Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Тульский государственный университет

КАФЕДРА АВТОМАТИКИ И ТЕЛЕМЕХАНИКИ

ТРАССИРОВКА НА ЯЗЫКЕ ПРОЛОГ

Лабораторная работа № 7
по курсу «Функциональное и логическое программирование»

Вариант № 4

Выполнил: студент группы 220601 _____ Белым А.А.
(подпись)
Проверил: д. ф.-м. н., проф. каф. АТМ _____ Двоенко С.Д.
(подпись)

Тула 2013

Цель работы

Изучить функцию трассировки в Турбо-Прологе и оттрассировать программу.

Задание

Программа удаляет заданный элемент в списке.

```
domains
  names=symbol*
predicates
  del(symbol,names,names)
clauses
  del(H,[H|T],T).
  del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Пример запроса: `del(x,[o,x,o,o,o],List) .`

Ход работы

Рассмотрим несколько запросов. Для начала попробуем удалить элемент из списка, в котором не содержится удаляемого элемента.

```
>> del("x",["o", "o"],L)

CALL:    del("x",["o", "o"],_)
REDO:    del("x",["o", "o"],_)
CALL:    del("x",["o"],_)
REDO:    del("x",["o"],_)
CALL:    del("x",[],_)
REDO:    del("x",[],_)
FAIL:    del("x",[],_)

>> No Solution
```

В данном случае Пролог на последнем этапе не смог разбить пустой список во втором аргументе предиката на голову и хвост, что требуется в обоих клозах – поэтому происходит ошибка, и из-за этого решение не было найдено.

Данную проблему можно обойти, добавив клоз, отслеживающий ситуацию для пустого списка, и пресекающий перебор:

```
del(_,[],[]):-!,fail.
```

Кроме того, в некоторых случаях более желаемым является поведение, при котором в случае отсутствия удаляемого элемента в списке просто возвращается исходный список. Для этого достаточно просто добавить следующий клоз:

```
del(_,[],[]):-!.
```

Теперь попробуем удалить элемент, стоящий в конце списка.

```
>> del("x",["o", "o", "x"],L)

CALL:    del("x",["o", "o", "x"],_)
REDO:    del("x",["o", "o", "x"],_)
CALL:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
CALL:    del("x",["x"],_)
RETURN:  *del("x",["x"],[])
RETURN:  del("x",["o", "x"],["o"])
RETURN:  del("x",["o", "o", "x"],["o", "o"])

>> L=["o", "o"]

REDO:    del("x",["x"],_)
CALL:    del("x",[],_)
REDO:    del("x",[],_)
FAIL:    del("x",[],_)

>> 1 Solution
```

Заметно, что в данном случае наблюдается та же проблема, что и предыдущем запросе – когда второй аргумент разбирается до пустого списка, происходит ошибка. Кроме вариантов решения, указанных к предыдущему запросу, в данном случае появляется возможность ограничить перебор при нахождении удаляемого элемента:

```
del(H,[H|T],T):-!.
```

Однако следует учесть, что в этом случае произойдет удаление лишь самого первого совпавшего элемента. Однако в текущем варианте это не так. Для этого рассмотрим случай, когда в исходном списке присутствует несколько элементов для удаления:

```
>> del("x",["x", "o", "x"],L)

CALL:    del("x",["x", "o", "x"],_)
RETURN:  *del("x",["x", "o", "x"],["o", "x"])

>> L=["o", "x"]

REDO:    del("x",["x", "o", "x"],_)
CALL:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
CALL:    del("x",["x"],_)
RETURN:  *del("x",["x"],[])
RETURN:  del("x",["o", "x"],["o"])
RETURN:  del("x",["o", "o", "x"],["o", "o"])
>> L=["x", "o"]
```

```
REDO:     del("x",["x"],_)
CALL:     del("x",[],_)
REDO:     del("x",[],_)
FAIL:     del("x",[],_)

>> 2 Solutions
```

В данном случае видно, что программа выводит все варианты исходного списка (**["x", "o", "x"]**, из которого однократно удаляется указанный элемент (**L=["o", "x"], L=["x", "o"]**). Это может являться желаемым поведением, а может и нет. Если такой вариант работы является подходящим, необходимо лишь добавить обработку случая, при котором в списке не содержится удаляемого элемента. Например, эта программа возвращает исходный список:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!.
 del(H,[H|T],T).
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

а эта сообщает об отсутствии решения:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],T).
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Или, например, более предпочтительным может быть удаление лишь первого совпавшего символа. Тогда для исправления программы необходимо использовать отсечение при совпадении головы и удаляемого элемента, плюс исправление для обработки случая, когда удаляемый элемент не находится в списке.

Напрнмер, в данной программе удаляется первый совпавший элемент, и она возвращает исходный список, если удаляемого элемента нет в списке:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
```

```
clauses
 del(_,[],[]):-!.
 del(H,[H|T],T):-!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

а эта сообщает об отсутствии решения, если в списке нет удаляемого элемента:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],T):-!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Далее, может возникнуть необходимость удалить все вхождения элемента в список. Тогда требуется добавить рекурсивный вызов в клозе, где отрабатывается совпадение удаляемого элемента и головы списка. Отсечение в этом месте также требуется для предотвращения входа во второй клоз.

И также получается два варианта, в зависимости от случая обработки случая, если в списке нет удаляемого элемента:

1. При отсутствии удаляемого элемента возвращается исходный список:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!.
 del(H,[H|T],L):-del(H,T,L),!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

2. При отсутствии удаляемого элемента решение отсутствует:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],L):-del(H,T,L),!;L=T,!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

В данном случае, при любом исходном списке так или иначе будет заход в первый клоз, который выдаст `fail`. Поэтому нужно в случае неуспеха связать оставшийся хвост обрабатываемого списка с возвращаемым значением.

Кроме того, можно рассмотреть все варианты удаления указанного элемента – лишь один, лишь два, так далее, и полностью. Тогда необходимо убрать отсечение в клозе, где отрабатывается совпадение удаляемого элемента и головы списка.

1. При отсутствии удаляемого элемента возвращается исходный список:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!.
 del(H,[H|T],L):-del(H,T,L).
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

2. При отсутствии удаляемого элемента решение отсутствует:

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],L):-del(H,T,L);L=T.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Все приведенные варианты являются корректными для своего случая. Выберем один из вариантов и покажем, что он выполняется корректно.

Например, оттрассируем программу, которая удаляет все вхождения элемента, и при этом возвращает «No Solution», если данного элемента нет в списке.

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],L):-del(H,T,L),!;L=T,!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Пример: несколько удаляемых элементов в списке.

```
>> del("x",["x", "o", "x"],L)


CALL:    del("x",["x", "o", "x"],_)
REDO:    del("x",["x", "o", "x"],_)
CALL:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
```

```
REDO:    del("x",["o", "x"],_)
CALL:    del("x",["x"],_)
REDO:    del("x",["x"],_)
CALL:    del("x",[],_)
REDO:    del("x",["x"],_)
         []=[]
RETURN:  del("x",["x"],[])
RETURN:  del("x",["o", "x"],["o"])
RETURN:  del("x",["x", "o", "x"],["o"])

>> L=["o"]
>> 1 Solution
```

Пример: удаляемый элемент в конце.

```
>> del("x",["o", "o", "x"],L)

CALL:    del("x",["o", "o", "x"],_)
REDO:    del("x",["o", "o", "x"],_)
REDO:    del("x",["o", "o", "x"],_)
REDO:    del("x",["o", "o", "x"],_)
CALL:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
REDO:    del("x",["o", "x"],_)
CALL:    del("x",["x"],_)
REDO:    del("x",["x"],_)
CALL:    del("x",[],_)
REDO:    del("x",["x"],_)
         []=[]
RETURN:  del("x",["x"],[])
RETURN:  del("x",["o", "x"],["o"])
RETURN:  del("x",["o", "o", "x"],["o", "o"])

>> L=["o", "o"]
>> 1 Solution
```

Пример: удаляемый элемент отсутствует в списке.

```
>> del("x",["o", "o"],L)

CALL:    del("x",["o", "o"],_)
REDO:    del("x",["o", "o"],_)
REDO:    del("x",["o", "o"],_)
REDO:    del("x",["o", "o"],_)
CALL:    del("x",["o"],_)
REDO:    del("x",["o"],_)
REDO:    del("x",["o"],_)
REDO:    del("x",["o"],_)
CALL:    del("x",[],_)

>> No Solution
```

Как видно, сообщения `FAIL:` отсутствуют в трассировках, следовательно, программа работает корректно.

Текст программы

Далее приводится текст исправленной программы на языке Turbo Prolog 2, которая удаляет все вхождения элемента, и при этом возвращает «No Solution», если данного элемента нет в списке.

```
trace
domains
 names=symbol*
predicates
 del(symbol,names,names)
clauses
 del(_,[],[]):-!,fail.
 del(H,[H|T],L):-del(H,T,L),!;L=T,!.
 del(El,[H1|T1],[H1|T2]):-del(El,T1,T2).
```

Тестовый пример

На рисунке 1 представлен пример работы и трассировки исправленной программы.

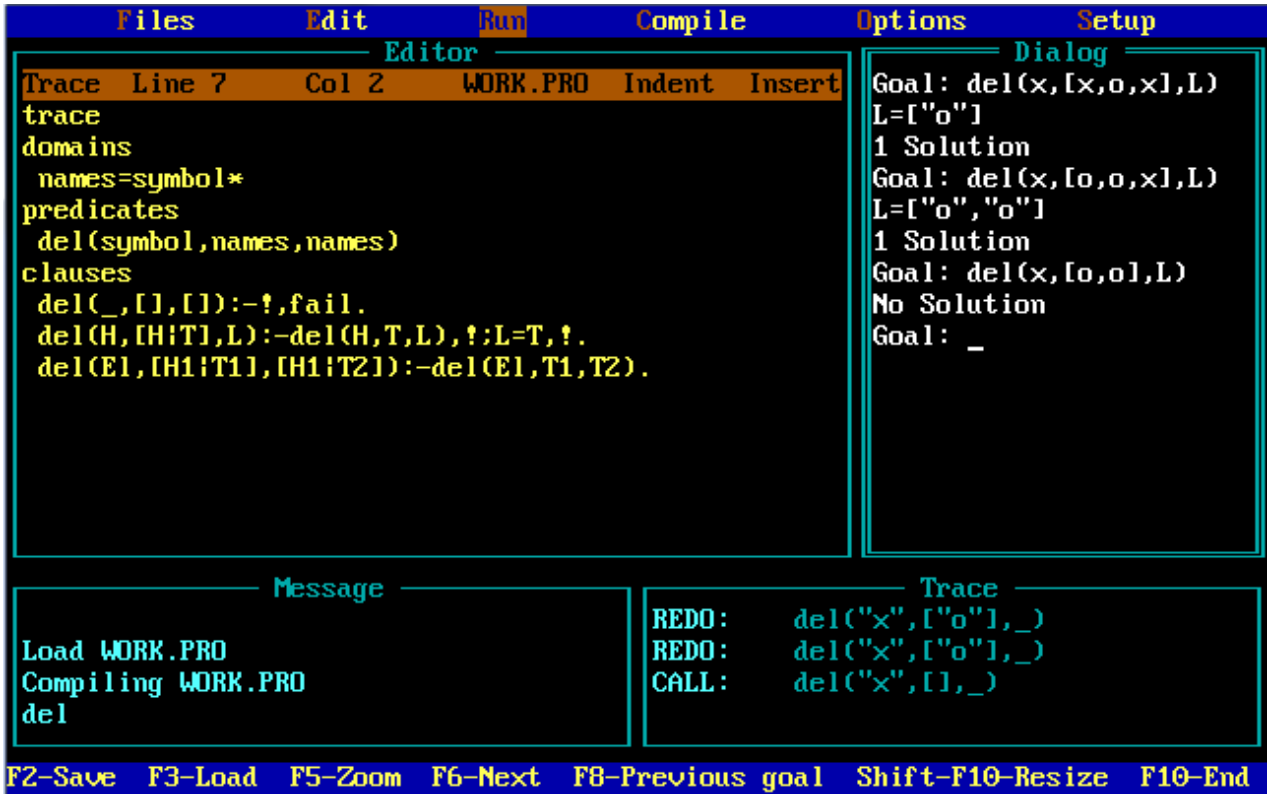

Рисунок 1— Пример работы и трассировки программы

Вывод

В данной работе я изучил трассировку программ в среде Turbo Prolog. Трассировка позволяет визуально пройти дерево поиска цели, найти и исправить проблемные места программы.