Logic Programming

Logic programming is a programming paradigm based on first order logic in which code is written in formal logic, each line expresses a rule or fact for a given problem. A solution is reached by running a query over these rules and facts.

Logic programming is based on 3 main concepts:

- Assertions – an assertion is a predicate which evaluates to true.
- Horn Clauses – a fact or a fact given all the clauses (facts/rules) evaluate to true.
- Relations – the way in which inputs and outputs are related, logic programs have a many to many relation while functional have a many to one relation.

Functional Programming

Functional programming is a programming paradigm based on lambda calculus in which code is written as pure functions, which avoids shared state, mutable data and thus any possible side effects. Functional programs reach a solution by recursively calling its own defined function repeatedly until a base case evaluates to true and then returns the results.

Functional programming is based on 5 main concepts:

- Expressions – calculates new values from old.
- Functions – functions are able to be passed as an argument to functions and evaluated from other functions.
- Parametric Polymorphism – allows a function to operate on a type family.
- Data Abstractions – simplified representation of data.
- Lazy Evaluation – only evaluates when the value is first needed.

Implementations

Functional

For the functional implementation of the longest common prefix problem I decided to use python. Python doesn't support all 5 of the concepts listed above, namely it doesn't support lazy evaluation or parametric polymorphism, but it is still enough to write a functional program for my purposes.

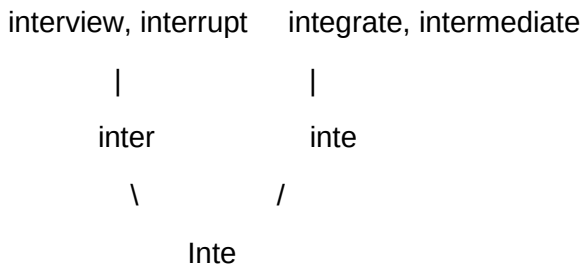
The program begins by defining the test case l, which is a list of strings whose longest common prefix is "inte", the program then calls the longestCommonPrefix function passing our list l as a parameter.

```
def main():
    l = ["interview", "interrupt", "integrate", "intermediate"]
    print(longestCommonPrefix(l))
```

The function then checks if the list l is empty, if so it returns the empty string stating that there is no common prefix. Else, the lcp() function is called passing the index of the lowest position in the list, 0, and the highest, the length of l – 1.

```
def longestCommonPrefix(l):
    if not l:
        return ""
    return lcp(l, 0, len(l)-1)
```

We then reach the most important function in our code lcp(). This function uses a divide and conquer approach by splitting the data as shown below:



lcp() recursively calls itself splitting the data into variables left and right, each of which contain at most two strings. Left and right are then passed consecutively to the prefix() function which finds the longest common prefix of two given strings.

In this way the problem is recursively divided into two subproblems to solve, the longest common prefix of “interview” and “interrupt” turns out to be “inter” and the longest common prefix of “integrate” “intermediate” is “inte”. We then find the longest common prefix of these two, “inter” and “inte”, which is “inte” and this is the answer for the longest common prefix of all the strings in the list.

```
def lcp(l, left, right):
    if left == right:
        return l[left]
    else:
        mid = (left + right) // 2
        return prefix(lcp(l, left, mid), lcp(l, mid + 1, right))
```

I will now define how the prefix function actually finds the longest common prefix. The functions recursively iterates the i index value until one of the two bases cases evaluate to true. One, if s1 is the same as s2 or for example, s1=”abc”, s2=”ab”, if we were to reach the elif statement show below and the index value is 2 the results would be an index error as s2 has no index s2 so the previous if statement remedies this issue. The second base cases simply returns once the characters of the string as the index i are no longer the same.

```
def prefix(s1, s2, i=0):
    if i == len(min(s1, s2)):
        return s1[:i]
    elif s1[i] != s2[i]:
        return s1[:i]
    return prefix(s1, s2, i + 1)
```

Logic

For the logical programming implementation of the longest common prefix I have decided to use prolog, as prolog supports all the concepts for logic programming as stated above.

The first line of the prolog implementation simple converts a string to a list of characters e.g. “test” becomes [t, e, s, t].

```
:- set_prolog_flag(double_quotes, chars).
```

If you were to query the program appropriately using:

lcp(["interview", "interrupt", "integrate", "intermediate"], P).

The predicate prefixes would return a list of possible prefixes in a Variable X given a list L.

In this case X = [[], [i], [i, n], [i, n, t], [i, n, t, e]]

Then, the list of prefixes X is given to pop which returns LCP the last value of the list of prefixes as that is the longest common prefix of all the strings in the list L.

Here, LCP = [i, n, t, e]

Atom_chars(P, LCP) converts a list into a string.

Here LCP converts to P = “inte”, the longest common prefix.

```
lcp(L, P) :-
    prefixes(L, X),
    pop(X, LCP),
    atom_chars(P, LCP),
    !.
```

As stated above prefixes returns a list of possible prefixes. It does this using the findall predicate which creates a list of objects if they satisfy the goal which in this case is the check() predicate.

```
prefixes(L,Prefixes) :-
    findall(P, check(P, L), Prefixes).
```

check() uses the maplist predicate to check if it is a common prefix.

```
check(P, L) :-
    maplist(add(P), L).
```

add concatenates P and _ into the list L using the append predicate.

```
add(P, L) :-
    append(P, _, L).
```

This process is repeated several times to produce the list X = [[], [i], [i, n], [i, n, t], [i, n, t, e]].

The list X is then given to the pop predicate, which recursively iterates through the list, passing the tail of the list until eventually there is one element remaining, which is the longest common prefix.

```
pop([Prefix], Prefix) :- !.
pop([_|T], Prefix) :-
    pop(T, Prefix).
```

Comparison

Both programs are written in a declarative manner but differ. The functional program uses functions to recursively iterate a list of strings dividing it into two subproblems, find the longest common prefix of those subproblems and repeat for the rest of the list. The logic program uses predicates composed of facts and rules to builds a list of all possible prefixes and then chooses the last element in the list as it is the longest common prefix.

Functional programming

Advantages:

- Recursion.
- Avoids changing state and mutable data as all data is passed and variables are kept within the scope of the function.
- Easier to debug and test.
- Modular, meaning they can easily be reused, unlike logic programming
- Good for parallel programming
- Often produces elegant solutions.

Disadvantages:

- Recursive calls are pushed to the stack leading to a larger memory overhead compared to logic programming.
- Recursion can seem unnatural to some people, making it difficult to code.

Logic programming

Advantages:

- Short elegant solutions.
- Efficient
- Flexible
- Good for mathematical proofs as based on first order logic.

Disadvantages:

- Harder to code
- Complex so it’s difficult to understand what is happening.
- Harder to perform mathematical computations.

References

Logic and functional programming notes:
https://www.computing.dcu.ie/~davids/courses/CA341/CA341.html