Affix Grammars for Programming Languages

C.H.A. Koster*
Department of Informatics, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract

Affix Grammars are members of the family of Two-Level Grammars, along with W-grammars, Metamorphosis Grammars and Attribute Grammars. In this tutorial we shall be concerned with the nature and rationale of Affix Grammars and their application in describing programming languages. Some parsing and affix evaluation methods for deterministic and nondeterministic Affix Grammars are discussed. By means of an example, a comparison is made with W-grammars and Attribute Grammars.

1 On describing Programming Languages

Informatics is full of artificial languages, not only many programming languages, specification languages and mathematical notations, but also the user interfaces of innumerable application systems. The greater part of informatics is paper work: the description and realization of the figments of the programmer's mind. The price of invention is description: what has been invented must be described, so that it can be communicated to others. All those artificial languages need some description, be it informal or formal.

The sixties of this century were an era of rapid development of algorithmic languages. The beginning and end of this period were marked by ALGOL 60 [23] and ALGOL 68 [30], respectively. Both languages were developed by a group of scientists as a lingua franca, to get away from the machine-oriented and vendor-oriented languages of the day. Both languages, by the formality of their description, set a new standard of precision. Both were accepted with mixed feelings [6], most readers finding their description far too complicated for ordinary programmers.

The definition of a programming language serves different (and rather contradictory) purposes:

- It serves as reference material for the more serious users of the language. To that end it has to have great expositionary and didactic value for human readers. A great language definition is a literary work of art.
- It should enable specialists to answer questions about the language without having to take recourse to its compiler. To that end, it will have to describe the syntax and semantics of the language in a very precise way. Although it is possible to be very precise in unformalized prose, this will imply in general that the description has to be formal.

*Visiting the Technical University of Budapest on a TEMPUS-grant.

- It will serve as the starting point for implementations of that language with the aid of automatic compiler generation tools (the old Compiler Compiler dream). Again, it is clear that only a formal description can be used to generate software automatically. The formality, this time, is of a different nature: it is intended for the eyes of a computer rather than a human.

In judging the relative merits of various formalisms for language description and their applications, a number of (objective and subjective) criteria can be applied.

The first is *comprehensiveness*: depending on the ambition of the language definer, four levels of increasing comprehensiveness can be aimed at:

1. A Context-Free description (like that of ALGOL 60)
2. A Context-Sensitive description which includes also identification, typing and scoping rules (like that of ALGOL 68)
3. Semantic description by transduction to another formalism
4. Semantic description by interpretation in the same formalism (the first such description was [26], the origin of the Vienna Development Method VDM).

A given language description is *generative* if it essentially shows how to generate (correct) programs. It is termed *analytic* if it rather describes how to recognize programs. Other qualifications that come to mind are: *operational* for a description in the form of an algorithm, and *expressive* for a description that is easily readable to a human — a very subjective qualification.

Finally, *conciseness* is an objective, as long as it does not come at the expense of clarity.

2 The family of Affix Grammars

Habent sua fata libelli ...

There are no fundamental differences between Affix Grammars (AGs) and Attribute Grammars (AtGs). The two formalisms differ in origin and notation, but they are both formalizations of the same intuition: the extension of parsers with parameters. One could speak of two different schools.

The differences in background and notation that exist between AG and AtG have had as effect that the research based on them has been characterized by different concerns, but most theoretical results achieved with respect to one of them can also be applied to the other.

Affix Grammars were invented (for linguistic applications) in 1962 [19], and were formalized in 1970 [14]. They form a family of two-level grammars

$$\left\{ \begin{array}{ccc} affixes & + & domains \\ CFgrammar & + & operations \end{array} \right\}$$

where the first or lower level consists of *CF rule-schemata*, CF rules extended with meta-variables (*affixes*) and the second level defines the domains of these affixes. Included among the nonterminals of the first level are some symbols standing for *operations*. The left column in the diagram is syntactic and the right column algebraic in nature.

In choosing a particular member of the family, a number of degrees of freedom are available:

- (domain and operations) The domains of the affixes may be chosen to be (restricted) Context-Free languages, or a more machine-oriented domain may be chosen, such as the integers of some computer, or the datatypes of some programming language. In each case, the domain is accompanied by a collection of operations applied to it on the first level of the AG. Still other choices of domains are possible, e.g. functional, or lattice [22] domains.

- (power of the underlying CF grammar) The underlying CF grammar may be chosen to be deterministic (e.g. LL(1)) for easy parseability, or it may be chosen to be nondeterministic (in which case a more powerful computational model will be needed).

- (wellformedness rules) It is customary to subsume, under this name, any restrictions concerning the consistency and directionality of the semantic flow (cf. Knuth's wellformedness rules in [13]). The semantic flow may be strictly left-to-right, so that all affixes can be evaluated during parsing, or it can be quite unrestricted, implying some form of delayed evaluation.

Affix Grammars are, because of this freedom, a large family, in which many different positions can be chosen.

2.1 Affix evaluation

In parsing AGs (or AtGs), three different attribute evaluation strategies can be distinguished:

- (during parsing) Computing the attribute values on-the-fly during parsing is only possible under strong restrictions on the attribute flow [2], since at the call of a semantic function all its inherited attributes must be available.

- (static) First a parse tree is constructed (using some suitable parsing algorithm) and then it is decorated with attributes by means of an algorithm, derived from the attribute dependencies, which walks (repeatedly) over the parse tree. This multi-pass approach has been studied extensively for AtGs.

- (dynamic) During the parsing, a dependency graph of the attributes in the parse tree is constructed, over which the computation takes place. This approach has been favoured in EAGs.

Evaluating affixes (attributes) during parsing is not only more economical, but it also enables *affix directed parsing*: the affixes may be used not only to carry information from one place to another and to check consistency, but their values may also directly influence the language described. Affix directed parsing is usually not allowed in AtGs.

2.2 Deterministic Affix Grammars

Choosing the underlying CF grammar to be LL(1) (or LR(1) for that matter) and the affix-flow to be strictly from left to right causes the grammar to be evaluable with a deterministic parser — the *one-pass* or *L-attributed* AGs. This "simplest" choice, which usually goes hand in hand with the choice of machine-oriented affix domains (integers or machine-words) and operations, was tried by a number of implementors (e.g. [11] and [5]).

I made this choice myself in 1969 in defining the Compiler Description Language CDL [15], and for the next ten years [4] found myself exploring the borderline between programming languages and syntactic formalisms. Apart from numerous academic applications, CDL has been used successfully in various commercial products, such as the portable COBOL compiler produced by the German softwarehouse MBP [12], the MPROLOG system [10] from the Hungarian SzKI and the software for the Mephisto Chess computer.

The related formalism of one-pass Attribute Grammars has been studied extensively (see e.g. [2]). It has also inspired hundreds of implementations, including the ubiquitous YACC — Yet Another Compiler Compiler. In spite of the opinion of Waite [25], there is no denying that compiler generators based on such one-pass formalisms have been quite successful in the production of serious compilers.

This simple type of grammar is however quite operational, the formalisms used are procedural languages in which, rather than a language, a parser or compiler is described.

2.3 Nondeterministic Affix Grammars

In describing formal and natural languages, the issue of ambiguity comes up. Affix Grammars (and their implementations) have been more concerned with this issue than Attribute Grammars.

For the application of Affix Grammars to natural languages, I developed in 1966 the Recursive Backup parsing method [16], which can be seen as an alternative to the Warren Abstract Machine. Its central idea is the extension of parsing procedures with *continuations*. In particular, by extending the well-known Recursive Descent algorithm and the lesser known Left Corner [1] and Recursive Ascent [17] algorithms with continuations, one obtains a family of nondeterministic parsing algorithms of increasing power:

type of grammar	*type of parser*	*nondeterministic extension*
LL (1)	Recursive Descent	Nondet. Recursive Descent (NDRD) [16]
LC (1)	Left Corner	Nondet. Left Corner (NDLC) [20]
LR (1)	Recursive Ascent	Nondet. Recursive Ascent (NDRA)

The NDRD parsers can cope with any Context Free grammar which is free from leftrecursion. It deals with ambiguity by backtracking. NDLC parsers can handle leftrecursion provided it is not hidden by an intervening empty production. NDRA parsers need no such restriction.

The generation of NDRD and NDLC parsers from a grammar is simple and fast. Furthermore these parsers have excellent modularity properties. This makes them especially suited for an experimental situation, in which the grammar is frequently modified.

In the AtG school, the study and use of nondeterministic grammars has been somewhat neglected. It might benefit in this respect from the research done in the AG school.

3 Comparing formalisms

In this section we shall present a bouquet of descriptions, in three different forms of two-level grammar, of one same not quite trivial mini-language, in order to allow a comparison. The example is an extended version of an example from the dissertation of Watt [27]

3.1 The problem

We describe a small example language, just consisting of a list of declarations and assignations, by means of the Context-Free syntax (in which the terminal symbols are underlined):

```
program :
   begin, statements, end.

statements :
   statement; statements, ; , statement.

statement :
   declaration; assignation.

declaration :
   define, identifier, declarer.

declarer :
   integer; boolean; array, of, declarer.

assignation :
   variable, := , variable.

variable:
   identifier; variable, [ , variable, ] .

identifier: letter; identifier, letter.
```

which allows us to write mini-programs like

```
BEGIN
  DEFINE FLAG BOOLEAN;
  DEFINE INDEX INTEGER;
  FLAG := Z [ INDEX ];
  DEFINE Z ARRAY OF BOOLEAN
END
```

Within those mini-programs, we want to impose the following (typical) context-conditions on the use of the variables:

- (definedness) Each identifier occurring in the program has a declaration.
- (uniqueness) No identifier is declared twice.

- (identification) All applied occurrences of an identifier possess the same type, dictated by its declaration.
- (type compatibility) The variables to the left and to the right of a becomes-symbol `:=` should have the same type.
- (type constraints) Only an integer can be used as a subscript. Only an array can be subscripted. The type of a subscription is that of an element of its subscriptum.

We expressly do not demand that all applications of an identifier come after its declaration. A declaration is valid in the whole program, even in the part textually before that declaration. This precludes the use of a simple one-pass analysis and demands more sophistication from the descriptional formalism. We shall see how various formalisms cope with this complication.

On the other hand, we do not try to describe block-structure, which would only make the example more complicated without contributing new insights.

3.2 A W-grammar

The two-level Van Wijngaarden grammars (W-grammars) were introduced by Aad van Wijngaarden in 1965 [29] for the formal description of ALGOL 68. They are a "pure" form of two-level grammars, without any admixture of functions.

A W-grammar consists of a first level of Context-Free *rule-schemata*, which may include free *meta-variables*, and a second level of Context-Free *meta-rules* defining the possible values for the meta-variables. Meta-variables will be written in capital letters, their terminal productions in small letters.

For this grammar, the meta-rules will be

```
MODE :: boolean; integer; array of MODE.

ENV :: ENV IDF has MODE; EMPTY.

EMPTY :: .

IDF :: letter LET; IDF letter LET.

LET :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q;
          r; s; t; u; v; w; x; y; z.
```

In the rule-schemata comprizing the first level of the grammar, by convention all strings of small letters ending in `...symbol` are terminal symbols. Any other string of small letters may serve as a nonterminal symbol.

A meta-variable may be eliminated from a rule-schema by *consistent substitution*, replacing it consistently throughout the rule-schema by one of its terminal productions. By eliminating all meta-variables from a rule-schema in this way, a Context-Free rule is obtained. By conceptually doing this in all possible ways to all rule-schemata of the W-grammar a (probably infinite) Context-Free grammar is obtained, describing the language of the W-grammar.

This sounds more complicated (and less constructive) than it is, so let's turn to the example.

```
program:
   begin symbol, statements ENV in context ENV, end symbol.
```

The two occurrences of ENV (which lists the declarations in the program) have to be replaced by the same terminal production. Actually this may seem a weird descriptional trick, the statements occurring in an environment being defined by themselves. In the next rules (let's drop the word schema) we shall see how the environment is built out of the contributions yielded by its statements.

We introduce two synonyms of ENV, which we shall use to denote the environments built at the point before and after some construct, respectively.

```
PRE :: ENV.    POST :: ENV.
```

A meta-variable may also be indexed by a number, to indicate another such meta-variable, with the same terminal productions.

```
statements POST in context ENV:
   statements POST1 in context ENV, semicolon symbol,
        statement POST1 yielding POST in context ENV;
   statement EMPTY yielding POST in context ENV.
```

A declaration contributes to the environment, whereas an asssignation doesn't.

```
statement PRE yielding POST in context ENV:
   declaration PRE yielding POST.

statement PRE yielding PRE in context ENV:
   assignation in context ENV.
```

The two variables in an assignation have to agree in type.

```
assignation in context ENV:
   MODE variable in context ENV,
      becomes symbol,
         MODE variable in context ENV.

declaration PRE yielding PRE IDF has MODE:
   define symbol, identifier IDF, MODE declarer,
      unless IDF defined in context PRE.
```

The unless-clause is a *predicate* to avoid double declarations. Notice how we have avoided the introduction of a predicate for building environments.

```
MODE variable in context ENV1 IDF has MODE ENV2:
   identifier IDF.
```

Observe how the right definition for the identifier is plucked from the environment in one fell swoop.

```
MODE variable in context ENV:
   array of MODE variable in context ENV,
      sub symbol, integer variable in context ENV, bus symbol.

integer declarer: integer symbol.
```

```
boolean declarer: boolean symbol.

array of MODE declarer: array of symbol, MODE declarer.
```

Even the lexical structure of identifiers can be defined in the W-grammar.

```
identifier letter LET: letter LET symbol.

identifier IDF letter LET: identifier IDF, letter LET symbol.
```

Finally we define the predicate `unless...defined...` within the same formalism. This involves inequality, which makes it a lot harder.

```
unless IDF defined in context ENV IDF1 has MODE1:
   unless IDF equals IDF1, unless IDF defined in ENV.
```

Notice the use made of indexed meta-variables to circumvent the consistent substitution.

```
unless IDF defined in context EMPTY: .
```

We need two auxiliary metanotions, one providing an ordering on the alfabet of small letters

```
ALPHABET :: "abcdefghijklmnopqrstuvwxyz".
```

and the other for denoting some part (possibly empty) out of the alfabet.

```
PART :: LET PART; .
```

The inequality can now be expressed:

```
unless letter LET1 IDF1 equals letter LET2 IDF2:
   where LET1 precedes LET2 in ALPHABET;
   where LET2 precedes LET1 in ALPHABET;
   where LET1 equals LET2, unless IDF1 equals IDF2.

unless letter LET1 equals letter LET2:
   where LET1 precedes LET2 in ALPHABET;
   where LET2 precedes LET1 in ALPHABET.

unless IDF equals EMPTY: .

unless EMPTY equals IDF: .

where LET1 precedes LET2 in PART1 LET1 PART2 LET2 PART3: .

where LET equals LET: .
```

On the whole, this grammar is generative rather than analytic, only the predicates have an operational flavour. Not an inkling is given of how to construct a parser.

W-grammars are a simple self-contained formalism, that is not based on any other semantics than rewriting. It does not take recourse to functions or operations in another formal system. Van Wijngaarden's aim was to achieve a maximum of generality with a minimum of concepts, so that ultimately (in his words) "we can be silent in full generality".

Because of its fine disdain for operational details, and especially because of the possibility to make rule-schemata read like (somewhat stilted) sentences in a natural language, this is the most expressive of the three formalisms which we will discuss.

There exist no implementations of unrestricted W-grammars, but a number of researchers (e.g. Małuszyński [18]) have defined restrictions under which direct implementations are possible.

Although the original Report [30] is a rewarding document, it is not easy to read. For a solid but readable introduction into the formal description of programming languages with W-grammars, the reader is referred to [7] or [24].

It may be mentioned in passing that the generality and expressiveness of W-grammars have made them (through Metamorphosis Grammars [8]) the precursor of PROLOG and logic programming.

3.3 An Extended Affix Grammar

Extended Affix Grammars (EAGs) are a form of Affix Grammars. The domains of the affixes are string domains, described by CF meta-rules. The extension from which they derive their name consists of the fact that affix expressions (composed by concatenation out of terminal and nonterminal affixes) are allowed at every affix position. EAGs need not be L-attributed and may be nondeterministic.

As in W-grammars, all predicates can readily be expressed in the formalism itself, without recourse to other formalisms. An informal description of the formalism can be found in [21].

The meta-grammar for the EAG differs mostly in details from that of the W-grammar.

```
MODE :: "boolean"; "integer"; "array" + "of" + MODE.
```

Note the explicit concatenation operator.

```
ENV :: ENV + "(" + MODE + IDF + ")"; EMPTY.
```

For clarity and to prevent unwanted ambiguities we enclose the components of a definition between brackets.

```
EMPTY :: .

IDF :: LET; IDF + LET.

LET :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q;
       r; s; t; u; v; w; x; y; z.

PRE :: ENV.    POST :: ENV.    POST1 :: ENV.
```

The rules in an EAG are quite similar to those in a W-grammar but, in contrast to W-grammars, in EAGs all parameter positions are demarcated by the traditional brackets and comma's. This gives the notation a PROLOG-like flavour. The terminal symbols are put between quotes.

```
program:
   "begin", statements (ENV, ENV), "end".
```

Again, the environment built up in the first parameter is passed to the second parameter, but this time we can see this quite explicitly from the *directions* specified in the grammar.

The sign `>` before or after a parameter in the heading of a rule indicates the direction of that parameter: (*inherited* if written before the parameter, and *derived* or *synthetized* if written after). In EAGs, directions are optional, but in this example we shall write them in order to make the relationship with the following AtG more obvious.

```
statements (POST>, >ENV):
   statements (POST1, ENV), statement (POST1, POST, ENV);
   statement (EMPTY, POST, ENV).

statement (>PRE, POST>, >ENV):
   declaration (PRE, POST).

statement (>PRE, PRE>, >ENV):
   assignation (ENV).

assignation (>ENV):
   variable (MODE, ENV), ":=", variable (MODE, ENV).
```

Again the type-agreement is ensured by the consistent substitution rule.

```
declaration (>PRE, PRE + "(" + MODE + IDF + ")">):
   "define", identifier (IDF), declarer (MODE),
       unless defined (IDF, PRE).
```

This heading contains an affix expression at a derived position, which saves us the introduction of a predicate for extending the environment.

```
declarer ("integer">): "integer".

declarer ("boolean">): "boolean".

declarer ("array" + "of" + MODE>):
   "array", "of", declarer (MODE).

variable (MODE>, >ENV):
   identifier (IDF), where defined (IDF, MODE, ENV);
   variable ("array" + "of" + MODE, ENV),
      "[", variable ("integer", ENV), "]".
```

In this last alternative, the affix expression `"array" + "of" + MODE` at the derived parameter position of the first `variable` achieves both a check (that the variable is an array) and the selection of the type of its elements.

Finally we define (within the same formalism) the two predicates `where defined` and `unless defined`, which were made into separate predicates mainly for descriptional clarity.

```
where defined (>IDF, MODE>, >ENV1 + "(" + MODE + IDF + ")" + ENV2):  .
```

The affix expression `ENV1 + "(" + MODE + IDF + ")" + ENV2` appearing at a derived position in the heading of this rule causes a *nondeterministic split* of the environment concerned. The rule succeeds once for each occurrence of the right `IDF`. From the way in which the environment is constructed, we know it will succeed only once.

```
unless defined (>IDF, >ENV + "(" + MODE + IDF1 + ")"):
    unequal (IDF, IDF1), unless defined (IDF, ENV).
```

The built-in predicate `unequal` saves us a laborious definition of inequality of identifiers.

```
unless defined (>IDF, >"(" + MODE + IDF1 + ")"):
    unequal (IDF, IDF1).
```

We shall not attempt to describe the lexical structure of identifiers, because it is customary to consider the lexical items as terminal symbols. This concludes our description.

EAGs were invented by Watt in his dissertation [27]. There exist a number of implementations. From an EAG, a parser and a syntax-directed editor can be generated automatically. EAGs have been used with success in the description (post factum) and implementation of a number of programming languages (for more information see [21]).

3.4 An Attribute Grammar

In trying to present an AtG for this minilanguage, we are faced with the problem that there exists no single canonical notation for AtG; every author seems to invent his own. In particular, the method for introducing semantic actions (usually in some programming language) may differ widely. We will do our best to follow the notation of Alblas [3] which is close to the one used originally by Knuth [13], apart from the fact that we shall use capital letters for types, in order to prevent confusion with attribute names.

We introduce the following attributes: `mode`, which is associated with variables, `idf` to remember an identifier, `pre` and `post`, the environment before and after this construct, and `env`, which stands for the completed environment.

nonterminals: `program, statements, statement, declaration,`
`assignation, declarer, variable.`
terminals: `begin, end, ;, :=, define, identifier, integer,`
`boolean, array, of, [, ].`
start symbol: `program.`

semantic domain:

```
type MODE = (INT, BOOL, ARRAY OF (MODE));
     ENV  = (NIL, RECORD idf: IDF, t: MODE, post : ENV END);
     IDF  = STRING
```

description of attributes:

```
pre:  ENV,  inh of statement, declaration;
post: ENV,  syn of statements, statement, declaration;
env:  ENV,  inh of statements, statement, assignation, variable;
mode: MODE,  syn of declarer, variable;
idf:  STRING,  syn of identifier;
```

We also need a number of semantic functions.

semantic functions:

```
emptyenv:               --> ENV
join:      (IDF, MODE)  --> ENV
           (ENV, ENV)   --> ENV
cons: (STRING, MODE)    --> MODE
head:             MODE  --> STRING
tail:             MODE  --> MODE
```

In AtG, the attributes and semantic functions are not included in the rules themselves, but each Context-Free rule is followed by its attribute relations.

```
program --> begin statements end.
   [env of statements:= post of statements]
```

This attribuation will make the AtG two-pass.

```
statements --> statement.
   [post of statements:= post of statement;
    env of statement:= env of statements]

statements_0 --> statements_1 ; statement.
   [post of statements_0:= post of statement;
    pre of statement:= post of statements_1;
    env of statements_1:= env of statements_0;
    env of statement:= env of statements_0]

statement --> declaration.
   [post of statement:=
         join (pre of statement, post of declaration);
    pre of declaration:= pre of statement]

statement --> assignation.
   [post of statement:= emptyenv;
    env of assignation:= env of statement]

declaration --> define identifier declarer.
   [post of declaration:=
        join (mode of declarer, idf of identifier);
    IF occurs (idf of identifier, pre of declaration)
    THEN error message ("identifier defined more than once")
    FI ]

declarer --> integer.
   [mode of declarer := integer]

declarer --> boolean.
   [mode of declarer := boolean]

declarer_0 --> array of declarer.
   [mode of declarer_0 := cons ("array of", mode of declarer_1)]
```

```
assignation --> variable1 := variable2.
   [IF mode of variable1 ≠ mode of variable2
    THEN error message ("mode incompatibility in assignation")
    FI ]

variable --> identifier.
   [mode of variable:= lookup (idf of identifier, env of variable)]

variable0 --> variable1 [ variable2 ].
   [IF mode of variable2 ≠ integer
    THEN error message ("subscript must be integer")
    ELIF head (mode of variable1) = "array of"
    THEN mode of variable0:= tail (mode of variable1)
    ELSE error message ("subscriptum must be array")
    FI ]
```

We will not attempt to define `identifier` in the same formalism, the gist of the example should be clear by now.

We have to rely on a rather large number of predicates defined outside the formalism. This *open-endedness* is a mixed blessing. It is also present in the original definition of AG [14] but need not be exploited in EAG.

The separation of the CF rules from their attribuation rules makes it hard to follow the fate of a particular attribute. The passing around of attributes takes a rather large amount of writing (which may be reduced through suitable default conventions). In the EAG notation, the semantical flow is denoted much more compactly and clearly.

The text is also rather operational, due to the strict directionality of attributes in conjunction with semantical functions, even though important algorithmic parts (such as the environment lookup) have been excluded from the grammar given. Exploiting this operational character, highly efficient implementations for various classes of Attribute Grammars have been made, which perform a very deep analysis and optimization of attribute flow. This is the strong point of AtGs.

3.5 Discussion

The diligent reader has certainly observed that the three descriptions given were remarkably similar. They not only describe the same language, but they do it in the same way. In fact, a computer program could translate the EAG to an AtG and vice versa, which would allow the free exchange of EAGs and AtGs between researchers.

The purpose of this whole exercise was to drive home the point that Attribute Grammars and Affix Grammars are two different notations for the same thing. Apart from the notation, all other differences are different choices made within the same choice space.

It should be mentioned that there exists a formalisation of AtGs using a notation with the same desireable properties as EAGs (and virtually indistinguishable from them): the Extended Attribute Grammars, introduced by Watt and Madsen [28].

4 Conclusion

In this overview we have tried to highlight the differences and similarities between three different branches of the family of Two-Level Grammars. The main points can be summarized in the following table (where for comparison we have also included PROLOG, which through its extension with DCGs can also be considered to belong in the same family).

property	W-GRAMMAR	EAG	ATG	PROLOG
meta-level	CF	CF	various	not explicit
affix-directed parsing	yes	yes	no	yes
non-determinacy	yes	yes	no	yes
flow directions necessary	no	optional	yes	no
open-ended	no	optional	yes	usually
left recursion allowed	yes	yes	yes	limited
has terminal symbols	yes	yes	yes	(DCG)
logical variables	no	no	no	yes

PROLOG is not so much a grammatical formalism as a fullfledged non-deterministic programming language. In its more descriptive moods, PROLOG with DCGs looks remarkably like EAG; only the use of cut's and (asymmetric) lists imparts a strong algorithmic flavour.

Of the three grammatical formalisms, W-grammars are the most descriptive and AtG the most operational. EAGs strike a balance between the two, being potentially as expressive as W-grammar and in particular cases as efficiently implementable as the corresponding Attribute Grammars.

EAGs are just a compact and convenient notation for AtGs, particularly suited for the formal description of programming languages.

References

[1] R. op den Akker, *Parsing Attribute Grammars.* PhD thesis, University of Twente, 1988.

[2] R. op den Akker, B. Melichar and J. Tarhio, *Attribute Evaluation and Parsing.* This volume.

[3] H. Alblas, *Attribute Evaluation methods.* This volume.

[4] M. Bayer et al., Software Development in the CDL2 Laboratory. In: H. Hünke (ed.), *Software Engineering Environments*, North Holland Publ. Cy., 1981.

[5] J. Beney and J.F. Boulicaut, *Starlet: Un langage pour une programmation logique fiable.* In *Actes Séminaire CNET de Programmation Logique*, 455-482, Trégastel, May 1986. In French.

[6] R.W. Bemer, *A politico-social history of* ALGOL. Annual Review in Automatic Programming 5, 1969.

[7] J. Cleaveland and R. Uzgalis, *Grammars for Programming Languages: what every programmer should know about grammar.* Elsevier, 1975.

[8] A. Colmerauer, *Metamorphosis Grammars.* In: L. Bolc (ed.),*Natural Language Communication with computers*, Springer-Verlag, Berlin, 133-189, 1978.

[9] P. Deransart and M. Jourdan (Eds.), *Attribute Grammars and their Applications*, Lecture Notes in Computer Science 461, Springer, 1990.

[10] B. Dömölki and P. Szeredi, PROLOG *in practice.* In: R.E.A. Mason (ed.), *Information Processing 83*, North Holland Publ. Cy., 1983.

[11] D. Grune, *On the Design of* ALEPH. PhD thesis, Universiteit van Amsterdam, September 1982.

[12] I.M. Kipps, *Experience with Porting Techniques on a* COBOL74 *Compiler.* In: *Proceedings of the* SIGPLAN 82 *Symposium on Compiler Construction*, Boston, June 1982.

[13] D.E. Knuth, *Semantics of context-free languages.* Mathematical Systems Theory, 2, 127-145, February 1968.

[14] C.H.A. Koster, *Affix Grammars.* In: J.E.L. Peck (ed.), ALGOL 68 *Implementation*, 95-109. North-Holland Publishing Company, Amsterdam, 1971.

[15] C.H.A. Koster, *Using the* CDL *Compiler Compiler.* In: F.L. Bauer and J. Eickel (eds.), *Compiler Construction: An Advanced Course*, Lecture Notes in Computer Science 21, 366-426, Springer 1975.

[16] C.H.A. Koster, *A technique for parsing ambiguous grammars.* In: D. Siefkes (ed.), *GI—4.Jahrestagung*, Lecture Notes in Computer Science 26, pages 233-246, Springer, 1975.

[17] F.E.J. Kruseman Aretz, *On a recursive ascent parser.* Information Processing Letters, 29, 201-206, 1988.

[18] J. Małuszyński, *Towards a Programming Language based on the Notion of two-level Grammar.* Theoretical Computer Science, 28, 13-43, 1984.

[19] L.G.L.T. Meertens and C.H.A. Koster, *Basic English, a generative grammar for a part of English.* In *Euratom Seminar "Machine en Talen"*, Amsterdam, 1962.

[20] H. Meijer, *Programmar: A Translator Generator.* PhD thesis, Katholieke Universiteit Nijmegen, 1986.

[21] H. Meijer, *The Project on Extended Affix Grammars at Nijmegen.* In: [9].

[22] M.P.G. Moritz, *Description and Analysis of Static Semantics by Fixed-Point Equations.* PhD thesis, Katholieke Universiteit Nijmegen, 1989.

[23] P. Naur et al., *Report on the Algorithmic Language* ALGOL 60. Communications of the ACM, Vol. 6, 1-17, 1960.

[24] F.G. Pagan, *Formal Specification of Programming Languages: a Panoramic Primer.* Prentice Hall, 1981.

[25] W.M. Waite, *Use of Attribute Grammars in Compiler Construction.* In: [9].

[26] K. Walk et al., *Abstract Syntax and Interpretation of PL/1.* Technical Report TR 25.082, IBM Laboratory Vienna, June 1968.

[27] D.A. Watt, *Analysis-Oriented Two-Level Grammars.* PhD thesis, University of Glasgow, January 1974.

[28] D.A. Watt and O.L. Madsen, *Extended Attribute Grammars*. The Computer Journal 26,2, 1983.

[29] A. van Wijngaarden, *Orthogonal Design and Description of a Formal Language*. Report MR76, Mathematisch Centrum, Amsterdam, 1965.

[30] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker, (eds.), *Revised Report on the Algorithmic Language* ALGOL 68, Acta Informatica 5, 1975.