STARLET : An Affix-Based Compiler Compiler designed as a Logic Programming System

Jean BENEY[1] Jean-François BOULICAUT[1-2]

Institut National des Sciences Appliquées de Lyon [1]
Laboratoire d'Informatique Appliquée, Batiment 502
F-69621 Villeurbanne Cedex (France)

Ecole Centrale de Lyon [2]
Département Mathématiques-Informatique-Systèmes, BP 163
F-69131 Ecully Cedex (France)

Abstract : We present STARLET, a new compiler compiler which compiles Extended Affix Grammars defining a translation into an executable program : the translator. We look at its operational semantics and we focus on the points which are close to or different from Prolog procedural semantics. We discuss the two interwoven issues which are Program Reliability (due to many static checks) and Program Efficiency (optimizations at compile time). Both are reached through a systematic use of grammatical properties.

I Introduction

Our research group has been working on *grammatical programming development* i.e. an approach to software construction based on compiling methods [Beney-86]. Compiler compilers are designed as high-level programming environments and if compiler writing is the major application field, we also investigate other application fields [Frécon 89].

Within the grammatical programming framework, the specification step must produce *abstract language definitions*. As context-free grammars are non-algorithmic descriptions for the syntax of languages, *two-level grammars* are grammatical formalisms which enable the semantic part to be defined. Among the best known types of two-level grammars are the *W-grammars* devised by A. van Wijngaarden and the *Attribute Grammars* devised by D.E. Knuth. To help language prototyping and also to improve its final implementations, the idea of analysis-oriented two-level grammars (and compiler compilers) leads to restricted classes of attribute grammars (L-attributed systems [E2S-84], SNC attribute grammars [Jourdan-88]...) or restricted classes of W-grammars (*Affix Grammars*, *Extended Affix Grammars*, RW-grammar and transparent W-grammars devised respectively by C.H.A. Koster [Koster-70], D.A. Watt [Watt-74] , M. Simonet [Simonet-81] and J. Maluszynski [Maluszynski-84]). CDL was the first compiler compiler based on affix grammars [Koster-77].

Our group has worked on affix grammars and one CDL-like implementation based on its own set of well-form conditions to deal with a deterministic top-down analysis [Beney-80]. This system is called LET (trademark of INSA de Lyon). Then, we needed to extend the class of grammar which was accepted as well as to improve the translator writing facilities. Independently of the research in Berlin or in Nijmegen (e.g. EAGLE [Franzen-77] or PROGRAMMAR [Meijer-86]), this has lead to a *shift from Affix Grammars to Extended Affix Grammars* and a *shift from an algorithmic language to a logic programming language*. We called this grammatical system and its metalanguage STARLET while the first implementation is called STARLET/GL.

II Background : Affix Grammars and related formalisms

Like W-grammars, Affix Grammars have two grammatical levels but there is a clear distinction between the notions (i.e. the non-terminals) and their parameters : the so-called *affix positions* which are *variables*. It introduces structural constraints on hypernotions and an *underlying context-free grammar* (UCFG). The affix level (affix rules) assign domains to affix positions. The referencing problem known to be undecidable for W-grammars is then decidable and one can make straightforward extensions of context-free parsers. As it is oriented towards analysis and metacompilation, *mode assignment* is introduced to specify the data flow. Another useful concept for programming purposes is the "*primitive predicate*". The primitive predicates, described in CDL and LET by some programming languages, are used to

define input/output operations (reading or writing of terminal symbols) or any algorithmic processing (e.g. symbol table management).

The class of grammars accepted by CDL or even LET is rather poor since the analysis method is an adaptation of the recursive descent (without any automatic backtracking and a one-pass left to right evaluation of every affix value). These systems are not good enough for general programming purposes since they look like algorithmic programming languages where coding and concrete data structure processing always interferes with higher-level analysis tasks.

D.A Watt proposed *Extended Affix Grammars* (EAG) [Watt-74] in order to eliminate primitive predicates and return to a more generative formalism. An EAG is "extended" since affix positions can be *affix expressions* and this allows all relationships between the affix in each rule to be implicitly defined. Thus, explicit evaluation rules (which are often simple copies) and constraints become unnecessary since the Uniform Replacement Rule combined with the type discipline (following affix rules specifications for affix positions) are enough. Therefore, one of the major problems in designing a compiler compiler which processes EAG is to implement a *grammatical unification algorithm* [Maluszynski-84] (see § III.2).

The question arises of the differences between Affix Grammars (Extended or not) and Attribute Grammars. Some answer that there is no distinction : EAG has become either Extended Affix Grammars or Extended Attribute Grammars in [Watt-83], assuming that the original W-grammar mechanism for context-sensitive requirements by predicative hyperrules could be applied to attribute grammars as constraints associated to each rule. We also notice that affix grammars which are processed by the LET compiler compiler are 1L-Attribute Grammars (see for example the MIRA compiler compiler [E2S-84] or [Deransart-88a] for a survey). This arises from the fact that in LET, affix rules (and thus two-level grammar mechanisms) are not really used since the user has to choose concrete data structures for affix variables and use them via the primitive predicates. In both cases, semantic values can be used to drive syntactic analysis (values may be computed during parsing).

However, there are differences between affix and attribute formalisms when no context-free basis is apparent. An attribute grammar ought to be defined from an underlying context-free language (which is independent from the attribute values). This means that the underlying language must be defined by a well-formed context-free grammar according to a particular parsing method (LL, LR ...).

Within the affix framework, the UCFG may have unwanted properties (e.g. ambiguities) that affix-driven parsing can allow because of the *contextual analysis method*. This problem has already been informally considered in [Beney-89].

On the other hand, many grammatical formalisms have been proposed under the generic term of "*Logic Grammars* ". *Metamorphosis Grammars* (MG) [Colmerauer-77] or *Definite Clause Grammars* (DCG) [Pereira-80] are grammatical formalisms designed for natural language processing which are easily compiled into (or interpreted as) logic programs. Furthermore, it is interesting to consider MG or DCG as programming languages and implement with them more sophisticated grammar processors. S. Saidi and J.F Boulicaut are working on such an implementation of transparent two-level grammars [Saidi-90].

Logic Programming features such as the tree data structure of terms, the variable instantiation by unification and the systematic backtracking mechanism are useful to handle multiple analysis and thus ambiguities in language processing. Given the SLD-resolution, interpreters turn Prolog into a real generator of non-deterministic top-down parsers.

The well-known shortcomings of top-down parsing (prefix sharing and local ambiguities, ε-productions, ambiguities) can easily be managed but it costs too much for many applications of practical interest. The *efficiency* of Prolog programs must be improved by means of *explicit control over resolution* (e.g. cut, freeze,diff, wait...), *mode* assignments or the *informed use of unification* (e.g. taking concatenation associativity into account). On the other hand, the production of more reliable logic programs is made easier by certain systems which have explicit *type* definitions [Borland-86].

EAG provided with operational semantics (given a compiler compiler) are very close to logic programs. We decided to execute an EAG defining an *unambiguous left to right translation* as a logic program. Thus, the rules must be selected by means of affix values to check whether the tried productions do not lead to *blind alleys* (we want only programs which deliver at least one result). This means that the affix values (or at least some of them) have to be computed while parsing. Local ambiguities are allowed since the right decision can be taken by inspecting known affix values (*contextual LL(k) parsing*). Each rule call can only have one successful completion as if there was an implicit "cut" at the end of each rule (see the recursive back-up scheme in [Koster-74]).

Two-level grammar features found in STARLET extend logic programming facilities for this class of problem (left to right unambiguous translations) by a well-motivated introduction of *types*, *modes* and *implicit control* of computations.

Other well-form conditions ensure the *grammar computability* i.e. the ability to recognize the axiom of the grammar while instantiating every affix variable. Therefore, our work on the reliability and the efficiency of translators developed with STARLET/GL contributes to the software engineering of logic programming on this restricted class of problem.

III THE STARLET/GL operational semantics

The reference manual of the STARLET/GL implementation is [Beney-87]. The STARLET/GL programming environment PLEIADE is running on a BULL DPX5000 under SPIX and the compiler can be used on other UNIX-like systems. This has been implemented by J. Beney (STARLET/GL compiler), A.N. Benharkat (Syntactic Editor, Pretty-Printer and Application Managment : [Benharkat-90]) and H. Harti (interpreter [Harti-89]).

III.1 Defining languages, translations and programs

The STARLET grammatical formalism is a set of conventions to specify EAG (see references for formal definitions). Our notations for EAG have been influenced by the van Wijngaarden style and also by our practical experiences with the LET language. Lexical features bring STARLET grammars very close to the natural language style of W-grammars.

A non-terminal symbol is made up of letters, spaces, quotes, minus signs and dollar signs (e.g. "do $ times $"). The number of dollar signs is the arity of the symbol (the number of the affix positions of the symbol). A hypernotion is written with parenthesized affix sentential forms in place of the dollar signs (e.g. "do (one N) times (ABC)"). We say that STARLET/GL allows *split identifiers.*

We illustrate the notations on a classic example : $L_1 = \{ \; a^n b^n c^n , n \geq 1 \}$

Example 1 :

```
ROOT : anbncn.          $    EAG axiom    $
AFFIXES :               $    The metarules      $
ABC : alpha ; beta ; gamma.  $ A rule which enumerates the available values for ABC $
N : zero ; one N.       $    An affix rule which defines the natural numbers  $
NOTIONS:                $    Rules defining the non-terminals $
anbncn :    do ( N ) times (alpha ), do ( N ) times (beta ), do ( N ) times ( gamma ).
```

```
do ( : zero ) times ( : _ABC ) : TRUE .
do ( : one N ) times ( : ABC) : an (ABC), do (N ) times ( ABC) .
an ( : alpha) : object("a").
an ( : beta) : object("b").
an ( : gamma) : object("c").
```

STARLET/GL keywords are in uppercase style. Modes are lexically defined with the symbol " : " in front of (resp. behind) a parameter to define the inherited mode (resp. the synthesized). An underscore in front of an affix variable denotes that its value will be unused by the computation.

This EAG is a *specification of the language* L_1.

We can consider this EAG as a generative grammar (assuming that "object" plays the role of "protonotions ending with symbol" in the two-level grammar scheme and forgetting mode assignment or underscores) but it is worth using it either to generate sentences of L_1 or parse some strings which must belong to L_1.

In fact, "object" is an external function which outputs its string parameter. A slight change to Example 1 gives a correct STARLET/GL program which generates sentences. We can give a value to N by an introduction of an inherited affix position to "anbncn" : anbncn (: N).

With "ROOT : anbncn (one one one zero) " the output is "aaabbbccc"

A *recognizer* of L1 could easily be implemented too. The effective terminal symbols are recognized by an external function "symbol". This function manages a buffer which allows source backtracking when necessary. The function "object" generates the object code with backtracking synchronized to source backtrack. A recognizer of L_1 (with the affix rules of Example 1) could be :

```
ROOT : parser.
NOTIONS : $ "write string" and "needed EOF" are external self-explanatory functions $
parser : anbncn, needed EOF, write string ("OK") ; write string ("KO").
?anbncn / N :          accept and count (N) times (alpha),
                              accept (N) times (beta), accept (N) times (gamma).
?accept and count ( one N :) times ( : alpha ) :
      there is an (alpha) on input, accept and count ( N ) times ( alpha ).
accept and count ( zero : ) times ( : alpha ) : TRUE.
accept ( zero : ) times ( : ABC ) : TRUE.
?accept ( one N : ) times ( : ABC ) : there is an ( ABC ) on input, accept ( N ) times ( ABC ).
```

?there is an (: alpha) on input : symbol ("a").
?there is an (: beta) on input : symbol ("b").
?there is an (: gamma) on input : symbol ("c").

Character "?" is used to specify that the rule is supposed to be a *Test (vs. Action)*. There is a (static) check of the *algorithmic consistency* of this specification : a test may fail while an action never fails. It is useful to help error recovery during parsing. Character "/" is followed by local variables used in the right hand side of a rule.

III.2 Grammatical unification and Uniform Replacement Rule

To execute a STARLET/GL program is to try to recognize the root of the EAG for a given input text. Its results are available as an object text. Data types are considered as intermediate languages defined by mean of the context-free affix rules. Affix sentential forms parameterize the non-terminal symbols and are generally manipulated via their derivation tree in this grammar of affixes. Thus, the general case of grammatical unification is the tree confrontation which is used to build and look at affix sentential form structures. Variables are considered as logic variables : the program tries to instantiate them (at present, every variable of a STARLET/GL program has to be instantiated).

A notion rule defines an analysis method for the non-terminal which is its left hand side. There may be several rules defining a given non-terminal. Alternatives in a rule development are considered from left to right.

A rule call introduces a *confrontation* between formal parameters and arguments i.e. an oriented type-sensitive grammatical unification. When the confrontation succeed, the result is either a *tree construction* or a *tree split* (expression in front of expression is not allowed). A *rule call* is a success if : *input confrontation* is a success, the *development of the rule* is a success and finally the *output confrontation* is a success.

Thus a rule call may fail either because of the confrontations or its development. The *Uniform Replacement Rule* is responsible for the correct propagation of the values already known. The confrontation of a variable and an affix expression may succeed only if this expression and one of the legal structures for the variable data type are compatible (e.g. an affix expression 'e' is compatible with the structure of an affix 'a' if affix rules exist such that 'e' can be derived from 'a' according to the affix rules).

A tree construction happens whenever an affix sentential form appears as an argument to be unified with an inherited variable affix position or as a synthesized affix position.

A tree split happens whenever an affix sentential form appears as an inherited affix position or as an argument to be unified with a synthesized affix position.

There are special cases of confrontation since we need predefined data types (INTEGER for integer, CHARAC for character and STRING for character strings) or enumerated types (every value is a single terminal affix : it allows the definition of the implicit intersection of types).

When possible, test which are needed during the confrontations are carried out at compile time and therefore rules which are able to succeed are statically sorted. It avoids useless attempts at confrontation (see § IV).

Application of the *Uniform Replacement Rule* combined with the *split identifiers* and *the confrontation* facilities give an idea of the concision and readability of the programs :

With these affix rules defining list of integers :

L : empty ; X L . X : INTEGER.

Test for the equality of two list If (: L) is (: L) : TRUE .

Extract the first element of a list The First of (: X _L) is (X :) : TRUE.

Test the first element of a list If the first of (: X _L) is (: X) : TRUE .

Unused variables are a convenient means of imposing type constraints over the values. We illustrate the use of STARLET as a logic programming language on a complete program that reads an integer list, builds it in memory and outputs it in the reverse order.

Example 2 : $ Input 3 4 2 outputs 2 4 3 $

ROOT : Reverse , next line.

$ "next line", "read integer", "write integer", "write charac" are external functions $

AFFIXES : L : empty ; X L . X : INTEGER. CAR : CHARAC .

NOTIONS :

Reverse / L, L1 : Read (L) , Reversed list of (L) in front of (empty) is (L1) , Write (L1) .

Reversed list of (: X L) in front of (: L1) is (L2 :) :

Reversed list of (L) in front of (X L1) is (L2) .

Reversed list of (: empty) in front of (: L) is (L :) : TRUE .

?Read (X L :) : read integer (X) , write integer (X) , write charac (' '),Read (L) .
Read(empty :) : next line .
Write(: X L) : write integer (X) , write charac (' ') , Write (L) .
Write (: empty) : TRUE .

III.3 A translator written in STARLET/GL

We illustrate STARLET programming with the example of expression translation. We obtain a list of identifiers and then an assignment instruction where the righthand side is an infix expression (input device is the standard one).

We parse it and check for the identifier use : each used identifier must have been declared once only. Then the tree representation of a correct "program" is translated into its postfix representation on the standard output device. If the expression turns out to be constant (each of its operands is a constant), the translator evaluates it.

Example 3 : *$ input "a, b, c; c := b+c*a" outputs "c b c a * + =" $*
*$ input "a ; a := 2 + 3 * 4" outputs "a 14 =" $*

```
ROOT : translate.
WITH : lexico. $    This program uses notions from this module : needed EOF, error,
                    symbol, ident, constant ... $
AFFIXES :                          $ Abstract syntax $
absy : absy op absy ; cste ; name.
cste: INTEGER .
name : STRING.
op : plus ; mult ; ass.
symbtab : name symbtab ; empty .   $A symbol table is a list of identifiers $
VARIABLES : symbtab.               $ Global variable to contain the symbol table $

NOTIONS :
translate : init(symbtab), declarations, instruction , needed EOF.
declarations : identifier list , needed symbol (";") .
identifier list : one variable , others identifiers.
others identifiers: symbol (",") , identifier list ; TRUE.
one variable / name1 :
        ident(name1) , check and enter (name1) in (symbtab) gives (symbtab) ;
        error("identifier expected").
instruction / absy : assignment (absy) , postfix (absy ).
```

```
assignment( name1 ass absy1 : ) / absy :
        variable reference (name1), needed symbol(":="),
            expression(absy ) , optimization of (absy) gives (absy1).
?variable reference (name:) : ident(name) , check for (name) .
variable reference ("foo":): error("identifier expected").
    $ "foo" replaces the missing identifier in symbtab $

expression (absy1:) / absy2 : term (absy2) , rest of expression (absy2,absy1) .
?rest of expression (: absy,absy1:) / absy2 :
    symb("+"), term(absy2), rest of expression (absy plus absy2,absy1).
rest of expression ( : absy,absy : ) : TRUE.
term(absy:)/absy1: factor(absy1), rest of term (absy1,absy).
?rest of term (:absy1,absy:) / absy2 :
    symb("*"), factor(absy2), rest of term (absy1 mult absy2,absy) .
rest of term ( : absy , absy :) : TRUE.
?factor(cste : ): constant (cste).
?factor(absy : ): symb("("), expression (absy), needed symbol ( ")").
?factor(name : ): variable reference (name).
factor (0:) : error ("operand is needed").

postfix(: cste ): write integer (cste).
postfix(: name) : write string (name).
postfix(: absy1 op absy) : postfix(absy1), postfix(absy), write op (op).
write op (: plus) : write string (" + ").
write op (: mult) : write string (" * ").
write op (: ass) : write string (" = ").

$ Optimization $

?optimization of (:absy) gives (absy1:) : if expression (absy) is constant its value is (absy1).
optimization of ( : absy ) gives ( absy : ) : TRUE.
?if expression (: absy1 _op absy2 ) is constant its value is ( absy30 : ) / absy10, absy20 :
    if expression (absy1) is constant its value is ( absy10 ) ,
    if expression (absy2) is constant its value is ( absy20 ) ,
    eval (absy10, op, absy20, absy30).
if expression (: cste) is constant its value is ( cste : ) : TRUE.
?eval (: absy1, : plus, : absy2, cste : ) : add (cste1,cste2,cste).
?eval (: absy1, : mult, : absy2, cste : ) : mul (cste1,cste2,cste).
```

$ Symbol table management $

```
init(empty :) : TRUE.
?check and enter (: name) in (: symbtab) gives (symbtab : ) :
        look for (name) in (symbtab) , error("identifier already declared").
check and enter (: name) in (: symbtab) gives ( name symbtab :) : TRUE.
look for (: name) in ( : name _symbtab) : TRUE.
?look for (: name) in (: _name1 symbtab) : look for (name) in (symbtab) .
check for (: name) :    look for (name) in (symbtab) ;  error("identifier not declared").
```

IV STARLET/GL for grammatical debugging

Optimizations consist of cutting useless pieces of the generated program. During this process, the translation of some rule-call or some rules may become impossible. Therefore, optimizations can lead to error messages that help to debug the program without having to execute it. We introduce some of the optimizations made by the STARLET/GL compiler which are related to grammatical properties. It helps either language design (grammatical debugging) or translator debugging .

First, for each rule call, we select only the rules whose parameters are of the "right type". Note that with Prolog this cannot be done and that some rules are always rejected when tried during execution.

This optimization may show logical errors since we only work on programs which have at least one solution :

- if, for a call, not one single rule has good parameters ; the call will always fail so that the rule that contains this call is a *systematic blind alley.*
- if a rule is never used because of its parameter types

This is checked at compile time by means of affix rules parsing. To sort the rules before runtime enables the user to give the same name to rules which process objects of different types without useless attempts at confrontation.

Example :

Given the affix rules

```
LE : empty ; e  LE.          e : INTEGER.
LN : empty ; n  LN.          n : STRING
```

write (: e LE) : ...
write (: n LN) : ...
write (: empty) : ...
A : ... write (LE) , write (LN) ...

In trying the development of "A", the first "write" call will not use the second "write" rule while the second will not use the first "write" rule. A single rule can be used to process values which are not of the same type but which share some structures (e.g. "empty"). The third rule will be used for each of the "write" call in the development of "A".

Static checks ensure that tree construction will always succeed. At run time, in the case of tree split, one must also check that the actual value for variables have the right structures. Thus, for the rule call "write (LE)", there is a test which tells if the structure of the variable "LE" is "e LE" (the first "write" rule is tried)) or "empty" (the third "write" rule is tried).

Note that it may reveal problems when the affix rules define an ambiguous grammar [Maluszynski-84] (as the "absy" affix rule in Example 3). As only one successful parse is retained (parse of affix sentential forms given the affix rules), one may not split some trees. However, Example 3 implements an unambiguous translation since the grammar which is used to parse infix expressions is unambiguous (operator precedence is well-defined).

Secondly, we reject some rules by checking the context of a call i.e. the type or value constraints fixed by the previous and the following calls in the development. It enables optimizations but if every possibility is eliminated it shows that there are logical errors.

Here we also find some errors : a succession of calls may always fail or a rule is no longer used because of the context of its calls.

Example :
Given the affix rule L : x L ; empty .
Read (x L :) : ....
Read (empty :) : ....
Write (: x L) : ....
Write (: empty) : ....
Try / L : Read (L) , write (L) .

Inside "Try" development, if the first rule "Read" is a success (resp. the second), we should try only the first rule "Write" (resp. the second).

The declarations of the possible failure of the rules (test rules) allow another optimization to be made when unreachable rule calls can be cut. This can lead to another logical error detection when a rule is always unreachable because of a previous rule that always succeeds (action rules).

Lastly, dataflow checking is very strict in the current implementation since every variable may receive a value. Global variables are not considered as logical variables : they are assigned by a side-effect and their use is not checked.

V Conclusion

We are working on an efficient implementation of Extended Affix Grammars which provides high-level debug facilities (grammar debugging) through numerous static checks. Previous experiments with an algorithmic implementation of affix grammars (LET) were made and and we already appreciate the ease of use of the STARLET predicative interpretation since it eliminates many of the coding difficulties previously encountered.

We have few experimental results since STARLET/GL is currently used in a research program on the automatic analysis of French (3000 lines, M. De Brito) or has been used for an expert system generator (5000 lines, L. Coudouneau). Moreover, our students learn compiling techniques with this compiler compiler (150 per year).

On the other hand, we still have a lot of work with the associated programming tools. It includes not only further developments and maintenance on the PLEIADE environment but also the design of new tools as soon as we are able to explicit grammatical programming methodologies (method-driven programming tools).

References

Many references are only available in French (see "-F-"). INSAL denotes Institut National des Sciences Appliquées de Lyon. BULLET is a local journal (6 issues since 1986) which can be obtained upon request to the authors. Most of the LET and STARLET applications are reported in it.

Beney-80 -F- BENEY (J.), FRECON (L.) . Langage et Systèmes d'Ecriture de Transducteurs. RAIRO Informatique, 1980, Vol.14, n°4, p. 379-394.

Beney-86 -F- BENEY (J.). La notion de programmation grammaticale. BULLET, juin 1986, n°1, p. 16-21.

Beney-87 -F- BENEY (J.). Présentation de STARLET/GL. Rapport Interne INSAL, Laboratoire d'Informatique Appliquée, 1987.

Beney-89 BENEY (J.), BOULICAUT (J.F.). Transfer of Expertise between Two-Grammars and Logic Programs Through an Affix Implementation. in : Proceedings of Informatika'89, (E. Tyugu Ed.), Tallin (URSS), May 29th - June 2nd 1988, 18 p.

Benharkat-90 -F- BENHARKAT (A.N.). Atelier Logiciel PLEIADE : Edition des modules et suivi des applications STARLET. Ph-D thesis : INSAL, 1990, 228 p.

Borland-86 BORLAND International. Turbo Prolog Owner's Handbook, Scott Valley, USA, 1986.

Colmerauer-77 COLMERAUER (A.). Metamorphosis grammars. in : Natural Communication with Computer, Springer-Verlag, 1977, LNCS 63, p.133-189.

Deransart-88a DERANSART (P.), JOURDAN (M.), LOHRO (B.). Attribute Grammars : Definitions, Systems and Bibliography. Springer-Verlag, LNCS 323, August 1988, 232 p.

Deransart-88b DERANSART (P.), MALUSZYNSKI (J.).A Grammatical view of Logic. in : Proceedings of PLILP 88, Orléans, France, May 1988, Springer-Verlag, LNCS 348, 1989, p. 219-251.

E2S-84 Expert Software System. MIRA user's guide. E2S Building "de Schelde" Moutstaat 100 B-9000 GHENT.

Franzen-77 FRANZEN (H.), HOFFMANN (B.), POHL (B.), SCHMIEDECKE (I.R.). The EAGLE Parser Generator : an Experimental Step towards a Practical Compiler-Compiler using Two Level Grammars. in : 5th Annual III Conference, Guidel, France, May 1977, p.397-420.

Frécon-89 -F- FRECON (L.). Grammaires affixes : applications et questions ouvertes. Document de travail, Atelier Lyon-Nimègue sur les grammaires affixes, 24-26 juin 1989 (Organisation INSAL), 25 p.

Harti-89 -F- HARTI (H.). Exploitation prédicative des grammaires affixes : interprète et machine virtuelle STARLET. Thèse de Doctorat : INSAL, 1989, 168p.

Jourdan-88 JOURDAN (M.), PARIGOT (D.). The FNC-2 System : advances in attribute grammars technology. INRIA, april 1988, RR-834, 28 p.

Koster-70 KOSTER (C.H.A.).
Affix Grammars. in : ALGOL 68 Implementation,
J.E.L. PECK Ed., North-Holland, 1970, p.95-109.

Koster-74 KOSTER (C.H.A.). A technique for parsing ambiguous grammars.
in : Proceedings of GI-4 Jahrestagung, (D. Siefkes Ed.),
Springer-Verlag, 1975, LNCS 26, p. 233-246.

Koster-77 KOSTER (C.H.A.). CDL : a Compiler Implementation Language.
in : Methods of Algorithmic Language Implementation,
Springer-Verlag, 1977, LNCS 47, p. 341-351.

Maluszynski-84 MALUSZYNSKI (J.). Towards a programming language based on the notion of two-level grammars.
Theoretical Computer Science, 1984, Vol.28, p.13-43.

Meijer-86 MEIJER (H.). PROGRAMMAR : a translator generator.
Ph-D thesis : University of Nijmegen, 1986, 225 p.

Pereira-80 PEREIRA (F.C.N.), WARREN (D.H.D.). Definite Clause Grammars for Language Analysis : a survey of the formalism and a comparison with Augmented Transition Networks.
Artificial Intelligence, 1980, Vol. 13, p. 231-278.

Saidi-90 SAIDI (S.) , BOULICAUT (J.F.).
Checking and Debugging Transparent Two-Level Grammars.
ECL : Research Report 90-10, July 1990, 24 p.

Simonet-81 -F- SIMONET (M.). W-grammaires et logique du premier ordre pour la définition et l'implantation des langages.
Thèse d'état : Grenoble , 1981, 329 p.

Watt-74 WATT (D.A.). Analysis-oriented two-level grammars.
Ph.D. thesis : University of Glasgow, january 1974.

Watt-83 WATT (D.A.), MADSEN (O.L.). Extended Attribute Grammars.
The Computer Journal, 1983, Vol. 26, n° 2, p. 142-153.
