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) as a part of the output of XLE, 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 system 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.

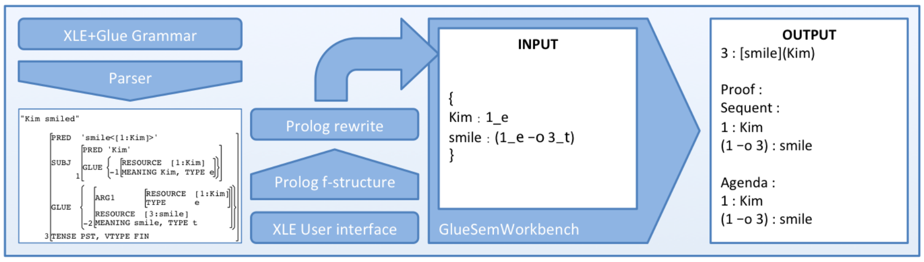
The XLE+Glue system is available in this repository:

`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 (GSWB) 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 4 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. The result is followed by some details about the computation of the proof. Specifically, the sequent (i.e. the input as parsed by the GSWB) and the agenda that is used to derive the proof. 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).



```
"Kim smiled"

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

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

```
3 : [smile](Kim)

Proof :
Sequent :
1 : Kim
(1 -o 3) : smile

Agenda :
1 : Kim
(1 -o 3) : smile
```

Figure 1: The glue prover is embedded in our XLE+Glue pipeline.

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

The input to the GSWB prover is a set of premises based on Glue semantics meaning constructors (Dalrymple, 1999). Glue meaning constructors consist of a semantic side (any semantic formalism) and a glue side (a linear logic expression of linguistic resources of a given type). The up and down 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 the first part of this manual we make several assumptions for ease of presentation. First, we assume simple untyped meaning representations such as *Kim* and *smile*; see Section 4 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 3) or other linguistic levels. However, it is also possible for meaning constructors to refer to other linguistic levels in our system (as in our sample grammar `glue-basic-semstr.lfg`, which includes a semantic projection).
Our system provides two ways of encoding meaning constructors. Both methods rely on the presence of a special GLUE attribute whose value is a set of meaning constructors in AVM format. The pipeline shown on the left-hand side of the diagram in Figure 1 illustrates the embedded encoding: meaning constructors are encoded in an attribute-value matrix (AVM) format in which embedding in the AVM mirrors the structure of the glue side of the meaning constructor. We describe this method in Section 2. In the string-based method, the meaning constructor is represented as a sequence of substrings which are values of an ordered set of attributes, as we describe in Section 7.2. In both encodings, 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 meaning constructors as AVMs with embedding

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

In the attribute-value encoding, each glue meaning constructor minimally consists of three attributes. The value of MEANING is the semantic (left-hand) side of the meaning constructor, while RESOURCE and TYPE specify the glue (right-hand) side: RESOURCE points to the relevant linguistic resource, while TYPE specifies RESOURCE's type. In XLE notation, the f-structure constraints contributed by a proper name like "Kim" are:

(3)
```
(^ PRED)='Kim'
(%mc MEANING)=Kim
(%mc RESOURCE)=^
(%mc TYPE)=e
%mc $ (^ GLUE)
```

`%mc` is a local name[2] used to construct an attribute-value structure containing attributes specifying the glue meaning constructor (`MEANING`, `RESOURCE`, `TYPE`). This f-structure is added to the `GLUE` set by specification of the constraint `%mc $ (^ GLUE)`.
A glue meaning constructor may also involve implication, as for a verb like *smile* in (4), where a resource is 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 RESOURCE and TYPE attributes.

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

b. $$s : \left[\begin{array}{ll} \textsc{pred} & \text{‘}\textsc{smile<subj>}\text{’} \\ \textsc{subj} & k :[\;] \\ \textsc{glue} & \left\{ \left[\begin{array}{ll} \textsc{meaning} & \textsc{smile} \\ \textsc{arg1} & \left[\begin{array}{ll} \textsc{resource} & k \\ \textsc{type} & e \end{array}\right] \\ \textsc{resource} & s \\ \textsc{type} & t \end{array}\right] \right\} \end{array}\right]$$

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

(5)
```
(^ PRED)='smile<(^ SUBJ)>'
(%mc MEANING)=smile
(%mc RESOURCE)=^
(%mc TYPE)=t
(%mc ARG1 RESOURCE)=(^ SUBJ)
(%mc ARG1 TYPE)=e
%mc $ (^ GLUE)
```

[2]`https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#N4.1.6`. See the discussion of scope of local names in 2.4.4.

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.

(6)
$$s : \left[\begin{array}{ll}
\textsc{pred} & \text{‘SMILE<SUBJ>’} \\
\textsc{subj} & k : \left[\begin{array}{ll}
\textsc{pred} & \text{‘Kim’} \\
\textsc{glue} & \left\{\left[\begin{array}{ll}
\textsc{meaning} & \text{Kim} \\
\textsc{resource} & k \\
\textsc{type} & e
\end{array}\right]\right\}
\end{array}\right] \\
\textsc{glue} & \left\{\left[\begin{array}{ll}
\textsc{meaning} & \text{SMILE} \\
\textsc{arg1} & \left[\begin{array}{ll}
\textsc{resource} & k \\
\textsc{type} & e
\end{array}\right] \\
\textsc{resource} & s \\
\textsc{type} & t
\end{array}\right]\right\}
\end{array}\right]$$

(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 on the linear logic side of meaning constructors.

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

b.
$$m : \left[\begin{array}{ll}
\textsc{pred} & \boxed{1}\ p \\
\textsc{glue} & \left\{ e : \left[\begin{array}{ll}
\textsc{meaning} & \textit{every} \\
\textsc{forall} & F \\
\textsc{arg1} & \left[\begin{array}{ll}
\textsc{arg1} & \left[\begin{array}{ll}
\textsc{resource} & \boxed{1} \\
\textsc{type} & e
\end{array}\right] \\
\textsc{resource} & \boxed{1} \\
\textsc{type} & t
\end{array}\right] \\
\textsc{arg2} & \left[\begin{array}{ll}
\textsc{arg1} & \left[\begin{array}{ll}
\textsc{resource} & m \\
\textsc{type} & e
\end{array}\right] \\
\textsc{resource} & F \\
\textsc{type} & t
\end{array}\right] \\
\textsc{resource} & F \\
\textsc{type} & t
\end{array}\right]\right\}
\end{array}\right]$$

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

[3]To display the attribute FORALL in XLE, select “constraints” in “Views” menu (or press “c”) in the window containing the glue premises in AVM format.

to a common noun meaning.[4] The value of the ARG2 attribute encodes an 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. $$m : \left[\begin{array}{ll}
\text{PRED} & \boxed{1}\ \textit{everyone} \\
\text{GLUE} & \left\{\begin{array}{l}
e : \left[\begin{array}{ll}
\text{MEANING} & \textit{every} \\
\text{FORALL} & F \\
\text{ARG1} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & \boxed{1}\\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & \boxed{1} \\ \text{TYPE} & t \end{array}\right] \\
\text{ARG2} & \left[\begin{array}{ll} \text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & m\\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & F \\ \text{TYPE} & t \end{array}\right] \\
\text{RESOURCE} & F \\
\text{TYPE} & t
\end{array}\right] \\
n : \left[\begin{array}{ll}
\text{MEANING} & \textit{person} \\
\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & \boxed{1}\\ \text{TYPE} & e\end{array}\right] \\
\text{RESOURCE} & \boxed{1} \\
\text{TYPE} & t
\end{array}\right]
\end{array}\right\}
\end{array}\right]$$

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

[4] Our sample grammars make 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: $(\uparrow \text{PRED})_e \multimap (\uparrow \text{PRED})_t$

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 3.

expect for the common noun *person*.

2.4 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 3 and in the comments in the grammar files.

2.4.1 The basic definitions

All of the templates which are used in defining meaning constructors in the sample grammars call the two basic templates `GLUE-RESOURCE` and `GLUE-MEANING`. The template `GLUE-RESOURCE` specifies an attribute-value structure `TypedRES` as having the value `R` for the attribute `RESOURCE` and the value `TY` for the attribute `TYPE`. The attributes `RESOURCE` and `TYPE` and their values must appear in all AVM meaning constructors and argument specifications, to identify the relevant linguistic resource and its type.

```
(10)    GLUE-RESOURCE(R TypedRES TY) =  (TypedRES RESOURCE) = R
                                        (TypedRES TYPE) = TY.
```

The template `GLUE-MEANING` specifies an attribute-value structure `TypedRES` as having the value `M` for the attribute `MEANING`. 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(TypedRES M) =  (TypedRES MEANING) = M.
```

2.4.2 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 `MEANING` attribute in the AVM meaning constructor. If it is desirable for the f-structure

[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`

PRED value to be different from the MEANING 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 MEANING 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(R TY M) =  @(GLUE-RESOURCE R %mc TY)
                                @(GLUE-MEANING %mc M)
                                %mc $ (R GLUE).
```

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

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

2.4.3 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 MEANING 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 `MEANING` 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 R TY M) =   @(GLUE-RESOURCE R %mc TY)
                                         @(GLUE-RESOURCE A1 (%mc ARG1) A1TY)
                                         @(GLUE-MEANING %mc M)
                                         %mc $ (R 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 `%mc`, that it has an attribute `RESOURCE` with value `R`, 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 `RESOURCE` 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).

2.4.4 Scope of local names

It is important to be aware of the scope of local names (variables prefixed with `%`) in XLE – it is limited to the c-structure category in which the variable is used.[6] This means that every time a given local name (for example: `%test`) is used within the same c-structure category, it refers to the same object – if there are two template calls using the same local name (`%test`) within one lexical entry, these template calls will impose constraints on the same object (one that corresponds to `%test`).
Depending on the intended effect, such behaviour of local names with respect to their scope may be a feature (when it is the intention to constrain the same object by separate template calls) or it may be undesired (when the intention is to impose constraints on two distinct objects by separate template calls) – this is why it is crucial to be aware of this when using local names.
This practical issue arises in sample grammars when a given c-structure category contributes more than one glue premise. For instance, as explained in the sample grammar `glue-basic.lfg`, the template `GLUE-NOUN0-MC` providing the meaning constructor for (common) nouns uses the local name `%mcn`, because it must be different from the local name `%mc` used by the template `GLUE-QUANT-MC` – both templates are called by the template `QUANT` called in the lexical entry of the quantifier "everyone". Another example can be found in the template `GLUE-ADJ0` which provides two meaning constructors for prenominal adjectives

[6]`https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#N4.1.6`: "A local name can be used as a variable whose scope is limited to the schemata associated with a particular category or lexical item."

– the call to the template `GLUE-REL1-MC` uses the local name `%mc` to provide the meaning constructor for the basic meaning of the adjective, while the call to the template `GLUE-ADJ-MODIFIER` provides the meaning constructor combining the adjective with the noun by calling the template `GLUE-MODIFIER1` which uses the local name `%mcm` to build this meaning constructor.

3 Sample grammars

The repository contains several XLE+Glue sample grammars. These build on the templates discussed in Section 2.4, 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:[7]

`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 5.3. When parsing with this grammar, please follow the instructions in Section 5.3 to modify the `xlerc` file to activate the semantic parser.

`glue-basic-flat-encoding.lfg` Similar to `glue-basic.lfg`, but instead of encoding meaning constructors as AVMs, it uses the alternative string-based, flat encoding described in Section 7.2.

4 Formatting for glue premises: General representational issues

4.1 Using `CONCAT`

XLE provides the built-in template `CONCAT`[8] 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 the local name `%OUTPUT`.
To avoid potential issues, it is safest to pass all special characters listed in section 4.2 as separate arguments to `CONCAT`.[9]

[7]Further grammars are discussed in section 7. However, these grammars have various shortcomings compared to the grammars presented here, and are, thus, not presented as full-fledged example grammars.
[8]`https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#CONCAT`
[9]Minimally, it seems the following must be passed as separate arguments to `CONCAT`:

4.2 Special characters in XLE

The following characters need to be escaped in XLE using a backquote (‘):[10]

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

4.3 Encoding the meaning side in XLE

We briefly present a possible way of encoding appropriate semantic representations in XLE in accordance with the semantic parser discussed in section 5.3. As mentioned above, 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.

(20)
```
GLUE-REL1-MC(A1 A1TY R TY P) =   @(GLUE-RESOURCE R %mc TY)
                                 @(GLUE-RESOURCE A1 (%mc ARG1) A1TY)
                                 @(CONCAT ‘[ ‘/x A1TY ‘.  P ‘(x‘) ‘] %MEANING)
                                 @(GLUE-MEANING %mc %MEANING)
                                 %mc $ (R GLUE).
```

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

4.4 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 4.2 with a backquote (‘), so that the resulting string can be used as the value of the attribute `MEANING`.
- `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 4.1).

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")
```

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

[10]Source: John Maxwell, PC.

5 Semantic representations and the prover

As described in the introduction, the Glue Semantics Workbench takes as input a set of premises, i.e., instantiated meaning constructors encoded in a specific string format, as input to calculate the semantics of an utterance. In this section, we describe the required input for the GSWB. First, we describe the encoding of linear logic formulas, and in sections 5.2 to 5.4, we describe different possibilities to encode the meaning side of a meaning constructor.

5.1 Parsing linear logic formulas

The Glue Semantics Workbench encodes linear logic formulas in a simple string format that is visually similar to the actual symbols used in linear logic. For example, the $\multimap$ symbol is replaced by `-o`. This is illustrated in (21), where some sample formulas from the fragment of linear logic that is covered by the parser are shown. As shown in (22), formulas can also be stated without type declarations.[11]

(21)
a. `g_e`
b. `(g_e -o (d_e -o (i_s -o h_t)))`
c. `((i_s -o h_t) -o (t_s -o f_t))`
d. `AX_t.((d_e -o X_t) -o X_t)`
e. `((g_e -o g_t) -o AX_t.((d_e -o X_t) -o X_t))`
f. `AX_t.AY_s.((d_e -o (Y_s -o X_t)) -o(Y_s -o X_t))`

(22) `((g -o g) -o AX.((d -o X) -o X))`

Examples (23) and (24) describe the formation of linear logic formulas that are well-formed from the perspective of the parser implemented in the prover. Most importantly, the parentheses around linear logic formulas are obligatory. Type declarations should either be applied to all constants and variables, or to none. Type declarations are indicated by an underscore, e.g., `_t`. Universal quantification over linear logic formulas is encoded in terms of an upper-case A followed by some variable (e.g. X) and a dot. As of now, the scope of a linear logic quantifier is the rest of the formula and does (and should) not need to be indicated via parentheses or brackets.[12] This means, linear logic quantifiers behave differently from the first-order logic quantifiers introduced in the next section.

(23) Atomic elements:
a. *Constants:* String of lower-case alphanumeric characters; optionally with type declaration
b. *Variables:* String of upper-case alphanumeric characters; optionally with type declaration

(24) Linear logic formulas:
a. *Linear implication:* (ϕ -o ψ), where ϕ, ψ are well-formed formulas
b. *Linear Quantification:* AX. ϕ, where ϕ is a well-formed formula

[11]An element without type declaration is treated as an element of type t.
[12]This behaviour is currently investigated and might change in the future.

5.1.1 Semantic types

The GSWB uses a parser for semantic types which is shared between the semantic parser (see section 5.3) and the linear logic parser. The available atomic types are shown in (25).[13] Complex types consist of atomic types, commas and angular brackets. This is illustrated in (26).

(25) **Atomic types:** e, s, v, t

(26) **Complex types**

a. `<e,<e,<s,t>>>`
b. `<<s,t>,<s,t>>`
c. `<<e,t>,t>`
d. `<<e,t>,<<e,t>,t>>`
e. `<<e,<s,t>>,<s,t>>`

5.2 System without the semantic parser

The GSWB, as of now, supports three modes for encoding meaning representations. Each mode needs to specify a procedure for encoding functional application, and abstraction. The default mode of the GSWB is a simple concatenation mode.
In this mode, any (string-based) format of semantic representation is compatible with XLE+Glue. This is the mode used by the demo grammars `glue-basic.lfg` and `glue-basic-semstr.lfg`, which include semantic representations that are treated as unanalysed strings by the workbench. In this simple format, functional application is expressed in the output by wrapping the argument in parentheses and concatenating functor and argument as in (27-a). Abstraction is handled by introducing a corresponding lambda binder as in (27-b).

(27) a. Combining $(1 \multimap 0)$: smile and 1 : Kim
to: 0 : smile(Kim)

b. sleep(x) to λx.sleep(x)

5.3 Using the semantic parser

The semantic parser provided by the GSWB is able to parse lambda expressions in accordance with a GSWB-internal semantic fragment, supporting alpha- and beta-conversion. This section describes how to use this semantic parser and the guidelines for writing lambda expressions that can be parsed by it.
Integrating the semantic parser into a grammar is simple: In the `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. Furthermore, 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

[13]Types need to be specifically declared in the code of the GSWB. It is, thus, not straightforward to introduce new types. This is an issue that we intend to address in future iterations of the GSWB. In the meantime, you can contact the authors to get help with the implementation of new types.

in the future.
The semantic parser parses lambda expressions and first-order logic terms. First-order predicates are encoded in the classic prefix notation. There is no convention with respect to the casing of predicates or constants, thus, the FOL terms in (28) all express a *liking* relation between two constants.

(28) a. `like(mary,semantics)`
b. `LIKE(mary,semantics)`
c. `like(MARY,SEMANTICS)`

5.3.1 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 a type as specified in section 5.1.1.
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. In (29) some examples for lambda expressions are shown.

(29) 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))]]]`

5.3.2 Logic operators

The basic logic operators $\wedge, \vee$ and $\rightarrow$ can be used as infix operators (see (30)a - (30)c), although their scope has to be defined via brackets or parentheses. Brackets can indicate operator scope and quantifier scope simultaneously (see (30)d). Other operators must be encoded as first-order logic predicates, i.e. in prefix notation (see (30)e).

(30) 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)`

5.3.3 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 as shown in (31).[14]

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

[14]Since `A` and `E` are reserved for quantifiers, these letters should not be used to encode other terms, e.g., variables, or constants.

5.3.4 Functional application

Functional application steps such as in the semantic terms for quantifiers are determined contextually. Consider `P(x)` and `Q(x)` in example (32). The P and Q variables over predicates followed by the x variable as an argument are translated as functional application steps (apply P/Q to x), rather than as a one-place predicate with a bound variable ($P(x)$).

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

Abstraction is handled in the same way as in the previous mode by adding a lambda binder to semantic formula, that is represented in terms of the λ symbol.

5.4 External semantic representations: Prolog mode

The third mode, supported by the GSWB, is the so called Prolog mode, which serves to illustrate how the GSWB can be made compatible with other semantic resources. It can be accessed by setting the `semParser` value to 2 in the `xlerc` file. This mode implements an alternative string encoding for semantic objects based on the system presented in (Blackburn and Bos, 2006). Using this system means, that all constants are expressed in terms of lower-case letters and all variables are encoded as (starting with) upper-case letters. Functional application is expressed in terms of the two-place predicate `app/2`, where the first argument is the functor and the second argument is the argument. Lambda expressions, and, thus, lambda abstraction, are introduced by wrapping a term in the two-place predicate `lam/2`. The first argument denotes the variable that is bound by the lambda and the second argument of `lam/2` denotes the body of the function. An example lambda expression in Prolog notation is given in (33). This corresponds to the functional application shown in (33-b). In the complex argument of this formula, x is combined with $\lambda v.bone(v)$, which is visually indicated as a function in terms of the square brackets. The variable x is then abstracted over by adding λx to combine with the quantifier.

(33) a. `app(lam(R,lam(S,every(Y,imp(app(R,Y),app(S,Y))))),`
`lam(X,app(lam(V,bone(V)),X)))`

b. $\lambda R.\lambda S.\forall y[R(y) \to S(y)](\lambda x.[\lambda v.bone(v)](x))$

6 Prolog rewrite component

The Prolog rewrite component takes the Prolog output of an XLE parse as input and translates it into a set of premises based on the specifications introduced in the previous section.[15] It does not rely on any particular assumptions about where the GLUE attributes must appear; as discussed in Section 3, 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

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

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 by the rewrite component.

Each element of the GLUE set provides one premise: as explained above, the value of MEANING provides the semantic side, while RESOURCE, TYPE, and the ARG1…N attributes provide the glue side.[16]

The values of the RESOURCE attributes 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 is used to encode linear quantification. Different quantified variables are distinguished by combining the label F with the unique f-structure index.

Example (34-a) shows the output produced by the rewrite component for a generalized quantifier encoded in AVM format as (34-c) (as discussed in example (8)), corresponding to the standard format meaning constructor in (34-b). All of the conventions discussed above are illustrated in (34-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.

(34) a. `AF11_t.((s1_e -o s1_t) -o ((4_e -o F11_t) -o F11_t))`

b. $\forall F11_t.((s1_e \multimap s1_t) \multimap ((4_e \multimap F11_t) \multimap F11_t))$

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

[16]In fact, only the attributes MEANING, RESOURCE, 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 MEANING, RESOURCE, TYPE, FORALL.

7 Illustrating the flexibility of the system

In this section, we present a variety of different modifications of the XLE+Glue system, that illustrate the possibility to use different semantic formalisms, as well as alternative encodings of Glue premises in XLE. We also show, how additional (semantic) resources can be added to the pipeline. Through this, we demonstrate how to enhance the functionality and coverage of the system.

7.1 Different semantic representations

7.1.1 Event semantics (with semantic parser)

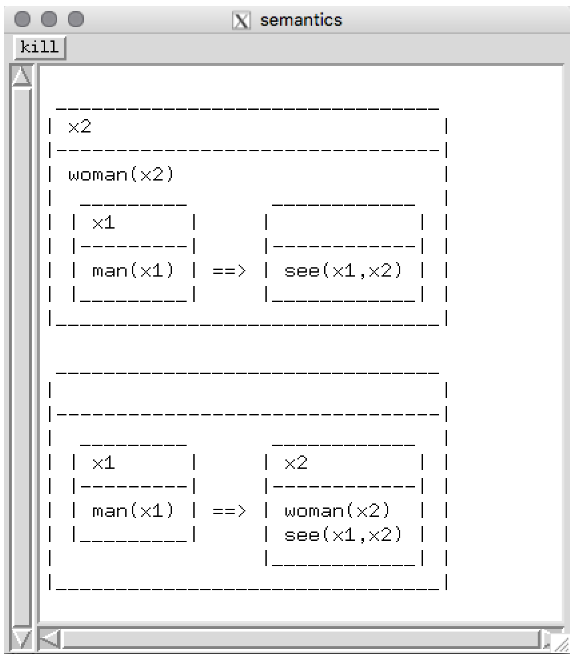
While `glue-basic-semparser.lfg` is the sample grammar using the semantic parser (see Section 5.3), `glue-basic-semparser_ND.lfg` is its modified version using event, Neo-Davidsonian semantics (Parsons, 1990).
Rather than using predicates with variable arity (depending on the number of arguments of the predicate), in event semantics the predicate has only one argument, the event variable, while the dependents of the predicate are related to it using separate predicates whose name corresponds to the semantic role of the given dependent (such as *agent*, *theme*, etc.).
The examples below provide semantic representations of "Kim smiled" produced by `glue-basic-semparser.lfg` and `glue-basic-semparser_ND.lfg`, respectively: in (35-a) the predicate *smile* has one argument (Kim), while in (35-b) the only argument of *smile* is the event variable (here: z), while Kim is related to the event z using the *agent* predicate (Kim is the *agent* of z).

(35) a. `smile(Kim)`
b. `exists([λz_v.and(smile(z),agent(z,Kim))])`

7.1.2 DRT semantics

In section 5.4, we have shown, that the GSWB supports Prolog-style encoding of semantic formulas as output. Using this mode, we provide a DRT-mode in XLE+Glue to illustrate the possibility to interact with different semantic resources. This mode is invoked by another variable in the `xlerc` file, `processDRT`, which needs to be set to 1. This mode is dependent on the Prolog mode. It is, thus, necessary to also set the variable `semParser` to 2.
The DRT-mode makes use of the Boxer DRT system (Blackburn and Bos, 2006; Bos, 2008, 2015) optimized to interpret λ-DRT formulas as described in Gotham and Haug (2018).[17] To include those two components, we extended the λ-DRT system with some simple wrapper code to execute it from within XLE+Glue. Thanks to the Prolog-style encoding option of the GSWB, the lambda DRT component can directly process the GSWB's output and produces a graphical boxer-style representation of the Prolog term that is presented in XLE+Glue's as output window. This is shown in Figure 2.

[17]We are very grateful to Johan Bos and Matthew Gotham for making their λ-DRT tools available to us.



```
 _________________________________
| x2                              |
|---------------------------------|
| woman(x2)                       |
|   _________       ____________  |
|  | x1      |     |            | |
|  |---------|     |------------| |
|  | man(x1) | ==> | see(x1,x2) | |
|  |_________|     |____________| |
|_________________________________|

 _________________________________
|                                 |
|---------------------------------|
|   _________       ____________  |
|  | x1      |     | x2         | |
|  |---------|     |------------| |
|  | man(x1) | ==> | woman(x2)  | |
|  |_________|     | see(x1,x2) | |
|                  |____________| |
|_________________________________|
```

Figure 2: Boxer-style DRT representation: *Every man saw a woman*

7.2 Meaning constructors encoded as strings

Section 2 describes the f-structure encoding of meaning constructors as AVMs, using the attributes RESOURCE, TYPE, MEANING, and ARG1...ARGN. In this encoding, the embedding in the AVM representation reflects the structure of the linear logic expression that is encoded, since material on the left-hand side of a linear implication is represented as the f-structure value of an attribute such as ARG1.

This section describes an alternative encoding: the string-based, flat encoding. In this encoding, the substrings of the meaning constructor are encoded as values of the attributes in a single AVM, which are concatenated together to produce the input to the prover. This is in some ways a simpler encoding, since it does not require the construction of a complex AVM to reflect the structure of the linear logic term. However, it requires a detailed understanding of the input format required by the GSWB prover, and it is also easier to make mistakes in the encoding, which can make it harder to use.

The flat encoding is illustrated in (36), where 1 is the label assigned by XLE to the outermost f-structure:

(36) a. `Kim:1_e`

b. $$1 : \left[\begin{array}{ll} \text{PRED} & \text{`Kim'} \\ \text{GLUE} & \left\{ \left[\begin{array}{ll} \text{A1} & \text{Kim} \\ \text{A2} & : \\ \text{A3} & 1 \\ \text{A4} & \_ \\ \text{A5} & \text{e} \end{array}\right] \right\} \end{array}\right]$$

```
c.   Kim   :   1   _   e
     a1   a2   a3  a4  a5
```

In this encoding, the substrings of the meaning constructor input to the GSWB prover are encoded directly. By convention, this encoding uses the attributes A1,A2,..., but in fact any attributes can be used except for the special attribute MEANING, which indicates to the transfer component that the embedded encoding is being used. It is possible to have meaning constructors encoded in the string-based format and the standard embedded format coexisting in the same f-structure, or even in the same GLUE set.
In the transfer component using the flat encoding, the attributes are sorted into alphabetical order, and the substrings of the meaning constructor are concatenated according to that ordering. (36-c) shows the correspondence between the attributes of the f-structure in (36-b) and the resulting meaning constructor. A cautionary note: if there are 10 or more meaning constructor substrings encoded via attributes `a1...a10...` in an AVM, the attribute `a10` will sort alphabetically between the attributes `a1` and `a2`; in that case, therefore, the single-digit attributes should be prefixed with 0 (`a01,a02,...a10,a11,...`) to ensure that the values of the attributes are concatenated in the correct order.
The lexical entry for *Kim* in the sample grammar `glue-basic-flat-encoding.lfg` calls the `PROPERNOUN` template as shown in (12), and `PROPERNOUN` is defined in terms of the template `GLUE-RELO-MC` as shown in (13). In the string-based, flat encoding, `GLUE-RELO-MC` is defined as follows:

```
(37)   GLUE-RELO-MC(R TY M) =  (%mc A1) = M
                               (%mc A2) = `:
                               (%mc A3) = R
                               (%mc A4) = _
                               (%mc A5) = TY
                               %mc $ (R GLUE).
```

The value of the attribute `a1` is the meaning term, which is the first component of the meaning constructor for *Kim*. The value of `a2` is the colon separating the meaning side of the meaning constructor from the glue side, which must be quoted with a backquote. The value of `a3` is the f-structure for *Kim*, the value of `a4` is the underscore separating the f-structure from its type, and the value of `a5` is its type, as specified by `TY` in the template call to `GLUE-RELO-MC`. Further examples can be found in the sample grammar `glue-basic-flat-encoding.lfg`.

8 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 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 (last updated: May 2020).

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 8.

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

Blackburn, Patrick and Johan Bos. 2006. *Working with Discourse Representation Theory*. unpublished draft.

Bos, Johan. 2008. Wide-coverage semantic analysis with boxer. In *Semantics in Text Processing 2008 conference proceedings*, Pages 277–286.

Bos, Johan. 2015. Open-domain semantic parsing with boxer. In B. Megyesi, editor, *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, Pages 301–304.

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. 1999. *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach*. MIT Press.

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.

Gotham, Matthew and Dag Trygve Truslew Haug. 2018. Glue semantics for Universal Dependencies. In M. Butt and T. H. King, editors., *Proceedings of the LFG'18 Conference, University of Vienna*, Pages 208–226. Stanford, CA: CSLI Publications.

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.

Parsons, Terence. 1990. *Events in the Semantics of English: A Study in Subatomic Semantics*. Cambridge, MA: The MIT Press.