AMALIA – A Unified Platform for Parsing and Generation*

Shuly Wintner†
Seminar für Sprachwissenschaft
Universität Tübingen
Kl. Wilhelmstr. 113
72074 Tübingen, Germany
shuly@sfs.nphil.uni-tuebingen.de

Evgeniy Gabrilovich and **Nissim Francez**‡
Laboratory for Computational Linguistics
Computer Science Department
Technion, Israel Institute of Technology
32000 Haifa, Israel
{gabr,francez}@cs.technion.ac.il

Abstract

Contemporary linguistic theories (in particular, HPSG) are declarative in nature: they specify constraints on permissible structures, not how such structures are to be computed. Grammars designed under such theories are, therefore, suitable for both parsing and generation. However, practical implementations of such theories don't usually support bidirectional processing of grammars. We present a grammar development system that includes a compiler of grammars (for parsing and generation) to abstract machine instructions, and an interpreter for the abstract machine language. The generation compiler inverts input grammars (designed for parsing) to a form more suitable for generation. The compiled grammars are then executed by the interpreter using one control strategy, regardless of whether the grammar is the original or the inverted version. We thus obtain a unified, efficient platform for developing reversible grammars.

1 Introduction

The popularity of contemporary linguistic formalisms such as Lexical Functional Grammar (Kaplan & Bresnan 82), Categorial Grammar (Haddock *et al.* 87) or Head-Driven Phrase-Structure Grammar (HPSG) (Pollard & Sag 94), and especially their mathematical and formal maturity, have led to the development of various frameworks, applying different methods, for their implementation.

This paper focuses on a computational framework in which HPSG grammars can be developed. A wide spectrum of implementation techniques for HPSG exist: one extreme is direct interpretation of grammars. For parsing, this involves a program that accepts as input a grammar and a string and parses the string according to the grammar. For generation, the input is a semantic formula from which a phrase is generated. The earliest HPSG parsers (e.g., (Prudian & Pollard 85; Franz 90)) were designed in this way. A slightly more elaborate technique is the use of some high-level, unification-based logic programming language (e.g., Prolog or LIFE (Aït-Kaci & Podelski 93)) for specifying the grammar. Further along this line lies compilation of grammars directly *into* Prolog, using Prolog's internal mechanisms for performing unification. This is the implementation technique of, e.g., Profit (Erbach 94). Systems such as ALE (Carpenter 92a; Carpenter & Penn 95) also compile grammars into Prolog. However, ALE compiles grammar descriptions directly into Prolog code, rather than into (a Prolog representation of) feature structures. At run time, ALE executes the code that was compiled for the rules. Parts of the unifications (resulting from type-unification) are performed at compile-time to increase the efficiency of the generated code.

In this paper we advocate a further step along the same spectrum. We propose AMALIA, an abstract machine specifically designed for executing ALE grammars (without relational extensions). AMALIA includes a compiler of input grammars into the abstract machine language and an interpreter for the abstract instructions. This implementation technique was proved useful for many programming languages, most notably Prolog itself[1], and as we show below, it improves the efficiency of parsing with ALE grammars considerably. We emphasize in this paper the more practical aspects of the system, focusing on the integration of parsing and generation, as its theoretical infra-structure has been presented elsewhere (Wintner & Francez 95; Wintner 97).

From the point of view of grammar engineer-

* In R. Mitkov, N. Nicolov and N. Nikolov, eds., *Proceedings of "Recent Advances in Natural Language Processing" (RANLP'97)*, pp. 135-142, Tzigov Chark, Bulgaria, 11-13 September 1997

† Supported by the Minerva Stipendien Komitee.

‡ Supported by a grant from the Israeli Ministry of Science: "Programming Languages Induced Computational Linguistics" and the Fund for the Promotion of Research in the Technion. We thank an anonymous reviewer for useful comments.

[1] Recently, such techniques were used for implementing the new programming language Java.

ing, the abstract machine approach has an additional advantage. AMALIA's compiler incorporates an algorithm, based on (Samuelsson 95), for inverting grammars (designed for parsing) into a form more suitable for generation. The compiler then produces code for the inverted grammar, using exactly the same machine language. Thus, the same grammar can be compiled to two different object programs for the two different tasks. The interpreter executes both kinds of programs in the same way – only the initialization of the machine's state and the format of the final results differ. We thus obtain a uniform platform for developing grammars serving both for parsing and for generation.

We discuss the use of abstract machine techniques for compilation in the next section, and sketch the algorithm that inverts a grammar for generation in Section 3. Section 4 explains the dual operation of the abstract machine, and Section 5 lists some implementation details.

2 Why abstract machines?

High-level programming languages with dynamic structures have always been hard to develop compilers for. A common technique for overcoming the problems involves the notion of an *abstract machine*. It is a machine that, on one hand, captures the essentials of the high-level language in its architecture and instruction set, such that compiling from the source language to the (abstract) machine language becomes relatively simple. On the other hand, the architecture must be simple enough for the abstract machine language to be easily interpretable on common machines. This technique also facilitates portable front ends for compilers: as the machine language is abstract, it can be easily interpreted on different (concrete) machines/platforms.

Abstract machines were used for various procedural and functional languages, but they became prominent for logical programming languages since the introduction of the Warren Abstract Machine (WAM) (Warren 83; Aït-Kaci 91) for Prolog. While Prolog has gained a recognition as a practical implementation of the idea of programming in logic, a method for interpreting the declarative logical statements was needed for such an implementation to be well-founded. Even though there were prior attempts to construct both interpreters and compilers for Prolog, it was the WAM that gave the language not only a good, efficient compiler, but, perhaps more importantly, an elegant operational semantics.

The WAM immediately became the starting point for many compiler designs for Prolog. The techniques it delineates serve not only for Prolog proper, but also for constructing compilers for related languages: parallel Prolog compilers, variants of Prolog that use different resolution methods, extend Prolog with types or with record structures, etc. An additional advantage of abstract machines is that they are a useful tool in formally verifying the correctness of compilers.

3 Inverting grammars for generation

One of the attractions of declarative linguistic theories such as HPSG is that a single grammar, formulated in the theory, can be used both for parsing and for generation. While this is true in theory, not many practical implementations of linguistic formalisms support bidirectional grammar processing. Many advantages of bidirectional natural language systems are listed in (Strzalkowski 94), where three options for *reversibility* are considered (pp. xiii-xxi): (1) A grammar is compiled into two separate programs, parser and generator, requiring a different evaluation strategy; (2) The parser and the generator are separate programs, executed using the same evaluation strategy; (3) The parser and the generator are one program, and the evaluation strategy can handle it being run in either direction. Our solution falls into the second category: there is only one input grammar, which is compiled into two different (abstract machine) object programs; these two programs are executed using exactly the same mechanism, the interpreter, and hence employ the same strategy. This guarantees both ease of grammar development and maintenance and no loss of efficiency.

Grammars are usually 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 desirable to transform the grammar in order to enable systematic reflection of any given logical form in the productions. To this end we apply an *inversion* procedure, based upon[2] (Samuelsson

[2]Samuelsson's inversion algorithm was developed for definite clause grammars (Pereira & Warren 80). We ported it to a typed feature-structure framework.

95), to render the rules with the nested predicate-argument structure, corresponding to that of input logical forms. 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.

Figure 1 depicts a simple example grammar in ALE format[3] (`prd` stands for predicate, `a` for argument, `var` for variable, `rst` for restriction and `conn` for connective). The first rule creates a sentence (S) out of a noun phrase (NP) and a verb phrase (VP). The semantics of the S (denoted by the variable `R6`) is obtained by applying the semantics of the NP ($\lambda R5.R6$) to that of the VP. In the same way, the second rule, combining a determiner (DET) with a noun (N) to obtain an NP, applies the meaning of the DET to that of the N to obtain (after two β-reductions that are incorporated into the rule) the meaning of the NP. The lexical entries of three words are shown as well.

Figure 2 depicts (part of) the same grammar after inversion. The inverted grammar reflects the semantic argument structure, not the phrase structure. For example, the first rule creates a sentence, whose *sem* feature corresponds to $\forall R5.(R8(R5) \rightarrow R10(R5))$, from three components: an N ($R8$), a VP ($R10$) and a *semantic head*, $R3$. The string generated by the S, encoded as the value of the *str* feature (see below), is the concatenation of the strings generated by the head, the N and the VP. For such rules to be applicable, the lexicon has to be inverted, too: the "words" of the inverted grammar are atomic semantic formulae. The last three rules add syntactic information to the semantics encoded in the primitives. In addition to these inverted rules, a *semantic knowledge base* is generated, associating semantic primitives with words. It is used in the final stage of the generation, when the actual words are generated.

Grammars must satisfy certain requirements in order for them to be invertible. However, the requirements are not overly restrictive and allow encoding of a variety of natural language grammars. In particular, the semantics must be encoded by predicate-argument structures. What the inversion in fact achieves is restructuring of a grammar; this enables effective treatment of the nested structure of logical forms, so that the resulting grammar is inherently suitable for generation.

Grammar inversion is performed as part of the compilation. 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 (*str* in the example grammar). 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.

Among the other advantages of the abstract machine approach mentioned above, this technique gives an express solution for the termination problem. It is usually difficult to define when generation terminates, but once the query is given as a sequence of semantic components, they are consumed in a linear manner. While generation, just like parsing, is not guaranteed to terminate, the termination criteria of parsing apply for our generation scheme. In other words, generation in our system can be viewed as parsing ('consuming') input sequences of meaning components.

4 Unified parsing and generation

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 complete edges. The latter are represented as pointers to feature structures; the former consist of a sequence of such pointers (for the part of the edge prior to the dot) and a pointer to the compiled code (for the part succeeding the dot). 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, and the effect of the machine instructions is the same for both tasks.

AMALIA's operation for generation differs from parsing only in initialization and interpretation of the results. For parsing, the input is a string of words. Each word is looked up in the lexicon, and its associated feature structure (or feature structures, in case the word is ambiguous) is entered in the main diagonal of the chart as a complete edge. Thus, for the example gram-

[3] The signature is omitted for lack of space.

```
(phrase, syn:(syn, cat:s), sem:(R6, sem))
===>
cat> (phrase, syn:(syn, cat:np), sem:(lambda, (var:R5, rst:(R6, funct)))), % head
cat> (phrase, syn:(syn, cat:vp), sem:(lambda, (var:R7, rst:(R5, funct)))).

(phrase, syn:(syn, cat:np), sem:(R6, sem))
===>
cat> (phrase, syn:(syn, cat:det), sem:(lambda, (var:R5, rst:(R6, funct)))), % head
cat> (phrase, syn:(syn, cat:n), sem:(lambda, (var:R7, rst:(R5, funct)))).

every --->
(word, syn:(syn, cat:det),
 sem:(lambda, var:R5,
      rst:(lambda, var:R6,
           rst:(prd:(forall, var:R2, form:(bool, conn:if,
                                           wff1:(R5, a1:R2),
                                           wff2:(R6, a1:R2))),
                a1:R5, a2:R6)))).
boy --->
(word, syn:(syn, cat:n), sem:(lambda, var:R5, rst:(prd:boy, a1:R5))).

sleeps --->
(word, syn:(syn, cat:vp), sem:(lambda, var:R5, rst:(prd:sleep, a1:R5))).
```

Figure 1: A simple grammar

```
(phrase, syn:(syn,cat:s), str:[R3,R19,R22],
         sem:(R3, prd:(forall, var:R5, form:(conn:if,
                                             wff1:(R8,a1:R5),
                                             wff2:(R10,a1:R5))),
                  a1:R8, a2:R10))
===>
(phrase, syn:cat:n, sem:(lambda, rst:R8), str:R19),
(phrase, syn:cat:vp, sem:(lambda, rst:R10), str:R22),
(lambda, var:R8, rst:(lambda, var:R10, rst:R3)).

(word, syn:cat:n, sem:R3, str:[R5])
===>
(R3, lambda, var:R4, rst:(R5, prd:noun, a1:R4)).

(word, syn:cat:vp, sem:R3, str:[R5])
===>
(R3, lambda, var:R4, rst:(R5, prd:v_intrans, a1:R4)).

(word, syn:cat:det, sem:R3, str:[R10])
===>
(R3, lambda, var:(R4,a1:R6),
             rst:(lambda, var:(R8, a1:R6),
                          rst:(R10,prd:(forall, var:R6, form:(conn:if, wff1:R4, wff2:R8)),
                                   a1:R4, a2:R8))).
```

Figure 2: The inverted grammar (partial)

mar and the input "every boy sleeps", the items in the $[0,1],[1,2],[2,3]$ entries of the chart are as depicted in Figure 3.

For generation, the input is a semantic form, represented as (an ALE description of) a feature structure. The chart is initialized with (complete) edges that correspond to elements in the input semantic form, rather than to words. For example, if the input is (a feature structure encoding of) $\forall x(boy(x) \rightarrow sleep(x))$, the items in the $[0,1],[1,2],[2,3]$ entries of the chart are as depicted in Figure 4. The first item encodes $\lambda x.boy(x)$; the second – $\lambda x.sleep(x)$; and the third – $\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$.

It must be clear that there doesn't have to be a $1-1$ correspondence between the initial states of the chart in both tasks. The semantic input is scanned and its elements are (recursively) selected in a pre-defined order that is induced by the restructuring of the grammar rules (in particular,

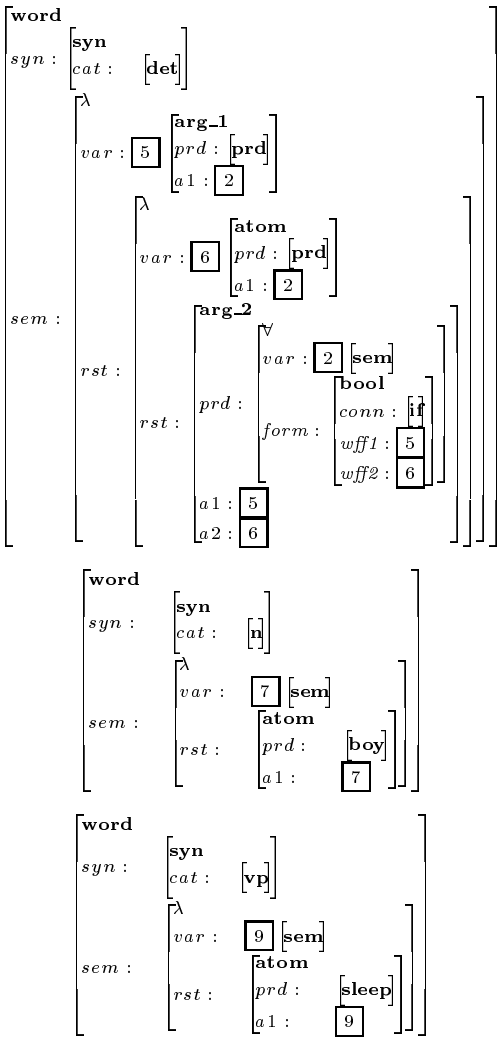

Figure 3: Initial chart entries, parsing

arguments precede the predicate).

Once the chart is initialized, the same processing strategy is applied independently of the task: the compiled program is executed on the input. The basic operation performed by the object programs is unification, which is needed for both tasks. Unification implements the *dot movement* operation that lies in the heart of chart-based parsing and generation. However, dot movement is interpreted differently for both tasks, since the (compiled) grammar rules are different: for parsing, dot movement goes over a sub-part of the input phrase; for generation, it covers a part of the input logical form.

Consider the effect of dot movement for parsing: assume that an active edge corresponding to the second rule with the dot in the initial position is applied to the lexical entry of "every", present

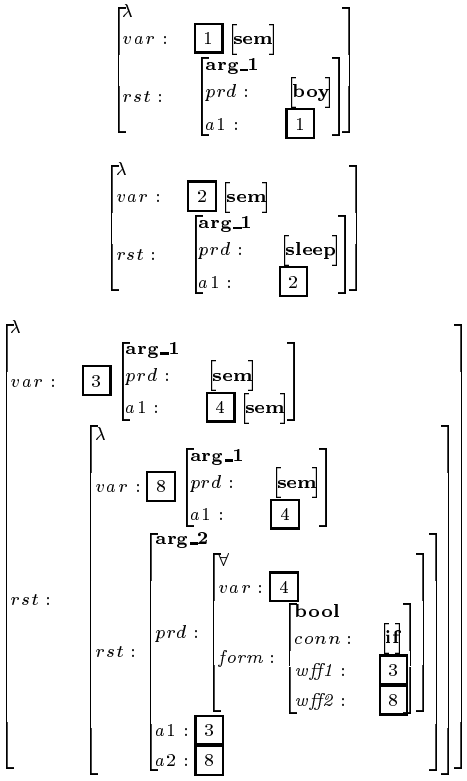

Figure 4: Initial chart entries, generation

in $[0, 1]$. The compiled code of the second rule is executed on "every"; some trivial unifications take place, but the more interesting ones bind `R5` of the rule to the value of the tag $\boxed{5}$ in the lexical entry, and `R6` – to the value of the path *sem:rst*. A new active edge is created, with these bindings recorded, and entered in $[0, 1]$. The part of the edge following the dot points to the second category in the body of this rule. Assume further that this edge is combined with (the complete edge that is) the lexical entry of "boy". Several trivial unifications take place, but the interesting ones bind `R7` in the rule to the tag $\boxed{7}$ in "boy", and `R5` of the rule to the value of *sem:rst* in "boy". Due to reentrancies among the rule's constituents, the obtained (complete) edge (spanning $[0, 2]$), whose *sem* feature indeed encodes the semantics of "every boy" ($\lambda Q.\forall x(boy(x) \rightarrow Q(x))$), is as depicted in Figure 5.

Next, we give a scenario of a generation process. It is easy to see how the last three rules of the inverted grammar are applicable to the three lexical entries of Figure 4, respectively. Assume an active edge corresponding to the first rule is present in $[0, 0]$, with the dot in the initial position. Two

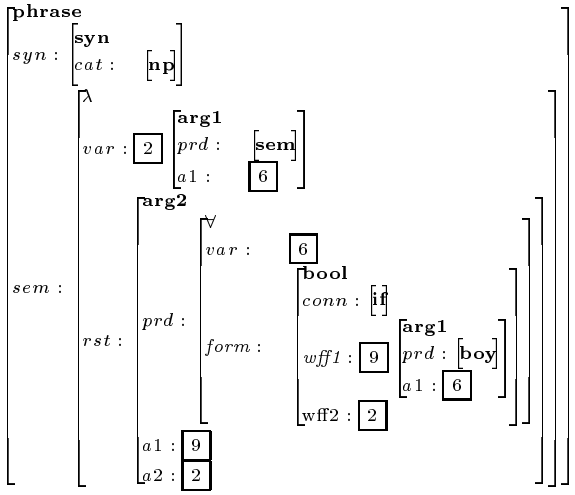

Figure 5: Parsing (intermediate) result

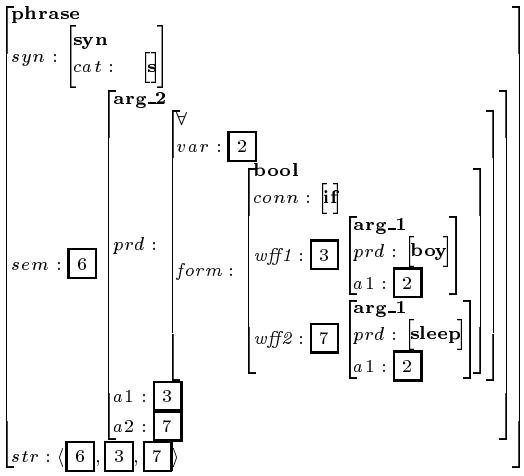

Figure 6: Generation result

dot movements, over the first two elements in the body of this rule, bind `R8` to the value of *rst* in the lexical entry of $\lambda(x).boy(x)$, and `R10` – to the value of *rst* in $\lambda(x).sleep(x)$. An active edge, with the dot in the penultimate position, is obtained in $[0, 2]$. The next dot movement applies (the code that was generated for) the last body element of the rule to the lexical entry residing in $[2, 3]$. `R8` of the rule is unified with the value of the tag [3] in this entry; since `R8` was bound by previous unifications, the value of *prd* is set to *boy*. `R10` of the rule is unified with the value of [8], and the second predicate is set to *sleep*. Finally, `R3` is unified with the value of *rst:rst* in the lexical entry; the complete edge created, spanning the entire input, is depicted in Figure 6.

The chart algorithm ends up with a (possibly empty) set of feature structures, spanning the entire input: these are all 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). For parsing, the results depict different structures of the input string. Ideally, they contain some representation of the string's semantics. This is also true for generation, with a slight difference: according to the grammar inversion algorithm, each resultant structure is guaranteed to have a feature (namely, *str*) that encodes a list of words, comprising the phrase generated. As can be seen in the example (Figure 6), the value of this feature is not a list of words but rather a list of feature structures, each of which corresponds to (i.e., is subsumed by) a lexical entry in the inverted grammar. A final post-processing stage generates all the possible strings using this list and the semantic knowledge base.

5 Implementation

This section describes the input language for AMALIA grammars and touches on some implementation details. In particular, it discusses the differences between AMALIA and ALE in terms of expressiveness and efficiency.

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.

Appropriateness, too, is specified using ALE's syntax, by listing 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 is appropriate for all its sub-types. However, AMALIA allows appropriateness loops[4] in the type hierarchy. Type constraints are not supported by AMALIA.

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

$\mathcal{A}$MALIA uses a subset of ALE's syntax for describing feature structures. As a rule, whenever $\mathcal{A}$MALIA supports ALE's functionality, it uses the same syntax. In general, $\mathcal{A}$MALIA supports totally well-typed, possibly cyclic, non-disjunctive feature structures. Set values, as in ALE, are not supported, but list values are. $\mathcal{A}$MALIA does not respect the distinction between *intensional* and *extensional* types (Carpenter 92b, Chapter 8). Also, feature structures cannot incorporate inequality constraints.

The semantics of the logical descriptions, as well as the operator precedence, follow ALE. As in ALE, partial descriptions are expanded at compilation time. $\mathcal{A}$MALIA'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.

ALE includes a built-in definite logic programming language; $\mathcal{A}$MALIA does not. The entire power of definite clause specifications is missing in $\mathcal{A}$MALIA. 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.

$\mathcal{A}$MALIA preserves ALE's syntax in describing lexical entries. Multiple lexical entries may be provided for each word, separated by semicolons. It also keeps ALE's syntax in the definition of *empty categories* (or ϵ-rules). In contrast to ALE, $\mathcal{A}$MALIA 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.

$\mathcal{A}$MALIA 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 $\mathcal{A}$MALIA. Lexical rules are not supported in this version of $\mathcal{A}$MALIA. $\mathcal{A}$MALIA's syntax for phrase structure rules is similar to ALE's, with the exception of the `cats>` specification (permitting a list of categories in the body of a rule) which is not supported.

The design details of the abstract machine are outside the scope of this paper; the reader is referred to (Wintner & Francez 95; Wintner 97) for more information on the machine itself and to (Gabrilovich 97) for a detailed description of the grammar inversion. A practical description of $\mathcal{A}$MALIA, its deviations from ALE and a complete user's guide, are given in (Wintner *et al.* 97).

$\mathcal{A}$MALIA is implemented in *C*, augmented by *yacc*, *lex* and *Tcl/Tk* (Ousterhout 94). It was tested on various Unix platforms and on IBM PCs. Two versions of $\mathcal{A}$MALIA exist: an interactive, easy-to-use, graphically interfaced system 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. In addition, the system functions as a graphical development framework for grammar engineers by providing some tracing and debugging options. The user can direct the system to execute a program in its entirety, to break at a certain instruction or to proceed in steps, stopping after each executed instruction. Throughout the process of grammar execution, the abstract machine's internal state is displayed for the user to inspect. The main data structure upon which feature structures are being built, the *heap*, is displayed, along with the machine's general-purpose and special-purpose *registers*. Moreover, the contents of the chart can be graphically displayed at any time and the derived structure can be recovered. Grammar development becomes an easier, simpler process.

The system was tested with a wide variety of grammars, mostly adaptations of existing ALE grammars. While most of the example grammars are rather small, we believe that the system can handle real-scale grammar quite efficiently; however, to accommodate large type hierarchies some major space optimizations must be introduced. It is important to emphasize that $\mathcal{A}$MALIA does not provide the wealth of input specifications ALE does. On the other hand, development of grammars in $\mathcal{A}$MALIA is made easier due to the GUI

and the improved performance over ALE. The support of generation is unique to our system.

To compare $\mathcal{A}$MALIA with ALE we have used a few benchmark grammars. The first is an early version of an HPSG-based Hebrew grammar described in (Wintner 97). It consists of 4 rules and one empty category; the type hierarchy contains 84 types and 32 features, and the lexicon contains 13 words. The second is an HPSG-based grammar for a subset (emphasizing relative clauses) of the Russian language described in (Gabrilovich & Estrin 96). It consists of 8 rules and 76 lexical entries; the type hierarchy contains 151 types and 31 features. The third example is a simple grammar generating the language $\{a^n b^n \mid n > 0\}$. Both systems were used to compile the same grammar and to parse the same strings. The results of a performance comparison of $\mathcal{A}$MALIA and ALE are listed in Figure 7 (all times are in seconds; n indicates the input string's length and r – the number of results). While the execution times for the last grammar are less impressing, the differences in compilation time indicate a major advantage in using $\mathcal{A}$MALIA for instructional purposes; in such cases grammars are compiled over and over again, while they are usually executed only a few times. Limited experiments we have conducted reveal that generation (as well as compilation for generation) is 40%–100% slower than parsing (we do not know of good benchmarks for generation).

task	ALE	$\mathcal{A}$MALIA
Grammar 1		
Compilation	35.0	1.4
Parsing, n=6, r=2	0.5	0.5
Parsing, n=10, r=8	3.2	0.8
Parsing, n=14, r=125	140.0	9.0
Grammar 2		
Compilation	68.0	2.3
Parsing, n=2, r=2	0.5	0.8
Parsing, n=4, r=2	2.4	0.9
Parsing, n=7, r=2	5.1	1.1
Parsing, n=8, r=2	7.8	1.2
Parsing, n=12, r=2	17.0	1.5
Grammar 3		
Compilation	6.5	0.2
Parsing, n=4	0.1	0.2
Parsing, n=8	0.8	0.3
Parsing, n=16	2.8	1.1
Parsing, n=32	26.0	16.0

Figure 7: Performance comparison of ALE and $\mathcal{A}$MALIA

References

(Aït-Kaci & Podelski 93) Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195–234, July-August 1993.

(Aït-Kaci 91) Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming. The MIT Press, Cambridge, Massachusetts, 1991.

(Carpenter & Penn 95) Bob Carpenter and Gerald Penn. Compiling typed attribute-value logic grammars. In Harry Bunt and Masaru Tomita, editors, *Current Issues in Parsing Technologies*, volume 2. Kluwer, 1995.

(Carpenter 92a) Bob Carpenter. ALE – the attribute logic engine: User's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213, December 1992.

(Carpenter 92b) Bob Carpenter. *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

(Erbach 94) Gregor Erbach. ProFIT: Prolog with features, inheritance and templates. CLAUS Report 42, Computerlinguistik, Universität des Saarlandes, D-66041, Saarbrücken, Germany, July 1994.

(Franz 90) Alex Franz. A parser for HPSG. Report CMU-LCL-90-3, Laboratory for Computational Linguistics, Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA 15213, July 1990.

(Gabrilovich & Estrin 96) Evgeniy Gabrilovich and Arkady Estrin. An HPSG grammar for the Russian language. To appear as a technical report, Laboratory for Computational Linguistics, the Technion, 1996.

(Gabrilovich 97) Evgeniy Gabrilovich. Natural language generation by abstract machine. M.Sc. thesis, Technion, Israel Institute of Technology, Haifa, Israel, 1997. In preparation.

(Haddock *et al.* 87) Nicholas Haddock, Ewan Klein, and Glyn Morill, editors. *Categorial Grammar, Unification and Parsing*, volume 1 of *Working Papers in Cognitive Science*. University of Edinburgh, Center for Cognitive Science, 1987.

(Kaplan & Bresnan 82) R. Kaplan and J. Bresnan. Lexical functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, Mass., 1982.

(Ousterhout 94) John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

(Pereira & Warren 80) Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

(Pollard & Sag 94) Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, 1994.

(Prudian & Pollard 85) Derek Prudian and Carl Pollard. Parsing head-driven phrase structure grammar. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, IL., 1985. University of Chicago.

(Samuelsson 95) Christer Samuelsson. An efficient algorithm for surface generation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

(Strzalkowski 94) Tomek Strzalkowski, editor. *Reversible Grammar in Natural Language Processing*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1994.

(Warren 83) David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA., August 1983.

(Wintner & Francez 95) Shuly Wintner and Nissim Francez. 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 1995.

(Wintner 97) Shuly Wintner. *An Abstract Machine for Unification Grammars*. PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, January 1997.

(Wintner *et al.* 97) Shuly Wintner, Evgeniy Gabrilovich, and Nissim Francez. AMALIA – Abstract MAchine for LInguistic Applications – user's guide. Laboratory for Computational Linguistics, Computer Science Deparmtent, Technion, Israel Institute of Technology, 32000 Haifa, Israel, January 1997.