1 reduce.py

To reduce a set of LFs run `python reduce.py filename` where `filename` is the name of a file that contains type specifications for constants, and the LFs. In Windows, you can do `reduce.exe filename`. This input file should have the following format:

```
const1 : Type # Type specifications for constants
const2 : Type
...
echo Here is some comment about LF1
LF1 # LFs that are to be reduced
LF2
...
```

In the input file all and only *constants* (denotational and attitudinal) must be specified. All variables will be assigned a type automatically, and must not be specified.

Arguments Usage: `reduce.py [-h] [-v {0,1,2,3}] [-p] filename`

`-h`	display help message
`-p`	pause at every reduction step. 'Enter' does the next reduction step, 'q' aborts the reduction and moves to the next LF (if any).
`-v` n	verbosity level: 0 = display all steps, 3 = display only CF [default = 0]
`filename`	(required) input file

2 Type specifications

The grammar for the human input type specification is as follows:

`type :: s | e | t | E | T | (type > type)`

Types `E` and `T` abbreviate `s>e` and `s>t`, respectively. Extra spaces are ignored by the parser. Type specification is right associative, i.e. `e>e>t` is interpreted as `(e>(e>t))`.

A constant specification consists of a name, a semicolon, a type specification and optionally an indication that the object is attitudinal. If no attitudinal specification is given, denotational is the default.

⟨name⟩ `: type␣X`
`X :: a | d`

A valid name for a constant contains only uppercase and lowercase letters, numerals, and the underscore, and is not interpretable as a name of a variable. A variable name is anything matching `[Aa-Zz][0-9]*`, such as `u`, `x1`, `P3`, etc. In-line comments can be given after a `#` sign.

Examples:

```
John : E   # this is a name
loves : E>E>T
and : (t>(t>t))
knows : E > T  a    # a for attitudinal
```

3 Logical forms

Human-input logical forms mostly follow LCMS notation. The parser allows for extra brackets and extra spaces.

- Function applications require brackets. Arguments are separated by commas.

  `func(arg1,arg2,arg3)`
  `func(arg1)(arg2)(arg3)`

- Assignments in the where-construct are written with a simple `=` symbol (rather than :=), e.g. '`P1 = John`'.

- The where-construct follows LCMS notation:

  `A␣where␣{P1=term1,P2=term2,P3=term3}`

- Lambda expressions are written with the string `lambda`. A multiple lambda term is written with variables separated by spaces, not commas. The body of a lambda function may but need not have enclosing brackets.

  `lambda␣x␣y␣z␣term`

  Operator precedence is `where` > `lambda` > application:

$$\begin{aligned} \texttt{lambda x A(B)} &\mapsto \lambda x(A(B)) \\ &\not\mapsto (\lambda x A)(B) \\ \texttt{lambda x B where \{B = Bob\}} &\mapsto (\lambda x B) \text{ where } \{B = \mathsf{Bob}\} \\ &\not\mapsto \lambda x(B \text{ where } \{B = \mathsf{Bob}\}) \end{aligned}$$

- That-constructs are written with the string `that`:

  `knows(John, that flat(earth))`

Examples from LCMS:

```
(lambda x (loves(x,x)))(J) where {J = John}
every(man)(lambda u (danced(u,wife(u))))
```