Tutorium to Introduction to AI, 3rd week - Nicolas Höning

Nicolas Höning

April 28, 2006

organizational issues

- sorry for the late homework results. we're having some technical problems...
  almost all of them were really fine, so don't worry :-)
  we need to get all of you in groups, so what about these people:
  Anna-Antonia Pape, Benjamin Wulff, Janine Yvonne Willbrand, Da Sheng Zhang, Annett Wegner, Gunther Baumgartner, Arthur Legler, Jonas Volger, Yvonne Eberl, Johannes Emden

organizational issues

- sorry for the late homework results. we're having some technical problems...
  almost all of them were really fine, so don't worry :-)
  we need to get all of you in groups, so what about these people:
  Anna-Antonia Pape, Benjamin Wulff, Janine Yvonne Willbrand, Da Sheng Zhang, Annett Wegner, Gunther Baumgartner, Arthur Legler, Jonas Volger, Yvonne Eberl, Johannes Emden
- we also found out yesterday that the Prolog system on VIPS didn't always show all error messages :-(

organizational issues

- I am here to make your work easier.
  So if there is anything you want to talk about or that should be done differently, don't hesitate to tell me.

organizational issues

- I am here to make your work easier.
  So if there is anything you want to talk about or that should be done differently, don't hesitate to tell me.
- that also includes repititions. if we need to reconsider some basic concepts in order for you to really get them, then that is really worth the time. Ask me!

built-in predicates are not for free

- this week's homework suggests to have a look at the manual to find a built-in predicate that appends a list to another list (it's uploaded in Stud.IP and called "learn_prolog.pdf" and it's really readable. check it out.)

built-in predicates are not for free

- this week's homework suggests to have a look at the manual to find a built-in predicate that appends a list to another list (it's uploaded in Stud.IP and called "learn_prolog.pdf" and it's really readable. check it out.)
- you should especially read chapter 6. It might help with that exercise, but mostly it helps to really grasp that damn recursion thing.

built-in predicates are not for free

- you would also learn that append is inefficient, because it always works up and down the same list. As we will later deal with efficiency a lot, this is good to understand right at the beginning.
  Average programmers think of using a library function as one call, good programmers care about the implementation of that library function.

built-in predicates are not for free

- you would also learn that append is inefficient, because it always works up and down the same list. As we will later deal with efficiency a lot, this is good to understand right at the beginning.
  Average programmers think of using a library function as one call, good programmers care about the implementation of that library function.
- if you have time on the bus, read this brilliant essay by Joel Spolsky about that topic (not Prolog-related, but a good read).

base cases: "once" vs "every time"

- we already said that a base case is, most of the time, just the simplest case imaginable

base cases: "once" vs "every time"

- we already said that a base case is, most of the time, just the simplest case imaginable
- now, if your predicate is asked to do something once, it is even easier: you don't want the predicate to proceed to the simplest case, but stop once something is done the first time. Right?

base cases: "once" vs "every time"

- we already said that a base case is, most of the time, just the simplest case imaginable
- now, if your predicate is asked to do something once, it is even easier: you don't want the predicate to proceed to the simplest case, but stop once something is done the first time. Right?
- a situation where that something is done, is your base case.

base cases: "once" vs "every time"

- we already said that a base case is, most of the time, just the simplest case imaginable
- now, if your predicate is asked to do something once, it is even easier: you don't want the predicate to proceed to the simplest case, but stop once something is done the first time. Right?
- a situation where that something is done, is your base case.
- a base case returns true and does not proceed. perfect.

base cases: "once" vs "every time"

- we already said that a base case is, most of the time, just the simplest case imaginable
- now, if your predicate is asked to do something once, it is even easier: you don't want the predicate to proceed to the simplest case, but stop once something is done the first time. Right?
- a situation where that something is done, is your base case.
- a base case returns true and does not proceed. perfect.
- the base case can be the distinction between "once" and "every time"

last weeks Gauss: the limitations

- do you remember last week's gauss(X,Y)-predicate to calculate this formula?

last weeks Gauss: the limitations

- do you remember last week's gauss(X,Y)-predicate to calculate this formula?
- $$\sum_{i=0}^{x} i = \frac{x}{2}(x+1)$$

last weeks Gauss: the limitations

```
gauss(X,_) :- X < 0, !, fail.
gauss(0,0).
gauss(X,Y) :-
  X1 is X - 1,
  Y1 is Y - X,
  gauss(X1,Y1).
```

last weeks Gauss: the limitations

- ```
  gauss(X,_) :- X < 0, !, fail.
  gauss(0,0).
  gauss(X,Y) :-
     X1 is X - 1,
     Y1 is Y - X,
     gauss(X1,Y1).
  ```
- it needed both X and Y instantiated. Why?
  ```

last weeks Gauss: the limitations

- ```
  gauss(X,_) :- X < 0, !, fail.
  gauss(0,0).
  gauss(X,Y) :-
     X1 is X - 1,
     Y1 is Y - X,
     gauss(X1,Y1).
  ```
- it needed both X and Y instantiated. Why?
- When you do not know X, and of course you don't yet know X1, the term $X1isX - 1$ has infinitely many solutions. The same holds for $Y1isY - X$
  ```

Argument usage

- So you (we) should always care about this issue when we document our program: What terms need to be instantiated?

Argument usage

- So you (we) should always care about this issue when we document our program: What terms need to be instantiated?
- from the lecture: Argument usage
  \+ means: value must be provided
  \- means: must be free, value will be computed
  ? can be either free or a value

Argument usage

- So you (we) should always care about this issue when we document our program: What terms need to be instantiated?
- from the lecture: Argument usage
  + means: value must be provided
  - means: must be free, value will be computed
  ? can be either free or a value
- so last week's Gauss was *gauss(+X,+Y)*

Argument usage

- So you (we) should always care about this issue when we document our program: What terms need to be instantiated?
- from the lecture: Argument usage
  + means: value must be provided
  - means: must be free, value will be computed
  ? can be either free or a value
- so last week's Gauss was *gauss(+X,+Y)*
- let's think about *gauss(+X,-Y)* now

gauss(+X,-Y)

- our only base case is still gauss(0,0). The problem is that we cannot substract from Y till we reach zero, because we have no idea what Y could be in the first place.

gauss(+X,-Y)

- our only base case is still gauss(0,0). The problem is that we cannot substract from Y till we reach zero, because we have no idea what Y could be in the first place.
- can't we add every X up to reach Y while we decrement X to zero? How could we tell Prolog to do that?

gauss(+X,-Y)

- our only base case is still gauss(0,0). The problem is that we cannot substract from Y till we reach zero, because we have no idea what Y could be in the first place.
- can't we add every X up to reach Y while we decrement X to zero? How could we tell Prolog to do that?
- How can we decrement X to zero, from the first call down to the base case, while we add all those Xes up to Y, beginning at the base case?

left recursion: a simple example

- ok, take a break, look at this simple predicate here:

```
recurse([]).
recurse([H|Rest]) :-
    writeln('right... H is '+H),
    recurse(Rest),
    writeln('left.... H is '+H).
```

left recursion: a simple example

- ok, take a break, look at this simple predicate here:
```
recurse([]).
recurse([H|Rest]) :-
    writeln('right... H is '+H),
    recurse(Rest),
    writeln('left.... H is '+H).
```
- it does nothing but recurse down a list until it is empty. Besides, it tells you what is the the actual head of the list. Twice.

left recursion: a simple example

- ok, take a break, look at this simple predicate here:
```
recurse([]).
recurse([H|Rest]) :-
   writeln('right... H is '+H),
   recurse(Rest),
   writeln('left.... H is '+H).
```
- it does nothing but recurse down a list until it is empty. Besides, it tells you what is the the actual head of the list. Twice.
- Once in right-recursion-style and once in left-recursion-style. Now what will be the output of *recurse([a,b,c,d]).*?

left recursion: a simple example

- this is the output of *recurse([a,b,c,d]).*:
  right... H is +a
  right... H is +b
  right... H is +c
  right... H is +d
  left.... H is +d
  left.... H is +c
  left.... H is +b
  left.... H is +a

left recursion: a simple example

- this is the output of *recurse([a,b,c,d]).*:
  right... H is +a
  right... H is +b
  right... H is +c
  right... H is +d
  left.... H is +d
  left.... H is +c
  left.... H is +b
  left.... H is +a
- we see the way to the base case, and then we see the way back from it.
  down the recursion tree and up again.

left recursion: a simple example

- this is the output of *recurse([a,b,c,d]).*:
  right... H is +a
  right... H is +b
  right... H is +c
  right... H is +d
  left.... H is +d
  left.... H is +c
  left.... H is +b
  left.... H is +a
- we see the way to the base case, and then we see the way back from it.
  down the recursion tree and up again.
- Now, right recursion is the usual way to go, but left recursion seems to make sense for some problems...

gauss(+X,-Y)

- ok, we should change our gauss example, but just a little:

gauss(+X,-Y)

- ok, we should change our gauss example, but just a little:
-
```
/* gauss_with_X(+X,-Y) */
gauss_with_X(X,_) :- X < 0, !, fail.
gauss_with_X(0,0).
gauss_with_X(X,Y) :-
    X1 is X - 1,
    gauss_with_X(X1,Y1),
    Y is Y1 + X.
```

gauss(+X,-Y)

- ok, we should change our gauss example, but just a little:
-
```
/* gauss_with_X(+X,-Y) */
gauss_with_X(X,_) :- X < 0, !, fail.
gauss_with_X(0,0).
gauss_with_X(X,Y) :-
    X1 is X - 1,
    gauss_with_X(X1,Y1),
    Y is Y1 + X.
```
- the only changes are switching the last two lines, so we compute Y in left recursion (after it has been instantiated to zero by the base case), and using addition to compute Y instead of substraction.

gauss(-X,+Y)

- ok, now what about *gauss(-X,+Y)*? Can we do it the same way?

gauss(-X,+Y)

- ok, now what about *gauss(-X,+Y)*? Can we do it the same way?
- the problem is: we cannot decrement Y just as easy as X. X was decremented by one, Y would be decremented by an X we don't yet know.

gauss(-X,+Y)

- ok, now what about *gauss(-X,+Y)*? Can we do it the same way?
- the problem is: we cannot decrement Y just as easy as X. X was decremented by one, Y would be decremented by an X we don't yet know.
- I'll use another interesting technique to solve that one: the accumulator.

accumulators: why?

- ok, the problem again: if we have Y but no X, we cannot decrement Y till we reach zero, because we don't know by what we should decrement. We only have an X parameter that should hold the X we are looking for but is not instantiated

accumulators: why?

- ok, the problem again: if we have Y but no X, we cannot decrement Y till we reach zero, because we don't know by what we should decrement. We only have an X parameter that should hold the X we are looking for but is not instantiated
- well... we could instantiate X with zero and increment it by one with every step. Then we could decrement Y by that X and if it comes down to zero, we incremented X up to the one we were looking for!

accumulators: why?

- ok, the problem again: if we have Y but no X, we cannot decrement Y till we reach zero, because we don't know by what we should decrement. We only have an X parameter that should hold the X we are looking for but is not instantiated
- well... we could instantiate X with zero and increment it by one with every step. Then we could decrement Y by that X and if it comes down to zero, we incremented X up to the one we were looking for!
- But if we instantiate X with zero in the first place, we will never get to see that incremented X that comes up in the base case :-(

accumulators: why?

- ok, the problem again: if we have Y but no X, we cannot decrement Y till we reach zero, because we don't know by what we should decrement. We only have an X parameter that should hold the X we are looking for but is not instantiated
- well... we could instantiate X with zero and increment it by one with every step. Then we could decrement Y by that X and if it comes down to zero, we incremented X up to the one we were looking for!
- But if we instantiate X with zero in the first place, we will never get to see that incremented X that comes up in the base case :-(
- so how about introducing another dummy parameter?

accumulators: what?

- an accumulator is a name for another technique while using recursion.
  It adresses just this problem we had by introducing another parameter that is instantiated empty (say, [] or 0).

accumulators: what?

- an accumulator is a name for another technique while using recursion.
  It adresses just this problem we had by introducing another parameter that is instantiated empty (say, [] or 0).
- this parameter is then recursively changed until the base case is reached.

accumulators: what?

- an accumulator is a name for another technique while using recursion.
  It adresses just this problem we had by introducing another parameter that is instantiated empty (say, [] or 0).
- this parameter is then recursively changed until the base case is reached.
- there the parameter we want to instantiate with the solution (here: X) is instantiated with the accumulator, passed up the recursion tree, and we're done.

accumulators: what?

- an accumulator is a name for another technique while using recursion.
  It adresses just this problem we had by introducing another parameter that is instantiated empty (say, [] or 0).
- this parameter is then recursively changed until the base case is reached.
- there the parameter we want to instantiate with the solution (here: X) is instantiated with the accumulator, passed up the recursion tree, and we're done.
- This technique does no harm to the efficiency of your program (you'll find it again in that chapter 6 I talked about earlier).

gauss(-X,+Y)

- ok, let's do this: Z is our accumulator:

```
gauss_with_Y_2(_,Y,_) :- Y < 0, !, fail.
  gauss_with_Y_2(X,0,X).
  gauss_with_Y_2(X,Y,Z) :-
    Z1 is Z + 1,
    Y1 is Y - Z1,
    gauss_with_Y_2(X,Y1,Z1).
```

gauss(-X,+Y)

- ok, let's do this: Z is our accumulator:

```
gauss_with_Y_2(_,Y,_) :- Y < 0, !, fail.
  gauss_with_Y_2(X,0,X).
  gauss_with_Y_2(X,Y,Z) :-
    Z1 is Z + 1,
    Y1 is Y - Z1,
    gauss_with_Y_2(X,Y1,Z1).
```

- we'll add it up from zero to the value that X should have. Then we unify it with X and pass X up the recursion tree

gauss(-X,+Y)

- ok, let's do this: Z is our accumulator:

```
gauss_with_Y_2(_,Y,_) :- Y < 0, !, fail.
  gauss_with_Y_2(X,0,X).
  gauss_with_Y_2(X,Y,Z) :-
    Z1 is Z + 1,
    Y1 is Y - Z1,
    gauss_with_Y_2(X,Y1,Z1).
```

- we'll add it up from zero to the value that X should have. Then we unify it with X and pass X up the recursion tree
- we're back to good old right recursion again

gauss(-X,+Y): cleaning up

- ok, the user probably doesn't want to call *gauss_with_Y2(X,5050,0)*.

gauss(-X,+Y): cleaning up

- ok, the user probably doesn't want to call *gauss_with_Y2(X,5050,0)*.
- ```
  /* gauss_with_Y(-X,+Y)
  this pipes the problem to our
  special accumulator predicate */
  gauss_with_Y(X,Y) :-
    gauss_with_Y_2(X,Y,0).
  ```
  ```

gauss(X,Y): cleaning up

- and now we let the user call gauss(X,Y) and find out ourselves if X is in there or Y is:

gauss(X,Y): cleaning up

- and now we let the user call gauss(X,Y) and find out ourselves if X is in there or Y is:
- ```
  gauss(X,Y) :-
      number(X),
      gauss_with_X(X,Y).
  gauss(X,Y) :-
      number(Y),
      gauss_with_Y(X,Y).
  ```
  ```

the end

- questions?