AMALIA

Abstract MAchine for LInguistic Applications

User's Guide

Version 1.0

February 1998

Shuly Wintner, Evgeniy Gabrilovich, Nissim Francez

Laboratory for Computational Linguistics

Computer Science Deparmtent

Technion, Israel Institute of Technology

32000 Haifa, Israel

lcl@cs.technion.ac.il

Contents

Preface – Version 0.1

The system described herein was implemented as part of the PhD dissertation of the first author and the Masters thesis of the second author, under the supervision of the third author, at the Department of Computer Science of the Technion, Haifa. It was never meant to become a 'commercial' product, and it should be considered as a prototype whose main purpose is to validate the theoretical results of the dissertations. Still, we believe that it can prove useful for certain practical purposes, and that is why we make it publicly available.

We will appreciate any comments regarding the system. Please send any bug reports as soon as the bugs are encountered. We'd also appreciate any feedback on the usability and functionality of the system, as well as requests for additions, modifications and extensions. However, there can be no guarantee that any such requests are fulfilled, nor even that bugs are fixed. The system comes with absolutely no warranty.

We wish to acknowledge the help and support of Bob Carpenter throughout the entire span of the system's development period. We are also grateful to the staff at the Seminar für Sprachwissenschaft, Universität Tübingen.

The system is available without charge from the authors. It is designed to run under the Unix operating system and was tested on Sun and Silicon Graphics workstations. Adaptations to other platforms will be considered if the need arise.

Shuly Wintner
Haifa, 1997

Preface – Version 0.2

The only major change introduced in this version is the use of a newer version of Tcl/Tk. With Tcl 7.6 and Tk 4.2 it is possible to pre-compile the GUI code, hopefully resulting in better performance of the graphical version of $\mathcal{A}$MALIA. With the Plus Patch to Tcl/Tk, $\mathcal{A}$MALIA is now independent of the presence of a Tcl/Tk installation – the system should run even in sites that don't have Tcl/Tk.

Shuly Wintner
Tübingen, June 1997

Preface – Version 1.0

This version contains some bug fixes. More importantly, the User's Guide now contains an extended description of generation with $\mathcal{A}$MALIA.

Shuly Wintner
Tübingen, February 1998

Chapter 1

Introduction

AMALIA is an efficient, abstract-machine based implementation for a subset of the grammatical formalism ALE, described in (Carpenter, 1992a). It provides a unified environment for processing ALE-based grammars: grammars can be compiled, and then used for both parsing and generation of natural language phrases. The system also contains a simple debugger for easing the process of grammar development. Two versions of the AMALIA exist: an interactive, easy-to-use, system with a graphical user interface and a text-oriented, non-interactive one. The former is intended for developing prototype grammars; the latter is far more efficient but less user friendly, and is intended to be used for batch processing.

This user's guide describes AMALIA from a practical point a view. It provides information necessary for installing the system and using it. However, it does *not* describe the theory underlying the system. This can be found in (Wintner, 1997), and in less details – in (Wintner and Francez, 1995a; Wintner and Francez, 1995b). (Wintner, Gabrilovich, and Francez, 1997) describes AMALIA as a unified platform for parsing and generation, elaborating more on the way the two directions are integrated into a single system. We also assume acquaintance with ALE – refer to either (Carpenter, 1992a) or (Carpenter, 1992b) for more details. We do, however, list the differences between ALE and AMALIA in particular, those features of ALE that are not supported by AMALIA.

Chapter 2 describes the features provided by the system, emphasizing the differences from ALE. Chapter 3 guides users in operating the system. The appendices will (some time in the future) list the error and warning messages issues by the system, and formalize the syntax of input specifications.

Chapter 2

Functionality and Features

This chapter lists the main features of AMALIA. In particular, it details the differences between AMALIA and ALE in terms of the supported features.

2.1 Type Specification

AMALIA supports the same type hierarchies as ALE does, with exactly the same specification syntax. This means that the user can specify any bounded-complete partial order as the type hierarchy. Only immediate sub-types are specified, and the reflexive-transitive closure of the sub-type relation is computed automatically by the compiler. The special type `bot` must be declared as the unique most general type. In contrast to ALE, AMALIA does not issue a warning message when a type only has one subtype.

Appropriateness, too, is specified using ALE's syntax, by describing the features at the type they are introduced by. The feature introduction condition must be obeyed: every feature must be introduced by some most general type, and it is appropriate for all its sub-types. However, there are two differences between AMALIA and ALE in this respect. First, AMALIA allows appropriateness loops[1] in the type hierarchy; on the other hand, some ALE specifications are not handled correctly by AMALIA. If a feature `f` is introduced by the type `t` and was assigned a type `t'` as its appropriate value, then sub-types of `t` cannot 'redefine' the appropriate values of `f`. This limitation will be removed in subsequent versions of AMALIA.Type constraints are not supported by AMALIA.

2.2 Grammars

AMALIA uses a subset of ALE's syntax for describing feature structures. As a rule, whenever AMALIA supports ALE's functionality, it uses the same syntax. In general, AMALIA supports totally well-typed, possibly cyclic, non-disjunctive feature structures. Set values, as in ALE, are not supported, but list values are. AMALIA does not respect the distinction between *intensional* and *extensional* types (see (Carpenter, 1992b, Chapter 8)). Also, feature structures cannot incorporate inequality constraints.

[1] Appropriateness loops are handled by employing lazy evaluation techniques at run time.

The semantics of the logical descriptions, as well as the operator precedence, follow ALE. As in ALE, partial descriptions are expanded at compilation time. AMALIA's compiler performs type inference on partial descriptions, reports any inconsistencies, and then creates code for the expanded structures. To avoid infinite processing in the face of appropriateness loops (where no finite totally well-typed structure that satisfies the description might exist), the compiler stops expanding a structure if it is the most general structure of its type.

AMALIA supports macros in a similar way to ALE. The syntax is the same, and macros can have parameters or call other macros (though not recursively, of course). ALE's special macros for lists are supported by AMALIA. There are two differences between AMALIA and ALE with respect to macros: first, AMALIA expands macros at compile time, thus reporting inconsistencies earlier than ALE. Second, variables in the scope of a macro are treated exactly the same as feature structure variables.

In ALE, variables in the scope of a macro are not the same as ALE feature structure variables — they denote where macro-substitutions of parameters are made, *not* instances of reentrancy in a feature structure. If we employ the following macro:

```
blah(X) macro
  b,
  f: X,
  g: X.
```

with the argument `(c,h:a)` for example we obtain the following feature structure in ALE:

```
b
F c
  H a
G c
  H a
```

where the values of `F` and `G` are not shared (unless `c` and `a` are extensional). In AMALIA, the same specification would yield:

```
b
F [0] c
  H a
G [0]
```

ALE includes a built-in definite logic programming language; AMALIA does not. The entire power of definite clause specifications is missing in AMALIA. However, a few common functions that are external to the feature structure formalism were added to the system, and grammar specifications can use them. These features are referred to as *goals*, although it must be remembered that they are far weaker than ALE's goals.

In the current version of the system, only two goals are supported: `append`, which performs list concatenation, and `union`, which performs set union.[2] The syntax of using both goals is:

[2]We assume an encoding of sets as lists – there is no internal representation for sets in AMALIA.

```
goal> append(X,Y,Z).
goal> union(X,Y,Z).
```

where `X,Y` and `Z` are variables. `X` and `Y` must be bound when the goal is evaluated, and their values must be of appropriate types (`Y` must be a list for `append`, `X` and `Y` must be sets for `union`). `Z` is set by the execution of the goals, and its former value is lost. Note that goals are not guaranteed to work properly when AMALIA is used for generation (Section 3.5).

AMALIA employs a bottom-up chart based control unit, where rules are evaluated from left to right. The chart is used for storing active and inactive edges. For parsing, edges span a sub-sequence of the input string, assigning it some structure. For generation, edges span a sub-form of the input semantic form, also assigning it a structure that eventually determines a phrase whose meaning is that sub-form. It must be noted that at run time there is no notion of the particular task (parsing/generation) performed by the machine. Computations terminate after finding all of the complete edges derivable from the input and the grammar rules. There is no notion of an *initial symbol*. Of course, if the grammar is such that an infinite number of derivations can be produced, computations might not terminate. AMALIA does not incorporate a subsumption check to test for spurious ambiguity.

Rules can incorporate definite clause goals only *after* category specifications. While goals can occur anywhere in the body of a rule, they are evaluated after the entire rule's body was scanned. Therefore, goals can refer to variables that are only instantiated after the entire body of the rule was seen, and can instantiate variables that occur in the rule's head.

AMALIA preserves ALE's syntax in describing lexical entries. Multiple lexical entries may be provided for each word, separated by semicolons. The current version of the system does not represent large lexicons efficiently; this will be changed in future versions. AMALIA also keeps ALE's syntax in the definition of *empty categories* (or ϵ-rules). In contrast to ALE, AMALIA processes empty categories at compile time. Each empty category is matched by the compiler against each element in the body of every rule; if the unification succeeds, a new rule is added to the grammar, based upon the original rule, with the matched element removed. Some limitations apply for this process (which in the general case is not guaranteed to terminate), and therefore the resulting grammar might not be equivalent to the original one.

Lexical rules are not supported in this version of AMALIA. AMALIA's syntax for phrase structure rules is similar to ALE's, with the exception of the `cats>` specification that is not supported.

2.3 Program Organization

AMALIA's programs are composed of four parts: a type specification, macro definitions, grammar rules and a lexicon. If a grammar is to be used for generation as well, it might contain an optional fifth part - a Connective Registry (see Section 3.5.2). In contrast to ALE, AMALIA's syntax requires that the type hierarchy precede both the lexicon and the rules. The macros must be defined before they are used, and the grammar rules must precede the lexicon. Furthermore, each part of the program must be introduced by a designated keyword. The syntax of AMALIA's programs is the following:

```
<prog> ::=  %th <th> <macros> <grammar> <lexicon> <conn_registry>

<th> ::= <type_spec>
         | <th> <type_spec>

<macros> ::= /* epsilon */
             | %macros macro_defs

<macro_defs> ::= /* epsilon */
                 | macro_defs macro_def

<grammar> ::= /* epsilon */
              | %grammar rules

<rules> ::= rule
            | rules rule

<lexicon> ::= /* epsilon */
              | %lexicon lex_entries

<lex_entries> ::= /* epsilon */
                  | lex_entries lex_entry

<conn_registry> ::= /* epsilon */
                    | %connective_registry cr_entries

<cr_entries> ::= /* epsilon */
                 | cr_entries cr_entry
```

Chapter 3

User's Guide

3.1 Installation

The exact details of the abstract machine are outside the scope of this paper; the reader is referred to (Wintner and Francez, 1995a; Wintner, 1997) for more information on the machine itself, and to (Gabrilovich, 1997) for using AMALIA for generation and for a detailed description of grammar inversion. In this Section we list some implementation details along with some functional features of the system.

AMALIA is implemented in *C*, augmented by *yacc* and *lex*; the graphical user interface is designed in *Tcl/Tk* (Ousterhout, 1994). It was developed on a Silicon Graphics Indy workstation but was tested also on an IBM PC running Windows'95 and Linux. Currently, the system is available only for some Unix machines, but other platforms might be supported in the future.

The system comes in two varieties: a unified, graphical, user-friendly one and a more efficient yet far less friendly batch one. The first version, which includes a graphical user-interface, is referred to below as 'amalia'. The other version contains two executable files, referred to as the compiler and the interpreter.

To use the graphical version of AMALIA, you'll need either a Sun or an SGI Indy with a graphical display. The necessary files are `amalia`, which is the main executable file, and `gui` and `FSbox.tcl`, which must reside under the directory `source` (the tar file in which the system is supplied automatically places the files in their appropriate locations). To use the non-grahical version you'll need the files `comp` and `int`, which are the compiler and the interpreter, respectively.

The entire system uses about 1.5MB of memory, of which about 1.2MB are needed for the graphical version.

The system is obtainable from:

`http://www.sfs.nphil.uni-tuebingen.de/~shuly/amalia.uu`

(if you cannot access this file, send e-mail to `lcl@cs.technion.ac.il`). Install the file `amalia.uu` there. Then execute the following commands:

```
> uudecode amalia.uu
> gunzip amalia.tar.gz
> tar -xf amalia.tar
```

A new directory, `amalia`, will be created, with three sub-directories: `bin`, with the files `amalia, int` and `comp` under it, `doc`, with a PostScript version of this document in it, and `examples` with some example grammars. The system is ready for use.

3.2 Using the Graphical User Interface

AMALIA's graphical user interface is invoked by the command

```
> amalia [file]
```

where `file` is the name of the grammar file. Make sure the `DISPLAY` environment variable points to your display before invoking the system.

AMALIA comes up as a large window which is divided to four areas (listed here from the top down): a control panel with pull-down menus; an entry line for the input string; several windows for displaying the abstract machine's state; and a window for displaying messages.

The control panel includes four pull-down menus: *File, Compile, Parse* and *Generate*. The *File* menu allows one to select the grammar file and to quit the system. *Compile* provides for compiling the grammar, both for parsing and for generation. *Parse* and *Generate* enable the user to execute a (previously compiled) grammar. If *Parse* is used, the *Input string* window must contain the phrase to be parsed. For *Generate*, the *Input string* window must specify the name of the query file, which contains (an ALE description of) a feature structure representing the input semantic form.

Both *Parse* and *Generate* provide several operation options. In particular, both allow the user to execute the program in a *step* mode, by executing one instruction at a time. In case the program 'gets out of hand', its operation can be ceased by selecting the *break* option from either of the two menus.

Most of the control options described above are bound to accelerators ('hotkeys') so that they are operable from the keyboard, without using the mouse.

The central area of the interface is dedicated to the display of the machine's internal state. A thorough explanation of the abstract machine is outside the scope of this document, and the user is referred to (Wintner and Francez, 1995a; Wintner, 1997) for the details. In general, there are windows that display the values of all the special purpose registers, the general-purpose registers, the heap and the code area. The program counter (pointing to the next instruction to be executed) is represented by the symbol `=>` next to the instruction, in the code area. The chart cannot be displayed in its entirety, but a special window allows the user to ask for the display of either the current edge or all the edges in a certain chart entry. Complete edges are shown as feature structures; active edges are currently not displayed.

A breakpoint can be specified in a certain location of the code. Simply click the mouse left button on some instruction, and the interpreter will stop before executing it. To continue running, select *Run* again. Breakpoints in the code that was generated for the type hierarchy or the lexicon are not respected.

An additional feature provides an option to view the contents of the heap as a feature structure. By double-clicking on some heap cell, the system graphically displays the feature structure accessible from this cell. If the program terminates, the resultant feature structures (if any) are also displayed graphically.

The *Message* area is used by the system to report various messages, warnings and errors to the user. When AMALIA is used in the generation mode, the generated phrases (if any) are also displayed in this area.

3.3 Compiling Grammars

The compiler is invoked by the command

```
> comp [-g] file
```

where `file` is the name of the grammar file and the optional flag `-g` instructs the compiler to invert the grammar for generation (see Section 3.5 below) prior to compilation. The main product of the compilation is a file containing the compiled grammar, represented as a program in AMALIA's machine language; this file is always called `program`. In addition to the program, the compiler creates several other files which contain symbol tables that are necessary for the interpreter to operate correctly. Currently, these files are the following:

- `type_tbl`: contains information on types;
- `subtype_tbl`: contains information on subtypes;
- `feat_tbl`: contains information on features;
- `th`: contains pointers to the type unification functions;
- `words`: contains information extracted from the lexicon.

If the compiler is invoked with the `-g` flag, the file `words` is not created, but instead two additional files are created:

- `ut_tbl`: contains pointers to the type unification functions;
- `skb`: contains information extracted from the lexicon and the Connective Registry (see Section 3.5.2).

Grammar compilation is highly efficient in AMALIA and shouldn't take more than a few seconds, even on relatively large grammars.

There are many errors that AMALIA is able to detect at compile time. These errors will be flagged during compilation. Most errors indicate the line number in which they are found. Some errors may be serious enough to halt compilation before it is finished. In general, it is a good idea to fix all of the errors before trying to run a program, as the error messages only report serious bugs in the code, such as type mismatches, unspecified types, ill-formed rules, etc. Less serious problems are flagged with warning messages. The error and warning messages are listed in an appendix at the end of this report, along with an explanation.

3.4 Running Compiled Grammars

The interpreter is invoked by the command

```
> int [-g] file string
```

where `file` is the name of the *compiled* grammar file and `string` is the string to parse. The optional flag `-g` instructs the interpreter to *generate* rather than parse. In this case, `string` must be the name of the query file, which contains (an ALE description of) a feature structure representing the input semantic form for generation (see Section 3.5).

If the parsed string is a sentence of the language defined by the grammar, the parsing result is a set of feature structures that derive the string. In the case of generation, the program ends up with a set of feature structures which encode the generated phrases. In either case, the interpreter displays each feature structure encoded in the ALE specification language, which is appropriate for further processing but rather difficult for a human to read. For example, the feature structure

```
b
F [0] c
  H a
G [0]
```

might be displayed as:

```
[12]b(
  f:[14]c(
    h:[18]a),
  g:[14])
```

3.5 Using AMALIA for Generation

3.5.1 Overview

An input for the generation[1] task is a logical form which represents a meaning, and a grammar to govern the generation process. The output consists of one or more phrases in the language of the grammar whose meaning is (up to logical equivalence) the given logical form.

Logical forms specified as meanings by input grammars are given in a so-called predicate-argument structure. The predicate-argument structure is analogous to the familiar representation of semantic logical forms with first-order terms. Thus, meanings are built from basic units (feature structures), each having a predicate and (optionally) a number of arguments (see also Section 3.5.2 below). The formalism also allows λ-abstractions over predicate-argument constructs, as well as systematic encoding of second- and higher-order functions.

Grammars are usually designed in a form oriented towards the analysis of a string and not towards generation from a (usually nested) semantic form. In other words, rules reflect the phrase structure and not the predicate-argument structure. It is therefore useful to transform the grammar in order to enable systematic reflection of any given logical form in the productions. For this purpose, we apply to the input grammar an inversion procedure, based upon[2] (Samuelsson, 1995), to render

[1] In this work we mean by "generation" what is sometimes known also as "syntactic generation". Thus, no text planning, speaker intentions and the like are considered here.

[2] Samuelsson's inversion algorithm was originally developed for definite clause grammars. We adapted it to the Typed Feature Structure formalism.

the rules with the nested predicate-argument structure, corresponding to that of input logical forms. The resultant "inverted" grammar is thus more suitable for performing the generation task. Once the grammar is inverted, the generation process can be directed by the input semantic form; elements of the input are consumed during generation just like words are consumed during parsing. Grammars must satisfy certain requirements in order for them to be invertible (see (Gabrilovich, 1997, Section 3.2)). However, the requirements are not overly restrictive and allow encoding of a variety of natural language grammars. These requirements induce certain constraints on the type hierarchy of the input grammars. A fragment of the type hierarchy that every grammar must include in order to be invertible is presented in Section 3.5.3 below.

Grammar inversion is performed immediately prior to compilation for generation; in $\mathcal{A}$MALIA these two functions are merged into one and are always performed together. The given grammar is enhanced in a way that will ultimately enable to reconstruct the words spanned by the semantic forms. To achieve this aim, each rule constituent is extended by an additional special-purpose feature. The value of this feature for the rule's head is set to the concatenation of its values in the body constituents, to reflect the original phrase structure of the rule. Because of the nature of grammar inversion, it does not necessarily preserve *goals* associated with grammar rules (if any).

Figure 3.1 delineates an overview of AM-based generation. After the grammar is inverted, it is compiled into the abstract machine code. At run time, the given logical form is decomposed into meaning components, which initialize the AM chart, and then the generation program is invoked. If generation terminates, it yields a (possibly empty) set of feature structures; a grammar-independent post-processing routine analyzes these structures and retrieves the generated phrases per se.

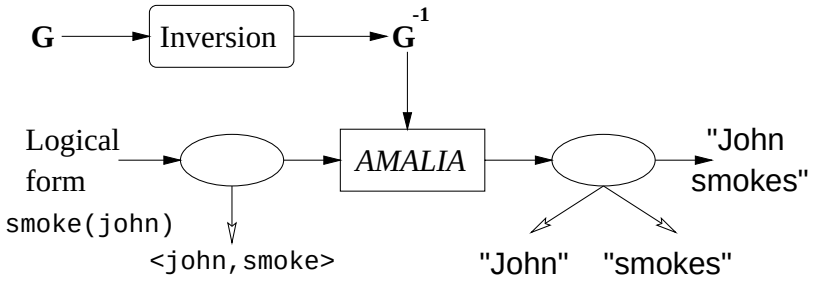

Figure 3.1: An overview of generation with Abstract Machine.

During generation, the graphical user interface is used as follows. The *Input string* field specifies the name of the query file, which contains (an ALE description of) a feature structure representing the input semantic form. The *Messages* window displays the phrases generated (if any). The feature structures that encode these phrases are also displayed graphically, in separate windows.

The above overview contains only a brief description of the generation process. For more details on Natural Language Generation with Abstract Machine, refer to (Gabrilovich, 1997) and (Wintner, Gabrilovich, and Francez, 1997).

3.5.2 Semantic primitives

Logical forms, which represent meanings of sentences derivable by a given grammar, are encoded with a set of semantic primitives. Such a set is fixed for each grammar and can be deduced from the grammar specification. The primitives may be of one of two kinds: most of them represent meanings of lexical constants (e.g., `john`, `smoke`, `today`), while the rest serve as *connectives*[3] which aid in building complex meaning forms (e.g., `mod`, which modifies its first argument with the second one, as in `mod( smoke(john), today )`). Primitives of the latter kind are not directly related to any lexicon entry; the role of connectives is to connect other meaning components together.

We therefore require that apart from the lexicon, a *Connective Registry* (CR) be supplied[4] with the grammar which encompasses the possible uses of the connectives (for instance, some may be applicable to constants, others to predicates etc.). A CR item is actually a connective *usage signature*; each CR entry is a feature structure giving the full representation of the primitive being defined.

For more details on encoding semantics in generation grammars, refer to (Gabrilovich, 1997, Sections 3.2 – 3.3)

3.5.3 Minimum required type hierarchy

The obligatory minimum type hierarchy in ALE notation is shown in Figure 3.2, and is depicted as an inheritance graph in Figure 3.3.

Pursuant to ALE's convention, `bot` serves the root of the type hierarchy. Each type has a (possibly empty) set of subtypes listed after the `sub` keyword following the type name. Furthermore, each type may introduce a list of features which appear after the `intro` keyword, while each feature is followed by an appropriate type.

The type hierarchy shown has provisions for up to three arguments per predicate (arg_i), though it can easily be extended to incorporate more, should this become necessary for some particularly sophisticated grammar. The `list` part of the type hierarchy allows for encoding lists and other ordered sequences used in *args* and *str* features.

The ellipsis (...) denotes values not stipulated by the minimum required type hierarchy, since inversion and generation do not assume anything about them. For example, specific syntactic categories (values of the *cat* feature) may vary in actual grammars, and therefore are not shown. On the other hand, the inversion algorithm assumes that constituents of input grammar rules have the feature *syn*, whose values in turn has the feature *cat*. Hence both features are required in the type hierarchy, while the particular values of the latter are not.

Currently, the names of the distinguished features and types (e.g., *syn*, *sem*, *pred*) are "hard-coded" into the type hierarchy. This way the compiler and the interpreter assume such features and types to be associated with known names. Should the need arise, these symbols can be turned into parametric, to be explicitly supplied with the input grammar.

[3] Not necessarily restricted to boolean connectives.

[4] In principle, the Connective Registry may be automatically deduced from the grammar specification. To simplify things, we currently supply it as a part of the input grammar.

```
bot sub [sign, syn, syn_term, sem, args, list].
  sign sub [phrase].
    phrase sub [word] intro [syn:syn, sem:sem, args:args, str:list].
      word sub [].
  syn sub [] intro [cat:syn_term].
  syn_term sub [...].
    ...
  sem sub [const, funct].
    const sub [...].
      ...
    funct sub [atomic, quant].
      atomic sub[arg_1] intro [pred:sem].
        arg_1 sub [arg_2] intro [arg1:sem].
          arg_2 sub [arg_3] intro [arg2:sem].
            arg_3 sub [] intro [arg3:sem].
      quant sub [l_bind] intro [var:sem].
        l_bind sub [] intro [rest:sem].
  args sub [] intro [larg:list].
  list sub [ne_list, e_list].
    ne_list sub [] intro [hd:bot, tl:list].
    e_list sub [].
```

Figure 3.2: Minimum required type hierarchy in ALE notation.

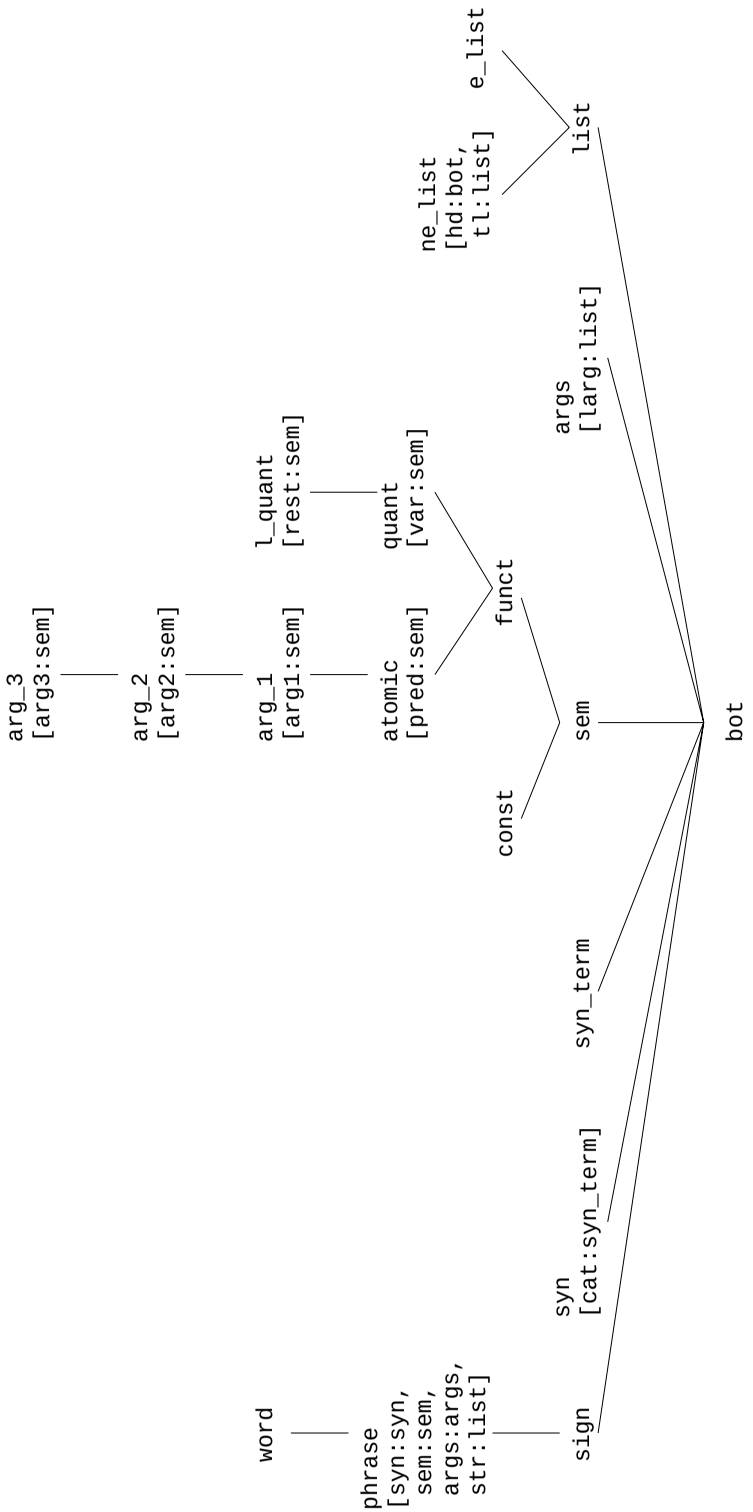

Figure 3.3: Minimum required type hierarchy depicted as a graph.

References

Carpenter, Bob. 1992a. ALE – the attribute logic engine: User's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213, December.

Carpenter, Bob. 1992b. *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Gabrilovich, Evgeniy. 1997. Natural language generation by abstract machine. Master's thesis, Technion, Israel Institute of Technology, Haifa, Israel. In preparation.

Ousterhout, John K. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series. Addison-Wesley.

Samuelsson, Christer. 1995. An efficient algorithm for surface generation. In *Proc. of the 14th International Joint Conference on Artificial Intelligence, Montreal, Canada*, pages 1414–1419. Morgan Kaufmann, August.

Wintner, Shuly. 1997. *An Abstract Machine for Unification Grammars.* Ph.D. thesis, Technion – Israel Institute of Technology, Haifa, Israel, January.

Wintner, Shuly and Nissim Francez. 1995a. An abstract machine for typed feature structures. In *Proceedings of the 5th Workshop on Natural Language Understanding and Logic Programming*, pages 205–220, Lisbon, May.

Wintner, Shuly and Nissim Francez. 1995b. Parsing with typed feature structures. In *Proceedings of the Fourth International Workshop on Parsing Technologies*, pages 273–287, Prague, September.

Wintner, Shuly, Evgeniy Gabrilovich, and Nissim Francez. 1997. AMALIA – a unified platform for parsing and generation. In R. Mitkov, N. Nicolov, and N. Nicolov, editors, *Proc. of "Recent Advances in Natural Language Processing" (RANLP'97)*, pages 135–142, Tzigov Chark, Bulgaria, September.