Informatics and Syntax

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

1 Informatics and Syntax

Informatics and linguistics have a common interest in syntax. This article is concerned with the historical development of the role of syntax in informatics and its relevance for linguistics.

Informatics is full of artificial languages, not only many programming languages, specification languages and mathematical notations, but also the user interfaces of innumerable program 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.

Syntax comes in whenever a concept has to be described. Syntax describes the form of things. Semantics ascribes meanings to things, depending on their form. It is natural for us to describe concepts recursively in terms of other concepts and, ultimately, in terms of some atomic concepts typical for the frame of reference in which the description is intended to be understood. Semantical description is based on syntactical description, through some homomorphism. Thus, semantics presupposes syntax.

1.1 On the Need for Formality

Let us look at an early example of an informal description: part of an old BASIC manual from a major computer manufacturer [6]:

```
EXPRESSIONS
Two types of expressions are considered by
BASIC, arithmetic and relational. Arithme-
tic expressions are rules for computing a
value. Arithmetic operators may not appear
in succession and must be stated explicit-
ly. The following are invalid arithmetic
expressions:
        x++y          (x+1)(y+2)

Expressions are evaluated from left to
right, except that the operators with
higher precedence are evaluated before
those with lower precedence. Parentheses
may be used to change the order of
evaluation. Parenthesized expressions are
evaluated first.
   . . .
OPERANDS
An operand itself is a valid expression.
```

This so-called definition raises more questions than it answers. Essentially it tries to define what an expression is by saying what it is not. It fails to state that an expression contains operands and does not forbid the use of a single operator as an expression.

The paragraph on operands is a masterpiece of ambiguity: does it define operands as expressions or expressions as operands? Are there any invalid expressions? And why the word 'itself'?

It should be pointed out that the lack of precision in this example is not a necessary and inherent problem of natural language, but mostly bad and inept usage. A better stylist would have produced a better definition, but unfortunately technical enthousiasm and linguistic ability do not often go hand in hand.

The first reason to desire formality in descriptions is that one can thus avoid verbosity and lack of clarity, and aim at consistency and completeness. To this end, the established theory of formal languages has turned out to be eminently useful. But as a bonus it has opened up the possibility of formal manipulation and automatic processing of the things described.

1.2 Context-Free Grammar as a Descriptional Aid

In the period 1958-1960, an international body of scientists attempted to define an International Algebraic Language. One of the innovative aspects of their work was the first recorded use of Context-Free (=CF) grammars in informatics. Let me quote the syntactic definition of expressions from section 3.3.1. of the Revised Report on ALGOL 60 [20].

```
<adding operator>::= + | -

<multiplying operator>::= * | /

<primary>::= <unsigned number> |
   <variable> | <function designator> |
   ( <arithmetic expression> )

<factor>::= <primary> |
   <factor> ↑ <primary>

<term>::= <factor> |
   <term><multiplying operator><factor>

<simple arithmetic expression>::=
   <term> | <adding operator><term> |
    <simple arithmetic expression>
       <adding operator><term>

<if clause>::= if <Boolean expression>
   then <arithmetic expression> else
   <simple arithmetic expression> |
   <if clause><simple arithmetic ex-
   pression> else <arithmetic
expression>
```

It is difficult to imagine today the electrifying effect that this formal syntactic definition had on the informatics of its time [3]. It was felt to be an exceedingly difficult formalism, the application of cruel formal methods to something as intimate as a programming language. It took people years to accept this style of definition as simple and straightforward, to realize that a precise definition helps in clarifying concepts.

For programmers this realization was helped by the fact that in working with CF grammars they learned to see them as programs, or rather as blueprints for programs, in their own right. Syntax-directed programming, syntax-directed parsing and syntax-directed editing were discovered.

We will follow some of the subsequent evolution of syntax in informatics.

1.3 Context-Sensitivity

Let us first concede that the notation for CF grammars used in the ALGOL 60 report, the 'Backus-Naur Form' BNF, was unwieldy, redundant and impractical. A simpler notation is obtained by replacing such curious tokens as '::= ' by the punctuation marks usually present on a typewriter, with more or less their conventional meaning, including a period to end a production rule (which eliminates some notational problems with empty productions). The sharp brackets around nonterminal symbols disappear, the terminal symbols are now enclosed in quotes.

```
arithmetic expression:
   simple arithmetic expression;
   'IF', boolean expression,
         'THEN', simple arithmetic expression,
         'ELSE', arithmetic expression.

simple arithmetic expression:
   term;
   adding operator, term;
    simple arithmetic expression, adding operator,
term.

term:
   factor;
   term, multiplying operator, factor.

factor:
   primary;
   factor, '**', primary.

primary:
   unsigned number;
   variable;
   function designator;
   '(', arithmetic expression, ')'.
```

This notation is certainly more readable than BNF, but it is still no more than a Context-Free grammar.

A Context-Free description of a programming language is imprecise in two senses:

- *lack of semantic definition*
  Syntax is not all that has to be defined, it has to be complemented by an equally formal definition of the semantics. This observation led to research on operational semantics, denotational semantics and axiomatic semantics, all closely linked to the syntax.

- *lack of context-dependency*
  The Context-Free grammar can not capture the notions of type and identification in context. The need for such context-dependencies is quite obvious in programming languages that might otherwise allow uncontrollable computations.

As an example, fig. 1 shows some context-dependencies in a program fragment. Ignoring the typing information would lead to incorrect program execution.

It would be natural to assume that the solution to this last problem lies in using Context Sensitive Phrase Structure grammars from the Chomsky hierarchy, rather than the Context Free grammars describable in BNF. As in linguistics, it was soon recognized in informatics that Context-Sensitive grammars are unintelligible for human beings, formally badly tractable and therefore practically useless.

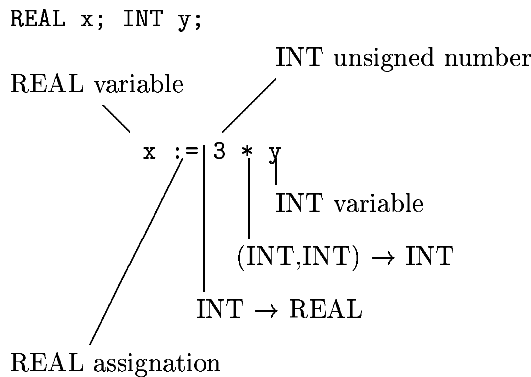

Figure 1: Context conditions

The use of transformational systems was tried in various ways (e.g. extendible languages) but again these do not form a closed and tractable formal system. Instead, the Context-Free rewriting rules were extended by various researchers with a second level.

2 Two level grammars

Two-level grammars can be seen as Context-Free grammars with an additional metasyntactic level of entities variously called affixes, metanotations or attributes, which are variables ranging over a specific domain.

As an example, we will give an Extended Affix Grammar (EAG) for a fragment of a programming language incorporating the previous example. First we define the affixes by CF metarules:

```
type :: INT; REAL; BOOL; ROW OF type.

t :: type.
```

Notice that these metarules define an unbounded domain of affix values.

The rules are parametrized with affixes in order to convey context information.

```
assignation:
   variable (type), ':=', expression (type).

expression (type):
   simple expression (type);
   'IF', expression (BOOL),
      'THEN', simple expression (type),
      'ELSE', expression (type).

simple expression (type):
   term (type);
   adding operator, term (type);
   simple expression (type), adding operator,
      term (type);
  simple expression (t), where convertible (t, type).

adding operator: '+'; '/'.

term (type):
   factor (type);
   term (type), multiplying operator, term (type);
   term (t), where convertible (t, type).

multiplying operator: '*'; '/'.

factor (type):
   primary (type);
   factor (type), '**', primary (INT).

primary (type):
   unsigned number (type);
   variable (type);
   function designator (type);
   '(', expression (type), ')'.

variable (type):
   identifier (type);
   variable (t), '[', expression (INT), ']',
       where subscripted (t, type).
```

Rewriting such parametrized production rules is constrained by the principle of *consistent substitution*, stating that all occurences of one specific affix should have the same value throughout the rule. In this way, the constituents of a construct can communicate and context conditions can be imposed.

Finally we introduce a number of *predicates*, functions expressed as epsilon rules in the grammar (containing no terminal symbols), where all the interest is in the computation on the metasyntactic level, which may involve the application of primitive *operations*, which may be defined by a mechanism outside the grammar. Predicates will succeed (i.e. produce epsilon) for some combinations of parameter values and fail for others. In this particular example two predicates are introduced, which can be defined as:

```
where convertible (INT, REAL): .

where subscripted (ROW OF type, type): .
```

It can be verified that this EAG for expressions captures all the typing information in fig. 1.

The semantics of two-level grammars are defined by a deceptively simple rewriting calculus with a solid basis in graph grammars and many-sorted algebras (see e.g. [10]). More formally, a two-level

grammar is a 7-tuple (M_n, M_t, V_n, V_t, P_m, P_v, Σ) whose components are the following:

M_n is a finite alphabet of meta-nonterminals (metanotions, attributes or affixes).

M_t is another, disjoint alphabet, whose members are called meta-terminals, attribute values or affix terminals.

V_n and V_t are two further disjoint alphabets of nonterminal symbols (including the primitive operations) and terminal symbols, respectively.

P_m is some computation scheme for computing attribute values, i.e. a mapping $M_n \mapsto M_t*$, specified in some manner which differs for the various types of two-level grammars.

P_v is another computation scheme, specifying direct productions for nonterminal symbols together with their associated attribute values, i.e. a mapping

$$V_n M_t* \mapsto (V_n M_t \cup V_t)*$$

again specified in a manner which differs for the various types of two-level grammars. Finally,

Σ is some symbol $\in V_n$, designated as the *initial symbol.*

The language defined by a two-level grammar is, as usual, the set of all terminal productions of the initial symbol.

The flow of a data through affixes during parsing can be studied, leading to the notions of *direction* (inherited and derived affixes) and *wellformedness* (ensuring e.g. decidability of the parsing problem).

As was already mentioned, there are various forms of two-level grammars:

- van Wijngaarden grammars [24, 4];
- Attribute grammars [11, 2];
- Affix grammars [12, 14];
- Extended Affix grammars [23, 17, 9];
- Metamorphosis grammars [5].

Although the various forms of two-level grammars differ in many respects (notation, domains, operations and wellformedness conditions), which determine to a large extent their usefulness as a descriptive device, they all have similar formal properties, parsing problems and implementation techniques.

2.1 Properties of Two-Level Grammars

The computational power of two-level grammars depends strongly on the domains of the second level (which may be chosen to be e.g. the strings over some alphabet, the natural numbers or even the finite integers of some computer) and on the set of primitive operations on that domain.

When choosing as domain the strings over a finite alphabet and as sole operation the concatenation (making the second level itself a CF grammar), the computational power will depend on the power of that second level. As a measuring scale we use the Chomsky hierarchy, slightly extended downwards:

Class 0	ST	Semi-Thue systems
Class 1	CS	Context-Sensitive grammars
Class 2	CF	Context-Free grammars
Class 3	FS	Finite State (Regular grammars)
Class 4	FC	Finite Choice grammars

We will use the notation $\frac{G2}{G1}$ to indicate a two-level grammar whose second level is of type G2 and whose first level is of type G1. We restrict ourselves to the case that G1 is Context-Free.

$\frac{CF}{CF}$ = Class 0 ([21])

$\frac{FS}{CF}$ = Class 0

$\frac{FC}{CF}$ = Class 2

Surprisingly, the computational power of two-level grammars with a context-free 'base' jumps from that of a context-free grammar to that of a Semi-Thue system when the second-level grammar is more powerful than a finite-choice system. It is an open question how the domain and operations should be chosen to obtain Class 1.

In spite of their Class 0 power, $\frac{CF}{CF}$ grammars have been used successfully in the definition of a number of programming languages. They have the desired generative power but are much more readable than usual Class 0 grammars. Restricted $\frac{CF}{CF}$ grammars have also quite attractive parsing properties.

3 Affix grammars over lattices

Experiences in applying Extended Affix grammars (EAG's) of type $\frac{CF}{CF}$ to the syntactic description of natural languages [1] show that most of the power of the formalism remains unused. The linguists concerned appear to restrict themselves naturally to a $\frac{FC}{CF}$ subset. In fact, such grammars have been used

extensively in classical linguistics, albeit in a non-formal way: the traditional conjugation and declination rules based on feature distinctions can be modelled immediately as a $\frac{FC}{CF}$ grammar, with such affixes as:

number:: singular; plural.

person:: first; second; third.

case:: nominative; genitive; dative; accusative; ablative; vocative.

gender:: masculine; feminine; neuter.

tense:: present; past; aorist; perfect; future.

animateness:: animate; inanimate.

and similar finite classifications.

Indeed the original motivation for Affix grammars was linguistic, and their first application was a generative grammar for a small part of English [15], which was presented to the Euratom-colloquium organized by prof. E.W. Beth in 1962.

Of course, any $\frac{FC}{CF}$ grammar may be expanded to an equivalent CF grammar, but this is not a good idea since the resulting CF grammar may be exceedingly large. The introduction of affixes in a CF grammar serves to shorten it considerably, making it possible to handle much more complicated grammars.

Parsing a sentence according to an Affix grammar can be accomplished by parsing the sentence according to the underlying CF grammar while computing the affix values either on-the-fly or after the parsing has been completed.

In order to weed out at an early stage those CF parsings that are impossible according to the affix values and context dependencies, it is desirable to compute the affix values during the CF scan. In scanning the sentence from left to right, we usually have a gradually increasing knowledge of the values of the affixes. We may take as an example the French sentence:

Je travaillais comme serveuse dans un restaurant.

It transpires half-way that the subject is feminine. In case the correspondence between the various parts of the sentence is seen as a (directed) dependency, this dependency may be from right to left as well as from left to right, as is the case for the subject and verb form in the Dutch sentences:

Ik ga naar school.
In de winter ga ik naar school.

Finally, a sentence may not uniquely specify the value of some affix, as in:

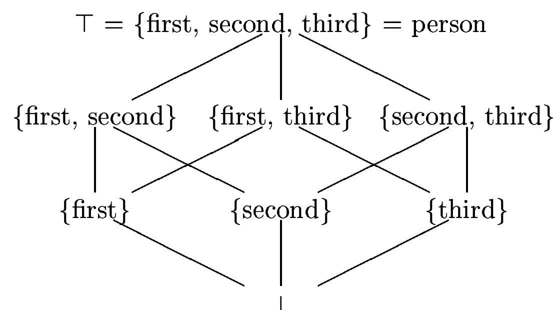

Figure 2: The lattice of values for the affix ‘person’

You are not satisfied.

in which the subject may be either singular or plural. This can be seen as a form of ambiguity (*affix-ambiguity*) which is qualitatively different from the structural ambiguity in the famous sentence:

They are flying planes.

The knowledge about the possible values of an affix can be seen as a mathematical object called a lattice. The lattice for an affix defined by

person :: first; second; third.

can be depicted as in fig. 2.

The *top*-element $\top$ of the lattice over three elements in 2 can be seen as the union of all possibilities (the value may be first or second or third or any combination). As we obtain more information, the number of possibilities may be narrowed down to any particular value, or even further to the *bottom*-element $\bot$, which indicates inconsistency.

The gradual acquisition of knowledge about an affix can be seen as a process in which a particular instance of the affix may initially have any value in its domain (represented by the set of all values in that domain). At each application of the affix, we obtain some information about its value which may serve to restrict the set of values it possesses (a simple form of *unification*).

If at any stage in the parsing process the set of values for an affix becomes empty, no consistent valuation of the affix is possible and the corresponding parsing output can be rejected.

Apart from such finite lattices, with the union and intersection as their operations, it is possible to introduce grammars over infinite lattices (see e.g. [18]). As an example, an enumeration of all verbs in a language, without any further operations, can be seen as a shallow lattice of interdeterminate size as in fig. 3. Indeed the set of sentences generated by a CF grammar may also be seen as a lattice, which need certainly not be finite.

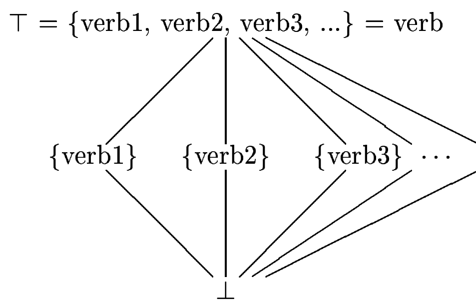

Figure 3: A shallow but infinite lattice

For many linguistic applications, Affix grammars over finite lattices (AGFL) may be adequate. Since they also admit of highly efficient implementations, they are of great practical value.

4 Parsing methods

The main advantage of the use of formal grammars over informal notations is the possibility to generate a correct parser by automatic means. Rather than laboriously writing a parsing program embodying a mixture of linguistic insights and technical considerations, one can concentrate on the linguistic aspects, leaving parser construction to a machine. A grammar written in a suitable formalism is much simpler to maintain (and extend) than a corresponding computer program.

Because of the important function of syntax in informatics, it is no wonder that in the Sixties and Seventies considerable effort went into the development of parsing methods for deterministic Context-Free grammars. Various deterministic top-down and bottom-up parsing methods were designed (LL(k), LR(k), SLR(k), LALR(k), ...), which are highly efficient in parsing very long sentences (e.g. programs) provided they are not ambiguous, as is the usual situation in informatics.

For linguistics, where ambiguity is an essential fact of life and sentences tend to be shorter, the work on nondeterministic parsers is more relevant:

- Earley's algorithm and related algorithms (for an overview see Harrison [8]).
- The Recursive Backup algorithm [13] and its Left Corner parsing variants [16].
- Multiple LR algorithms, e.g. Tomita's algorithm [22].

All of these algorithms can be generalized readily to the parsing of two level grammars.

5 Related formalisms

A number of syntactic formalisms widely used in linguistics are closely related to two-level grammars: various kinds of feature-based and unification-based grammars (e.g. GPSG [7], Augmented Transition Networks and DCG's [19]). The main difference with two-level grammars is that in those formalisms the metalevel is usually less explicit and controlled than in two-level grammars.

The differences in notation still obscure the fact that a concensus is growing about the nature and properties of syntactic formalisms for describing natural languages.

6 Afterword

The early work on formal languages and parsing methods in informatics has benefited greatly from the syntactic theories and techniques already developed in linguistics. Chomsky can be seen as one of the founding fathers of informatics. It is my conviction that linguistics can now benefit substantially from the syntactic technology developed in informatics.

The same holds for artificial intelligence: It is interesting to note that the first implementation of PROLOG by Colmerauer was based on the work he did in 1969-1970 on Metamorphosis grammars.

In informatics the interest in formal grammars for natural languages is growing, due to the present interest in expert systems. An *expert system* is a programmed system that, in important respects, behaves like an expert on some specific subject. It needs a fund of factual knowledge (in the fifth-generation jargon a *knowledge-base*), inference rules for making logical deductions from facts and also a linguistic component for input and output in natural language form, or in a form near to it.

It is in this area that informatics and linguistics will have increasing contact and, I hope, collaboration. In the coming years, linguists will have to provide relatively adequate and linguistically sound formal grammars for natural languages, suitable as a framework for attaching semantics (for me, there is no sense in discussing semantics without an adequate syntax: syntax is the backbone of semantics). This task can be best performed in direct collaboration between linguists (who provide the grammars) and informaticians (who provide the technology and the programming). The field of expert systems is too important to leave it to hobbyists.

References

[1] AARTS, J & HEUVEL, Th. VAN DE (1985), Computational tools for the syntactic analysis of corpora. In: *Linguistics* 23, 303, 335.

[2] ALBLAS, H. & MELICHAR, B. (1991), Attribute Grammars, Applications and Systems, Springer Lecture Notes in Computer Science 545.

[3] BEMER, R.W. (1969), A politico-social history of ALGOL. In: *Annual Review in Automatic Programming* 5.

[4] CLEAVELAND, J.C. & UZGALIS, R.C. (1977), Grammars for Programming Languages. *Elsevier/North Holland.*

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

[6] CONTROL DATA CORPORATION (1970), *Basic Language Reference Manual, Models 72, 73, 74 version 2, 6000 version 2.*

[7] GAZDAR, G., KLEIN, E., PULLUM, G. & SAG, I. (1985), Generalized Phrase Structure Grammar. Harvard University Press.

[8] HARRISON, M.A. (1987), Introduction to Formal Language Theory. *Addison-Wesley.*

[9] HEUVEL, Th. VAN DE et al. (1983), Extended Affix Grammars in linguistics. A Manual. *English Department, University of Nijmegen.*

[10] HOFFMANN, B. (1983), Compiler generation: From Language Definitions to Abstract Compilers. *PhD thesis, TU Berlin*; Bericht 5/83 Fachbereich Mathematik/Informatik, Universität Bremen.

[11] KNUTH, D.E (1968), Semantics of context-free languages. In: *Mathematical Systems Theory* 2(2), 127-145.

[12] KOSTER, C.H.A. (1965), On the Construction of ALGOL-Procedures for Generating, Analyzing and Translating Sentences in Natural Languages. *Report MR72, Mathematisch Centrum.*

[13] KOSTER, C.H.A. (1975), A technique for parsing ambiguous grammars, In: *GI 4. Jahrestagung*, D. Siefkes (Ed.), Springer Lecture notes in Computer Science 26.

[14] KOSTER, C.H.A. (1991), Affix Grammars for Natural Languages. In [2].

[15] MEERTENS, L.G.L.Th. & KOSTER, C.H.A. (1962), An affix grammar for an part of the English language, presented at Euratom Colloquium, University of Amsterdam, 1962, *not published* (reprints may be obtained from the authors).

[16] MEIJER, H. (1986), PROGRAMMAR a Translator Generator. *PhD thesis, University of Nijmegen.*

[17] MEIJER H. (1990), The project on Extended Affix Grammars at Nijmegen, In: *Attribute Grammars and their Applications, SLNC 461, 130-142.*

[18] MORITZ, M.P.G. (1989), Description and Analysis of Static Semantics by Fixed Point Equations. *PhD thesis, University of Nijmegen.*

[19] PEREIRA, F.C.N. & WARREN, D.H.D. (1980), Definite clause grammars for language analysis – A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13, 3, 231-278.*

[20] NAUR, P. (Ed.) (1963), Revised report on the language ALGOL 60. In: *Numerische Mathematik 4, 420-453.*

[21] SINTZOFF, M. (1967), Existence of Van Wijngaarden syntax for every recursively enumerable set. In: *Annales de la Société Scientific de Bruxelles 81*, 115-118.

[22] TOMITA, M. (1985), An efficient All-paths parsing Algorithm for Natural Language. *PhD thesis, Carnegie Mellon University.*

[23] WATT, D.A. (1977), The parsing problem for affix-grammars. In: *Acta Informatica* 8(1), 111-20.

[24] VAN WIJNGAARDEN, A., MAILLOUX, B.J.M., PECK, J.E.L. & KOSTER, C.H.A. (1969), Report on the algorithmic language ALGOL 68. In: *Numerische Mathematik* 14, 79-218.