ACGtk: A Toolkit for Developing and Running Abstract Categorial Grammars

Maxime Guillaume, Sylvain Pogodalla, Vincent Tourneur

ACGtk: A Toolkit for Developing and Running Abstract Categorial Grammars

Maxime Guillaume[1,2], Sylvain Pogodalla[1], and Vincent Tourneur[1]

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
{maxime.guillaume,sylvain.pogodalla,vincent.tourneur}@inria.fr
[2] YSEOP

Abstract. Abstract categorial grammars (ACGs) is an expressive grammatical framework whose formal properties have been extensively studied. While it can provide its own account, as a grammar, of linguistic phenomena, it is known to encode several grammatical formalisms, including context-free grammars, but also mildly context-sensitive formalisms such as tree-adjoining grammars or m-linear context-free rewriting systems for which parsing is polynomial. The ACG toolkit we present provides a compiler, `acgc`, that checks and turns ACGs into representations that are suitable for testing and parsing, used in the `acg` interpreter. We illustrate these functionalities and discuss implementation features, in particular the Datalog reduction on which parsing is based, and the magic set rewriting techniques that can further be applied.

Keywords: Abstract Categorial Grammars · Natural Language Processing · OCaml · Datalog.

1 Introduction

Abstract categorial grammars [5, ACGs] is an expressive grammatical framework, designed to account both for the syntax and the semantics of natural languages. ACGs derive from type-theoretic grammars in the tradition of [22,3,24]. While they can provide their own account of linguistic phenomena, they can also be considered as a framework in which several grammatical formalisms may be encoded [10,6,25], including context-free grammars, but also mildly context-sensitive formalisms such as tree-adjoining grammars [12,13] or m-linear context-free rewriting systems [30,31] for which parsing is polynomial. Its formal properties have been extensively studied [10,28,18,16].

The definition of an ACG is based on a small set of mathematical primitives from type theory, λ-calculus, and linear logic. These primitives combine via simple composition rules, offering ACGs a good flexibility. In particular, ACGs generate languages of linear λ-terms, which generalize both string and tree languages, but also allow for using higher-order logic to express semantic representations.

A key feature of ACGs is to provide the user direct control over the parse structures of the grammar, the *abstract language*, defined over an abstract vocabulary (i.e., a higher-order signature), which can be seen as the set of admissible

parse structures of the grammar. Such structures are later on interpreted by a morphism, the *lexicon*, to get the *object language*, defined over an object vocabulary (i.e., a higher-order signature). The process of recovering an abstract structure from an object term is called *ACG parsing* and consists in inverting the lexicon.

In this article, we present the ACG toolkit, ACG`tk`, which provides a compiler, `acgc`, that checks and turns ACGs into representations that are suitable for testing and parsing, used in the `acg` interpreter. It is implemented in OCaml and distributed under a free license. We illustrate its functionalities and discuss implementation features, in particular the Datalog reduction on which parsing is based, and the magic set rewriting techniques that can further be applied.

2 Abstract Categorial Grammars

The syntax we use in ACG`tk` to define grammars is very faithful to their mathematical definitions we introduce next. In order to illustrate the definitions, concepts, and functionalities, we elaborate on an example made very simple on purpose. More involved ones were for instance proposed in [23,25]. Formal definitions are illustrated with examples of ACG`tk` source listings, to be compiled using `acgc` (as in Grammar 1), and with ACG`tk` commands, to be run with `acg` (as in Commands 1).[3]

Definition 1 (Types). *Let A be a set of atomic types. The set $\mathscr{T}(A)$ of* implicative types *built upon A is defined with the following grammar:*

$$\mathscr{T}(A) ::= A \mid \mathscr{T}(A) \to \mathscr{T}(A) \mid \mathscr{T}(A) \Rightarrow \mathscr{T}(A)$$

where $\to$ is the linear implication and $\Rightarrow$ is the intuitionistic implication. They are usually denoted by $\multimap$ and $\to$, resp. However, because ACG`tk` use the ASCII `->` (or UTF-8 $\to$) and the ASCII `=>` (or UTF-8 $\Rightarrow$) arrows, we use this notation in this article.

Definition 2 (Higher-Order Signatures). *A* higher-order signature Σ *is a triple $\Sigma = \langle A, C, \tau \rangle$ where:*

- *A is a finite set of atomic types;*
- *C is a finite set of constants;*
- *$\tau : C \to \mathscr{T}(A)$ is a function assigning types to constants.*

As for the implication that comes with the two linear and intuitionistic flavors, the λ-terms also feature two abstractions: the linear one (`lambda` in ASCII and λ^o in UTF-8) and the regular one (`Lambda` in ASCII and λ in UTF-8).

Definition 3 (λ-Terms). *Let X be an infinite countable set of λ-variables. The set $\Lambda(\Sigma)$ of λ-terms built upon a higher-order signature $\Sigma = \langle A, C, \tau \rangle$ is inductively defined as follows (the typing rules are provided in Appendix A):*

[3] Grammars and command files used in this article are available at https://inria.hal.science/hal-04479621/file/acg-examples.zip.

- *if* $c \in C$ *then* $c \in \Lambda(\Sigma)$*;*
- *if* $x \in X$ *then* $x \in \Lambda(\Sigma)$*;*
- *if* $x \in X$ *and* $t \in \Lambda(\Sigma)$ *and* x *occurs free in* t *exactly once, then* $\lambda^{\circ} x.t \in \Lambda(\Sigma)$ *(*linear *abstraction);*
- *if* $x \in X$ *and* $t \in \Lambda(\Sigma)$*, then* $\lambda x.t \in \Lambda(\Sigma)$ *(abstraction);*
- *if* $t, u \in \Lambda(\Sigma)$ *then* $(t\, u) \in \Lambda(\Sigma)$*.*

Example 1 (Trees). Let us assume a ranked alphabet $A_1 = \{John, Mary, saw, everyone, \mathsf{NP}_1, \mathsf{Vt}_1, \mathsf{S}_2, \mathsf{VP}_2\}$ with the arity given by the subscript (the arity is 0 if there is no subscript). Trees such as the ones of Figure 1 can be expressed using this alphabet.

A signature describing the trees that can be built over this ranked alphabet is $\mathsf{Trees} = \langle\{\tau\}, \{John, Mary, saw, everyone, \mathsf{NP}_1, \mathsf{Vt}_1, \mathsf{S}_2, \mathsf{VP}_2\}, \tau_{\text{trees}}\rangle$, with

$$\tau_{\text{trees}} = \begin{cases} John \mapsto \tau & \mathsf{NP}_1 \mapsto \tau \to \tau \\ Mary \mapsto \tau & \mathsf{Vt}_1 \mapsto \tau \to \tau \\ saw \mapsto \tau & \mathsf{VP}_2 \mapsto \tau \to \tau \to \tau \\ everyone \mapsto \tau & \mathsf{S}_2 \mapsto \tau \to \tau \to \tau \end{cases}$$

Grammar 1 shows how such a signature is declared in ACG`tk`. For instance, in such a signature, the trees of Fig. 1 are encoded by:

$$t_{1a} = \mathsf{S}_2\,(\mathsf{NP}_1\ John)\,(\mathsf{VP}_2\,(\mathsf{Vt}_1\ saw)\,(\mathsf{NP}_1\ Mary)) : \tau$$
$$t_{1b} = \mathsf{S}_2\,(\mathsf{NP}_1\ John)\,(\mathsf{Vt}_1\ saw) : \tau$$

The interpreter `acg` can check they are well typed, as Commands 1 shows.

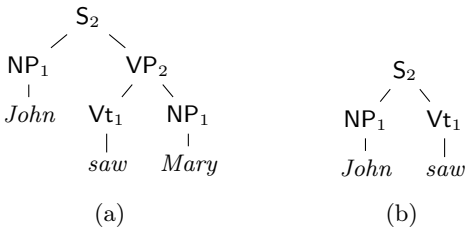


Fig. 1: Sample of trees built over the ranked alphabet A_1 of Example 1

Example 2 (Strings). Strings build over an alphabet T are encoded in a higher-order signature $\Sigma_{str} = \langle\{o\}, T, \tau_{\text{str}}\rangle$ where for all $s \in T, \tau_{\text{str}}(s) = o \to o$. It is then easy to check that if we define the type $\sigma \stackrel{\Delta}{=} o \to o$, the infix operator $+ \stackrel{\Delta}{=} \lambda x\, y.\, \lambda^{\circ} z.\, x\,(y\, z)$ and the empty string $\epsilon \stackrel{\Delta}{=} \lambda^{\circ} x.\, x$, $+$ is associative and ϵ neutral for $+$.

So we can define a string signature $\mathsf{Strings}$ corresponding to the yields of the trees built over the ranked alphabet A_1 as in Grammar 2 (a string constant is

```
signature Trees =
  τ          : type;                    John, Mary, saw, everyone : τ;
  NP1, Vt1 : τ → τ;                     S2, VP2                   : τ → τ → τ;
end
```

Grammar 1: A higher-order signature for representing the trees that can be built over the ranked alphabet A_1 of Example 1

```
ACGtk>"S2 (NP1 John) (VP2 (Vt1 saw) (NP1 Mary)) : τ" | check signature = Trees
1 term computed.
ACGtk>"S2 (NP1 John) (Vt1 saw) : τ" | check signature = Trees
1 term computed.
ACGtk>"S2 (NP1 John) (VP2 (Vt1 saw)) : τ" | check signature = Trees
[ERROR] Type error: line 1, characters 16-28:
The type of this expression is "τ" but is used with type "τ → τ".
```

Commands 1: Checking well-formedness and well-typedness of a term against a signature

introduced for each symbol of arity 0, while the other symbols are not present anymore). The string $John + saw + Mary$ will then be represented by the term $\lambda^{\circ} z.\, John\,(saw\,(Mary\,z))$.

```
signature Strings =
  o          : type;
  σ = o → o : type;                     John, Mary, saw, everyone  : σ;
  ε = λ°x. x  : σ;                      infix+ = λx y. λ°z. x (y z) : σ → σ → σ;
end
```

Grammar 2: A higher-order signature for representing strings built over the alphabet $\{John, Mary, saw, everyone\}$

Definition 4 (Lexicon). *Let $\Sigma_1 = \langle A_1, C_1, \tau_1\rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2\rangle$ be two higher-order signatures, a lexicon $\mathscr{L} = \langle F, G\rangle$ from Σ_1 to Σ_2 is such that:*

- *$F : A_1 \to \mathscr{T}(A_2)$. We also note $F : \mathscr{T}(A_1) \to \mathscr{T}(A_2)$ its homomorphic extension;*
- *$G : C_1 \to \Lambda(\Sigma_2)$. We also note $G : \Lambda(\Sigma_1) \to \Lambda(\Sigma_2)$ its homomorphic extension;*

– *F and G are such that for all $c \in C_1$, $\vdash_{\Sigma_2} G(c) : F(\tau_1(c))$ is provable.*

We also use $\mathscr{L}$ instead of F or G.

The lexicon is the interpreting device of ACGs. If $\mathscr{L}(t) = u$, also denoted by $t{:=}\, u$, we say that u is the *interpretation* or the *realization* of t.

Example 3 (Relating the signature Trees and the signature Strings). We can now relate the signature Trees and the signature Strings with the lexicon Yield to interpret the trees built over the ranked alphabet as strings. The lexicon we use is defined in Grammar 3. Note that the lexicon satisfies the condition ensuring that the type of the interpretation of a constant is the interpretation of the type of the constant.

It is then straightforward to check that the term $t_{1a} = \mathsf{S}_2\,(\mathsf{NP}_1\ \mathit{John})\,(\mathsf{VP}_2\ (\mathsf{Vt}_1\ \mathit{saw})\,(\mathsf{NP}_1\ \mathit{Mary})) : \tau$ is interpreted as the string $\mathit{John} + \mathit{saw} + \mathit{Mary} = \lambda^{\circ} z.\ \mathit{John}\,(\mathit{saw}\,(\mathit{Mary}\,z))$.

```
lexicon Yield (Rules) : Strings =
  τ          := σ;          John       := John;          Mary := Mary;
  saw        := saw;        everyone   := everyone;
  NP₁, Vt₁   := λ°x. x;     S₂, VP₂    := λ°x y. x + y;
end
```

Grammar 3: A lexicon for interpreting the trees as strings

The formal definition of ACGs is not used as such in ACG`tk` because it's enough to have a lexicon (hence two signatures) and an abstract type A in order to parse an (object) term and get an (abstract) term, if any, of type A. The `parse` command of the `acg` interpreter requires these two parameters (see Section 4). But for the sake of completeness, we provide here the definitions of ACGs and of the languages they generate.

Definition 5 (Abstract Categorial Grammar and vocabulary). *An* abstract categorial grammar *is a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, S\rangle$ where:*

– *$\Sigma_1 = \langle A_1, C_1, \tau_1\rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2\rangle$ are two higher-order signatures. Σ_1 (resp. Σ_2) is called the* abstract vocabulary *(resp. the* object vocabulary*) and $\Lambda(\Sigma_1)$ (resp. $\Lambda(\Sigma_2)$) is the set of* abstract terms *(resp. the set of* object terms*).*
– *$\mathscr{L} : \Sigma_1 \to \Sigma_2$ is a lexicon.*
– *$S \in \mathscr{T}(A_1)$ is the* distinguished type *of the grammar.*

Definition 6 (Abstract and Object Languages). *Given an ACG $\mathscr{G}$, and S its distinguished type, the* abstract language *of $\mathscr{G}$ is defined by*

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \vdash_{\Sigma_1} t : S \text{ is derivable}\}$$

The object language *of $\mathscr{G}$ is defined by*

$$\mathcal{O}(\mathscr{G}) = \{u \in \Lambda(\Sigma_2) \mid \exists t \in \mathcal{A}(\mathscr{G}) \text{ such that } u = \mathscr{L}(t)\}$$

Parsing with an ACG $\mathscr{G}$ any term u that is built over the object vocabulary of $\mathscr{G}$ amounts to finding the abstract terms $t \in \mathcal{A}(\mathscr{G})$ such that $u = \mathscr{L}(t)$. In other words, ACG parsing is morphism inversion.

3 Properties

3.1 Expressive Power

Two parameters are useful to describe the hierarchy of ACGs with respect to their expressive power: the *order* and the *complexity* of an ACG.

Definition 7 (Order and complexity of an ACG, and ACG hierarchy). *The order of a type α, $ord(\alpha)$, is defined inductively on α as follows:*

$$\begin{aligned} ord(a) &= 1 \\ ord(\alpha \to \beta) &= \max(ord(\alpha) + 1, ord(\beta)) \\ ord(\alpha \Rightarrow \beta) &= \max(ord(\alpha) + 1, ord(\beta)) \end{aligned}$$

The order *of an ACG is the maximum of the orders of its abstract constants. The* complexity *of an ACG is the maximum of the orders of the interpretations by the lexicon of its abstract atomic types.*

We call second-order ACGs *the set of ACGs whose order is at most 2. $ACG_{(n,m)}$ denotes the set of ACGs whose order is at most n and whose complexity is at most m.*

Second-order ACGs are a particular class of interest because of its polynomial parsing property [27]. Table 1, from [25], sums up some of the formal properties of second-order ACGs.

	String language	Tree language
$ACG_{(1,n)}$	finite	finite
$ACG_{(2,1)}$	regular	regular
$ACG_{(2,2)}$	context-free	linear context-free
$ACG_{(2,3)}$	non-duplicating macro well-nested multiple context-free	$\subset$ 1-visit attribute grammar
$ACG_{(2,4)}$	mildly context-sensitive (multiple context-free)	tree-generating hyperedge replacement gram.
$ACG_{(2,4+n)}$	$ACG_{(2,4)}$	$ACG_{(2,4)}$

Table 1: The hierarchy of second-order ACGs

ACG`tk` implements parsing of second-order ACGs that are almost linear, i.e., terms in which variables occur exactly once, except for variables of atomic type that occur at least once, but possibly several times.

Higher-order ACGs can generate languages that are NP-complete [32,28]. In general, the problem of parsing with higher-order ACGs is equivalent to the open problem of provability in multiplicative exponential linear logic [9,32]. Even if there is a semi-complete algorithm for parsing with such grammars (see Sect. 3.3), it is currently not implemented in ACG`tk`. However, the interpretation of terms is available for any ACG (not only second-order ones).

3.2 ACG Composition

Because both abstract and object languages are sets of λ-terms, ACGs have built-in support for *grammar composition*. Two ways of composing ACGs are available:

- ACGs can be composed by making the abstract structures of a grammar the object structures of another ACG. This corresponds to the *applicative* composition paradigm (function composition of the lexicons) of [5] and is illustrated in Fig. 2 by $\mathscr{L}_{Derive}$ (from $\Lambda(\Sigma_{Rules})$ to $\Lambda(\Sigma_{Trees})$) and $\mathscr{L}_{Yield}$ (from $\Lambda(\Sigma_{Trees})$ to $\Lambda(\Sigma_{str})$).
- ACGs can also be composed by having a shared abstract vocabulary. Two terms of the two object languages are related if they share a common abstract structure. This corresponds to the *transductive* composition paradigm of [5] and is typically used to relate a surface form (e.g., as a string) and a semantic form (e.g., as a logical formula) through a common parse structure. This composition is illustrated in Fig. 2 by $\mathscr{L}_{Yield} \circ \mathscr{L}_{Derive}$ and $\mathscr{L}_{Semantics}$ that share the same abstract vocabulary, Σ_{Rules}.

This composition ability makes ACG a modular framework.

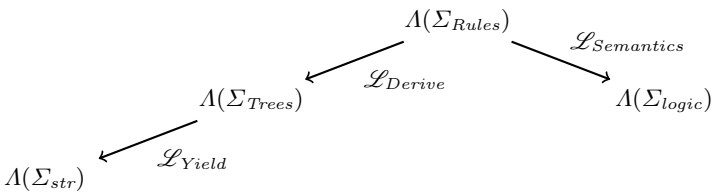


Fig. 2: An example of ACG composition. The structures are the ones used as examples in the article

Example 4 (Encoding a context-free grammar). Let $G = \langle T, N, P, \mathsf{S}\rangle$ be the context-free grammar such that:

- the set of terminals T is $\{\mathit{John}, \mathit{Mary}, \mathit{saw}, \mathit{everyone}\}$,

– the set of non terminals N is $\{\mathsf{S}, \mathsf{S}', \mathsf{NP}, \mathsf{VP}, \mathsf{Vt}\}$ (S' will be used when computing the semantic representation associated to a tree, see Sec. 3.2),
– the set of production rules P is $\{\rho_i \mid 0 \leq i \leq 6\}$ where:

$$\begin{array}{lll} \rho_0 = & \mathsf{S}' \to \mathsf{S} & \rho_4 = \quad \mathsf{NP} \to \mathit{Mary} \\ \rho_1 = & \mathsf{S} \to \mathsf{NP}\ \mathsf{VP} & \rho_5 = \quad \mathsf{NP} \to \mathit{everyone} \\ \rho_2 = & \mathsf{VP} \to \mathsf{Vt}\ \mathsf{NP} & \rho_6 = \quad \mathsf{Vt} \to \mathit{saw} \\ \rho_3 = & \mathsf{NP} \to \mathit{John} & \end{array}$$

A signature describing the derivation trees of this context-free grammar is $\mathsf{Rules} = \langle \{\mathsf{S}, \mathsf{S}', \mathsf{NP}, \mathsf{VP}, Vt\}, \{\rho_i \mid 0 \leq i \leq 6\}, \tau \rangle$ where:

$$\tau = \begin{cases} \rho_0 : \mathsf{S} \to \mathsf{S}' & \rho_4 : \mathsf{NP} \\ \rho_1 : \mathsf{NP} \to \ \mathsf{VP} \to \mathsf{S} & \rho_5 : \mathsf{NP} \\ \rho_2 : \mathsf{Vt} \to \ \mathsf{NP} \to \mathsf{VP} & \rho_6 : \mathsf{Vt} \\ \rho_3 : \mathsf{NP} & \end{cases}$$

Grammar 4 shows how such a signature is declared in ACG`tk`.

```
signature Rules =
  S, S', NP, VP, Vt : type;                          ρ0 : S → S';
  ρ1                : NP → VP → S;                   ρ2 : Vt → NP → VP;
  ρ3, ρ4, ρ5        : NP;                            ρ6 : Vt;
end
lexicon Derive (Rules) : Trees =
            S, S', NP, VP, Vt := τ;
  ρ1 := λ°l r. S2 l r;     ρ2 := λ°l r. VP2 l r;     ρ0 := λ°x. x;
  ρ4 := NP1 Mary           ρ5 := NP1 everyone        ρ3 := NP1 John
                                                     ρ6 := Vt1 saw
end
lexicon CFG = Yield << Derive
```

Grammar 4: A higher-order signature for representing the derivation trees of the CFG G and a lexicon to interpret them as syntactic (derived) trees, and as strings (by composition)

Example 5 (Relating the signature Rules and the signature Trees). Not all the trees built over the ranked alphabet A_1 correspond to derivation trees of the CFG G. For instance, the syntactic (derived) tree of Fig. 1b does not correspond to any derivation tree of G. By relating the signature Rules and Trees, we are able to discriminate between trees corresponding to actual derivations (admissible parse structures) and the other ones. The lexicon we use is defined in Grammar 4.[4]

[4] This might seem an overkill when dealing with context-free grammars, as the derivation trees are taken to be the syntactic trees. However, when encoding tree-adjoining grammars, it is not the case anymore and such a composed architecture is indeed used in [25].

It is then straightforward to check that $\rho_1\,\rho_3\,(\rho_2\,\rho_6\,\rho_4) : \mathsf{S}$ is interpreted as the term $t_{1a} = \mathsf{S}_2\,(\mathsf{NP}_1\,\mathit{John})\,(\mathsf{VP}_2\,(\mathsf{Vt}_1\,\mathit{saw})\,(\mathsf{NP}_1\,\mathit{Mary}))$ of type τ (tree). This term can in turn be interpreted by the lexicon Yield to get the term $\mathit{John}+\mathit{saw}+\mathit{Mary}$ as Commands 2 shows.

Grammar 4 also shows how to define a new lexicon CFG as the composition $\mathscr{L}_{CFG} = \mathscr{L}_{Yield} \circ \mathscr{L}_{Rules}$.

```
ACGtk>"ρ1 ρ3 (ρ2 ρ6 ρ4) : S" | realize lexicons = Derive
Term : S2 (NP1 John) (VP2 (Vt1 saw) (NP1 Mary)) : τ
1 term computed.
ACGtk>"S2 (NP1 John) (VP2 (Vt1 saw) (NP1 Mary)) : τ"| realize lexicons = Yield
Term : λ°z. John (saw (Mary z)) : o → o
1 term computed.
```

Commands 2: Different interpretations of a term representing a context-free derivation of G

Example 6 (Providing a semantic interpretation to the context-free derivations of G). We now introduce a signature to build logical terms and a new lexicon Semantics that shares its abstract vocabulary, Rules, with Derive (Grammar 5). It is beyond the scope of this article to explain why these semantic interpretations are meaningful, but they correspond to the ones of [2] that provide a continuation based semantics to natural language expressions.

For instance, the term $t_6 = \rho_0\,(\rho_1\,\rho_3\,(\rho_2\,\rho_6\,\rho_4)) : \mathsf{S}'$ is interpreted by Semantics as the term $t_{\text{sem}} = \mathsf{see\,j\,m} : t$. This allows us to relate, by the transductive composition mode, the string $\mathit{John} + \mathit{saw} + \mathit{Mary}$, which is the interpretation of t_6 by $\mathscr{L}_{CFG} = \mathscr{L}_{Yield} \circ \mathscr{L}_{Rules}$, to the logical term t_{sem}. The same holds for $\mathit{John} + \mathit{saw} + \mathit{everyone}$ and $\forall\, x.\mathsf{see\,j}\,x$.

Commands 3 illustrates these relations using the `parse` command in both direction (i.e., from strings to logical representations and from logical representations to strings). The ACGs we defined indeed are second order and parsing is available.

3.3 Parsing With Abstract Categorial Grammars and Datalog Reduction

Parsing with ACGs had first been studied as linear higher-order matching, for which several complexity results were established [4,7]. However, [14] showed how parsing of second-order ACGs reduces to Datalog querying, grounding parsing algorithms and optimization techniques on well-established fields, such as database

```
signature Logic =
  e, t : type;                                   j, m      : e;
  see : t → e → t;                               binder ∃ : (e ⇒ t) → t;
end
lexicon Semantics (Rules) : Logic =
  S′  := t;                  ρ0 := λ°P. P (λ°p. p);     ρ3 := λ°P. P j;
  S   := (t → t) → t;        ρ4 := λ°P. P m;            ρ5 := λ°P. ∀ x. P x;
  NP  := (e → t) → t;        ρ6 := λ°R. R saw;
  VP  := ((e → t) → t) → t;  Vt := ((e → e → t) → t) → t;
  ρ1  := λ°s v. λ°p. v (λ°P. s (λ°x. p (P x)));
  ρ2  := λ°v o. λ°s. v (λ°R. o (λ°y. s (λ°x. R x y)));
end
```

Grammar 5: A higher-order signature for representing higher-order logical formulas

```
ACGtk>"ρ0 (ρ1 ρ3 (ρ2 ρ6 ρ4)) : S′" | realize lexicons = CFG, Semantics
Term : λ°z. John (saw (Mary z)) : o → o
Term : see j m : t
2 terms computed.
ACGtk>"John + saw + Mary : σ" | parse lexicon = CFG type = S′ |
        realize lexicons = Semantics
Parsing time: 76.6μs
Parse forest building time: 6.31μs
Term : see j m : t
1 term computed.
ACGtk>"John + saw + everyone : σ" | parse lexicon = CFG type = S′ |
        realize lexicons = Semantics
Parsing time: 76.6μs
Parse forest building time: 6.31μs
Term : ∀ x.see j x : t
1 term computed.
ACGtk>"see j m : t" | parse lexicon = Semantics type = S′ |
        realize lexicons = CFG
Parsing time: 76μs
Parse forest building time: 5.27μs
Term : λ°z. John (saw (Mary z)) : o → o
1 term computed.
ACGtk>"∀x.see j x : t" | parse lexicon = Semantics type = S′ |
        realize lexicons = CFG
Parsing time: 74.7μs
Parse forest building time: 4.48μs
Term : λ°z. John (saw (everyone z)) : o → o
1 term computed.
```

Commands 3: Illustration of the transductive composition mode

and logic programming, and offering a general method for getting efficient tabular parsing algorithms [17]. This method applies whatever the object language: representing strings, trees, but also any kind of (almost linear) λ-terms as exemplified in Commands 3. This also allows for deriving algorithms with specific properties such as prefix correctness in a general way.[5] ACG`tk` implements the Datalog reduction with `acgc` and Datalog evaluation in `acg` to parse object terms as illustrated by Fig. 3.

Because we are interested in parse structures, and not only in membership, not only do we need the answer to the query corresponding to an object term to parse, but also the *Datalog derivations* (and all of them) that prove the query. Then, any Datalog derivation uniquely determines an abstract term, hence the abstract terms corresponding to the term being parsed. Standard engines, to our knowledge, either do not provide proof trees or do so only partially for debugging purposes and explainability. Therefore, ACG`tk` relies on our own implementation of a Datalog prover that enumerates all the possible derivations.

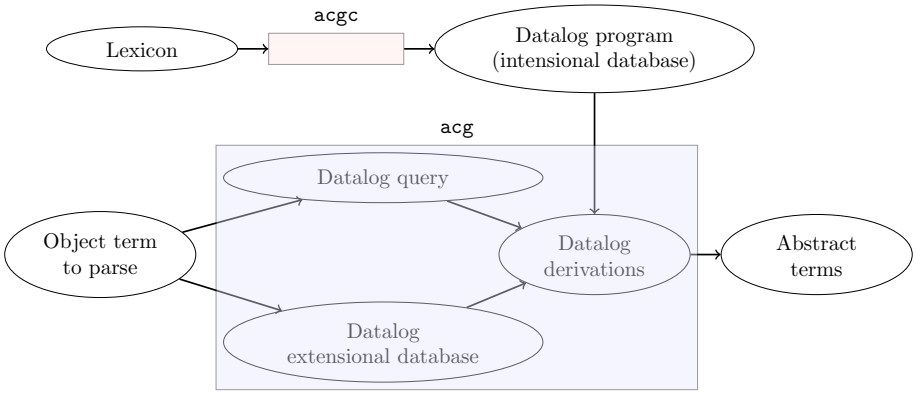


Fig. 3: Parsing process

For higher-order ACGs, [8] shows a similar reduction of ACG parsing to linear logic programming, leading to a semi-complete parsing algorithm (i.e., if the algorithm returns a solution, the term to parse belongs to the object language), not yet implemented.

4 ACG`tk`

ACG`tk` is implemented in OCaml, an efficient and expressive functional programming language (the object-oriented features are not used, though). This language provides some features that ease the implementation of the structures

[5] For a n^6 prefix-correct Earley recognizer for TAGs, see [15].

and algorithms of ACG`tk`, such as algebraic data types with pattern matching, an efficient compiler with tail-call optimization, a fast garbage collector.

It is also fully integrated with its development environment: using libraries provided by the OCaml package manager (opam), using the Dune build system, etc. It provides two pieces of software: `acgc` and `acg`.

The Grammar Compiler: `acgc` The main feature of `acgc` is to provide binary representations (including the associated Datalog reductions for the lexicons) of ACGs. Grammars are parsed, terms are statically type-checked against signatures and lexicon interpretations. Separate compilation is supported, so that different grammars may be coded in different source files.

The Interpreter: `acg` It provides a simple command language to use and test grammars. The most useful commands are:

- `check` to check well-formedness and well-typedness against a signature
- `realize` to interpret a term according to a lexicon
- `parse` to parse a term according to a second-order lexicon
- `list-terms` to (randomly or not) list terms of a given type built over a signature (including higher-order ones)
- `idb` to show the Datalog program associated to a lexicon
- `query` to show the query and the extensional database associated to a term to parse according to a second-order lexicon

5 Notable Implementation Features

5.1 Datalog Evaluation, Tabular Parsing and Shared Forest

As explained in Section 3.3, ACG`tk` implements parsing using proof search in Datalog. Currently, it uses the bottom-up seminaive algorithm of [1] together with a CYK-like chart parsing algorithm. This allows us to finitely represent possibly infinite derivations.

At a next stage, the derivations are mapped to a shared forest. Admissible parse structures of a second-order ACGs are indeed trees, and solutions to a parsing problem are therefore represented by forests that are shared in order to keep a finite representation of the possibly infinite set of solutions. Enumerating the solutions corresponds to traversing the forest. However, if we want the enumeration to be sorted according to a metric associated to the trees, we need to be able to easily move from the computation of a solution to the computation of another one for which the metric is better. To this end, we implemented a (purely applicative) forest context data structure that behaves with respect to the forest data structures similarly to how zippers [11] behave with respect to trees.

A forest is a list of forest trees, the latter being trees whose children are forests themselves. Consequently, the context of a forest tree needs to keep track: (i) of its context in the forest, i.e., the list of alternatives it belongs to, (ii) of the

context of the forest it belongs to as a child in the children list, (iii) of its upper context in the parent-children relation. Such a structure is enough to suspend and resume walks in the forest on demand. Sharing is an orthogonal feature and is currently implemented using relative path in the forest (but other solutions, for instance with association tables, or dictionaries, could be used).

The metric we currently use involves the size and the depth of the abstract terms, i.e., the number of nodes and the depth of the parse trees. The whole framework is, however, ready to have metrics based on weighted grammars inferred from corpora.

5.2 Magic Set Rewriting

We also introduced the ability to use magic set rewriting techniques [29,1] as an experimental feature for `acgc` and `acg`. Magic set techniques are logical rewriting techniques used to optimize query evaluation in deductive databases. These methods simulate the selection pushing that is characteristic of top-down query processing algorithms within a bottom-up evaluation framework. Programs are transformed into larger ones that include additional intensional database predicates. These extra predicates make the evaluation focus on those parts of the database that are relevant to the query, thus avoiding unnecessary computations.

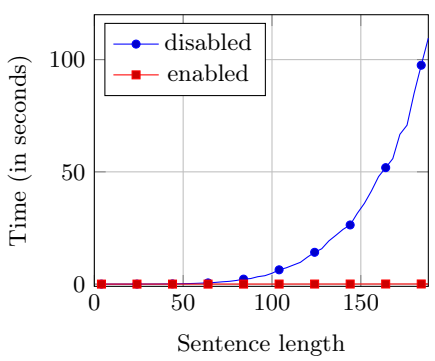
In the context of ACGs, the evaluation of rewritten programs leads to an Earley style parsing algorithm. For small grammars, it proves to be very efficient on large sentences as Figure 4 shows for parsing strings of the mildly context-sensitive language $\{a^n b^n c^n d^n\}$ with n ranging from 1 to 94.[6]

However, Datalog programs (intensional databases) resulting from ACG transformation are usually large compared to the extensional databases corresponding to the terms to parse, a somewhat different situation of what happens for usual databases. Magic set rewriting strengthens this contrast because it makes the number of rules increase (Fig. 4c shows the effect on parsing time of small sentences). We did not yet run extensive experiments with large ACG grammars, first because of their unavailability, and second because of optimization issues (in particular related to size and space) in the current implementation of the rewriting algorithm (not the parsing algorithm).

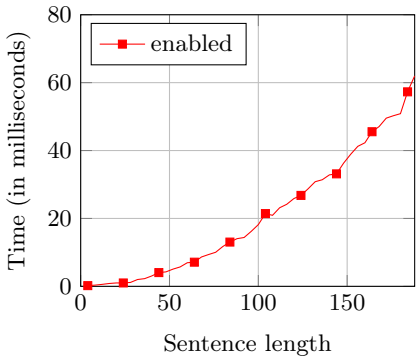
6 Related Work

Available symbolic parsers (possibly augmented with stochastic models for pruning search space) for natural language processing are usually dedicated to a specific grammar formalism, often tweaked to fit a specific extension of that formalism. See for instance Partage for tree-adjoining grammars, TuLiPA for tree-adjoining grammars and mildly context-sensitive variants such as multi-component tree-adjoining grammars with tree tuples, OpenCCG or NLTK for

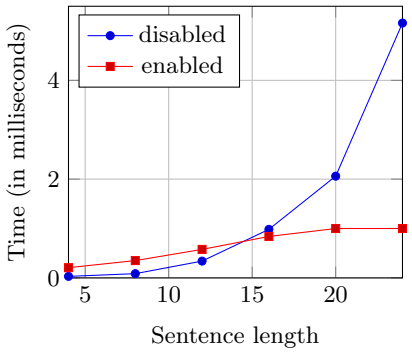
[6] Each parsing time is an average of 100 time measures for each length using 1 core on an Intel® Core™ i7-3520M CPU, 2.90 GHz, 16 GB RAM laptop.



(a) Comparing the parsing time with and without magic set rewriting enabled



(b) Focus on parsing time with magic set rewriting enabled



(c) Comparing the parsing time with and without magic set rewriting enabled for short sentences

Fig. 4: Parsing time of sentences $a^n b^n c^n d^n$ (length $= 4n$)

combinatory categorial grammars, the Babel-System, DELPH-IN tools, or Enju for HPSG. Moreover, getting semantic representations also usually requires using additional tools to process the parsing output, or targets a specific semantic representation language.

From a theoretical perspective, the flexibility provided by the ACG framework and its architecture (often coined as *synchronous* or *parallel* grammars) is shared by other formalisms. For instance, Interpreted Regular Tree Grammars [21] that also consider parsing by morphism inversion and is implemented in Alto. However, this formalism is not well suited to deal with semantics represented with logical formulas. To parse a term t indeed requires that the set of trees that are interpreted as t is regular. To parse a logical representation based on λ-calculus would mean to represent all terms that are β-equivalent to the term we want to parse by a regular tree grammar. It is not clear how it can be done.

Another related approach is Grammatical Framework [26] and its implementation GF. The type system of GF is based on dependent type theory and is not as flexible as ACG for composing grammars. However, GF is also considered as a programming language for grammar applications, and a lot of solutions have been developed to address grammatical (instead of software) engineering issues. It has been extensively used to develop grammars for many languages (38 languages so far) and is an important source of inspiration for features to add to ACGtk.

Finally, Applicative Abstract Categorial Grammars [19] and its conceptual simplification, Transformational Semantics [20] also come with an implementation in form of a domain-specific language embedded in Haskell.[7] It focuses more specifically on structure transformations that are needed to derive difficult semantic phenomena.

7 Conclusion

ACGtk provides an environment to develop, test, and use ACGs. It is used to implement syntax-semantics interface models for natural languages, possibly using complex composition architectures. It is implemented in a functional language, using interesting data structures such as zippers, and also implements a Datalog prover and optimization related techniques.

Directions for further developments are threefold:

- To integrate recent theoretical work on ACGs (mainly feature structures and weighted grammars).
- To improve the grammatical engineering facilities (for instance introducing functors).
- To use different optimization techniques, either related to Datalog evaluation (e.g., with precedence graph) or to the OCaml language (parallel programming).

[7] https://okmij.org/ftp/gengo/transformational-semantics/

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Assison-Wesley (1995), http://webdam.inria.fr/Alice/pdfs/all.pdf
2. Barker, C.: Continuations and the nature of quantification. Natural Language Semantics **10**(3), 211–242 (2002). https://doi.org/10.1023/A:1022183511876, https://cb125.github.io/Papers/barker-continuations.pdf
3. Curry, H.B.: Some logical aspects of grammatical structure. In: Jakobson, R. (ed.) Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics. pp. 56–68. American Mathematical Society (1961). https://doi.org/10.1090/psapm/012
4. de Groote, P.: Linear higher-order matching is NP-complete. In: Bachmair, L. (ed.) Rewriting Techniques and Applications. Lecture Notes in Computer Science, vol. 1833, pp. 127–140. Springer (2000). https://doi.org/10.1007/10721975_9
5. de Groote, P.: Towards Abstract Categorial Grammars. In: Proceedings of 39th Annual Meeting of the Association for Computational Linguistics. pp. 148–155 (7 2001). https://doi.org/10.3115/1073012.1073045, ACL anthology: P01-1033
6. de Groote, P.: Tree-Adjoining Grammars as Abstract Categorial Grammars. In: Proceedings of the Sixth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6). pp. 145–150. Università di Venezia (2002), ACL anthology: W02-2220
7. de Groote, P.: Proof-search in implicative linear logic as a matching problem. In: Parigot, M., Voronkov, A. (eds.) Logic for Programming and Automated Reasoning. Lecture Notes in Artificial Intelligence, vol. 1955, pp. 257–274. Springer (2000). https://doi.org/10.1007/3-540-44404-1_17
8. de Groote, P.: Abstract categorial parsing as linear logic programming. In: Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015). pp. 15–25. Association for Computational Linguistics, Chicago, United States (2015), HAL open archive: hal-01188632. ACL anthology: W15-2302
9. de Groote, P., Guillaume, B., Salvati, S.: Vector Addition Tree Automata. In: Proceedings of the Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2004). pp. 64–73. Turku, Finland, France (2004), https://inria.hal.science/inria-00100081, colloque avec actes et comité de lecture. internationale.
10. de Groote, P., Pogodalla, S.: On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms. Journal of Logic, Language and Information **13**(4), 421–438 (2004). https://doi.org/10.1007/s10849-004-2114-x, HAL open archive: inria-00112956
11. Huet, G.: The zipper. Journal of Functional Programming **7**(5), 549–554 (1997). https://doi.org/10.1017/S0956796897002864
12. Joshi, A.K., Levy, L.S., Takahashi, M.: Tree adjunct grammars. Journal of Computer and System Sciences **10**(1), 136–163 (1975). https://doi.org/10.1016/S0022-0000(75)80019-5
13. Joshi, A.K., Schabes, Y.: Tree-adjoining grammars. In: Rozenberg, G., Salomaa, A.K. (eds.) Handbook of formal languages, vol. 3, chap. 2. Springer (1997). https://doi.org/10.1007/978-3-642-59126-6_2
14. Kanazawa, M.: Parsing and generation as datalog queries. In: Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007). pp. 176–183. Association for Computational Linguistics, Prague, Czech Republic (6 2007), ACL anthology: P07-1023

15. Kanazawa, M.: A prefix-correct earley recognizer for multiple context-free grammars. In: Gardent, C., Sarkar, A. (eds.) Proceedings of the Ninth International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+9). pp. 49–56. University of Tübingen, Tübingen, Germany (6 2008), ACL anthology: W08-2307
16. Kanazawa, M.: Second-order abstract categorial grammars as hyperedge replacement grammars. Journal of Logic, Language and Information **19**(2), 137–161 (2009). https://doi.org/10.1007/s10849-009-9109-6
17. Kanazawa, M.: Parsing and generation as datalog query evaluation. IfCoLog Journal of Logics and their Applications **4**(4), 1103–1211 (2017), http://www.collegepublications.co.uk/downloads/ifcolog00013.pdf#page=307, special Issue Dedicated to the Memory of Grigori Mints
18. Kanazawa, M., Salvati, S.: Generating control languages with abstract categorial grammars. In: Penn, G. (ed.) Proceedings of the 12th conference on Formal Grammar (FG 2007). CSLI Publications (2007), https://makotokanazawa.ws.hosei.ac.jp/publications/control.pdf
19. Kiselyov, O.: Applicative abstract categorial grammar. In: Kanazawa, M., Moss, L.S., de Paiva, V. (eds.) NLCS'15. Third Workshop on Natural Language and Computer Science. EPiC Series in Computing, vol. 32, pp. 29–38. EasyChair (2015). https://doi.org/10.29007/s2m4
20. Kiselyov, O.: Applicative abstract categorial grammars in full swing. In: Otake, M., Kurahashi, S., Ota, Y., Satoh, K., Bekki, D. (eds.) New Frontiers in Artificial Intelligence. pp. 66–78. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-50953-2_6, https://okmij.org/ftp/gengo/applicative-symantics/AACG1.pdf
21. Koller, A., Kuhlmann, M.: A generalized view on parsing and translation. In: Proceedings of the 12th International Conference on Parsing Technologies. pp. 2–13. Association for Computational Linguistics, Dublin, Ireland (10 2011), ACL anthology: W11-2902
22. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly **65**(3), 154–170 (1958). https://doi.org/10.2307/2310058
23. Maskharashvili, A.: Discourse Modeling with Abstract Categorial Grammars. Ph.D. thesis, Université de Lorraine (12 2016), HAL open archive: tel-01412765
24. Montague, R.: The proper treatment of quantification in ordinary english. In: Hintikka, J., Moravcsik, J., Suppes, P. (eds.) Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics, pp. 221–242. Springer Netherlands, Dordrecht (1973). https://doi.org/10.1007/978-94-010-2506-5_10
25. Pogodalla, S.: A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars. Journal of Language Modelling **5**(3), 527–605 (2017). https://doi.org/10.15398/jlm.v5i3.193, HAL open archive: hal-01242154
26. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Studies in Computational Linguistics, CSLI Publications (2011)
27. Salvati, S.: Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites. Ph.D. thesis, Institut National Polytechnique de Lorraine (2005)
28. Salvati, S.: Encoding second order string ACG with Deterministic Tree Walking Transducers. In: Wintner, S. (ed.) Proceedings of The 11th conference on Formal Grammar (FG 2006). pp. 143–156. FG Online Proceedings, CSLI Publications, Malaga, Spain (2006), http://cslipublications.stanford.edu/FG/2006/salvati.pdf

29. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W. H. Freeman & Co., New York, NY, USA (1990)
30. Vijay-Shanker, K., Weir, D.J., Joshi, A.K.: Characterizing structural descriptions produced by various grammatical formalisms. In: Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL 1987). pp. 104–111. Stanford, CA (1987). https://doi.org/10.3115/981175.981190, ACL anthology: P87-1015
31. Weir, D.J.: Characterizing Mildly Context-Sensitive Grammar Formalisms. Ph.D. thesis, University of Pennsylvania (1988), http://users.sussex.ac.uk/~davidw/resources/papers/dissertation.pdf
32. Yoshinaka, R., Kanazawa, M.: The complexity and generative capacity of lexicalized abstract categorial grammars. In: Blache, P., Stabler, E., Busquets, J., Moot, R. (eds.) Logical Aspects of Computational Linguistics: 5th International Conference, LACL 2005, Bordeaux, France, April 28-30, 2005. Proceedings. LNCS/LNAI, vol. 3492, pp. 330–348. Springer (2005). https://doi.org/10.1007/11422532_22

A Typing Rules

Let $\Sigma = \langle A, C, \tau \rangle$ be a higher-order signature. The typing rules for terms of $\Lambda(\Sigma)$ use sequents of the form $\Gamma; \Delta \vdash_\Sigma t : \alpha$ where Γ is a *non-linear* context assigning types to variables and Δ is a *linear* context assigning types to variables. Whenever Δ is empty, e.g., in the (const.), the (var.) and the (app.) rules, such a sequent is written $\Gamma; \vdash_\Sigma t : \alpha$.

$$\frac{c \in C}{\Gamma; \vdash_\Sigma c : \tau(c)}\,(\text{const.})$$

$$\frac{}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha}\,(\text{lin. var.}) \qquad \frac{}{\Gamma, x : \alpha; \vdash_\Sigma x : \alpha}\,(\text{var.})$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^\circ x.\, t : \alpha \to \beta}\,(\text{lin. abs.})$$

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \to \beta \quad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma t\, u : \beta}\,(\text{lin. app.}) \text{ if } \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda x.\, t : \alpha \Rightarrow \beta}\,(\text{abs.}) \qquad \frac{\Gamma; \Delta \vdash_\Sigma t : \alpha \Rightarrow \beta \quad \Gamma; \vdash_\Sigma u : \alpha}{\Gamma; \Delta \vdash_\Sigma t\, u : \beta}\,(\text{app.})$$