Сошников Дмитрий Валерьевич
к.ф.-м.н., доцент
http://soshnikov.com

Лекция 5: Prolog & Mercury

Логическое программирование

https://soshnikov.com/courses/logpro/

Пролог-программа

```
speciality(X,tech_translator) :-
       studied_languages(X),studied_technical(X).
speciality(X,programmer) :-
       studied(X,mathematics),studied(X, compscience).
speciality(X,lit_translator) :-
studied_languages(X),studied(X,literature).

studied_technical(X) :- studied(X,mathematics).
studied_technical(X) :- studied(X,compscience).
studied_languages(X) :- studied(X,english).
studied_languages(X) :- studied(X,german).

studied(petya,mathematics).  studied(vasya,german).

studied(petya,compscience). studied(vasya,literature).
studied(petya,english).
```

Правила

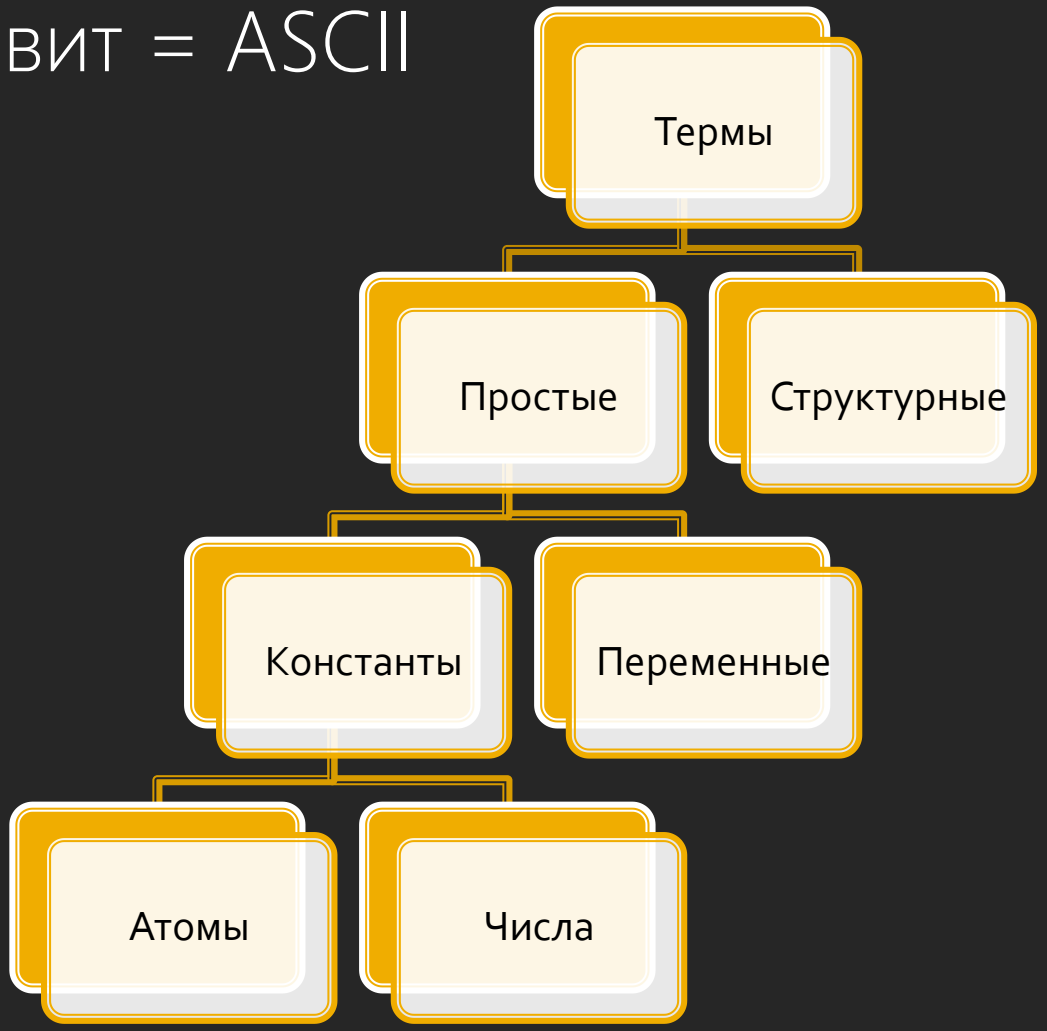
Факты

```
?- specialty(petya,X).
```

Запрос

Объекты программы

- Алфавит = ASCII



Атомы и числа

- Константы, соответствующие объектам предметной области
- Во всей программе одинаковые атомы соответствуют одному и тому же объекту
- Синтаксис:
  - Слово со строчной буквы petya
  - Последовательность спецсимволов <=
  - Символы в кавычках 'Petya Ivanov'

Переменные

- Синтаксис:
  - Начинаются с заглавной буквы или _
  - _ - анонимная переменная
- Область действия – одно правило
  - Две переменные с одним и тем же именем в разных правилах – разные

```
has_child(X) :- parents(X,Y,Z).
has_child(X) :- parents(Y,X,Z).
```

```
has_child(_123) :- parents(_123,_124,_125).
has_child(_126) :- parents(_127,_126,_128).
```

  - _ - означает все время разные переменные

```
has_child(X) :- parents(X,_,_).
has_child(X) :- parents(_,X,_).
```

Квантификация переменных

```
has_child(X) :- parents(X,Y,Z).
```

- Переменные, входящие в заключение правила, квантифицированы универсально
- Переменные, входящие только в посылку, квантифицированы экзистенциально

$$(\forall X)\ \texttt{has\_child}(X) \subset (\exists Y)(\exists Z)\ \texttt{parents}(X, Y, Z)$$

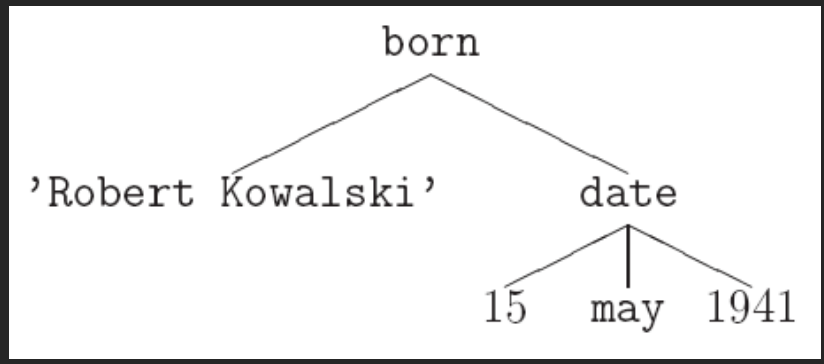
$$(\forall X)(\forall Y)(\forall Z)\ \texttt{has\_child}(X) \vee \neg\texttt{parents}(X, Y, Z)$$

Свободные и связанные переменные

- В каждый момент времени переменная может быть свободной или связанной
- Переменная связывается в процессе унификации
- Повторная унификация не изменяет значения переменной
- Переменная может изменить значение (перестать быть связанной) в процессе возврата (backtracking)

Структурные термы

- функтор(терм_1,...,терм_n)
  - функтор/n – арность=n

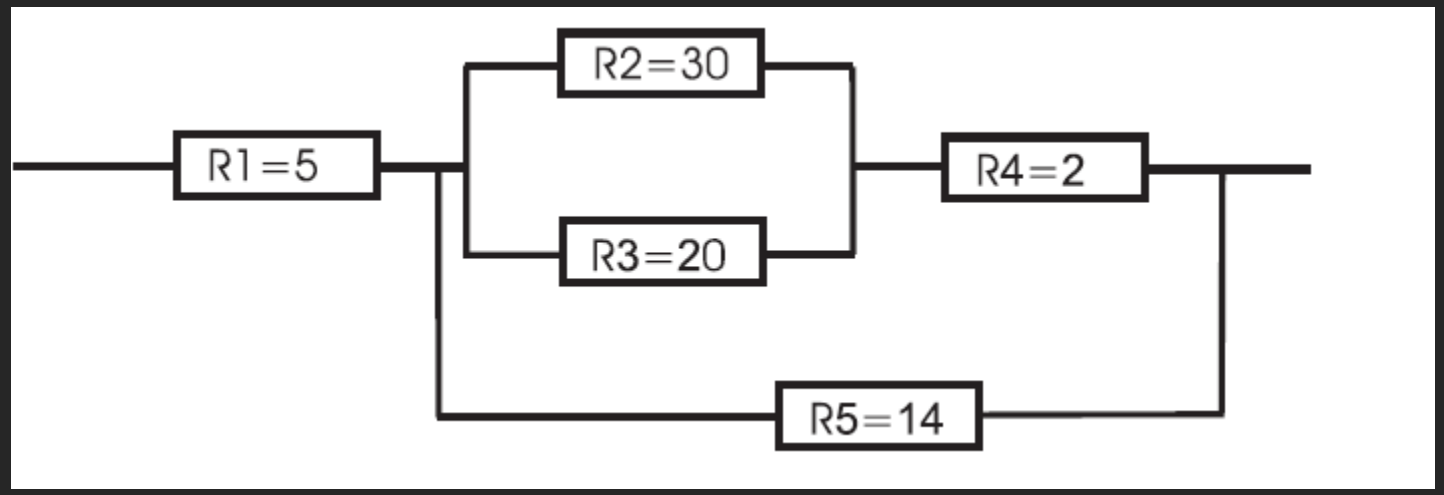


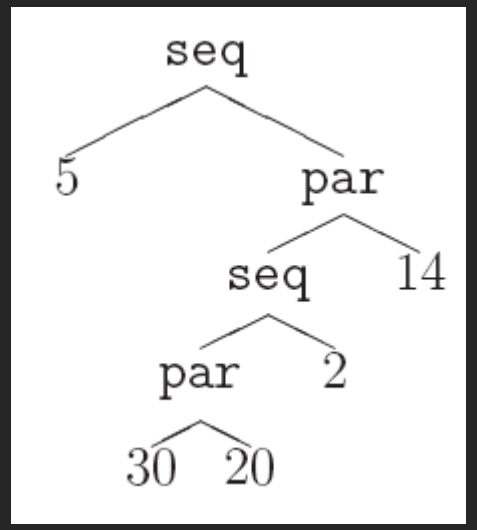

```
born(‘Robert Kowalski’,15,may,1941).

born('Robert Kowalski',date(15,may,1941)).

pension_age(X) :- born(X,Date),
      addyears(Date,60,Date1),
      current_date(Date2),
      date_less(Date1,Date2).
```

Унификация структурных термов

- Унификация в Прологе
  - Явная f(X,a) = f(b,Y)
  - Неявная в процессе поиска правил
    - pred(f(X,a)) :- ... <=> f(Z) :- Z=f(X,A), ...
- Правила унификации
  - Константа унифицируется с такой же константой, разные константы не унифицируются
  - Свободная переменная унифицируется с чем угодно и связывается
  - Связанная переменная унифицируется как значение, с которым она связана
  - Структурные термы унифицируются, если
    - У них одинаковый функтор и арность
    - Все аргументы попарно унифицируются

Структурные термы для представления бесконечных объектов

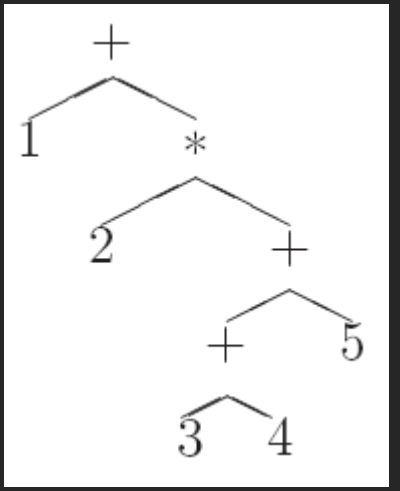





```
C = seq(5,par(seq(par(30,20),2),14)).
```

Операторная нотация

- С помощью структурных термов можно записывать арифметические выражения




```
Expr=+(1,*(2,+(+(3,4),5))).
Expr= 1+2*(3+4+5).


?- X is 1+2*(3+4+5).
X = 25
?- X = +(1,*(2,+(+(3,4),5))), Y is X.
X = 1+2*(3+4+5)
Y = 25
```

Пример

```
?- C = seq(5,par(seq(par(30,20),2),14)), resistance(C,X).
```

```
resistance(seq(X,Y),R) :-
       resistance(X,RX), resistance(Y,RY), R is RX + RY.
resistance(par(X,Y),R) :-
       resistance(X,RX), resistance(Y,RY),
       R is RX*RY/(RX + RY).
resistance(R,R).
```

```
resistance(seq(X,Y),R) :-
       resistance(X,RX), resistance(Y,RY), R = RX + RY.
resistance(par(X,Y),R) :-
       resistance(X,RX), resistance(Y,RY),
       R = RX*RY/(RX + RY).
resistance(R,R).
```

Пример символьной обработки

```
resistance(seq(X,Y),R) :-
        resistance(X,RX), resistance(Y,RY), R = RX + RY.
resistance(par(X,Y),R) :-
        resistance(X,RX), resistance(Y,RY),
        R = RX*RY/(RX + RY).
resistance(R,R).
```

```
?- resistance(seq(5,par(seq(par(30,20),2),14)),X).
X = 5+(30*20/(30+20)+2)*14/(30*20/(30+20)+2+14)
?- resistance(seq(5,par(seq(par(30,20),2),14)),X), Y is X.
X = 5+(30*20/(30+20)+2)*14/(30*20/(30+20)+2+14),Y = 12
```

```
?- resistance(seq(r1,par(seq(par(r2,r3),r4),r5)),X).
X = r1+(r2*r3/(r2+r3)+r4)*r5/(r2*r3/(r2+r3)+r4+r5)
```

Использование + и *

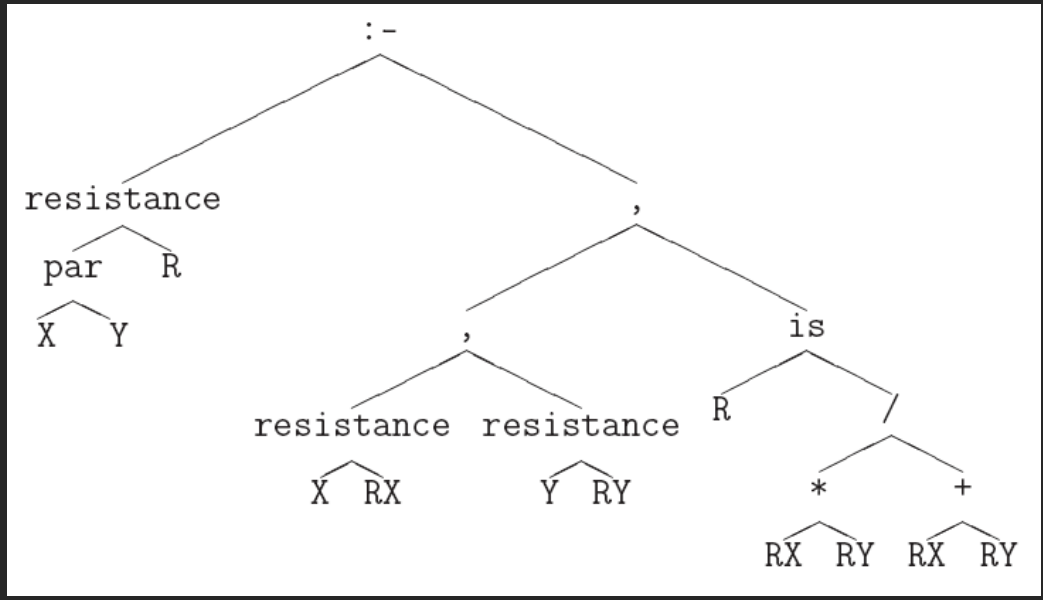
```
resistance(X+Y,R) :-
        resistance(X,RX), resistance(Y,RY), R is RX + RY.
resistance(X*Y,R) :-
        resistance(X,RX), resistance(Y,RY),
        R is RX*RY/(RX + RY).
resistance(R,R).
```

```
?- resistance(5+(30*20+2)*14,X).
X = 12.
```

В начале было дерево...

```
resistance(par(X,Y),R) :-
      resistance(X,RX), resistance(Y,RY),
      R is RX*RY/(RX + RY).
```

```
:-
(resistance(par(X,Y),R),
,(,
   (resistance(X,RX),
     resistance(Y,RY)),
is(R,/(*(RX,RY),+(RX,RY))
)))
```



Характеристики операторов

- Арность
  - одноместный not x
  - двухместный x + y
- Позициональность
  - инфиксный x + y
  - префиксный not x
  - постфиксный x!
- Ассоциативность
  - Левоассоциативный x◊y◊z = (x◊y)◊z
  - Правоассоциативный x◊y◊z = x◊(y◊z)
- Приоритет
  - 1+2*3 = 1+(2*3)

Пользовательские операторы

```
:- op(<приоритет>,<шаблон>,<оператор>).
```

- Приоритет – от 1 (высший) до 255 (низший)
- Оператор – послед.спецсимволов или функтор
- Шаблон – задает арность, позициональность и ассоциативность

Шаблон	Вид оператора	Ассоциативность
fx	унарный префиксный	неассоциативный
fy		ассоциативный
xfx	бинарный инфиксный	неассоциативный
xfy		правоассоциативный
yfx		левоассоциативный
xf	унарный постфиксный	неассоциативный
yf		ассоциативный

Пример

```
:- op(210, xfx, ===>).
:- op(200, yfx, !).
:- op(205, yfx, -).
```

```
X-Y ===> R :-
       X ===> RX, Y ===> RY, R is RX + RY.
X!Y ===> R :-
       X ===> RX, Y ===> RY, R is RX*RY/(RX + RY).
R ===> R.
```

```
?- 5-(30!20-2)!14 ===> X.
X = 12.
```

- Таким образом, мы определили Domain-Specific Language (DSL) для задания конфигурации цепи резисторов и операцию вычисления сопротивления.

Встроенные предикаты

- true – всегда завершается успешно
- fail – всегда завершается неуспешно

- Вопрос: как можно определить предикаты true и fail?

- true :- 1=1.
- fail :- 1=2.

Ввод-вывод

```
?- specialty(petya,X).
```

```
main :-
      write('Имя студента?'),
      read(Name),
      specialty(Name,Spec),
      write(Spec),
      nl.
```

```
main :-
  specialty(petya,X),
  write(X),
  nl, fail.
```

- write(X) – печать на текущий файл вывода
- read(X) – чтение терма (заканчивающегося .) из текущего файла ввода
- nl – newline, печать возврата строки
- see(filename), tell(filename) – открытие файла на ввод/вывод
- seeing(X), telling(X) – проверка в какие файлы идет ввод/вывод

Язык Mercury

- Разрабатывается университетом Мельбурна
- http://www.cs.mu.oz.au/research/mercury/
- Функционально-логический язык программирования
- Строгая типизация
- Компилятор
  - UNIX-платформы, .NET, C
- Если программа компилируется, она скорее всего будет работать правильно!

Пример программы

```
:- module factorial.
:- interface.
:- pred main(io__state,io__state).
:- mode main(di,uo) is cc_multi.

:- implementation.

:- import_module io.
:- import_module int.

:- pred fact(int,int).
:- mode fact(in,out) is cc_multi.

fact(1,1).
fact(N,R) :- N1 is N-1, fact(N1,R1), R is R1*N.

main(IN,OUT) :-
      fact(5,X), print(X,IN,I1), nl(I1,OUT).
```

Пример с цепью резисторов

```
:- type resistance == int.
:- type circuit -->
       r(resistance);
       seq(circuit,circuit);
       par(circuit,circuit).
```

```
:- pred res(circuit,resistance).
:- mode res(in,out) is det.

res(seq(C1,C2),R) :-
       res(C1,R1), res(C2,R2),
       R is R1+R2.
res(par(C1,C2),R) :-
       res(C1,R1), res(C2,R2),
       R is (R1*R2)/(R1+R2).
res(r(X),X).
```

Режимы предикатов (mode)

- Предикаты могут допускать различные конкретизации входные переменных
  - speciality(in,out), speciality(out,in), speciality(in,in), speciality(out,out)
  - resistance(in,out), resistance(in,in)
- Некоторые режимы являются частным случаем других
- Основные режимы определяют процесс доказательства
- Несколько вариантов детерминизма в каждом из режимов:
  - det – ровно одно решение (fact, …)
  - nondet – 0 и более решений (speciality(in,out),…)
  - multi – 1 и более решений (speciality(out,out),…)
  - …
- Mercury проверяет соответствие определения предиката и его режима и детерминизма

Пример

```
:- pred fact(int,int).
:- mode fact(in,out) is multi.
fact(1,1).
fact(N,R) :- N1 is N-1, fact(N1,R1), R is R1*N.
```

```
:- func fact(int) = int.
:- mode fact(in) = out is det.
fact(N) = R :- (N=<0 -> R=1 ; R is fact(N-1)*N).
```

Ввод-вывод

```
:- pred main(io__state,io__state).
:- mode main(di,uo) is det.

main(I,O) :-
      res(seq(r(5),par(seq(par(r(30),r(20)),r(2)),r(14))),X),
      write(X,I,I1), nl(I1,O).

main -->
    { res(seq(r(5),par(seq(par(r(30),r(20)),r(2)),r(14))),X) },
    write(X), nl.
```

DCG-нотация

- Часто в задачах грамматического разбора встречается ситуация, когда приходится использовать конструкции вида:

```
P(X,A,D) :- Q(X,A,B), R(B,C), S(X,C,D).

P(X) --> Q(X), R, S(X).
```

```
P(X,A,D) :- Q(X,A,B), R(B,C), T(X), S(X,C,D).

P(X) --> Q(X), R, {T(X)}, S(X).
```

Функциональная нотация

```
:- func fact(int) = int is det.
fact(N) = R :- (N=<0 -> R=1 ; R is fact(N-1)*N).

:- pred main(io__state,io__state).
:- mode main(di,uo) is det.
main -->
  {X=fact(5)},
  print(X),
  nl.
```

Функциональная нотация

```
:- func res(circuit) = resistance is det.

res(seq(C1,C2)) = res(C1)+res(C2).
res(par(C1,C2)) =
(res(C1)*res(C2))/(res(C1)+res(C2)).
res(r(X)) = X.

main --> write(res(par(r(20),seq(r(10),r(20))))),
nl.
```

Операторная нотация

```
:- func res(circuit) = resistance is det.

res(C1 `seq` C2) = res(C1)+res(C2).
res(C1 `par` C2) = (res(C1)*res(C2))/(res(C1)+res(C2)).
res(r(X)) = X.

main --> write(res(r(20) `par` (r(10) `seq` r(20)))), nl.
```

Мораль

- Пролог – классический язык логического программирования, удобный для изучения благодаря режиму интерпретации и широкой распространненности и доступности на всех платформах
- Mercury – современный исследовательский язык функционально-логического программирования, воплощающий последние идеи и использующийся в коммерческом программировании

Вопросы?

- http://t.me/log_pro
- https://soshnikov.com/courses/logpro/