Application Development with the FNC-2 Attribute Grammar System

Martin JOURDAN   Didier PARIGOT

INRIA*

Abstract

FNC-2 is an advanced attribute grammar system aiming at production-quality, currently under development at INRIA. After a brief tour through its internals and a short presentation of its input language OLGA, the talk will concentrate on how FNC-2 and its companions can be used to develop large language-processing applications. The key feature for enhancing programmers' productivity and supporting teamwork is FNC-2 constructs for modularity. This will be exemplified by the development of FNC-2 itself. Finally we'll present how FNC-2 can be combined with other tools under development at INRIA to form a complete, high-quality compiler production workbench.

1 Introduction

Since attribute grammars (AGs) have been introduced in Knuth's seminal paper [Knu68], they have become a method of choice for describing and implementing syntax-directed computations, in particular language-processing applications such as compilers, translators and syntax-directed editors. Many people have made much research work on various aspects of AGs, and many systems have been developed around the world which produce attributes evaluators [DJL88]. FNC-2 is one of the most advanced such systems, under development at INRIA since 1986. It aims at production-quality by providing efficiency, expressive power, ease of use and versatility. To achieve these qualities, it uses the latest results in AG implementation [JPJ90], which it combines with OLGA, a new language especially designed for the specification of AGs [JLP90].

This paper concentrates on the use of FNC-2 and OLGA for building realistic applications. After a brief presentation of the system and the language, including in particular a description of what an application is for them, we'll report on the experience we gained in using them, and especially in building FNC-2, since it is bootstrapped. We'll also show how FNC-2 can be combined with other tools being developed at INRIA to generate complete compilers from high-level specifications. We'll conclude with ideas for future work.

*Author's address: INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex (France). E-mail: `{jourdan,parigot}@minos.inria.fr`

Note that, for lack of space, this paper does not aim at a complete description of FNC-2 and OLGA; please refer to [JoP89, JPJ90, JLP90] for more details. For the same reason, no comparison with other work will be included.

2 A Tour through FNC-2 and OLGA

2.1 General overview

FNC-2 is a modern AG-processing system that aims at production-quality by providing the following qualities:

Efficiency: the generated evaluators, based on the visit-sequence paradigm [Kas80], are completely deterministic; furthermore, this paradigm allows to apply very effective space optimization techniques [Kas84, EnJ90, JuP90]. The evaluators are hence basically as efficient in time and space as hand-written programs using a tree as internal data structure.

Expressive power: this efficiency is not achieved at the expense of expressive power since FNC-2 accepts AGs in the very broad class of strongly non-circular AGs [CoF82].[1]

Easiness of use: OLGA, the input language of FNC-2, enforces a high degree of programming safety, reliability and productivity.

Versatility: the generated evaluators can be interfaced with many other tools and produced in many variations (e.g. exhaustive or incremental, sequential or parallel, in one of several implementation languages) from the same specification.

How we achieve both efficiency and expressive power has to do with the evaluation methods; ours are described in detail elsewhere [Par88, JPJ90, JuP90]. However, as far as programming and developing applications are concerned, this does not matter much, provided of course that the system does a reasonable job in implementing the specification. Much more important is the way the programmer interacts with the system, i.e. the input language and the general development paradigm. This is what this paper will concentrate on.

2.2 What an application is for FNC-2

In our opinion, one of the main reasons why AGs are only scarcely used in industrial settings is their lack of modularity, which forces to program a complete application as a single, monolithic file. As applications get larger and more complicated, this scheme becomes more and more unworkable; a striking example is the ADA front-end developed using the GAG system [UDP82]: it is a single AG of more than 500 *pages*! We clearly had to address this issue if we wanted FNC-2 to have a chance to become a production-quality system. Also, there are parts of some applications for which tools other than AGs might be more pleasant to use (e.g. attributed tree transformation systems are better than AGs

[1]These AGs are also called absolutely non-circular.

to describe the optimization phases in a compiler), so we had to provide for interfacing our attribute evaluators with these other tools.

The paradigm we chose is hence to consider that an AG specifies, and an attribute evaluator implements, an attributed-tree to attributed-tree mapping, i.e. an evaluator takes as input an attributed tree and produces as output another attributed tree (or zero or more than one, actually). This paradigm was called *attribute coupled grammars* (ACGs) by its inventors [GaG84]. It responds to the above-quoted problems as follows:

1. A large application can be split into a sequence of passes, each pass taking as input the intermediate representation produced by the previous one and transforming it into another intermediate representation to be fed to the next pass. Each pass can be described by an AG, of course, and each of them is thus much smaller and hence much more easily manageable than a single AG specifying the same translation as the whole sequence of passes.

2. In this sequence of passes, some can be described by AGs while others can be described and implemented by other tools. The tree-to-tree mapping paradigm is indeed quite general and many tools are based on it.

There are two kinds of AGs in OLGA, side-effect and functional, respectively roughly equivalent to classical AGs and ACGs. A side-effect AG is one in which there is exactly one output tree which is exactly the same as the input one except that it carries different attributes. A functional AG has zero, one or more output trees, generally different from the input one, which must be constructed piece by piece by semantic rules and carried by (synthesized) attributes. It must be noted here that a naive implementation of functional AGs involves the physical construction of the output tree(s), and hence suffers from some overhead, but actually, thanks to *descriptional composition* [GaG84], this needs not be the case: it is possible to mechanically construct, from two (functional) AGs coupled into a "pipe," a single AG which performs the same translation as the original sequence but without physically constructing the intermediate tree. We already know that descriptional composition is feasible with FNC-2 because the SNC class is closed under composition, however we have not yet implemented it.

Let us open a parenthesis here to note that, for FNC-2, trees are *abstract trees* in the well-known sense [ASU86], i.e. the grammars describing these trees express the structure, rather than the textual appearance, of the notions and constructs of the languages they model. This frees the grammar writer from cumbersome syntactic constraints which stem from a particular parsing method and, on the other hand, allows smaller trees since they do not include keywords, simple productions and such spurious things. In addition, our experience is that this allows to make the trees closer to the true semantics of a language, which makes the AGs shorter and simpler. (End of the parenthesis.)

In this scheme, the intermediate trees are described by grammars extended with attribute declarations (to avoid confusion we call such things *attributed abstract syntaxes* (AAS)). Each such AAS potentially "belongs" to more than one AG: it is the input AAS of one or more AGs and the output AAS of one or more other AGs. To avoid duplication of work and possible inconsistencies, and to facilitate modularity and reusability, we have decided to separate the specification of these AASes from the AGs themselves.

It should be clear by now that the most important components of (the specification of) an application, at least as far as FNC-2 is concerned, are attribute grammars, which specify the computations performed by the various passes, and attributed abstract syntaxes, which specify the input and output data (attributed abstract trees) of these passes.

In addition, FNC-2 comes with a number of companion processors that help build complete applications:

- a generator of abstract tree constructors driven by parsers (*atc*); two instantiations of *atc* have been implemented, one on top of SYNTAX,[2] which will be described with more details in section 4, and one on top of *Lex* and *Yacc*;
- a generator of unparsers of attributed abstract trees (*ppat*), based on the TEX-like notion of nested boxes of text;
- and a tool for describing the modules composing an application and managing their processing (*mkfnc2*).

Note that the input languages of all these tools have much in common with OLGA and *asx*, the language for describing AASes.

Fig. 1, which depicts the organization of a typical application as can be constructed by FNC-2 and its companions alone, should now be easily understandable. More details will be given in the next section, which briefly describes OLGA. Let us add only a few words: the application is the front-end of a compiler. The first pass checks that the contextual constraints of the source language are verified; this is a side-effect AG, since no translation is involved here, only information is computed and added to the tree. The second pass translates the source tree to some intermediate representation; this is a functional AG. The final unparsing pass would be used for debugging purposes and needs not be present if the IR is to be further processed by a complete back-end. However there exist applications, e.g. source-to-source translation, in which the output is some high-level language; in that case *ppat* can generate a suitable "back-end" (see Fig. 4).

2.3 The OLGA AG-description language

OLGA was designed for the description of all aspects of an attribute grammar. This of course involves constructs to declare attributes, access them in semantic rules, etc., but also constructs to describe pure calculations without resorting to a foreign language. Hence, in addition to being a specialized language for describing AGs, OLGA is also a general-purpose applicative language. We'll briefly present these two aspects now; more details can be found in [JLP90, JoP89].

2.3.1 OLGA as a general-purpose applicative language

First of all, OLGA is a *purely applicative* language, which means that there is no assignable variable and side effects, just pure expressions and functions. This is a strong step towards programming safety and reliability. However, OLGA is not (yet) a *functional* language in the sense of ML. The basic objects of OLGA are thus values and functions. There are no control-flow constructs but value-selection ones; iteration is replaced by recursion.

[2]SYNTAX is a trademark of INRIA

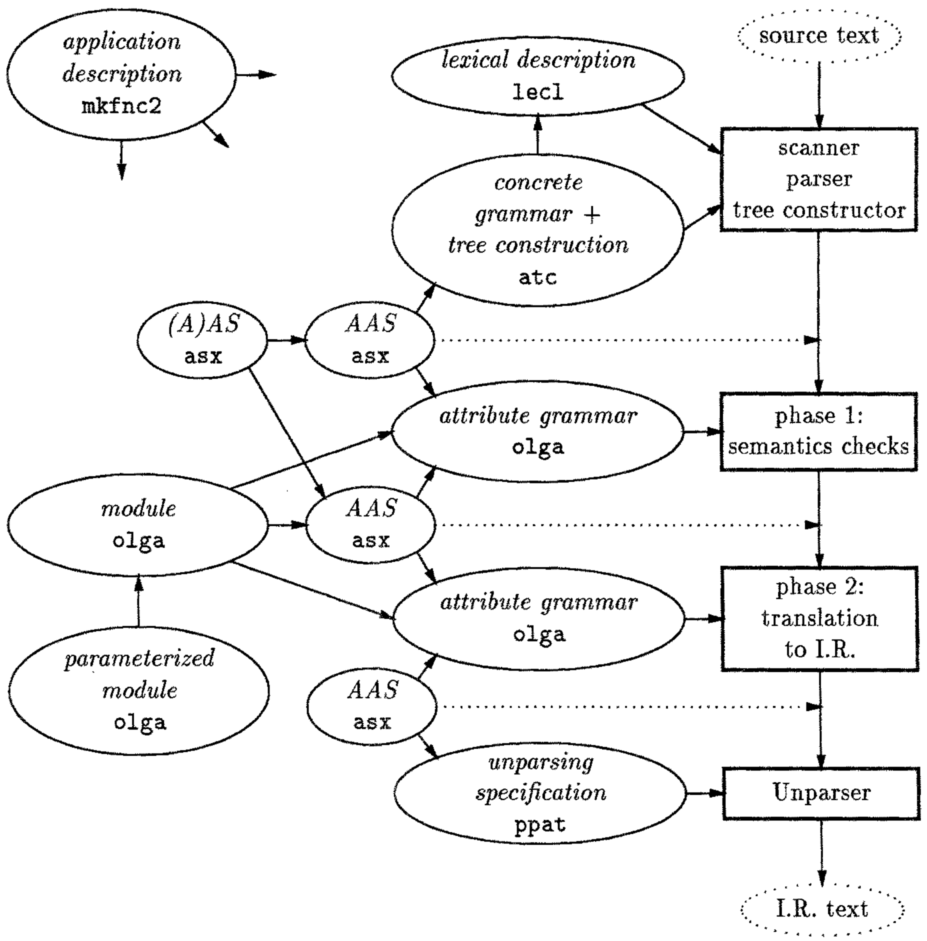
application description mkfnc2
lexical description lecl
source text
scanner parser tree constructor
concrete grammar + tree construction atc
(A)AS asx
AAS asx
attribute grammar olga
phase 1: semantics checks
module olga
AAS asx
attribute grammar olga
phase 2: translation to I.R.
parameterized module olga
AAS asx
unparsing specification ppat
Unparser
I.R. text

Figure 1: Structure of a typical application

OLGA is a *strongly typed* language, the other necessary condition for programming safety. The collection of predefined types is rather classical. Type constructors are enumerations, subranges of scalar types, records, discriminated unions, sets of scalar values and homogeneous lists. Structured types may be recursive. Each type definition involves the definition of corresponding construction or conversion functions. In addition, in a "functional" AG there are tree types and tree values (see below). Functions, including user-defined ones, may have a *polymorphic* profile as in ML, although OLGA polymorphism is restricted to set and list types. We also have a *type inference* algorithm.

OLGA is of course *block-structured*. The basic scope rules are similar to those of Ada. As in Ada also, function names can be *overloaded*; we found this feature very convenient to enhance the readability of OLGA programs.

OLGA supports the notion of *modules*, in which one can define a set of related objects (types, functions, constants, values, i.e. run-time constants, and exceptions). OLGA modules are similar to those found in e.g. Ada and Modula-2. A module is split in two actual compilation units, a *declaration module* (DCM) which declares the objects which are visible from outside, and a *definition module* (DFM) in which the actual implementation of these objects (and maybe other, non-visible objects) is given. Type checking is performed across module boundaries. Types and other objects in a declaration module may be specified as *opaque*. In addition, a module may be *parameterized* by types and/or functions. This supports the notion of abstract data types. Two such modules are depicted in Fig. 1.

OLGA provides a powerful *pattern matching* construct unifying structural matching, selection driven by a scalar value and test for a union tag. OLGA also provides for the declaration, activation and handling of *exceptions*. User-defined exceptions may return parameters. There also exist some constructs which somewhat violate the applicative character of OLGA but which we felt we had to include anyhow because they are so useful: production of *error messages*, interface with *external functions* written in some foreign language, and construction of *circular structures*.

2.3.2 OLGA as an attribute grammar description language

As said above the syntactic base of AGs written in OLGA is not a concrete syntax as in the classical framework [Knu68] but rather an *abstract syntax* as in [VoM82]. The abstract syntax formalism used for OLGA is very close to Metal [KLM83]. It provides for heterogeneous, fixed arity *operators*, similar to concrete productions, and homogeneous, variable-arity operators (*list nodes*). Operators are grouped into *phyla*, which roughly correspond to non-terminals. In a functional AG the phyla of the output AAS(es) are tree types whose constructors are the operators.

In addition to what we have already said regarding modularity with FNC-2 and the ACG concept, let us note that an AAS can be imported in another one. This is useful to describe the "profile" of a side-effect AG: since the (attribute-less) syntax is the same for both the input and output AAS, it needs be specified only once; this "base" syntax is then imported in the actual input and output AASes, which merely specify what attributes are attached to this syntax (see Fig. 1). The types of the attributes of an AAS must be defined in a separate DCM.

An AG is structured into *phases* and, of course, productions. A phase is purely a structuring construct which has no effect on the AG in the classical sense (i.e. it is "transparent" to productions and semantic rules), except that it is a block and hence may contain local declarations and import clauses. For instance, an AG describing the verification of the contextual constraints for some programming language might have a phase for name analysis and one for type analysis; each phase needs not know about the functions used in the other, and they communicate only through the attributes.

Each production is also a block; values local to a production may depend on attributes of this production and hence play the role of what is usually referred to as "local attributes." Thus, the condition that an AG be in normal form is not too constraining. A production may appear several times in each AG or phase when this improves the readability but, of course, a given attribute should be defined only once.

In each AG there are three kinds of attributes. The *input* and *output* attributes are those carried by the input and output AATs and are declared in the corresponding

```
where decls -> DECL* use
   $in-env(DECL) := case position is
            first: $in-env(decls);
            other: $out-env(DECL.left);
         end case;
   $out-env := case arity is
            0: $in-env;
            other: $out-env(DECL.last);
         end case;
   $correct := map left & value true
            other $correct(DECL)
         end map;
end where;
```

Figure 2: Examples of semantic rules for a list production

AASes. Input attributes are constants. In side-effect AGs, output attributes must be given a *direction* (inherited or synthesized) and be defined by semantic rules, whereas in a functional AG they have no direction and must be attached to the corresponding AAT while it is being built. *Working* attributes are "local" to the AG and correspond to classical ones; for instance they carry the output AATs during their construction. They must have a direction.

In OLGA, a semantic rule is written, as expected, as the assignment of some value to some attribute occurrence. The right-hand side expression can refer, in addition to attributes of the production at hand, to attributes of nodes upward in the tree and attributes of subtrees (after suitable pattern matching to control the access). OLGA provides several special constructs for semantic rules in list productions. Fig. 2 presents a (hopefully self-explanatory) example of these constructs; please refer to [JLP90, JoP89] for more details.

OLGA allows to leave the definition of some attribute occurrences unspecified in an AG, and FNC-2 will try to infer the corresponding semantic rules from the context. Automatically generating *copy rules* is a rather well-known technique now [Lor77, Jou84], and we have implemented it in FNC-2; in Fig. 2 for instance, with suitable declarations for `$in-env` and `$out-env`, the first two rules would have been automatically generated. However, there still remain a lot of semantic rules which "look the same" and that you have to write in whole. Thus, we have extended FNC-2's mechanism to provide for automatic generation of non-copy rules. The basic mechanism is the definition of *attribute classes*, which are sets of attribute occurrences, and associated templates to specify the semantic rules which define these occurrences. The templates are actually structured in two levels, a syntactic level used to specify in which productions the templates will be applied, and a semantic level which correspond to the actual rules. When some attribute occurrence is not explicitly defined in a given production, FNC-2 searches whether this attribute belongs to some class; if so, it tries to match some syntactic template in the class definition with the production at hand and the corresponding semantic template with some context conditions that space does not permit to explain in detail here; if all these conditions are verified, the actual rule is created.

In addition to the shortening of the text of an AG, the other benefit of this feature—

the most important in our opinion—is that this gives a more "semantic" view of the AG, because closely related semantic computations are grouped in a single place (the definition of a class) rather than being spread over several syntactic productions. The phase construct supplements this semantic structuring.

3 Application Development

Until now we have worked on two "pure" real-size applications:

- a compiler from the parallel logic language PARLOG to code for the SPM abstract machine [GJR87];
- a compiler from ISO-Pascal to P-code, which had been originally written in Lisp for the FNC/ERN system [Jou84] and was translated to OLGA (work not yet completed).

In addition, FNC-2 is used in the PAGODE code generator generator [DMR89, DMR90], both to develop the generator itself and to compile some of the modules it generates (see also section 4).

However, the largest application of our system, and the one with which we have the deepest experience, is the development of FNC-2 itself, through bootstrap, and of its companions. This explains why we have decided to use it as the "test case" for this discussion.

The most important component in FNC-2 is the OLGA compiler, whose structure is depicted in Fig. 3. The OLGA reader groups the scanner, the parser and the constructor for the first tree; it is generated by *atc* and SYNTAX. The checker is the largest component in FNC-2; it performs the verification of the contextual constraints, generates the missing semantic rules and splits the AG into an "abstract AG" to be input to the evaluator generator, containing only the syntax (list operators are expanded into a finite set of fixed-arity productions) and attribute dependencies, and the rest of the AG (types, functions and expressions in the right-hand side of the semantic rules) which is directly fed to the translator(s). It is written as a sequence of two AGs, a big one doing nearly all the job while the second one only optimizes the function bodies and semantic rules (tail recursion elimination, construction of a deterministic decision tree for the pattern matching construct, etc.). The big AG makes heavy use of out-of-line functions in separate modules which are also written in OLGA, including the generation of the missing semantic rules and the creation of the abstract AG. The evaluator generator is the other most important component in the system; since the basic data structures it manipulates are graphs, and since most of the algorithms it uses involve fixed-point computations—two things which are cumbersome or even impossible to specify with AGs—, it is written in C. The last components of FNC-2 are the translators to the various implementation languages; presently there exist two of them, one to C and one to Lisp. They are also written in OLGA (and soon in *ppat*). The library manager does not exist yet; its job is performed out of line by *mkfnc2*.

The organization of FNC-2 into three parts (front-end, evaluator generator and back-end(s)) with well-defined interfaces allows to use and test these parts separately. For instance, the evaluator generator was running long before the rest of the system; we

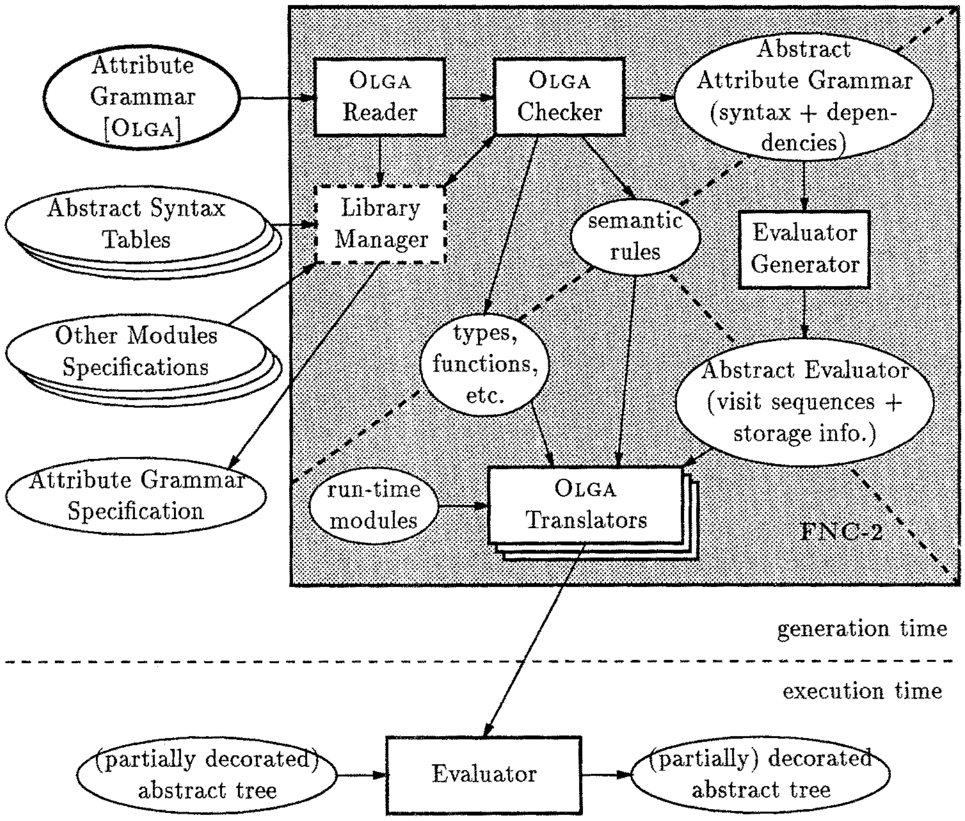
Attribute Grammar [OLGA]
OLGA Reader
OLGA Checker
Abstract Attribute Grammar (syntax + dependencies)
Abstract Syntax Tables
Library Manager
semantic rules
Evaluator Generator
Other Modules Specifications
types, functions, etc.
Abstract Evaluator (visit sequences + storage info.)
Attribute Grammar Specification
run-time modules
OLGA Translators
FNC-2
generation time
execution time
(partially decorated) abstract tree
Evaluator
(partially) decorated abstract tree

Figure 3: The OLGA compiler of the FNC-2 system

constructed a crude interface (textual version of our abstract AGs) which allowed to debug it and can still be used to operate it when you don't want to use OLGA; the internal representations produced by the crude interface and by the OLGA front-end have the same structure. In addition, the separation of concerns makes each part simpler; for instance, the "high-level optimization" AG quoted above acts only on the non-AG part of the OLGA language, and this is reflected in the input AAS. Lastly, the ACG paradigm makes it easy to add new features; for instance the optimization pass was not included in the first design but was very easy to insert later without major reworking of the other components.

Even in the presently incomplete status of OLGA—the most important missing features are full polymorphism, parameterized modules and exceptions—, we found it generally satisfying. In particular, there are very few parts of the system which could not be written in OLGA; the most prominent are the evaluator generator and the parts which read from or write to files (*ppat* will provide a solution for the latter). This shows that the expressive power of the language is sufficient for our purposes. The automatic generation of most copy rules and many non-copy rules greatly improves the readability of the AGs, especially for code-generation-like applications.

	#	# lines			
	files	min.	max.	total	ave.
AGs	7	354	3,212	10,118	1,445
AASes	8	8	381	779	97
DCMs	15	28	391	2,891	193
DFMs	15	55	3,188	13,404	894
atc	4	60	2,089	2,575	644
total	49	8	3,212	29,767	607

Table 1: Source files in the FNC-2 system

We would like to point out here that AGs are, to the best of our knowledge, the only programming method which really supports incremental development: you may freely add new attributes and semantic rules, and then test your AG without having to completely specify it. This is made easier by the great expressive power of FNC-2, i.e. the SNC class: many AGs we have written were, at some time during their development, not ordered and not even *l*-ordered (and our bigger AG still isn't). Of course, with more or less work, it would be possible to make them actually ordered, but using FNC-2 gives us more freedom in our development. Also, a great expressive power is quite useful when AGs are automatically produced by other systems, such as PAGODE: designers of these systems can ignore "mundane" issues such as evaluation order.

But the single most beneficial feature of FNC-2 and OLGA is their support for modularity: presently there are about 30,000 lines of OLGA, *asx* and *atc* code in the system, summarized in Table 1 (these exclude the *ppat* subsystem, which is still under development, the files processed by SYNTAX, and the C files). As can be seen, the files are rather small and hence very readable. Even the 3,212-lines AG (the OLGA static semantics checker) is easy to understand because it is split in several phases (name analysis, type checking, check for well-definedness of the AG and generation of missing semantic rules, generation of the abstract AG) which address only one problem. We would have had much greater problems if we had had to specify the FNC-2 system in a single AG. Furthermore, separate compilation comes with (true) modularity, and this saves much time in the development process. As said above, another benefit is that it is easy to add new components in the system. Yet other benefits are that this makes teamwork ("programming in the large") easier, and that it is easy to reuse modules in different applications; for instance, the two instantiations of *atc* share most of their stuff.

As we gained experience, we learnt to push the usage of AGs and modularity very far (the OLGA compiler itself is not the best example in this respect). For instance, Fig. 4 depicts the organization of the *ppat* subsystem and the pretty-printers it generates, with all the files involved in the process. Files X`.asx` and X`.ppat` are written by the user; they respectively specify the input attributed abstract trees and their textual representation. File X`-ppat.olga` is generated from those by *ppat* and then combined with files `boxes*` to form the actual unparser. Apart from these, all the files need be written only once. Each one of them is rather small and easy to read. Furthermore, it is easy to reuse them for other purposes (e.g. `olga.asx` is the AAS for OLGA, which is borrowed from FNC-2). This figure also illustrates our bootstrap philosophy (file `olga.ppat`).

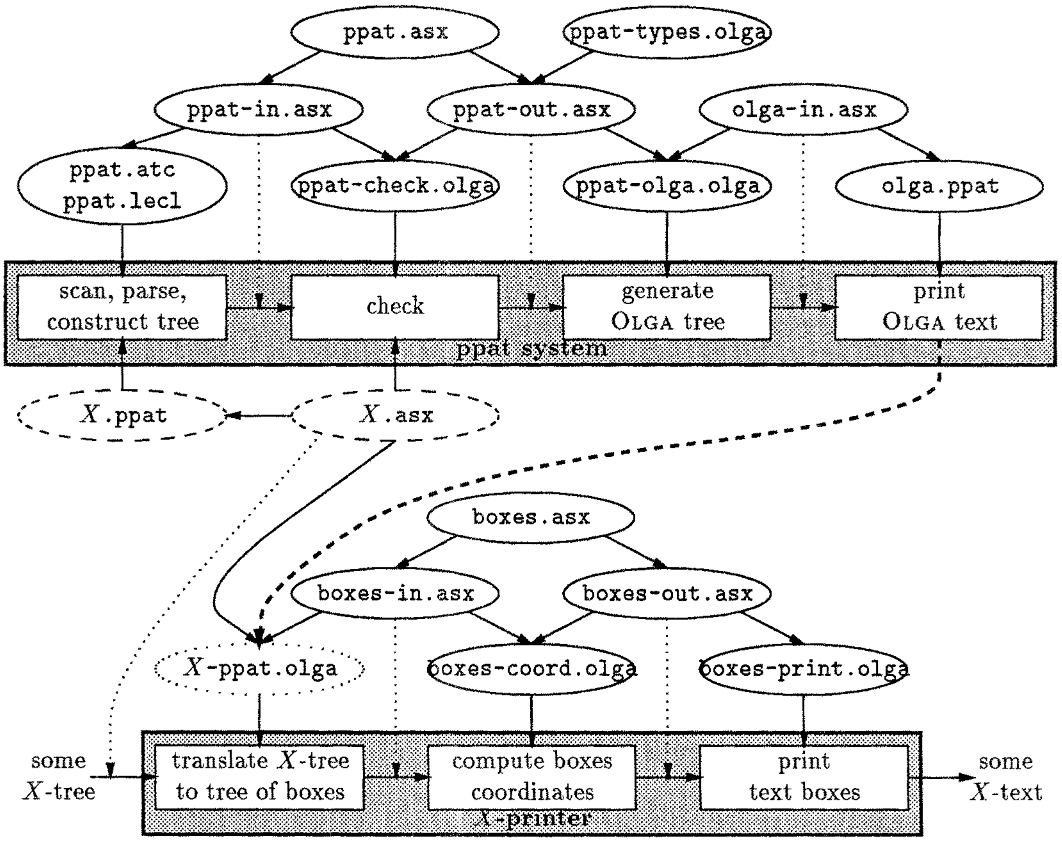

Figure 4: Generation and use of the *ppat* subsystem

It is quite hard to measure the efficacy of a programming language, i.e. the relative productivity of a programmer using this language. Let us however quote a single figure regarding OLGA: the above-described 30,000 lines, together with other support files, were written and tested by one man in not much more than one year (the development team grew quite a bit recently, so our experience with teamwork is real).

4 Towards a Compiler Construction Workbench

In addition to FNC-2, several other language-processing tools are under development at INRIA.

SYNTAX [BoD88] is a powerful generator of efficient scanners and parsers. From the description of language as a set of regular expressions (for the lexical level) and a BNF grammar (for the syntactic level), SYNTAX produces, using LR-based techniques, automata that recognize this language. SYNTAX is a production-quality system, which integrates the result of several years of theoretical research work and practical experience. Some of its advantages are:

- SYNTAX can solve analysis conflicts by using an unbounded number of look-ahead characters (in the scanner) or tokens (in the parser), while keeping a linear analysis time [Bou84].
- You can influence the analyzers through user-defined predicates and actions.
- The tables representing the automata are typically compressed by more than 95% while keeping a constant access time.
- Keywords are recognized in constant time by a perfect and optionally minimal hash function constructed by SYNTAX.
- Above all, SYNTAX provides a very powerful and easily tunable error repair and recovery mechanism [BoJ87].

In addition, SYNTAX comes with a complete library of modules that facilitate the development of language-processing applications: source text manager, character strings manager, error messages messages, ... FNC-2 makes heavy use of this library.

TRANSAT [Sou90] is an attributed tree transformation system based on the same paradigm as the well-known OPTRAN system [LMW88]. However it has a much cleaner semantics; in particular it is able to detect when a transformation system is confluent, i.e. when the result of the transformation is the same whatever actual sequence of transformation steps is used. This eliminates the need for the user to specify the transformation strategy, which is automatically inferred by TRANSAT. It offers several other advantages, however its development has begun only recently so we have no experience with it yet.

PAGODE [DMR89, DMR90] is a code generator generator based on tree rewriting. From the description of the intermediate language (an AAS) and of the target machine—very easy to write from the manufacturer's manual—, it produces an instruction selector which produces locally-optimal code and a register allocator; the latter is in the form of an OLGA AG. Special attention is paid to the management of temporaries.

In addition, work has been done and is still in progress at INRIA on the detection of parallelism in programs written in classical programming languages (FORTRAN) and its exploitation for producing highly efficient code for architectures involving fine-grain parallelism (RISC, VLIW, vector machines). Although we have reached very good results for particular cases, this work is not "ripe" enough yet to be embodied into real compiler generators, but we're working on this.

The association of FNC-2 with all these tools already forms a highly usable compiler construction workbench, and will become more and more effective as we improve them.

5 Conclusion and Future Work

We have presented the FNC-2 attribute grammar system and its input language OLGA, and we have described how they can be used to develop large applications. Of paramount importance for the efficacy of the development is FNC-2's support for modularity. Our experience is not very deep yet but it already shows that we are on the right track.

Apart from all the work that remains to make FNC-2 a real production-quality system (see [JPJ90, JuP90] for more details), an important item on our "to do" list that is particularly relevant to application development is the improvement of the modularity features. The authors of the MARVIN system [GGV86] have extended the ACG paradigm by unifying the syntactic (the trees) and semantic (the attributes) domains; this allows to define functions by means of AGs and offers real support for modularizing an AG into (reusable) sub-AGs. We're thinking of implementing such a scheme in FNC-2. Another longer-term research direction is to extend the expressive power of AGs to computation on graphs, fixed-point computations, etc.; our work on space optimization [JuP90] offers new opportunities therefore.

Also, we are thinking of using OLGA and FNC-2 to design specialized languages and systems for compiler generation based on AGs. The problem with AGs is that it is a general-purpose programming method, while one would rather have specialized tools with a knowledge of a particular application domain and of the algorithms to use in that domain. We think that AGs are a promising technique both for developing the tools and as their target language. GIGAS [Fra89] is a good example of this methodology: it is a language/system for specifying graphical pretty-printers of abstract trees and is based on the notions of boxes. Its output is an AG, with semantic rules for e.g. computing the coordinates of the boxes. When compiled by the Synthesizer Generator [ReT89], it produces a syntax-directed editor with incremental redrawing of the graphic boxes. A GIGAS specification is translated into an AG which is typically ten times larger; this shows the interest of having specialized tools rather than general-purpose ones! We would like to apply the same methodology to compilation problems such as data flow analysis [Kil73] and name analysis [Rei83]. With such a good AG-processing system as FNC-2 as base, and with the other compiler construction tools under development at INRIA, our dream of a complete, powerful and efficient compiler development workbench will become a reality.

References

[ASU86] A. V. Aho, R. Sethi & J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986.

[Bou84] P. Boullier, "Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques," thèse d'État, Univ. d'Orléans, Dec. 1984.

[BoD88] P. Boullier & P. Deschamp, *Le système SYNTAX—Manuel d'utilisation et de mise en œuvre sous Unix*, INRIA, Rocquencourt, Sept. 1988.

[BoJ87] P. Boullier & M. Jourdan, "A new Error Repair and Recovery Scheme for Lexical and Syntactic Analysis," *Sci. Comput. Programming* **9**, 4 (Dec. 1987), 271–286.

[CoF82] B. Courcelle & P. Franchi-Zannettacci, "Attribute Grammars and Recursive Program Schemes," *Theoret. Comput. Sci.* **17**, 2 and 3 (1982), 163–191 and 235–257.

[DJL88] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. **#323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[DMR89] A. Despland, M. Mazaud & R. Rakotozafy, "Using Rewriting Techniques to Produce Code Generators and Proving them Correct," Rapport RR-1046, INRIA, Rocquencourt, June 1989. To appear in *Sci. Comput. Programming.*

[DMR90] ———, "Pagode: A Back-end Generator using Attributed Abstract Syntaxes and Term Rewritings," in this volume, Oct. 1990.

[EnJ90] J. Engelfriet & W. de Jong, "Attribute Storage Optimization by Stacks," *Acta Inform.* (1990).

[Fra89] P. Franchi-Zannettacci, "Attribute Specifications for Graphical Interface Generation," in *Information Processing '89*, San Francisco, CA, G. X. Ritter, ed., 149–155, North-Holland, Amsterdam, Aug. 1989.

[GaG84] H. Ganzinger & R. Giegerich, "Attribute Coupled Grammars," in *ACM SIGPLAN '84 Symp. on Compiler Construction*, Montréal, published as *ACM SIGPLAN Notices* **19**, 6 (June 1984), 157–170.

[GGV86] H. Ganzinger, R. Giegerich & M. Vach, "MARVIN: a Tool for Applicative and Modular Compiler Specifications," Forschungsbericht 220, Fachbereich Informatik, Univ. Dortmund, July 1986.

[GJR87] J. Garcia, M. Jourdan & A. Rizk, "An Implementation of PARLOG Using High-Level Tools," in *ESPRIT '87: Achievements and Impact*, Brussels, Commission of the European Communities—DG XIII, ed., 1265–1275, North-Holland, Amsterdam, Sept. 1987.

[Jou84] M. Jourdan, "Les grammaires attribuées: implantation, applications, optimisations," thèse de Docteur-Ingénieur, Univ. Paris VII, May 1984.

[JLP90] M. Jourdan, C. Le Bellec & D. Parigot, "The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation," in *Attribute Grammars and their Applications (WAGA)*, Paris, P. Deransart & M. Jourdan, eds., 222–237, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990.

[JoP89] M. Jourdan & D. Parigot, *The FNC-2 System User's Guide and Reference Manual*, INRIA, Rocquencourt, Feb. 1989. This manual is periodically updated.

[JPJ90] M. Jourdan, D. Parigot, C. Julié, O. Durin & C. Le Bellec, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," in *ACM SIGPLAN '90 Conf. on Programming Languages Design and Implementation*, White Plains, NY, published as *ACM SIGPLAN Notices* **25**, 6 (June 1990), 209–222.

[JuP90] C. Julié & D. Parigot, "Space Optimization in the FNC-2 Attribute Grammar System," in *Attribute Grammars and their Applications (WAGA)*, Paris, P. Deransart & M. Jourdan, eds., 29–45, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990.

[KLM83] G. Kahn, B. Lang, B. Mélèse & É. Marcos, "Metal: a Formalism to Specify Formalisms," *Sci. Comput. Programming* **3** (1983), 151–188.

[Kas80] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.* **13**, 3 (1980), 229–256.

[Kas84] ———, "The GAG-System—A Tool for Compiler Construction," in *Methods and Tools for Compiler Construction*, B. Lorho, ed., 165–182, Cambridge Univ. Press, Cambridge, 1984.

[Kil73] G. Kildall, "A unified approach to global program optimization," in *1st ACM Symp. on Principles of Progr. Languages*, 194–206, Jan. 1973.

[**Knu68**] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2**, 2 (June 1968), 127–145. Correction: *Math. Systems Theory* **5**, 1 (Mar. 1971), 95–96.

[**LMW88**] P. Lipps, U. Möncke & R. Wilhelm, "OPTRAN – A Language/System for the Specification of Program Transformations: System Overview and Experiences," in *Compiler Compilers and High Speed Compilation*, Berlin, D. Hammer, ed., 52–65, Lect. Notes in Comp. Sci. #**371**, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1988.

[**Lor77**] B. Lorho, "Semantic Attributes Processing in the System DELTA," in *Methods of Algorithmic Language Implementation*, A. Ershov & C. H. A. Koster., eds., 21–40, Lect. Notes in Comp. Sci. #**47**, Springer-Verlag, New York–Heidelberg–Berlin, 1977.

[**Par88**] D. Parigot, "Transformations, évaluation incrémentale et optimisations des grammaires attribuées: le système FNC-2," thèse, Univ. de Paris-Sud, Orsay, May 1988.

[**Rei83**] S. P. Reiss, "Generation of Compiler Symbol Processing Mechanisms from Specifications," *ACM Trans. Progr. Languages and Systems* **5**, 2 (1983), 127–163.

[**ReT89**] T. Reps & T. Teitelbaum, *The Synthesizer Generator*, Springer-Verlag, New York–Heidelberg–Berlin, 1989.

[**Sou90**] A. Souah, "Contribution à la sémantique déclarative des systèmes de transformation d'arbres attribués," thèse, Univ. d'Orléans, Sept. 1990.

[**UDP82**] J. Uhl, S. Drossopoulos, G. Persch, G. Goos, M. Daussmann, G. Winterstein & W. Kirchgäßner, *An Attributed Grammar for the Semantic Analysis of ADA*, Lect. Notes in Comp. Sci. #**139**, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[**VoM82**] A. O. Vooglaid & M. B. Méristé, "Abstract Attribute Grammars," *Progr. and Computer Software* **8**, 5 (Sept. 1982), 242–251.