Manual for XLE+Glue system

Mary Dalrymple[1] and Agnieszka Patejuk[1,2] and Mark-Matthias Zymla[3]
[1]Centre for Linguistics and Philology, University of Oxford
[2]Institute of Computer Science, Polish Academy of Sciences
[3]University of Konstanz
`mary.dalrymple@ling-phil.ox.ac.uk`
`agnieszka.patejuk@gmail.com`
`mark-matthias.zymla@uni-konstanz.de`

Contents

The computational glue system XLE+Glue allows the user to specify meaning constructors (Dalrymple, 2001; Dalrymple et al., 1993, 2019) in an AVM format for input to a theorem prover. The system relies on the Xerox Linguistic Environment (XLE; Crouch et al. 2017), a platform for development of computational LFG grammars, and the Glue semantics workbench (GSWB; Meßmer and Zymla 2018), a recently developed and maintained glue prover. The system provides the means to test and explore co-descriptive approaches to glue semantics computationally.

The XLE+Glue pipeline integrates functionalities from various programming languages. First, Tcl provides the functionality for the XLE user interface. We provide an `xlerc` and a Tcl file that set up all components of the pipeline: the grammar, the rewrite component, and the GSWB. The UI is modified to allow the user to call the XLE+Glue analysis from the XLE user interface. Second, SWI-Prolog is used to rewrite the AVM encoding of meaning constructors into the appropriate string format.[1] Third, Java in which the GSWB is implemented.

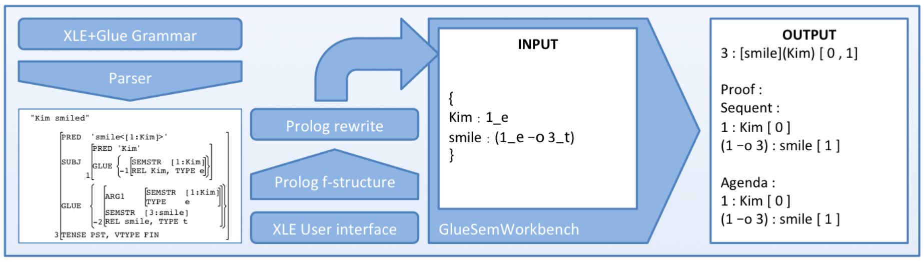
The XLE+Glue system is available in this repository:

`https://github.com/Mmaz1988/xle-glueworkbench-interface`

Please consult the README file in the repository for an overview of the system.

1 Overview

The Glue Semantics Workbench is a glue prover written in Java that can be used out of the box (Meßmer and Zymla, 2018). It takes as input a set of glue premises that are written in a simple string format which adheres to the representational conventions of linear logic (specifically, the quantified implicational fragment of linear logic; see Section 5 for a discussion of the notation used). A schematic example of the input and output of the prover is given on the right-hand side of the picture below. The result, given in the first line of the output, consists of the resulting glue semantics representation and a list of indices that refer to the glue premises listed in the agenda (the agenda is separate from the sequent since the sequent may need to be modified to mimic implication introduction steps for assumptions; see Meßmer and Zymla 2018 for details). Thus, the list of indices indicates whether all resources have been used during the search for a valid proof.



```
"Kim smiled"

 [PRED   'smile<[1:Kim]>'
          [PRED 'Kim'
 SUBJ     [GLUE {-1[SEMSTR  [1:Kim]
                    REL Kim, TYPE e]}]
         1
 GLUE    {-2[ARG1     [SEMSTR  [1:Kim]
                       TYPE    e]
             SEMSTR  [3:smile]
             REL smile, TYPE t]}
3TENSE PST, VTYPE FIN]
```

```
INPUT

{
Kim : 1_e
smile : (1_e -o 3_t)
}
```

```
OUTPUT
3 : [smile](Kim) [ 0 , 1]

Proof :
Sequent :
1 : Kim [ 0 ]
(1 -o 3) : smile [ 1 ]

Agenda :
1 : Kim [ 0 ]
(1 -o 3) : smile [ 1 ]
```

The glue prover is embedded in our XLE+Glue pipeline. This pipeline, shown on the left-hand side of the picture, converts glue premises encoded in an attribute-value matrix (AVM) format to the GSWB prover input format. This is made

[1]We did not use the XLE transfer system, since it is not available for newer XLE versions.

possible by the introduction of a special GLUE attribute whose value is a set of meaning constructors in AVM format. Each meaning constructor is rewritten from the attribute-value format to a format suitable for input to GSWB. The Prolog rewrite script and GSWB run as separate processes, integrated into XLE. As a result, XLE+Glue allows the user to call the semantic analysis directly from the XLE output windows.

2 Encoding of glue premises as AVMs

Glue meaning constructors consist of a semantic side (any semantic formalism) and glue side (expression of linear logic over linguistic structures of a given type). F-structure metavariables in lexical entries are instantiated to indices representing particular f-structures, for instance:

(1) a. $\mathit{Kim} : \uparrow_e \quad \Rightarrow \quad \mathit{Kim} : k_e$
b. $\mathit{smile} : (\uparrow \textsc{subj})_e \multimap \uparrow_t \quad \Rightarrow \quad \mathit{smile} : k_e \multimap s_t$

In this section, we make several assumptions for ease of presentation. First, we assume simple untyped meaning representations such as *Kim* and *smile*; see Section 5 for discussion of the ways in which meanings can be represented in the system, including as terms of the typed lambda calculus. Second, we assume that the glue side of meaning constructors refers to f-structures rather than their semantic projections (as in our sample grammar `glue-basic.lfg`, described in Section 4). However, it is also possible for meaning constructors to refer to other linguistic structures in our system (as in our sample grammar `glue-basic-semstr.lfg`, which includes a semantic projection).

2.1 Meaning constructors and their AVM encoding

(2-a) illustrates the meaning constructor for the proper name *Kim* in the standard format, and (2-b) illustrates the corresponding attribute-value encoding, where the AVM encoding the meaning constructor appears as a member of the GLUE set. In (2) and the rest of this section, we follow the usual notational convention of referring to f-structures by means of letters like k and s, but in later sections of this document we will use numbers instead, since the Prolog rewrite script relies on the numeric indices assigned to f-structures in the Prolog output format of XLE. In the examples in this section, each GLUE set contains only one meaning constructor, but in other cases several meaning constructors appear as members of the GLUE set, as we show in Section 2.3.

(2) a. Kim: k_e

b. $$k : \left[\begin{array}{ll} \textsc{pred} & \text{‘}\textsc{Kim}\text{’} \\ \textsc{glue} & \left\{ \left[\begin{array}{ll} \textsc{mr} & \textsc{Kim} \\ \textsc{semstr} & k \\ \textsc{type} & e \end{array}\right] \right\} \end{array}\right]$$

In the attribute-value encoding, each glue meaning constructor minimally consists of three attributes. The value of MR is the semantic (left-hand) side of the meaning constructor, while SEMSTR and TYPE specify the glue (right-hand) side: SEMSTR points to the relevant f-structure, while TYPE specifies SEMSTR's type (e or t). In XLE notation, the f-structure constraints contributed by a proper name like "Kim" are:

(3)
```
(^ PRED)='Kim'
(%sstr MR)=Kim
(%sstr SEMSTR)=^
(%sstr TYPE)=e
%sstr $ (^ GLUE)
```

`%sstr` is a local name[2] used to construct an f-structure containing attributes specifying the glue meaning constructor (`MR`, `SEMSTR`, `TYPE`). This f-structure is added to the `GLUE` set by specification of the constraint `%sstr $ (^ GLUE)`.

A glue meaning constructor may also involve implication, as for a verb like *smile* in (4), where a resource needs to be consumed in order to produce another resource. The standard meaning constructor for the verb *smile* is given in (4-a), and the AVM translation is given in (4-b). In the AVM encoding, ARG1 is the first resource to be consumed, ARG2 the second, etc. The resources to be consumed are specified using the SEMSTR and TYPE attributes.

(4) a. *smile*: $k_e \multimap s_t$

b. $$s:\left[\begin{array}{ll}\text{PRED} & \text{`SMILE<SUBJ>'}\\ \text{SUBJ} & k:[\ ]\\ \text{GLUE} & \left\{\left[\begin{array}{ll}\text{MR} & \text{SMILE}\\ \text{ARG}1 & \left[\begin{array}{ll}\text{SEMSTR} & k\\ \text{TYPE} & e\end{array}\right]\\ \text{SEMSTR} & s\\ \text{TYPE} & t\end{array}\right]\right\}\end{array}\right]$$

This means that the constraints contributed by the verb *smile* are:

(5)
```
(^ PRED)='smile<(^ SUBJ)>'
(%sstr MR)=smile
(%sstr SEMSTR)=^
(%sstr TYPE)=t
(%sstr ARG1 SEMSTR)=(^ SUBJ)
(%sstr ARG1 TYPE)=e
%sstr $ (^ GLUE)
```

The f-structure for the sentence "Kim smiles" is shown in (6). The Prolog rewrite component collects up all of the premises in the GLUE sets, rewrites each premise into a format suitable for input to the prover (as shown in (7)), and passes the complete set of premises to the prover.

[2]`https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#N4.1.6`

(6) $$s : \left[\begin{array}{ll} \text{PRED} & \text{‘SMILE<SUBJ>’} \\ \text{SUBJ} & k : \left[\begin{array}{ll} \text{PRED} & \text{Kim} \\ \text{GLUE} & \left\{ \left[\begin{array}{ll} \text{MR} & \text{Kim} \\ \text{SEMSTR} & k \\ \text{TYPE} & e \end{array}\right] \right\} \end{array}\right] \\ \text{GLUE} & \left\{ \left[\begin{array}{ll} \text{MR} & \text{SMILE} \\ \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & k \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & s \\ \text{TYPE} & t \end{array}\right] \right\} \end{array}\right]$$

(7)
```
{
Kim :  k_e
smile :  k_e -o s_t
}
```

2.2 Universal quantification over meaning constructors

Example (8) illustrates the use of the attribute FORALL[3] for universal quantification in meaning constructors.

(8) a. *every*: $\forall F.(p_e \multimap p_t) \multimap (m_e \multimap F_t) \multimap F_t$

b. $$m : \left[\begin{array}{ll} \text{PRED} & \boxed{1}\ p \\ \text{GLUE} & \left\{ e : \left[\begin{array}{ll} \text{MR} & \textit{every} \\ \text{ARG1} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & t \end{array}\right] \\ \text{ARG2} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & m \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & F \\ \text{TYPE} & t \end{array}\right] \\ \text{FORALL} & F \\ \text{SEMSTR} & F \\ \text{TYPE} & t \end{array}\right] \right\} \end{array}\right]$$

In (8)b, the f-structure labeled e is the attribute-value encoding of the meaning constructor for “every” given in (8)a. It has two arguments: ARG1 represents the restriction of the quantifier, and ARG2 represents its scope. The value of the ARG1 attribute encodes the implication $(p_e \multimap p_t)$ (where p is the value of the PRED attribute of the noun phrase, as shown in (8)), which corresponds to a common noun meaning.[4] The value of the ARG2 attribute encodes an

[3]To display the attribute FORALL in XLE, select “constraints” in “Views” menu (or press “c”) in f-structure window (or other relevant window, for instance s-structure).

[4] Our sample grammar `glue-basic.lfg` makes the non-standard assumption that the meaning of a common noun is a function from its PRED value of type e to its PRED value of type t; that is, a common noun like “person” has a lexical entry of the following form:

person: $(\uparrow \text{PRED})_e \multimap (\uparrow \text{PRED})_t$

implication from m_e to F_t, where F is a variable bound by a universal quantifier, representing the scope of the quantifier, which is freely chosen. At the top level we have a new attribute FORALL, which encodes the universal quantifier $\forall$ in (8)a.

2.3 Multiple meaning constructors contributed by a single word

Example (9) illustrates the encoding of the meaning constructor for the quantifier *everyone*, decomposed into a meaning constructor contributing the "every" part of the meaning and another meaning constructor contributing the "person" part. This allows for the modification of the restriction of the quantifier in examples like *everyone who smiled*, and illustrates the possibility for a single word to contribute more than one meaning constructor to the GLUE set.

(9) a. *every*: $\forall F.(p_e \multimap p_t) \multimap (m_e \multimap F_t) \multimap F_t$

person: $p_e \multimap p_t$

b. $$m : \left[\begin{array}{ll} \text{PRED} & \boxed{1}\ \textit{everyone} \\ \text{GLUE} & \left\{ \begin{array}{l} e : \left[\begin{array}{ll} \text{MR} & \textit{every} \\ \text{ARG1} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & t \end{array}\right] \\ \text{ARG2} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & m \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & F \\ \text{TYPE} & t \end{array}\right] \\ \text{FORALL} & F \\ \text{SEMSTR} & F \\ \text{TYPE} & t \end{array}\right] \\ n : \left[\begin{array}{ll} \text{MR} & \textit{person} \\ \text{ARG1} & \left[\begin{array}{ll} \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & e \end{array}\right] \\ \text{SEMSTR} & \boxed{1} \\ \text{TYPE} & t \end{array}\right] \end{array} \right\} \end{array}\right]$$

In (9)b, the f-structure labeled e is the same as the attribute-value encoding of the meaning constructor for *every* given in (8) in the previous section. The second member of the set, labeled n, is the same as the contribution we would expect for the common noun *person*.

This is done for simplicity, to avoid the introduction of attributes encoding VAR and RESTR as in standard treatments, and is not a necessary feature of the implementation. We discuss our sample grammars in more detail in Section 4.

3 Templates for meaning constructors

The sample grammar files `glue-basic.lfg`, `glue-basic-semstr.lfg`, and `glue-basic-semparser.lfg` provide a set of templates[5] for encoding meaning constructors which may be generally useful, though it is of course possible for grammar writers to develop their own set of templates or to modify the sample templates as needed. We describe the basic templates here; more discussion of the sample grammars can be found in Section 4 and in the comments in the grammar files.

3.1 The basic definitions

All of the templates which are used in defining meaning constructors in the sample grammars call the two basic templates `GLUE-SEMSTR` and `GLUE-MEANING`. The template `GLUE-SEMSTR` specifies an attribute-value structure `SSTR` as having the value `FSTR` for the attribute `SEMSTR` and the value `TY` for the attribute `TYPE`. The attributes `SEMSTR` and `TYPE` and their values must appear in all AVM meaning constructors and argument specifications, to identify the relevant linguistic resource and its type. (The argument names `FSTR` and `SSTR` are merely mnemonic; the linguistic resource need not be an f-structure.)

```
(10)    GLUE-SEMSTR(FSTR SSTR TY) =  (SSTR SEMSTR) = FSTR
                                     (SSTR TYPE) = TY.
```

The template `GLUE-MEANING` specifies an attribute-value structure `SSTR` as having the value `P` for the attribute `MR`. This attribute corresponds to the left-hand (meaning) side of the meaning constructor, and must appear once, at the top level of all AVM meaning constructors.

```
(11)    GLUE-MEANING(SSTR P) =  (SSTR MR) = P.
```

3.2 Non-implicational meaning constructors: Proper names

In the sample grammar `glue-basic.lfg`, the lexical entry for the proper name *Kim* is as in (12):

```
(12)    Kim   N   *  @(PROPERNOUN Kim).
```

The sample grammar `glue-basic.lfg` provides the template in (13) for proper names. It defines the f-structure `PRED` value, and calls the template `GLUE-PROPERNOUN` to define the meaning constructor in AVM format, passing in the argument `P`.

```
(13)    PROPERNOUN(P) =  (^ PRED) = 'P'
                         @(GLUE-PROPERNOUN P).
```

In `glue-basic.lfg`, the argument `P` of `PROPERNOUN` is used to construct the f-structure semantic form as well as appearing as the value of the `MR` attribute in the AVM meaning constructor. If it is desirable for the f-structure `PRED` value to

[5]General documentation of the use of templates in XLE is available here: `https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/walkthrough.html#W.templates`

be different from the MR value of the AVM meaning constructor, the PROPERNOUN template would have to be defined to take two arguments, one providing the PRED value and the other providing the MR value.

GLUE-PROPERNOUN simply calls GLUE-REL0-MC (mnemonic for "meaning constructor for relation of arity 0": in other words, a meaning constructor that requires no arguments). It specifies the first and second arguments of the template as ^ and e for all proper nouns, and passes in the value of P as the third argument.

```
(14)    GLUE-PROPERNOUN(P) =  @(GLUE-REL0-MC ^ e P).
```

In the glue-basic.lfg grammar, it would also have been possible for the PROPERNOUN template to call GLUE-REL0-MC directly, providing the arguments ^ and e. The intermediate template GLUE-PROPERNOUN allows for the possibility that in scaling up to a more complete grammar, additional semantic specifications may be associated with the GLUE-PROPERNOUN template, besides the definition of the meaning constructor.

The definition of GLUE-REL0-MC is:

```
(15)    GLUE-REL0-MC(FSTR TY P) =  @(GLUE-SEMSTR FSTR %sstr TY)
                                   @(GLUE-MEANING %sstr P)
                                   %sstr $ (FSTR GLUE).
```

This template calls the two basic templates GLUE-SEMSTR and GLUE-MEANING. The call to GLUE-SEMSTR specifies properties of the AVM %sstr: it has an attribute SEMSTR whose value is FSTR, and it has an attribute TYPE whose value is TY. The call to GLUE-MEANING provides the value P for the attribute MR in %sstr. The final line requires the f-structure %sstr to appear as a member of the GLUE set in the f-structure FSTR.

When the template GLUE-REL0-MC is called with arguments ^, e, and Kim, an AVM %sstr is created which corresponds to the simple meaning constructor *Kim*:$\uparrow_e$. This AVM has three attributes: SEMSTR, whose value is ^; TYPE, whose value is e; and MR, whose value is Kim. The final line of this template specifies that %sstr is a member of the GLUE set in the f-structure FSTR. Thus, the template call @(PROPERNOUN Kim) produces the f-description given in (3).

3.3 Meaning constructors requiring arguments: Intransitive verbs

In glue-basic.lfg, the lexical entry for the intransitive verb *smiled* is:

```
(16)    smiled   V   *  @(VERB-SUBJ smile)
                        @VPAST.
```

The template VPAST specifies a past tense feature in the f-structure; we do not discuss this template here. The template VERB-SUBJ is defined as:

```
(17)    VERB-SUBJ(P) =  (^ PRED) = 'P<(^ SUBJ)>'
                        @(GLUE-VERB-SUBJ P).
```

As with the proper name template described in the previous section, the template argument P is used to define both the f-structure semantic form and the

MR value of the AVM meaning constructor. If this is not desirable, the template VERB-SUBJ should be defined to take two arguments, one specifying the semantic form and the other the value of the MR feature in the AVM meaning constructor.

The template GLUE-VERB-SUBJ is defined as:

```
(18)    GLUE-VERB-SUBJ(P) =  @(GLUE-REL1-MC (^ SUBJ) e ^ t P).
```

As with the GLUE-PROPERNOUN template, the GLUE-VERB-SUBJ template simply calls GLUE-REL1-MC (mnemonic for "meaning constructor for relation of arity 1": in other words, a meaning constructor that requires one argument). In scaling up to a more complete grammar, there may be additional semantic specifications associated with GLUE-VERB-SUBJ.

The template GLUE-REL1-MC is defined as:

```
(19)    GLUE-REL1-MC(A1 A1TY FSTR TY P) =    @(GLUE-SEMSTR FSTR %sstr TY)
                                             @(GLUE-SEMSTR A1 (%sstr ARG1) A1TY)
                                             @(GLUE-MEANING %sstr P)
                                             %sstr $ (FSTR GLUE).
```

The first, third, and fourth lines of this template are the same as for the template GLUE-REL0-MC: they specify that the meaning constructor in the GLUE set of this verb is called %sstr, that it has an attribute SEMSTR with value FSTR, and that it has an attribute TYPE with value TY. The additional specification in the second line adds an attribute ARG1 to the structure, whose value for the SEMSTR feature is A1, and whose value for the TYPE feature is A1TY. When the template GLUE-REL1-MC is called with arguments (^ SUBJ), e, ^, t, and smile, the resulting f-description is as in (5).

4 Sample grammars

The repository contains several XLE+Glue sample grammars. These build on the templates discussed in Section 3, and they also provide templates for further basic constructions, in particular, adjectival modifiers and verbs taking sentential complements. Each sample grammar is documented by means of comments describing its rules, lexical entries, and templates. The sample grammars that are currently available in the repository are:

glue-basic.lfg Corresponds closely to the discussion in this manual. GLUE attributes appear at f-structure, and meaning constructors refer to f-structures.

glue-basic-semstr.lfg Similar to glue-basic.lfg, but GLUE attributes appear at semantic structure and refer to semantic structures.

glue-basic-semparser.lfg Similar to glue-basic.lfg in that GLUE attributes appear at f-structure. Meaning representations are typed formulas of the lambda calculus, as described in Section 5.5. When parsing with this grammar, please follow the instructions in Section 5.5 to modify the xlerc file to activate the semantic parser.

5 Format for semantic representations

5.1 Special characters in XLE

The following characters need to be escaped in XLE using a backquote (`` ` ``):[6]

- bound: `< > ( ) { } [ ]`
- oper: `` * + - & | \ / ~ ^ $ ` # ' ``
- space: [space] `\t` [tab] `\r` [carriage return]
- mark: `, : _ ? = ! % @ . ;`

5.2 Using `CONCAT`

XLE provides the built-in template `CONCAT`[7] which is used to concatenate an arbitrary number of elements – `CONCAT` joins all its arguments except the last one which is a local name (variable) containing the result of `CONCAT`. For instance: `@(CONCAT a b c %OUTPUT)` results in `abc` as the value of `%OUTPUT` local name.

To avoid potential issues, it is safest to pass all special characters listed in section 5.1 as separate arguments to `CONCAT`.[8]

5.3 Using `backquote-region.el`

`backquote-region.el` is an emacs module aimed at making it easier to format strings for use with XLE+Glue. It provides two commands:

- `backquote-region` escapes characters listed in section 5.1 with a backquote (`` ` ``), so that the resulting string can be used as the value of the attribute `MR`.
- `backquote-space-region` escapes the same characters as `backquote-region`, but it also surrounds them with whitespace, so that the resulting string can be passed to `CONCAT` template without issues (see section 5.2).

To use `backquote-region.el`, put it in a directory (for instance: `~/elisp`), and add the following two lines to your `.emacs`:

```
(add-to-list 'load-path "~/elisp")
(load-library "backquote-region")
```

[6]Source: John Maxwell, PC.

[7]`https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#CONCAT`

[8]Minimally, it seems the following must be passed as separate arguments to `CONCAT`:

- every underscore (_)
- `[` when it is the first character of the first argument of `CONCAT`

5.4 System without the semantic parser

The demo grammars available on github serve illustrative purposes in that the semantic representations used therein are simply strings and treated as such by the workbench. This is to show that any (string-based) format of semantic representation is compatible with XLE+Glue. The simple format expresses functional application between arbitrary strings by wrapping the argument in parentheses and concatenating functor and argument.

(20) Combining $(1 \multimap 0)$: smile [0] and 1 : Kim[1]
to: 0 : smile(Kim)[0, 1]

Semantic parsing, discussed in section 5.5, requires additional machinery within the lexical entries/templates to encode the desired meaning representations, since the semantics are based on several special characters which need to be escaped within the XLE. However, this interferes with the unification of variables. More details can be found in the next section.

5.5 Using the semantic parser

The Glue semantics workbench provides a semantic parser for lambda expressions supporting alpha- and beta-conversion. This section describes how to activate semantic parsing for your grammar and the guidelines for writing lambda expressions that can be parsed by the semantic parser.

5.5.1 Integrating the semantic parser

Integrating the semantic parser into your grammar is simple: In your `xlerc` file change the value for the variable `semParser` to `1` instead of `0`. **Warning:** Once activated, unparsable input on the meaning side will result in a parsing error.

5.5.2 Specifications for semantic representations

The semantic parser parses lambda expressions that scope over first order logic formulas. First-order predicates are encoded in the classic prefix notation. There is no convention with respect to the casing of predicates or constants. Furthermore, constants and predicates are not themselves typed.

(21) `like(mary,semantics)`

5.5.2.1 Lambda expressions:. Lambda expressions are introduced via a scope defining bracket and a slash, followed by the variable that the lambda operator binds. Variables require a type declaration to be distinguished from constants. This is done by using an underscore and one of the following types.[9]

(22) **Available types:** e, s, v, t

Bound occurrences of a variable should not be typed again. The scope of the lambda function is separated from the binder via a dot. It can be any kind of well-formed (lambda) expression.

[9]Contact Mark-Matthias Zymla if you require support for a new type.

(23) a. `[/x_e.sleep(x)]`
b. `[/x_e.[/w_s.sleep(x,w)]]`
c. `[/P_<e,t>.[/Q_<e,t>.[/x_e.(P(x) & Q(x))]]]`

5.5.2.2 Complex types:. Complex types consist of atomic types, commas and angular brackets, e.g.:

(24) a. $< e, t >$
b. $<< e, t >, t >$
c. $< e, < s, t >>$

5.5.2.3 Logic operators:. The basic logic operators $\wedge, \vee$ and $\rightarrow$ can be used as infix operators (see (25)a - (25)c), although their scope has to be defined via brackets or parentheses. Brackets can indicate operator scope and quantifier scope simultaneously (see (25)d). Other operators must be encoded as first-order logic predicates, i.e. in prefix notation (see (25)e).

(25) a. **Logical 'and' (&):** `(P(x) & Q(x))`
b. **Logical 'or'(v):** `(P(x) v Q(x))`
c. **Logical 'implication'(->):** `(P(x) -> Q(x))`
d. **Variant with brackets:** `Ex_e[P(x) & Q(x)]`
e. **Prefix notation:** `equals(x,y)`

5.5.2.4 Quantifiers:. Quantifiers are introduced via the upper case letters `A` and `E`. Their scope variable is introduced with an appropriate typing. The scope is defined via brackets.

(26) a. `Ex_e[dog(x) & bark(x)]`
b. `Ax_e[cat(x) -> sleep(x)]`

5.5.2.5 Functional application:. Functional application steps such as in the lambda terms for quantifiers are determined contextually, e.g., `P(x)` and `Q(x)` in the example below.

(27) `[/P_<e,t>.[/Q_<e,t>.Ex_e[P(x) & Q(x)]]]`

5.5.2.6 Lexical entries in XLE:. We briefly present a possible way of encoding appropriate semantic representations in XLE. As pointed out in section 5.2, the challenge is to overcome the issue of escaping certain characters while preserving variable unification. One way to achieve this is to use the `@CONCAT` template to separate variables and string parts of a meaning representation.

(28)
```
GLUE-REL1-MC(A1 A1TY FSTR TY P) =   @(GLUE-SEMSTR FSTR %sstr TY)
                                    @(GLUE-SEMSTR A1 (%sstr ARG1) A1TY)
                                    @(CONCAT '[ '/x A1TY '.  P '(x') '] %MEANING)
                                    @(GLUE-MEANING %sstr %MEANING)
                                    %sstr $ (FSTR GLUE).
```

This approach is still being tested. Further feedback for encoding meaning representations according to the specifications given above is welcome.

5.5.2.7 Final note:. The current version of the semantic parser is completely independent of the glue side, which means that type restrictions need to be manually added. Furthermore, eta-conversion is not possible. This may change in the future.

6 Prolog rewrite component

The Prolog rewrite component takes the Prolog output of an XLE parse as input.[10] It does not rely on any particular assumptions about where the GLUE attributes must appear; as discussed in Section 4, GLUE attributes and their values are a part of f-structure in our sample grammar `glue-basic.lfg`, while our sample grammar `glue-basic-semstr.lfg` places them at s-structure. Indeed, the system works even if some GLUE attributes appear at f-structure, and others appear in other structures. It is also not necessary for the meaning constructors to be distributed in any particular way in the structure in which they appear; the system simply gathers up all members of every GLUE set in the input representation, rewrites them into the standard format, and passes the resulting set of standard-format meaning constructors to the prover. Thus, decisions about which structure hosts the GLUE attribute and its values should be made on the basis of linguistic considerations, and are not determined by properties of the implementation. Attributes other than the GLUE attributes and their values are ignored and discarded.

Each element of the GLUE set provides one premise: as explained above, the value of MR provides the semantic side, while SEMSTR, TYPE, and the ARG1...N attributes provide the glue side.[11]

The SEMSTR labels are instantiated to the numeric labels provided by XLE. Because the numeric indexing for semantic forms in the Prolog output format is independent of the numeric indexing for other structures (for example, there may be an f-structure with numeric index 1 and also a semantic form with numeric index 1 in the same f-structure), the numeric index of a semantic form is additionally prefixed with an S, e.g., S1, to ensure uniqueness of indices. As described above, the FORALL attribute F is used as reference to linear quantification. Different linear quantifiers in a computation are distinguished by combining the label F with its unique f-structure index.

Example (29-a) shows the output produced by the rewrite component for a generalized quantifier encoded in AVM format as (29-c) (as discussed in example (8)), corresponding to the standard format meaning constructor in (29-b). All of the conventions discussed above are illustrated in (29-a): `F11` is bound by a universal quantifier, `s1` refers to the semantic form whose index is 1, and `4` refers to the f-structure whose index is 4.

(29) a. `AF11_t.((s1_e -o s1_t) -o ((4_e -o F11_t) -o F11_t))`
b. $\forall F11_t.((s1_e \multimap s1_t) \multimap ((4_e \multimap F11_t) \multimap F11_t))$

[10] https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle.html#Prolog_Output

[11] In fact, only the attributes MR, SEMSTR, TYPE, and FORALL have a special status. All other attributes are assumed to represent arguments, which are consumed according to alphabetical order. It would also be possible to use A, B, C; A1, A2, A3; or any other alphabetically ordered series of attributes for arguments. It is not possible to substitute other names for the special attributes MR, SEMSTR, TYPE, FORALL.

c. $$4{:}\left[\begin{array}{ll}\textsc{pred} & \texttt{1} \\ \textsc{glue} & \left\{\left[\begin{array}{ll}\textsc{mr} & \textit{every} \\ \textsc{arg1} & \left[\begin{array}{ll}\textsc{arg1} & \left[\begin{array}{ll}\textsc{semstr} & \texttt{1} \\ \textsc{type} & e\end{array}\right] \\ \textsc{semstr} & \texttt{1} \\ \textsc{type} & t\end{array}\right] \\ \textsc{arg2} & \left[\begin{array}{ll}\textsc{arg1} & \left[\begin{array}{ll}\textsc{semstr} & \texttt{4} \\ \textsc{type} & e\end{array}\right] \\ \textsc{semstr} & \texttt{F11} \\ \textsc{type} & t\end{array}\right] \\ \textsc{forall} & \texttt{F11} \\ \textsc{semstr} & \texttt{F11} \\ \textsc{type} & t\end{array}\right]\right\}\end{array}\right]$$

7 Setting up and running the system

For full details on setting up and running the XLE+Glue system, please consult:
`https://github.com/Mmaz1988/xle-glueworkbench-interface/blob/master/README.md`
The basic steps are:

- Set up XLE according to these instructions:
  `https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle.html#SEC1.5`
- Download the XLE+Glue system and sample grammars.
- Navigate to the XLE+Glue folder that you downloaded and run XLE. The `xlerc` file that comes with the system automatically loads the sample grammar `glue-basic.lfg`, but you can edit it to load another sample grammar or your own grammar.
- The `xlerc` file sets up the new functionalities. It requires on the directory `src`, which is part of the github repository. It is easiest, to just keep the structure of the github repository to run the system.
- You will find an additional entry “Semantics” in the f-structure (or s-structure) “Commands” menu. Choose the “Semantics” command to start up the prover and produce a proof from the meaning constructors contributed by the sentence you parsed.

A Appendix: Running XLE in Windows 10 Subsystem for Linux

We have found that using the Windows Subsystem for Linux is an easy way of running the Linux version of XLE in Windows 10. Here is an overview checklist for this means of running Linux in Windows 10.

1. Follow the instructions on this webpage for installing a Linux distribution in Windows 10:

`https://docs.microsoft.com/en-us/windows/wsl/install-win10`

(Note that your Windows drives can be accessed from within Linux via the Linux directory `/mnt`: for example, the C drive can be accessed as `/mnt/c`.)

2. Install the Xming X-server to allow display of graphics, including XLE windows.

   (a) Download Xming here: `http://www.straightrunning.com/XmingNotes/` Be sure to install both Xming and Xming-fonts.

   (b) Run Xlaunch to create and save your profile. There is no need to change the default values. Save the profile to the directory where the XLaunch application is located by clicking on "Save configuration" at the end of the XLaunch dialogue and navigating to the XMing directory.

   (c) Add "`export DISPLAY=:0.0`" to the end of your Linux .profile file if you are using the Bash shell. (If you are not using the Bash shell, try "`setenv DISPLAY 0.0`" in your .login file.)

   (d) Depending on how you have set up XMing, you may have to launch XMing whenever you start up a Linux process.

3. You may find it convenient to allow copy/paste to and from Linux windows. You can get this by right-clicking on the white bar at the top of a Linux window and choosing "Properties", and making sure that the check box next to "Use Ctrl+Shift+C/V as Copy/Paste" is ticked.

4. Install the Linux version of XLE and the XLE+Glue system according to the instructions in Section 7.

5. To run the XLE+Glue system, you will also need to install SWI-Prolog and Java on your Linux system. For Ubuntu, the following works:

   (a) SWI-Prolog:
   ```
   sudo apt-add-repository ppa:swi-prolog/stable
   sudo apt-get update
   sudo apt-get install swi-prolog
   ```

   (b) Java:
   ```
   sudo apt install default-jdk
   ```

References

Crouch, Dick, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John T. Maxwell III, and Paula Newman. 2017. *XLE Documentation*. Palo Alto Research Center.

Dalrymple, Mary. 2001. *Lexical Functional Grammar*, volume 34. New York: Academic Press.

Dalrymple, Mary, John Lamping, and Vijay A. Saraswat. 1993. LFG semantics via constraints. In *Proceedings of the 6th Meeting of the EACL*, Pages 97–105. European Association for Computational Linguistics, Utrecht.

Dalrymple, Mary, John J. Lowe, and Louise Mycock. 2019. *The Oxford Reference Guide to Lexical Functional Grammar*. Oxford: Oxford University Press.

Meßmer, Moritz and Mark-Matthias Zymla. 2018. The Glue Semantics Workbench: A Modular Toolkit for Exploring Linear Logic and Glue Semantics. In M. Butt and T. H. King, editors., *Proceedings of the LFG'18 Conference, University of Vienna*, Pages 249–263. Stanford, CA: CSLI Publications.