MoSMath
A MOdeling System for MATHematics

Arnold Neumaier, Peter Schodl, Kevin Kofler
University of Vienna, Austria

February 5, 2010

Support by the Austrian Science Foundation FWF under contract number P20631 is gratefully acknowledged.
The MoSMath project

**Goal:** creation of a software package that is able to understand, represent and interface optimization problems posed in a controlled natural language.
1. The semantic matrix
2. Typing in the semantic matrix
3. The semantic Turing machine (STM)
4. An experimental grammar
5. Interface to the TPTP and the OR-Lib
6. Interface to Naproche
The semantic matrix

A user-friendly representation of information.

Designed to be human intelligible and clear (akin to the Semantic Web), and easily processable for a machine.
The semantic matrix: Definitions

The semantic matrix represents information in terms of nodes.

Kinds of nodes, not necessarily disjoint: **Booleans**, **counts**, **fields**, **handles**, **strings** and **values**.

The only Booleans: **TRUE**, **FALSE**.
Counts represent the natural numbers.
**EMPTY** is not a count, not a handle, not a string, and not a value.
Nodes are countable.
The **hash** # followed by some alphanumerical string can stand for any node.

We use suggestive strings, e.g., for a handle, we use `#handle` or `#h`.
The semantic matrix: Dot notation

A **semantic mapping** assigns to every handle #h and every field #f a node #h.#f.

a.b.c stands for (a.b).c etc., i.e., the semantic mapping is left associative.

An equation a.b=c with a, b, and c not EMPTY is a **semantic unit** or **sem**.

Here, a is called the **handle**, b the **field**, and c the **entry of the sem**.
Example: $7 + 5 = 12$ represented via the semantic mapping:

```
#h.RHS = 12
#h.LHS = term_lhs
#h.OP = EQUAL
term_lhs.1 = 7
term_lhs.2 = 5
term_lhs.OP = PLUS
```
Interpreted as a **semantic matrix**:

<table>
<thead>
<tr>
<th></th>
<th>LHS</th>
<th>RHS</th>
<th>OP</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#h</td>
<td>term_lhs</td>
<td>12</td>
<td>EQUAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>term_lhs</td>
<td></td>
<td>PLUS</td>
<td>7</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Interpreted as a semantic graph:
A node \(#e\) is **reachable** from a handle \(#h\) if there is some path starting at \(#h\) and ending in \(#e\).

Information in the semantic matrix is organized in records: A **record** is some handle \(#h\), the nodes and the sems reachable from \(#h\) **belong** to the record \(#h\).
The semantic matrix: Example

\[ \left\{ x \in \mathbb{R} \mid 0 \leq x \leq 1 \lor x = 2 \right\} \]

<table>
<thead>
<tr>
<th>record</th>
<th>MATLAB construction code</th>
</tr>
</thead>
<tbody>
<tr>
<td>#h.FREE = #3 .OP = SET .FORMULA.1.1.FREE = #4 .OP = CHAINLINK .RHS.FREE = #5 .OP = CON .ENTRY = &quot;1&quot; .RELATION = LEQ .FREE.#1 = #1 .OP = CHAIN .LHS.FREE.#1 = #1 .OP = LEQ .LHS.FREE = #6 .OP = CON .ENTRY = &quot;0&quot; .RHS = #1 .NARG = 1 .2.FREE.#1 = #1 .OP = EQUAL .LHS = #1 .RHS.FREE = #7 .OP = CON .ENTRY = &quot;2&quot; .FREE.#1 = #1 .OP = OR .NARG = 2 .SCOPE.FREE = #8 .OP = CON .ENTRY = &quot;\backslash Rz&quot; .BINDS.#1 = #1 #1.FREE = #1 .NAME = &quot;x&quot; .OP = VAR</td>
<td></td>
</tr>
</tbody>
</table>

\[ x = \text{mkvar}('x'); \]
\[ R = \text{mkcon}('\backslash Rz'); \]
\[ \text{zero} = \text{mkcon}('0'); \]
\[ \text{one} = \text{mkcon}('1'); \]
\[ \text{two} = \text{mkcon}('2'); \]
\[ f1 = \text{mkexp}('LEQ', \{\text{zero}, x\}); \]
\[ f1b = \text{mkexp}('CHAINLINK', \{\text{LEQ}, \text{one}\}); \]
\[ f1 = \text{mkexp}('CHAIN', f1, \{f1b\}); \]
\[ f2 = \text{mkexp}('EQUAL', \{x, \text{two}\}); \]
\[ \text{exor} = \text{mkexp}('OR', \{f1, f2\}); \]
\[ \text{ex} = \text{mkexp}('SET', \text{exor}, \{\text{x}, R\}); \]
The typing system

When using a record, we need information about the structure of this record.

⇒ we assign types.

A sem can have some type.
A record can be well-typed.

Our type system is suited for the typing of:
− usual data structures
− grammatical categories.
Type declaration: description of a type as requirements on the out-edges.

General form of a type declaration:

#TD: #T1 + #T2 + ... +
ALLOF > #N1=#T1, #N2=#T2, #N3=#T3, ...
ONEOF > #N1=#T1, #N2=#T2, #N3=#T3, ...
SOMEOF > #N1=#T1, #N2=#T2, #N3=#T3, ...
OPTIONAL > #N1=#T1, #N2=#T2, #N3=#T3, ...
INDEX > #T
SOMEOF.TYPE > #T1=#t1, #T2=#t2, #T3=#t3, ...
#TD: #T1 + #T2 + ... +

Type #TD has to fulfill all requirements on #T1, #T2, etc., simultaneously.

**Example:** The type declaration COMMENTEDVECTOR requires everything that is required of a COMMENTEDNODE and of a VECTOR.

COMMENTEDVECTOR: COMMENTEDNODE + VECTOR
ALLOF > #N1=#T1, #N2=#T2, #N3=#T3, ...

Requires all of the fields with entries of a certain type.

**Example:** Type declaration LEQ requires that the children both in LHS and RHS are counts.

**LEQ:**

ALLOF > LHS=COUNT, RHS=COUNT
Typing: ONEOF

ONEOF > #N1=#T1, #N2=#T2, #N3=#T3, ...  

Requires exactly one of the fields with entries of a certain type.

**Example:** An integral must have either a field FROMTO or a field OVER, but not both. The node in OVER must be a set, the node in FROMTO must be an expression.

**INTEGRAL:**
- ALLOF > INTEGRAND=EXPR
- ONEOF > FROMTO=PAIR, OVER=SET
SOMEOF > #N1=#T1, #N2=#T2, #N3=#T3, ... 

Requires at least one of the fields with entries of a certain type.

**Example:** Type declaration INDICES requires at least one of #rec.SUB, #rec.SUP, #rec.LSUB and #rec.LSUP to be an expression.

**INDICES:**
SOMEOF > SUB=EXPR, SUP=EXPR, LSUB=EXPR, LSUP=EXPR
OPTIONAL > #N1=#T1, #N2=#T2, #N3=#T3, ...

Requires the if a record has certain fields, to have entries of a certain kind.

**Example:** A variable can, but need not have an assigned name. Type declaration `VAR` requires that if `#rec.NAME` not `EMPTY`, then it must be a string.

**VAR:**

OPTIONAL > NAME=STRING
INDEX > #T

Requires entries of a certain kind in the fields 1, . . . , n and a count $n$ in the field NARG.

Example: Type declaration PLUS requires an expression in the position #handle.$i$ for $i = 1, \ldots, n$ and count $n$ in #handle.NARG.

PLUS:
INDEX > EXPR
SOMEOPFTYPE > #T1=#t1, #T2=#t2, #T3=#t3, ...

Requires entries of a certain type in fields of a certain type.

**Example:** Type declaration ONLYCOUNTS requires a count in every field.

**ONLYCOUNTS:**
SOMEOPFTYPE > NODE=COUNT
A supertype is a collection of types.

The type declaration of a supertype #ST has the general form:

#T1, #T2, ... < #ST

Example: We want to define the supertype NUMBER containing the subtypes REAL, FLOAT and INTEGER.

REAL, FLOAT, INTEGER < NUMBER
Also, we allow variables in declarations, of the form:

\[
\#TD<x,y,\ldots>:\n\text{ALLOF} > \#N1=x
\text{INDEX} > y
\ldots
\]

We call these templates.

The string \#TD<#T1,#T2,\ldots> is then a type declaration.

**Example:**

\[
\text{ARRAY}<x>:\n\text{INDEX} > x
\]

\[
\text{RVECTOR} = \text{ARRAY}<\text{REAL}>
\]
Type declarations are texts!

They define a declared type which is stored as a record in the semantic matrix.

For record #rec, the declared type in #rec.OP applies to #rec.
Intrinsic type: every node has intrinsic type NODE, every count has intrinsic type COUNT, etc.

A sem 
\[ #\text{rec.} #f=#n \]

is called typed if 
\[ #\text{rec.} . OP \]

is a declared type and 
\[ #\text{rec} \]

meets the requirements.
Typed sems have the type as required in this type declaration.

A declared sem is a reachable sem, mentioned in the type declaration.

A record is well-typed:
either 
\[ #\text{rec.} . OP = \text{EMPTY} \], or every declared sem of 
\[ #\text{rec} \]

has a unique type.
Typing: Example

The following semantic graph displays the relevant part of the record encoding the expression $1/(x + 1)$. 
Is the record #\( h \) well-typed?

The type declarations are:

\[
\begin{align*}
\text{DIV:} & \quad \text{ALLOF} > \text{DEN}=\text{EXPR}, \text{NUM}=\text{EXPR} \\
\text{PLUS:} & \quad \text{INDEX} > \text{EXPR} \\
\text{PLUS,CON,VAR} & \quad < \text{EXPR} \\
\text{VAR:} & \quad \text{OPTIONAL} > \text{NAME}=\text{NODE} \\
\text{CON:} & \quad \text{OPTIONAL} > \text{ENTRY}=\text{NODE}
\end{align*}
\]
First step:

Gather all declared sems.
Typing: First step

Declared sems:
#h.DEN=#den
#h.NUM=#num

DIV:
ALLOF > DEN=EXPR, NUM=EXPR
Typing: First step

Declared sems:
#h.DEN = #den
#h.NUM = #num
#den.1 = #varx
#den.2 = #num
#den.NARG = 2

PLUS:
INDEX > EXPR
Declared sems:

- \( \#h.DEN = \#\text{den} \)
- \( \#h.NUM = \#\text{num} \)
- \( \#\text{den}.1 = \#\text{varx} \)
- \( \#\text{den}.2 = \#\text{num} \)
- \( \#\text{den}.\text{NARG} = 2 \)
- \( \#\text{num.ENTRY} = 1 \)

CON:

\text{OPTIONAL} \rightarrow \text{ENTRY=}\text{NODE}
Typing: First step

Declared sems:
#h.DEN = #den
#h.NUM = #num
#den.1 = #varx
#den.2 = #num
#den.NARG = 2
#num.ENTRY = 1
#varx.NAME = #str_x

VAR:
OPTIONAL > NAME=NODE
Second step:

Determine the type of all the declared sems. Order is irrelevant!

We remind:
A sem \( #_{rec}.#f=#n \) is called **typed** if \( #_{rec}.OP \) is a declared type and \( #_{rec} \) meets the requirements. Typed sems have the type as required in this type declaration.
Typing: Second step

Declared sems:

- \#h.DEN = \#den
- \#h.NUM = \#num
- \#den.1 = \#varx
- \#den.2 = \#num
- \#den.NARG = 2
- \#num.ENTRY = 1
- \#varx.NAME = \#str_x

DIV:

\texttt{ALLOC} > \texttt{DEN} = \texttt{EXPR}, \texttt{NUM} = \texttt{EXPR}

\texttt{PLUS}, \texttt{CON}, \texttt{VAR} < \texttt{EXPR}

\Rightarrow \#h.DEN = \#den \textit{has type} \texttt{EXPR}.
DIV:  
ALLOF > DEN=EXPR, NUM=EXPR

PLUS, CON, VAR < EXPR

⇒ #h.NUM = #num has type EXPR.
Declared sems:
#h.DEN = #den
#h.NUM = #num
#den.1 = #varx
#den.2 = #num
#den.NARG = 2
#num.ENTRY = 1
#varx.NAME = #str_x

PLUS:
INDEX > EXPR

PLUS, CON, VAR < EXPR

⇒ #den.1 = #varx has type EXPR.
Typing: Second step

Declared sems:

\#h.DEN = \#den
\#h.NUM = \#num
\#den.1 = \#varx
\#den.2 = \#num
\#den.NARG = 2
\#num.ENTRY = 1
\#varx.NAME = \#str_x

PLUS:
INDEX > EXPR

PLUS, CON, VAR < EXPR

⇒ \#den.2 = \#num has type EXPR.
Typing: Second step

Declared sems:

\#h.DEN = \#den
\#h.NUM = \#num
\#den.1 = \#varx
\#den.2 = \#num
\#den.NARG = 2
\#num.ENTRY = 1
\#varx.NAME = \#str_x

PLUS:
INDEX > EXP

2 has intrinsic type COUNT.

⇒ \#den.NARG = 2 has type COUNT.
Typing: Second step

Declared sems:
\#h.DEN = \#den
\#h.NUM = \#num
\#den.1 = \#varx
\#den.2 = \#num
\#den.NARG = 2
\#num.ENTRY = 1
\#varx.NAME = \#str_x

CON:
OPTIONAL > ENTRY=NODE

1 has intrinsic type NODE.

⇒ \#num.ENTRY = 1 has type NODE.
VAR:
OPTIONAL > NAME=NODE

#str_x has intrinsic type NODE.

⇒ #varx.NAME = #str_x has type NODE.
Third step:

#rec.OP is not EMPTY and every declared sem of #h has a unique type

⇒ #h is well-typed
The STM: Motivation

The semantic Turing machine (STM)

Foundation is a theoretical machine:
- works on the semantic matrix,
- assembler-style programming language,
- basis for higher programming languages,
- Turing-complete,
- formalized I/O,
- transparent and simple,
- verifiable.
STM can perform 33 different commands, each is a comprehensible action.

Four groups of commands:
– commands that structure the program, no influence at runtime
– commands for flow control
– commands for alterations in the memory of the STM
– commands for handling I/O and external processors.
A **process** is a sequence of commands, the first has to be `process(#proc)`. Every process ends with a command that either halts the STM or calls another process.

A process can be entered only at its first command, but can be left before its last command.

**STM program** is a sequence of processes, the first line has to be `program(#prog)`.

<table>
<thead>
<tr>
<th>STM command</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>program #1</td>
<td>first line of the program #1</td>
</tr>
<tr>
<td>process #1</td>
<td>first line of the process #1</td>
</tr>
<tr>
<td>start #1</td>
<td>start with process #1</td>
</tr>
</tbody>
</table>
To enable a program to call another program as a subroutine, each program has its own **core**: a record reserved for temporary data.

Core is the most important record for a program
→ simplify notation:

The **caret** ^ abbreviates reference to the current core. Hence ^a means #core.a, where #core is the core of the program under consideration. In any STM program, ^ means always the core of this program.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~#1=(~#2==~#3)</code></td>
<td>sets <code>~#1</code> to 'T if <code>~#2</code> equals <code>~#3</code>, else to 'F</td>
</tr>
<tr>
<td><code>~#1=copy of ~#2</code></td>
<td>makes a copy of record <code>~#2</code> to the node <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=copy of #2</code></td>
<td>makes a copy of record <code>#2</code> to the node <code>~#1</code></td>
</tr>
<tr>
<td><code>#1=copy of ~#2</code></td>
<td>makes a copy of record <code>~#2</code> to the node <code>#1</code></td>
</tr>
<tr>
<td><code>~#1.#2=~#3</code></td>
<td>assigns <code>~#3</code> to <code>~#1.#2</code></td>
</tr>
<tr>
<td><code>~#1.#2=const #3</code></td>
<td>assigns <code>#3</code> to <code>~#1.#2</code></td>
</tr>
<tr>
<td><code>~#1.^#2=const ^#3</code></td>
<td>assigns <code>^#3</code> to <code>~#1.^#2</code></td>
</tr>
<tr>
<td><code>~#1=^#2.#3</code></td>
<td>assigns <code>^#2.#3</code> to <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=^#2.^#3</code></td>
<td>assigns <code>^#2.^#3</code> to <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=const #3</code></td>
<td>writes the constant <code>#3</code> to <code>~#1.#2</code></td>
</tr>
<tr>
<td><code>~#1.^#2=const ^#3</code></td>
<td>writes the constant <code>^#3</code> to <code>~#1.^#2</code></td>
</tr>
<tr>
<td><code>~#1=fields of ~#2</code></td>
<td>increments the count <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=fields of #2</code></td>
<td>decrements the count <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=exist(#2.#3)</code></td>
<td>assigns some free node to <code>~#1</code></td>
</tr>
<tr>
<td><code>~#1=exist(~#2.^#3)</code></td>
<td>assigns nonempty fields of <code>~#2</code> to <code>~#1.1</code>, <code>~#1.2</code>,...</td>
</tr>
<tr>
<td><code>clean ~#1</code></td>
<td>sets <code>~#1</code> to 'T if <code>#1.#2</code> exists, else to 'F</td>
</tr>
<tr>
<td><code>~#1=exist(~#2.^#3)</code></td>
<td>sets <code>~#1</code> to 'T if <code>~#1.^#2</code> exists, else to 'F</td>
</tr>
<tr>
<td><code>clean ~#1</code></td>
<td>deletes the nodes reachable only from <code>~#1</code></td>
</tr>
</tbody>
</table>
The STM: Flow control

The (changing) STM command currently executed is called the focus.

Focus is stored in ^process.^line, where ^process contains the process currently executed, and ^line is a count. Incrementing #line means to proceed one line forward in the program.

We need a record containing information concerning flow control called a frame.

Local frame for every program (in its core), global frame to jump between programs (keeps track of cores).
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto #1</td>
<td>sets the focus to the first line of process #1</td>
</tr>
<tr>
<td>goto ^#1</td>
<td>sets the focus to the first line of process ^#1</td>
</tr>
<tr>
<td>if ^#1 goto #2</td>
<td>sets the focus to the first line of process #2 if ^#1='T, and to the next line if ^#1='F</td>
</tr>
<tr>
<td>function: #1(^#2,^#3)</td>
<td>starts execution of the STM program #1 in library ^#3 with context ^#2</td>
</tr>
<tr>
<td>stop</td>
<td>ends a program</td>
</tr>
</tbody>
</table>
Each node can have an external value: arbitrary data stored outside the memory.

Information about how to represent the external value in the semantic matrix is called the protocol.

Special commands to access external processors, i.e., the facilities of the physical device it is implemented on.

Example, printing external value on the screen is a common external processor. Also: existing well-trusted programs.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>external #1(^#2)</code></td>
<td>starts execution of the external processor #1 with context ^#2</td>
</tr>
<tr>
<td><code>external ^#1(^#2)</code></td>
<td>starts execution of the external processor ^#1 with context ^#2</td>
</tr>
<tr>
<td><code>in ^#1 as #2</code></td>
<td>imports VALUE(^#1) into ^#1 by protocol #2</td>
</tr>
<tr>
<td><code>out ^#1 as #2</code></td>
<td>exports ^#1 into VALUE(^#1) by protocol #2</td>
</tr>
<tr>
<td><code>in ^#1 as ^#2</code></td>
<td>imports VALUE(^#1) into ^#1 by protocol ^#2</td>
</tr>
<tr>
<td><code>out ^#1 as ^#2</code></td>
<td>exports ^#1 into VALUE(^#1) by protocol ^#2</td>
</tr>
<tr>
<td><code>^#1='&lt;string&gt;'</code></td>
<td>sets VALUE(^#1) to &lt;string&gt;</td>
</tr>
<tr>
<td><code>^#1=vcopy of ^#2</code></td>
<td>sets VALUE(^#1) to VALUE(^#2)</td>
</tr>
</tbody>
</table>
Example for an STM program:

```plaintext
program test
process setup
  ~core.x=const 1
  ~core.five=const 5
goto loop
process loop
  ~x ++
  ~test=(~x==~five)
  if ~test goto end
  goto loop
process end
  stop
start setup
```
program test
process setup
    ^core.x=const 1
    ^core.five=const 5
    goto loop
process loop
    ^x ++
    ^test=(^x==^five)
    if ^test goto end
    goto loop
process end
    stop
start setup
program test
process setup
  \^core.x=const 1
  ^core.five=const 5
  goto loop
process loop
  \^x ++
  ^test=(^x==^five)
  if ^test goto end
  goto loop
process end
  stop
start setup
The STM: Example

program test
process setup
  ^core.x=const 1
>  ^core.five=const 5
goto loop
process loop
 ^x ++
^test=(^x==^five)
if ^test goto end
goto loop
process end
 stop
start setup
program test
process setup
  ^core.x=const 1
  ^core.five=const 5
> goto loop
process loop
  ^x ++
  ^test=(^x==^five)
  if ^test goto end
  goto loop
process end
  stop
start setup
The STM: Example

program test
process setup
    ^core.x=const 1
    ^core.five=const 5
    goto loop
process loop
    > ^x ++
    ^test=(^x==^five)
    if ^test goto end
    goto loop
process end
    stop
start setup
The STM: Example

program test
process setup
  ^core.x=const 1
  ^core.five=const 5
goto loop
process loop
  ^x ++
  > ^test=(^x==^five)
  if ^test goto end
  goto loop
process end
  stop
start setup
program test
process setup
  ^core.x=const 1
  ^core.five=const 5
goto loop
process loop
  ^x ++
  ^test=(^x==^five)
> if ^test goto end
  goto loop
process end
  stop
start setup
program test
process setup
  ^core.x=const 1
  ^core.five=const 5
  goto loop
process loop
  ^x ++
  ^test=(^x==^five)
  if ^test goto end
>  goto loop
process end
  stop
start setup
program test
process setup
  \textasciitilde core.x=const 1
  \textasciitilde core.five=const 5
  goto loop
process loop
  \textasciitilde x ++
  > \textasciitilde test=(\textasciitilde x==\textasciitilde five)
  if \textasciitilde test goto end
  goto loop
process end
  stop
start setup
program test
process setup
  ^core.x=const 1
  ^core.five=const 5
goto loop
process loop
  ^x ++
  ^test=(^x==^five)
> if ^test goto end
  goto loop
process end
  stop
start setup
The STM: Example

program test
process setup
  ^core.x=const 1
  ^core.five=const 5
  goto loop
process loop
  ^x ++
  ^test=(^x==^five)
  if ^test goto end
  goto loop
process end
>  stop
start setup
We have a STM program that simulates a usual TM.

It has 54 lines, uses 15 different commands.
The universal semantic Turing machine (USTM) is a program that can simulate every other STM program.

We can trust an implementation of the STM: – check correctness of USTM (232 lines), and – program and simulated program have same output.
Simulation is straightforward:

Program to simulate is data in its core. The USTM identifies the command and simulates changes in the memory, flow control etc.

The USTM simulates any STM-program in linear time.

Simplicity of the USTM was design criterion for STM language.
Towards a controlled natural language for arbitrary mathematics

A first step towards our controlled natural language: a context-free grammar from a textbook on calculus and linear algebra.
The TeX-file was automatically processed (formulas replaced by the word FORMULA, etc.)

→ A list of about 4000 unique sentence templates.
→ A lexicon of about 1500 German basic words.

By identifying common structures in the sentences: sentence grammar with about 1000 production rules.

From the list of words: a simple morphological grammar (later to be replaced by the Grammatical Framework?),
ALA-grammar: Example

"defsentence = "v heisst "o "o.
"defsentence = "o "v heissen "o.
"defsentence = "v heisst dann "o.
"defsentence = "v heisst dann "v.
"defsentence = "o ist "o mit "f
"defsentence = "o "o ist "o "o.
"defsentence = "o von "v heissen "o.
"defsentence = fuer "f heisst "v "o.
"defsentence = "v heisst "p, "if "f.
"defsentence = "qt solche "o heisst "o.
"defsentence = wir schreiben "f falls "f.
"defsentence = "v bezeichnet "o aller "v.
"defsentence = "o sind "o der form "f.
"defsentence = man nennt "r "o von "v.
"defsentence = "if "f heisst "v ein "o.
"defsentence = "o der form "f heisst "o.
"defsentence = "v heisst "o 'art "o "v.
"defsentence = ein "o heisst "p, "if "f.
"defsentence = statt "v schreibt man auch "v.
"defsentence = "o wird kurz als "o bezeichnet.
"defsentence = "qt "o mit "o "v ist "o.
OR-Library:

Problem Formulation

Parsing

Representation

Processing

Reformulation

Variety of solvers

To figure out representation of optimization problems: OR-Library (test data sets for Operations Research problems).

The OR-Library contains data sets and references to detailed problem descriptions.
OR-Lib: Process

Process (at present):

- extract essential parts (by human)
- represent in semantic matrix (by human, but computer-aided)
- create readable output (by machine)
Description quoted from
E. Falkenauer:
A *Hybrid Grouping Genetic Algorithm for Binpacking*

A Bin Packing Problem (BPP) is defined as follows ([Garey and Johnson, 79]): given a finite set $O$ of numbers (the item sizes) and two constants $C$ (the bin’s capacity) and $N$ (the number of bins), is it possible to ’pack’ all the items into $N$ bins, i.e. does there exist a partition of $O$ into $N$ or less subsets, such that the sum of the elements in any of the subsets doesn’t exceed $C$?
binp.type = optimization problem

binp.problem.generic = bin packing problem
    .attribute = 1-dimensional
    .ref = [GareyJohnson]

binp.instance.nconc = 4
    .1.concept = bin capacity $
        .name = C
        .in = "NN"
    .2.concept = number $ of items
        .name = L
        .in = "NN"
    .3.concept = item size $
        .name = m
        .in = "NN"
    .4.name = M
        .is_a = set
        .elements.in = "NN"
            .concept = item size
        .property.nprop = 1
            .1 = "|M| = L"

binp.feasible.nconc = 1
    .1.concept = packing $
        .name = P
        .property.nprop = 2
            .1 = "P is a partition of M"
            .2 = "forall B in P: sum_{m in B} m <= C"
        .elements.concept = bin $
            .name = B

binp.optimal.nopt = 1
    .1.name = Formulation 1
        .mode = min
        .variables = P
        .objective = N
        .property.nprop = 1
            .1 = "N = |P|"
1-dimensional bin packing problem

An instance of a 1-dimensional bin packing problem [GareyJohnson] is defined by two positive integers, the bin capacity \( C \), the number \( L \) of items, together with a set \( M \) with \( |M| = L \). The elements of \( M \) are called the item sizes.

Given an instance, a packing \( P \) satisfies: \( P \) is a partition of \( M \) and for all \( B \in P \): \( \sum_{m \in B} m \leq C \). An element \( B \) of \( P \) is called a bin.

Problem:
Find a \( P \) that minimizes \( N \) under the constraint \( N = |P| \).
To further test the universality of the semantic representation: parse problems from the TPTP library ("Thousands of Problems for Theorem Provers") and represent them in semantic matrix.

A second translator takes the result and produces readable \texttt{\LaTeX} documents.
cnf(unordered_pair_3,axiom,
    ( ~ member(X,unordered_pair(Y,Z))
    | X = Y
    | X = Z )).

cnf(ordered_pair,axiom,
    ( ordered_pair(X,Y) =
       unordered_pair(singleton_set(X),unordered_pair(X,Y)) )).
Name: unordered_pair_3
Form: cnf
Domain: SET
Role: axiom
File: SET016-1.p

¬(X ∈ {Y, Z}) ∨ X = Y ∨ X = Z

Name: ordered_pair
Form: cnf
Domain: SET
Role: axiom
File: SET016-1.p

(X, Y) = {{X}, {X, Y}}
Interface to Naproche

The grammar of the Naproche-language enables us to represent Naproche-texts.

From represented texts, we can create Naproche-texts again or output in our controlled natural language.
Axiom.
There is no $y$ such that $y \in \emptyset$.

Axiom.
For all $x$ it is not the case that $x \in x$.

Define $x$ to be transitive if and only if for all $u$, $v$, if $u \in v$ and $v \in x$ then $u \in x$.
Define $x$ to be an ordinal if and only if $x$ is transitive and for all $y$, if $y \in x$ then $y$ is transitive.
Naproche: Aided creation of records

\begin{verbatim}
var_u=mkvar('u');
var_v=mkvar('v');
var_x=mkvar('x');
var_y=mkvar('y');
emptyset=mkcon('\emptyset');
ord=mkcon('Ord');
trans=mkcon('Trans');
transy=mkexp('OF',\{trans,var_y\});
form_def1 = mkexp('FORALL',imp,\{var_u,var_v\});
form_def2 = mkexp('IMPLIES',\{and,transy\});
form_th1 = mkexp('OF',\{ord,emptyset\});
ass1 = mkassumption(form_ass1);
s1 = mksentence(ass1);
ass2 = mkassumption(form_ass2);
s2 = mksentence(ass2);
def1 = mkdefinition(form_trans,\[],labsearch('property'),\[],form_def1);
s3 = mksentence(def1);
def2 = mkdefinition(form_ord,\[],labsearch('property'),\[],form_def2);
s4 = mksentence(def2);
s5 = mksentence(form_th1);
\end{verbatim}
Assume that $\neg \exists y : y \in \emptyset$. Assume that $\forall x : \neg x \in x$. Define $\text{Trans}(x)$ if and only if $\forall y : y \in x \rightarrow \text{Trans}(y)$. Define $\text{Ord}(x)$ if and only if $\text{Trans}(x) \wedge \forall y : y \in x \rightarrow \text{Trans}(y)$.
We assume that there is no $y$ such that $y \in \emptyset$. We assume that for all $x$, not $x \in x$. We say that $Trans(x)$ if for all $u$ and $v$, $u \in v$ and $v \in x$ implies $u \in x$. We say that $Ord(x)$ if $Trans(x)$ and for all $y$, $y \in x$ implies $Trans(y)$.
Thank you for your attention!

More information, these slides and preprints can be found:

http://www.mat.univie.ac.at/~schodl/
http://www.mat.univie.ac.at/~neum/FMathL.html#MoSMath