GUPU: A Prolog course environment and its programming methodology

Ulrich Neumerkel
Institut für Computersprachen
Technische Universität Wien
ulrich@complang.tuwien.ac.at

Problems learning Prolog

* previous knowledge/skills not helpful
* many things suggest imperative understanding
  - frequently used imperative names
    suggest imperative meaning; e.g. append/3
  - overwhelming majority of built-ins produce side effects*
  - imperative programming environments
    - imperative I/O required, often misused for "debugging"
    - tracers show imperative not declarative meaning
    - debuggers produce/require too much detail

Solution

avoid imperative references by focusing on language skills
=
reading techniques + programming environment

GUPU

Gesprächsunterstützende Programmierübungsumgebung

Conversation supporting programming course environment

http://www.complang.tuwien.ac.at/ulrich/gupu/

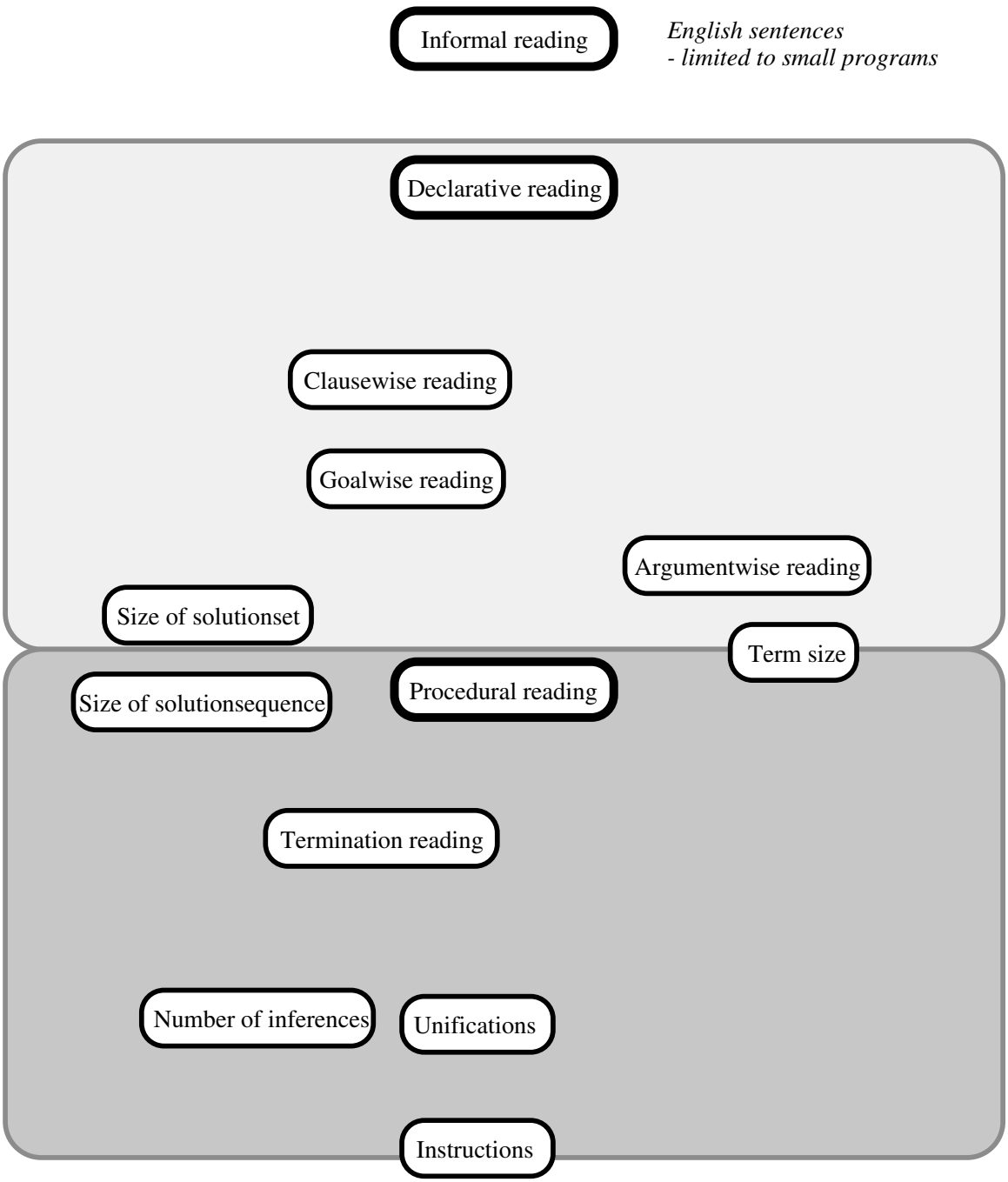
* specialized for Prolog courses
* side effect free, no toplevel shell
* subset of Prolog (e.g. layout and spelling significant)
* fast querying and testing

* *Built-in predicates according to Mixtus:*
56 with side effects,
15 sensitive to instantiations e.g. var
16 logical

Reading programs

Family of reading techniques

* read few properties at once
  by covering parts of program
* no execution traces, no proof trees



Informal and declarative reading

Informal reading

```
ancestor_of(Ancestor, Person) :-
   child_of(Person, Ancestor).
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```

Someone is an ancestor of a descendant, if he is the parent of that descendant, or if he is the parent of another ancestor of the descendant.

Incomprehensible

read only parts at once
(add remark that something is missing)

Declarative reading

Clausewise reading

```
ancestor_of(Ancestor, Person) :-
   child_of(Person, Ancestor).
~~ancestor_of(Ancestor, Descendant) :-~~
   ~~child_of(Person, Ancestor~~),
   ~~ancestor_of(Person, Descendant~~).
```

Someone is an ancestor of a person, if he is the parent of that person. (But there may be other ancestors as well.)

```
~~ancestor_of(Ancestor, Person) :-~~
   ~~child_of(Person, Ancestor~~).
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```

Someone is an ancestor of a descendant, if he is the parent of another ancestor of the descendant. (But there may be other ancestors as well.)

Goalwise reading

```
ancestor_of(Ancestor, Descendant) :-
   ~~child_of(Person, Ancestor~~),
   ~~ancestor_of(Person, Descendant~~).
```

Anyone is ancestor of anyone.

```
ancestor_of(Ancestor, Descendant) :-
   ~~child_of(Person, Ancestor~~),
   ancestor_of(Person, Descendant).
```

Someone is an ancestor of a descendant, if the descendant descends from (another) person. (But maybe more is required)

```
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ~~ancestor_of(Person, Descendant~~).
```

Someone is an ancestor of a descendant, if the ancestor has a child. (But maybe more is required)

Argumentwise reading

```
ancestor_of(Ancestor~~, Descendant~~) :-
   child_of(~~Person~~, Ancestor),
   ~~ancestor_of(Person, Descendant~~).
```

Someone is an ancestor if he has a child. (But maybe more is required)

Detecting errors with declarative reading

most errors can be located by
reading only part of a program

Wrong definition

```
ancestor_of(Ancestor, Person) :-
   child_of(Ancestor, Person).
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```

Someone is an ancestor of a person,
if the ancestor is a child of that person.

The hidden clause cannot "undo" the error.
It can be ignored, if the remaining program is already wrong.

* errors can be located statically
* debuggers not helpful,
  because they provide irrelevant detail
  (e.g. procedural aspects)

Estimating efficiency with declarative reading

size of solutionset
size of terms in solution

Procedural reading

* special case of declarative reading
* uncover goal in fixed order
* consider variables, estimate size of solutions

```
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```

```
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```


Free Variables in head.

```
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```


Person always free.
Descendant has no influence on child_of/2.

```
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```


Descendant is passed through.
ancestor_of/2 depends on child_of/2.

Termination reading

Hide parts that do not influence termination.
If remaining predicate terminates (for a particular goal), also the original will terminate.

```
ancestor_of(Ancestor, Person) :-
   child_of(Person, Ancestor).
ancestor_of(Ancestor, Descendant) :-
   child_of(Person, Ancestor),
   ancestor_of(Person, Descendant).
```

If this rule terminates (and fails), also original predicate terminates.

If Person= Ancestor it does not terminate (similarly for larger cycles)

The programming environment GUPU

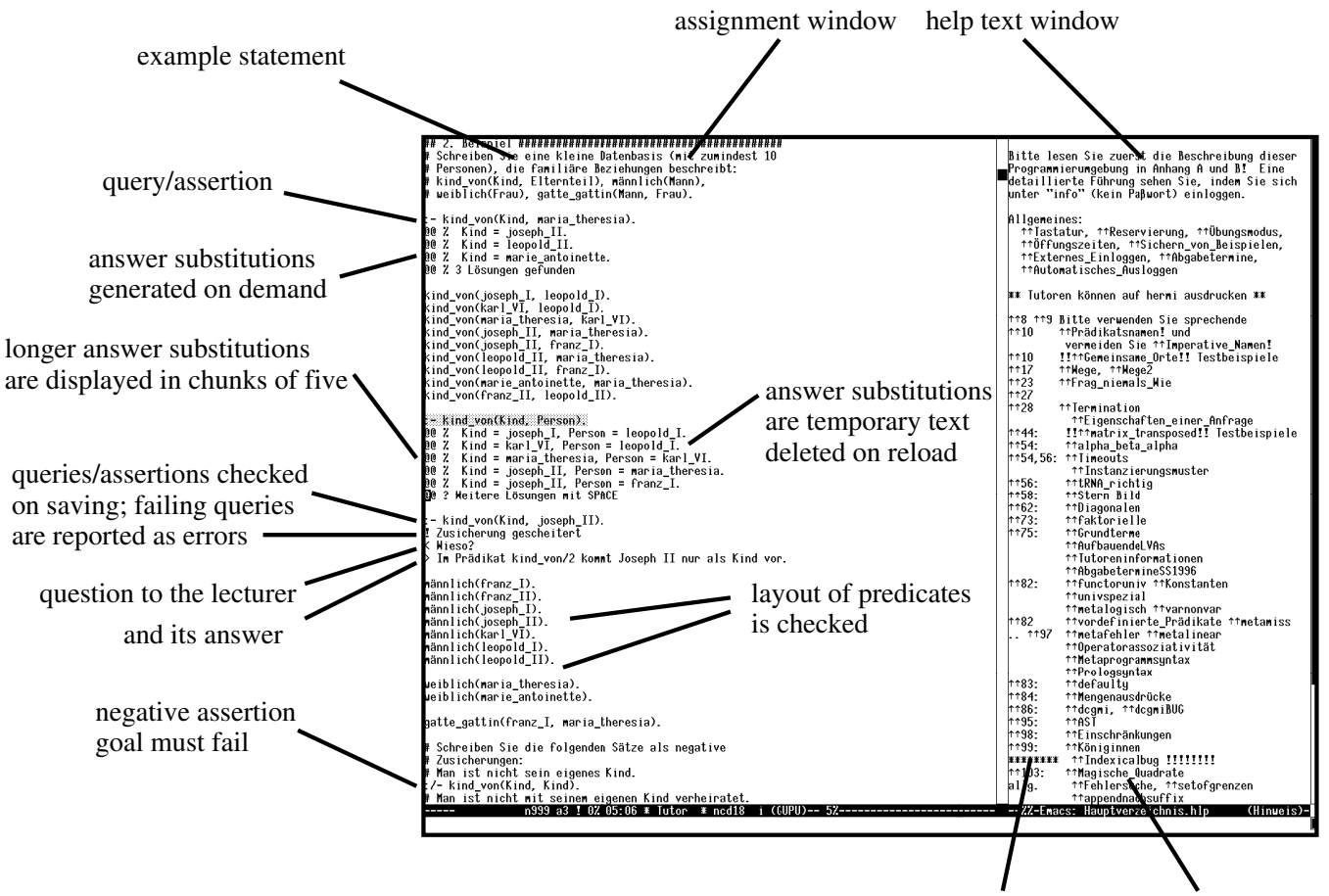
Gesprächsunterstützende Programmierübungsumgebung
Conversation supporting programming course environment

http://www.complang.tuwien.ac.at/ulrich/gupu/

* specialized for Prolog courses
* side effect free, no toplevel shell
* subset of Prolog (e.g. layout and spelling significant)
* fast querying and testing

Supported approach to write predicates:

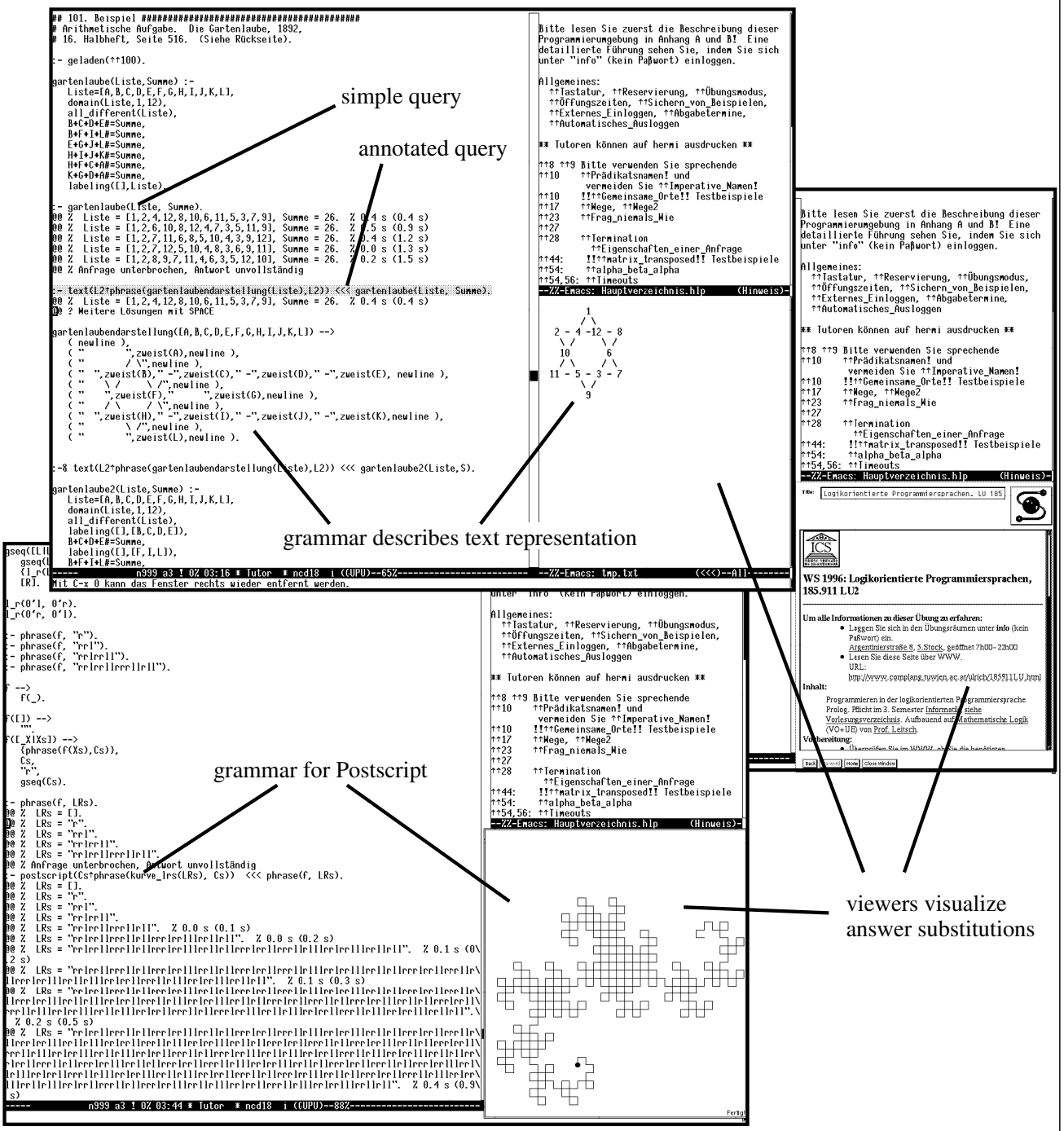
1) find relation
2) find nonimperative, descriptive names
   names must not be verbs in imperative (checked)
3) write goals for which relation should hold and should not hold
4) code predicate



Viewing answer substitutions

side effect free "output" using query annotations
based on elementary viewers more complex viewers are built

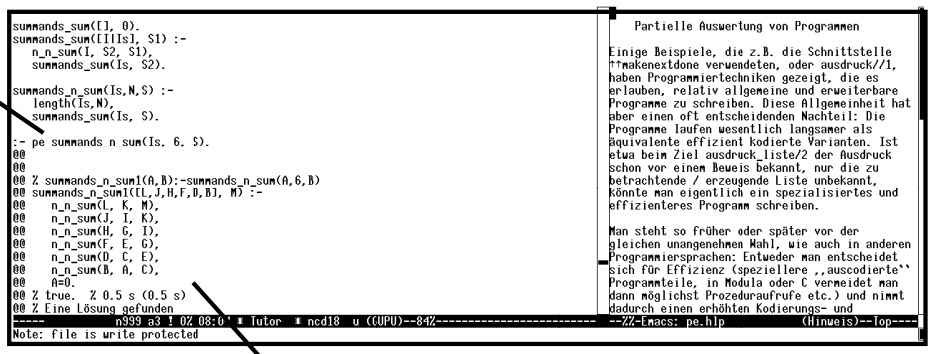
`:- text(Cs) <<< Query.`	View characterlist as unstructured text.
`:- postscript(Cs) <<< Query.`	View characterlist as postscript document.
`:- html(Cs) <<< Query.`	View characterlist as html document.



Partial evaluation

* Mixtus integrated
* simple usage by annotating query
* instant response
* often helpful for finding errors

annotated query



residual program
can be included into source

Abstract.
GUPU is a programming environment specialized for Prolog programming courses which supports a novel way to teaching Prolog. The major improvement in teaching Prolog concerns how programs are read and understood. While the traditional approach covers Prolog's execution mechanism and its relation to mathematical logic we confine ourselves to reading programs informally as English sentences. The student's attention remains focused on a program's meaning instead of details like proof trees or execution traces. Informal reading is limited to short predicates. Larger predicates translate into incomprehensible sentences cluttered with referents and connectives. To overcome this problem a simple reading technique is presented that does not translate the whole predicate at once into English. Only parts of a predicate are considered. The remainder (e.g. some clauses, goals, arguments) is neglected for the moment. In this manner incomprehensible sentences are avoided. Our reading technique extends well to the more procedural aspects of Prolog like termination and resource consumption. The reading technique allows to reason about a program (e.g. understanding, detecting errors) in an efficient static manner while avoiding reference to superfluous details of the computation.

GUPU supports this approach with a side effect free programming environment. Programs are subject to restrictions which ease informal reading and catch many mostly syntactic and stylistic errors. The cumbersome ''type and forget''-style top-level shell is replaced by a side effect free mode of interaction which also improves coding style by allowing to write tests before coding a predicate. The partial evaluator Mixtus is seamlessly integrated into GUPU.