Chat

A program to make Prolog input more English like

A project from Clocksin and Mellish, page 244 third edition

The main program – chat

◊ The rule repeats itself until the user enters exactly "**Stop.**"

chat :- repeat

> **Get a sentence from the user**

> **, readLine (Sentence)**

> **Obtain the semantic form, Clause, from the external form, Sentence.**

> **, parse (Clause , Sentence , _)**

> **Determine the appropriate response.**

> **, respondTo (Clause)**

> **chat succeeds when the semantic form is stop**

> **, Clause = stop .**

readLine (Sentence)

- ◊ Read a sentence as a list of words, where each word is the list of characters in ASCII numeric code.
- ◊ Split off the periods, question marks and apostrophes
- ◊ Create the corresponding list of atoms

```
readLine ( Sentence )  :-  readCharLists ( Words )
                  , morphs ( Words , Sentence)  ,  !  .
```

- ◊ User types **John is a person.**
- ◊ Words ==> **[[74, 111, 104, 110], [105, 115], [97], [112, 101, 114, 115, 111, 110, 46]]**
- ◊ Sentence ==> **[John , is , a , person , .]**

  > **John is a constant not a variable**

readCharLists (Words)

◊ Read in a list of words from the keyboard and convert each word to a list of character lists

```
readCharLists ( [ Word | MoreWords ] )  :-
```

> Read a word

```
readWord ( Word , TerminatingChar )
```

> end of line (ASCII 10 is newLIne) signals the end of the list of words

```
, ( (TerminatingChar = 10 ) , MoreWords = []
  ;  readCharLists ( MoreWords ) ).
```

◊ MoreWords is a hole

> see parts assembly example

readWord(Word, CharList)

◊ Read in a word from the keyboard

```
readWord ( Word , TerminatingChar )  :-  get0 ( C )
```

> Check for end of line or space character

```
        , ( ( C = 10 ; C = 32 )
```

> Handle eol and space character cases

```
          , TerminatingChar = C , Word = []
```

> Character in a word, get the rest of the word

```
        ;  readWord ( RestOfWord ,
                      TerminatingChar )
          , Word = [ C | RestOfWord ] ) .
```

Morphs (WordList , AtomList)

◊ Convert list of words (as character lists from readCharLists, for example) to list of atoms, applying morphological rules to split off punctuation and the possessive " 's ".

```
morphs ( [] , [] ).
morphs ( [ Word | RestOfWords ] , Atoms )  :-
    morph ( Word , Atom )
  , morphs ( RestOfWords , RestOfAtoms )
  , append ( Atom , RestOfAtoms , Atoms ) .
```

morph (Word , ItsAtoms)

◊ Convert one word, as a list of characters, to its corresponding atoms.

> More than one atom occurs when punctuation is split off, as punctuation is treated as an atom separately from a word.

```
morph ( [] , [] ) .
morph ( Word , ItsAtoms )  :-
```

> Use the available rules for morphing a word to a list of component character lists

```
    morphrules ( Word , WordComponents )
```

> Convert each list of character codes to its corresponding atom

```
  , maplist ( name , ItsAtoms , WordComponents ) .
```

morphrules (CharList , ComponentLists)

- ComponentLists is a sequence of sublists of CharList determined by the **splitOff** rules

```
morphrules ( CharList , ComponentLists ) :-

  > Do any split off rules apply?
  ( append ( X , Y , CharList )
    , splitOff ( Y )
    , ComponentLists = [ X , Y ]  )

  > Nothing to split off so only one sublist
  ; ComponentLists = [ CharList  ] .
```

splitOff (String)

◊ List of strings that are to be split off from words

> Apostrophe s

splitOff ("'s") .

> Question mark

splitOff ("?") .

> Period

splitOff (".") .

maplist (P , Arg1 , Arg2)

◊ **maplist** is a predicate that is the equivalent to the Lisp mapcar but restricted to exactly one argument

◊ **maplist** applies the predicate **P** to every item in **Arg1** and the result is the corresponding item in **Arg2**.

```
maplist ( _ , [] , [] ).
maplist ( P , [ H1 | T1 ] , [ H2 | T2 ]  )  :-
```

> **Q is the predicate P (H1 , H2). The operator =.. defines the correspondence of the compound term Q with the list form on the the right.**

```
  Q  =..  [ P, H1, H2 ]
, call ( Q )
, maplist ( P , T1 , T2 ) .
```

Parse rules

◊ The **parse** rules analyse the list of atoms in a sentence. The relevant parts are extracted and rearranged for the **respondTo** rules.

```
parse ( semantic_sentence_representation
      , the_sentence_to_parse
      , remainder_of_sentence )
```

> **First rule creates the term stop to terminate the program.**

```
parse ( stop ,  [ 'Stop' ,  '.'  ] ,  [] ) .
```

> **Last rule matches everything to create the term noparse for the "Can't parse that" response**

```
parse ( noparse , _  , _  ) .
```

Parsing "_ is a _."

◊ A rule to parse sentences of the form

John is a person.

◊ The parsing part of the rule

```
parse ( Clause )  -->
        thing ( Name ) , [ is , a ] , type ( T ) , [ '. '] .
```

◊ Where

```
thing ( Name )  -->  [ Name ] .
type ( T )  -->  [ T ] .
```

◊ This does not look like Prolog syntax

◊ What is happening?

Parse rule translations

◊ The previous syntax is in the library of predicates that comes with Edinburgh Prolog

◊ The predicates define a correspondence with the previous syntax and pure prolog syntax

Why do we need the predicates?

◊ Writing parsing rules in pure Prolog is tedious

Parsing "P is a T."

◊ Syntax as entered in chat **Looks fairly straight forward**

parse (Clause) --> [P] , [is , a] , [T] , ['.'] .

◊ Its equivalent in Prolog **compared to the translation**

parse (Clause , S , Srem) :- det1 (S , S0)
, det2 (S0 , S1) , det3 (S1 , S2) , det4 (S2 , Srem) .

◊ Query: **parse(Clause, [John, is, a, person, '.'], _)**

det1 ([P | St] , St). **P = John** **St = [is , a , person , '.']**

det2 ([is , a | St] , St). **St = [person, '.']**

det3 ([T | St] , St). **T = person** **St = ['.']**

det4 (['.' | St] , St). **St = [ ] ==> Srem = []**

Parsing "_ is a _." and translation

Looks fairly straight forward

```
parse ( Clause )  -->
        thing ( Name ) , [ is , a ] , type ( T ) , [ '. '] .
thing ( Name )  -->  [ Name ] .
type ( T )  -->  [ T ] .
```

◊ In Prolog is the following

compared to the translation

```
parse ( Clause , S , Srem )  :-
      thing ( Name , S , S0 ) , det5 ( S0 , S1 )
     , type ( T , S1 , S2 ) , det6 ( S2 , Srem ).
thing ( Name , S , Srem )  :-  det7 ( S , Srem ).
type ( T , S , Srem )  :-  det8 ( S , Strem ).

det5 ( [ is , a ] I St ] , St).        det6 ( [ '.' ] I St ] , St ).
det7 ( [ Name I St ] , St ).          det8 ( [ T I St ] , St ).
```

Semantic representation of a parse

- We can parse a sentence. So what?
- Need to get a **semantic representation** for the parse so the **respondTo** can work.
- That is the role played by the **Clause** variable in the parse rules

Parsing "_ is a _." and semantics

◊ Query:

parse (Clause , [John , is , a , person , '.'] , _).

◊ The parsing part of the rule

```
parse (Clause) -->
    thing ( Name ) , [ is , a ] ,
    type ( T ) , [ '.' ]
```

> Makes the binding
> Name = John
> T = person

◊ The semantic part of the rule

```
, { Clause =.. [ T , Name ]
    , ! } .
```

> Makes the binding
> Clause
> = person (John)

{...} indicates do not translate ..., keep as it is, in the translated rule

thing (X) & type (X)

- For things we want to check they begin with an upper case letter (capital letter)

  thing (Name) --> [Name] , { capital (Name) } .

- For types we want to check that it begins with a lower case letter.

  type (T) --> [T] , { not (capital (T)) } .

- Rule for determining if a letter is a upper case (capital) letter or not.

  > **Character withASCII code less than 96 means it is an upper case letter.**

  capital (Name) :- name (Name , [F |_]) , F < 96 .

Parsing "A _ is a _."

◊ The complete rule for parsing sentences like the following

A woman is a person.

> **The parsing part**

parse(Clause) --> ['A'] , type (T1) , [is , a]
, type (T2) , ['. ']

> **The semantic part**

, { Head =.. [T2, X] , Condition =.. [T1, X]
, Clause = (Head :- Condition) , ! } .

◊ The following bindings occur

T1 = woman T2 = person **parse**
Head = person (X) **semantics, X is a variable**
Condition = woman (X) **semantics, same X**
Clause = person (X) :- woman (X) **semantics**

Parsing "Is _ a _?"

◊ The complete rule for parsing sentences like the following

Is Mary a person?

> **The parsing part**

```
parse( Clause )  -->  [ 'Is' ] , thing( Name ) , [ a ]
                      , type( T ) , [ '?' ]
```

> **The semantic part**

```
, { Goal =.. [ T, Name ] , Clause = ( '?-' ( Goal ) ) , ! } .
```

◊ Using the example the following bindings occur

Name = Mary T = person parse
Goal = person (Mary) semantics
Clause = ?-(person (Mary)) semantics

◊ **?-** makes Clause functor unique, correct **respondTo** is used.

RespondTo

◊ The following two clauses are the response to stopping the program and to not finding a parse.

> **The argument is the semantic representation formed in the semantic part of parse rules**

```
respondTo ( stop )  :-  write ( 'All done.' ) , nl , ! .

respondTo ( noparse )  :-
            write ( 'Can''t parse that.' ) , nl , ! .
```

RespondTo – enter into database

- The following matches all clauses, so it would be last on the list
  - **It adds the clause to the database – at the beginning**

```
respondTo ( Clause )  :-  asserta ( Clause )
                          , write ( 'Ok' ) , nl , ! .
```

- **assertz(Clause)** – add at the end of the database
- **retract(X)** – find a clause in the database that matches the argument and remove it from the database

RespondTo – Yes/No query

- ◊ Match functor **?-** and argument Goal.
  - > **?- is used to provide a respondTo to correspond to a particular parse rule.**
  - > **The operator -> tries to establish the goals to its left. If they succeed, then the goals to its right are attempted**

```
respondTo ( '?-' ( Goal ) )  :-
        ( Goal -> write ( 'Yes' )  ;  write ( 'No' ) )
        , ! , nl , nl .
```

- ◊ In the case of the "**Is Mary a person?**" query we only need a yes and no answer.