotel, near="Tower")
so you can't find anything	confirm(name=none)
but i don't want a restaurant	deny(type=restaurant)
no i want indian food not italian	deny(food="Italian", food="Indian")
no not in the centre of town near the river	deny(area="central", near="River Jay")
no not italian	deny(food="Italian")
hello i'm looking for a three star hotel	hello(task=find, type=hotel, stars="3")
hiya i'm looking for a wine bar in the south of town	hello(task=find, type=bar, drinks="wine", area="south")
help	help()
what are the options	help()
a bar in the north of the city	inform(type=bar, area="north")
a bar near the shopping centre	inform(type=bar, near="Westside Shopping")
a restaurant where i can eat pizza or spaghetti	inform(type=restaurant, food="Italian")
any kind of music	inform(music="dontcare")
anything but russian	inform(food!="Russian")
anything except jazz	inform(music!="Jazz")
anywhere	inform(area="dontcare")
can you help me with a three star restaurant	inform(type=restaurant, stars="3")
cheap price range please	inform(pricerange="cheap")
cheap to moderate	inform(pricerange!="expensive")
five star hotel please	inform(type=hotel, stars="5")
how about one close to the cinema	inform(near="Cinema")
i am looking for a bar	inform(task=find, type=bar)
i am looking for a bar near the shopping centre	inform(task=find, type=bar, near="Westside Shopping")
i am looking for something near the south part	inform(task=find, area="south")
i am looking for somewhere to eat a snack in the south	inform(task=find, type=restaurant, food="snacks", area="south")
i don't mind	inform(="dontcare")
i don't mind the kind of music	inform(music="dontcare")
i don't mind what kind of music it is	inform(music="dontcare")
i don't want any drinks	inform(drinks="dontcare")
i would like a bar preferably in the north	inform(type=bar, area="north")
i would like a cheap italian restaurant	inform(type=restaurant, pricerange="cheap", food="Italian")
i would like something near a cinema	inform(near="Cinema")
i'll eat anything except indian	inform(food!="Indian")
i'll have jazz music please	inform(music="Jazz")
i'm looking for somewhere that serves beer	inform(task=find, drinks="beer")
in the centre by the post office	inform(area="central", near="Post Office")
in the centre near the post office	inform(area="central", near="Post Office")

in the cheap	inform(pricerange=”cheap”)
in the riverside area a five star hotel please	inform(type=hotel, area=”riverside”, stars=”5”)
in the riverside area of the city	inform(area=”riverside”)
it’s not important	inform(=”dontcare”)
luxurious five star hotel	inform(type=hotel, pricerange=”expensive”, stars=”5”)
moderate price range	inform(pricerange=”moderate”)
moderately priced bar that s	inform(type=bar, pricerange=”moderate”)
near the shopping centre	inform(near=”Westside Shopping”)
not pizza or pasta please	inform(food!=”Italian”)
ok i’d like a bar that’s mid price range please	inform(type=bar, pricerange=”moderate”)
ok i’ll have classical music then please	inform(music=”Classical”)
somewhere in the south of the city	inform(area=”south”)
no	negate()
no a five star hotel near the museum please	negate(type=hotel, stars=”5”, near=”Museum”)
no a hotel in the south of town	negate(type=hotel, area=”south”)
that’s not correct i am looking for a hotel	negate(type=hotel, task=find)
could you repeat that	repeat()
are there any hotels in any price range	reqalts(type=hotel, pricerange=”dontcare”)
are there any other bars playing jazz	reqalts(type=bar, music=”Jazz”)
are there any others	reqalts()
i want a different restaurant	reqalts(type=restaurant)
tell me more about murphys	reqmore(venue.name=”Murphys”)
tell me more please	reqmore()
but what’s the music like	request(music)
can i get the phone number please	request(phone)
can i have the address	request(addr)
can i please have the address of the taj mahal	request(addr, venue.name=”Taj Mahal”)
can you give me the address	request(addr)
can you tell me its telephone number	request(phone)
how expensive is it	request(pricerange)
how expensive is the regent	request(pricerange, venue.name=”The Regent”)
how much will that cost	request(price)
what kind of music does it play	request(music)
what kind of music does it play though	request(music)
start again	restart()
is that a bar or a hotel	select(type=bar, type=hotel)
thank you very much	thankyou()