On the borderline between grammars and programs *

C. H. A. Koster
Informatics department
University of Nijmegen
The Netherlands

J. G. Beney
Informatics department
INSA de Lyon
France

Abstract

We describe some of the engineering considerations and trade-offs in the design of a new Compiler Description Language, CDL3. The language is based on Extended Affix Grammars, where the affix rules are used to define tree types. The execution model is deterministic and depth-first, except that part of the work can be delayed until a second pass over the implicit parse tree. It is checked statically whether the program can indeed be executed in two passes without backtracking. A simple module structure allows separate compilation in a safe way.

As a running example, we treat the translation of a block-structured language.

1 Introduction

Programmers have always [17] been fascinated by the close kinship between grammars and related software, such as parsers, interpreters and compilers. This fascination led on the one hand to tools for the automatic generation of software from grammars (Compiler Compilers, Meta-compilers, Environment Generators) and on the other hand to the development of syntactic formalisms amenable to the automatic generation of software, including the various forms of two-level grammars (like W-Grammars [20], Attribute Grammars [10] and Affix Grammars [12]).

In the work on meta-compilation based on two-level grammars two different directions can be distinguished [4]:

- (*grammatical specification*) On the one hand, some researchers [15, 9] want to allow maximum freedom in the writing of grammars and therefore in the order of evaluation of the attributes. They strive for minimal well-formedness conditions that permit this evaluation, even at the price of multiple traversals of a parse tree and nondeterminism.

- (*grammatical programming*) On the other hand, the restriction to L-attributed grammars made in e.g. [11, 1, 2, 14] leads to efficient context-sensitive parsers, where the attribute evaluation can guide the parsing and without the need to build a parse

*Presented at the PLILP conference in Passau, August 1991; published in Springer Lecture Notes in Computer Science 528, p. 219-230

tree. But since there is only one deterministic left-to-right pass, in case another order of evaluation is needed the programmer must still explicitly build an abstract tree that is later used to traverse the same structure.

For describing and implementing programming languages, which is our aim, a deterministic formalism is to be preferred. But many problems (such as forward referencing) require more than one pass over the parse tree. Some attributes are to be calculated using information that comes later in the source text. For that purpose part of the attibute evaluation may have to be delayed. For many applications, a two-pass approach is sufficient: to collect information in the first pass, and use it in the second. That is why we chose to base our language on 2-pass L-attributed Affix Grammars.

Another major difference is in the choice of a second grammatical level defining the semantic objects. In W-Grammars and Affix Grammars, the domains of the meta-variables are specified by means of Context-Free meta-grammars, which also serve as a typing system. Various effective affix representations have been explored: strings [15], trees [2], sets [13] and lattices [18]. For reasons of expressivity as well as efficiency, we choose to interpret the affix domains as trees, similar to the functor structures in PROLOG, rather than as strings.

A third choice is to be made between top-down and bottom-up analysis. The construction of the table for a bottom-up parser must take the whole grammar into account, while a top-down parser is easily divided into several sub-grammars that do not need to be recompiled when only one is modified. Also a top-down parser can more readily be extended with parameters (affixes) than a table-driven one. That is why we chose to implement a two-pass variant of recursive descent parsing.

2 Grammar and program

The goal of the CDL3 project is, as the acronym suggests, to design a new Compiler Description Language. In order to combine a high degree of expressiveness and formal tractability with efficient implementation, we choose a notation based on the syntactic formalism of Extended Affix Grammars (EAGs) [19, 15] while adhering to well-understood principles of Software Engineering.

2.1 The language

A CDL3 program can be seen as a sugared version of an EAG, which is quite similar to PROLOG extended with a meta-grammar and without the logical variable, all affix values being ground. We shall describe the notation only superficially, since we are mainly interested in bringing out the design decisions; details and examples follow in later sections.

A program consists of meta-rules (see 2.3), which serve as *type-declarations*, and syntax rules playing the role of *procedures*. The order of these declarations is free.

A procedure can have affixes as *input*- (= inherited) and *output*- (= synthesized) parameters, as indicated by the *direction* signs > in its heading, e.g.:

```
ACTION p (>INPAR, OUTPAR>):
```

Procedures may be *generic* in the sense that any number of procedures can have the same name provided they differ in name and/or type of parameters.

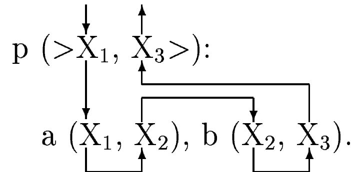
A CDL3-program is L-attributed: the *flow-of-information* within a procedure body is strictly left-to-right. No information can be passed from one (failed) alternative to the next.

The *control-structures*, which are of syntactic origin, are those which have been made familiar by PROLOG: The semicolon as a nondeterministic OR-operator between alternatives and the comma as a MacCarthy AND-operator within an alternative. Right-recursion is used to express repetition. The execution is deterministic and sequential (but an alternative can have a *second pass*, see 4).

An alternative may end in an *enclosed group*, a group of alternatives between brackets, that can be seen as an anonymous procedure. This allows procedure bodies to be left-factored.

A procedure may end with one or more `WHERE`-clauses, which serve as *refinements*. The refinements of a procedure are implicitly parametrized with the same parameters as the procedure itself. This convention eliminates some stereotypical parameters.

Another kind of stereotypical parameters can be seen in examples like



As a shorthand for such pairs of parameters, winding their way through the calls, we also admit *transient* parameters, written

```
p (>X>): a(X), b(X).
```

At a transient parameter position we also allow a *pair of affixes* separated by a `|`-sign (a special kind of comma). As an example, the call `p(0|OFFSET)` calls `p` with `0` as input parameter and `OFFSET` as output parameter.

2.2 Grammatical programming

By these notational conventions (and a few more pragmatic decisions, e.g. concerning built-in operations), the grammatical formalism has turned into a programming language. The programming style resulting from the abovementioned control structures gives meaning to the hackneyed term "structured programming". The refinements and procedures support Top-Down programming. In conjunction with a simple module system (see 5), the generic procedures support Bottom-Up programming.

In fact, various researchers have introduced the notion of *grammatical programming* [3, 8] to describe programming based on syntactic structure.

Grammatical programming can be seen as the exploitation of some homomorphism: between syntax and input, (abstract) syntax and output, input and output (the Jackson method), input and computation (interpreters) or even between different computations and one same datastructure. A formal basis for this observation can be found in [7].

2.3 The meta-level

The meta-rules of a CDL3-program are CF rules, with affixes (written in capital letters) as the meta-nonterminals and small-letter words as meta-terminals. They serve to define

datastructures and the domains of the affixes.

In W-Grammars and EAGs, the values of the attributes are considered as strings over the meta-alphabet which are concatenated. This approach leads to concise formulations, but any ambiguity in the meta-grammar will necessitate backtracking.

A way to avoid this backtracking as well as some dynamic patern-matching is to represent such meta-values as trees, that is to keep track of the way those values were built, and to require that each meta-alternative is marked by a different terminal affix.

To be more precise, the meta-rules of the CDL3-program form a Context-Free Grammar in (near) Greibach Normal form, every alternative (except maybe for the last one) starting with a distinct terminal symbol (its *marker*). Only the last meta-alternative is allowed to have no marker.

As an example, we describe the environment structure for a simple block-structured language by the following meta-rules:

```
RANGE:: empty; RANGE DEF.
```

A `RANGE` is in fact a linear list of `DEF`s. There is no predefined list-type or tree-type.

```
DEF:: where IDF has OFFSET in BNO.
```

In the example language to be described, the affixes `BNO` and `OFFSET` will denote a static block number and the offset of a variable within that block, respectively.

```
BNO, OFFSET:: INT.
```

The affixes `BNO` and `OFFSET` are introduced as synonyms for `INT`. Another way to introduce a synonym for an affix is to index it with an integer — so `OFFSET1` is also a synonym for `OFFSET`.

```
IDF:: TEXT.
```

The types `INT` and `TEXT` are the only predefined types.

Trees can be built and decomposed by means of *guards*. A guard is a confrontation between an affix and one of its productions (an *affix expression*), or between two affixes with the same domain. The guards serve as elementary actions and tests.

Four kinds of guards can be distinguished:

`[BNO = BNO1]`	*equality* of (ground) affix values
`[RANGE =: RANGE DEF]`	*split* combined with conformity test
`[where IDF has OFFSET in BNO =: DEF]`	*join*
`[RANGE =: RANGE1]`	*assignment* to `RANGE1`

As a further extension, we allow affix expressions to occur at both inherited and derived positions, standing for implicit guards. Letting `E` stand for some affix expression, `A` for an affix of which it is a production and `p` for a procedure having one `A` parameter, in a procedure heading

- `p(>E):` is interpreted as `p(>A):` with `[A =: E]` inserted at the beginning of every alternative of `p`, and
- `p(E>):` as `p(A>):` with `[E =: A]` inserted at the end of every alternative.

Similarly, in a procedure call

- `p(E)` is interpreted as `[E =: A], p(A)` for an input-parameter, and

- `p(E)` as `p(A), [A =: E]` for an output-parameter.

The same sugaring conventions apply to the components of a transient parameter. By these conventions, most guards may remain implicit.

3 Static Semantic Checks

We want to distinguish statically between *effects* of the execution of a program, i.e. desired side effects, and *defects*, i.e. spurious side effects. To that end we consider the program as a grammar and require it to be LL(1) (and therefore deterministic).

Each procedure is classified according to two binary criteria:

- (has it an effect) Does an execution of the procedure cause a change in state (of the input, the output or the variables global to the procedure)?
- (does it yield a control value) Can the execution of the procedure fail (and thereby influence the flow of control) or does it always succeed?

The programmer specifies a type for each procedure, as follows:

`PREDICATE`	has effect, may fail
`ACTION`	has effect, always succeeds
`TEST`	no effect, may fail
`FUNCTION`	no effect, always succeeds.

This specification is checked on the one hand against the body of the procedure, and on the other hand against the use made of the procedure in the program.

Consider a production rule of a CF grammar satisfying the LL(1) restriction

$$r: \; m_1, \; m_2, ..., \; m_n; \; \textit{other alternatives}.$$

and an associated recursive descent recognition procedure

`PREDICATE r:` $\mathtt{m}_1$`,` $\mathtt{m}_2$`, ...,` $\mathtt{m}_n$`; other alternatives.`

where each of the members m_i of the CF grammar similarly has a corresponding recognition procedure $\mathtt{m}_i$. Since the grammar satisfies the LL(1) restriction, none of the *other alternatives* has a starter in common with $m_1, m_2, ..., m_n$. Assume none of the m_i produces empty (all of the $\mathtt{m}_i$ are predicates, in the sense given above).

The predicate $\mathtt{m}_1$ can only succeed upon recognizing one of the starters of $\mathtt{m}_1$. Consequently, successful return from $\mathtt{m}_1$ indicates that none of the `other alternatives` could have succeeded — even if one of $\mathtt{m}_2$, ..., $\mathtt{m}_n$ fails. There is therefore no point in backtracking to the `other alternatives` if $\mathtt{m}_1$ succeeds but $\mathtt{m}_2$, ..., $\mathtt{m}_n$ fails. We will therefore insist (and verify) that $\mathtt{m}_2$, ..., $\mathtt{m}_n$ must succeed. Any later failure is forbidden.

More generally, an alternative may have a ‘point of no return’, viz. the first member which is a predicate or action; before this point there can be any number of tests or functions, after it only functions or actions. Any failure after this point leads to a *defect* — a side effect upon failure. A defect can be removed by backtracking explicitly, or by the introduction of an *error alternative*, as in

```
PRED is parameter list(PLIST>):
   is open symbol,
     ( is parameter list(PLIST),
         ( is close symbol;
           error("missing close symbol"))
       [nil =: PLIST]).
```

In CDL3, defects are forbidden!

A defect corresponds to a place where the program might have to backtrack. A program without defects will never need to backtrack.

Experience from CDL2 [6] shows that this semantic consistency check (together with a number of checks based on the directions of parameters) is very effective both in catching deep errors and in imposing a rigorous discipline on the use of side effects.

3.1 Example: a simple language

We want to recognize a simple block-structured language with the following CF syntax:

```
program: block.
block: "begin", units, "end".
units: unit, units; unit.
unit: application; definition; block.
application: "apply", identifier.
definition: "define", identifier.
```

By turning the rules of this grammar into procedures and suitably extending them with affixes we obtain a recursive descent parser, which builds for each block a local symbol table (`RANGE`) as a result of parsing.

```
R:: RANGE.
ACTION program:
  block(0).
ACT block(>BNO):
  should be token("begin"),
    units(BNO, 0|OFFSET, empty|RANGE),
      should be token("end").
ACTION units(>BNO, >OFFSET>, >R>):
  application, units tail;
  definition(BNO, OFFSET, R), units tail;
  block(BNO+1), units tail
WHERE units tail:
  is token(";"), units(BNO, OFFSET, R); +.
```

The refinement `units tail` is implicitly parametrized with the affixes `BNO`, `OFFSET` and `R` of the procedure `units`. The `+` indicates the empty alternative (which always succeeds) and `-` will be used to indicate an alternative that always fails.

```
PRED application:
  is token("apply"),
    ( is identifier(IDF);
      error("identifier expected")).
```

```
PRED definition(>BNO, >OFFSET|OFFSET+1>, >R>):
  is token("define"),
    ( is identifier(IDF), [R where IDF has OFFSET in BNO =: R];
      error("identifier expected")).
```

4 The second pass

An important property of Attribute Grammars (and Affix Grammars) is the freedom they give in the order of evaluation of components and their attributes, a freedom however which may have to be bought at the expense of a rather expensive mechanism for multipass or delayed evaluation. All kinds of subclasses (n-pass, n-visit, etc.) have been introduced to allow more efficient evaluation models.

In CDL3 we have chosen to support two-pass evaluation. Some actions from an alternative can be delayed until a second pass over the execution tree (the *second-pass elements*). These actions are meant to evaluate and use attributes that could not be known before the end of the first pass (the *second-pass parameters*). After the first pass is completed, the second pass of each successful procedure is executed, starting at the root and proceeding in *postfix* order: the second passes of all children are executed in textual order before the second pass of the parent. The second pass of a procedure can access both its first- and second-pass parameters, as well as the second pass output-parameters of its children.

The second pass shows its usefulness in many compilation problems where it obviates the need for an explicit abstract tree.

This execution model can also be seen as a form of partial evaluation of the parse tree, where the second pass builds the subtrees that were left unevaluated during the first pass. Since such a delayed subtree might itself have a second pass, this notion could be applied recursively, but we shall not pursue this possibility here.

Of course some compiler tasks imply a form of comparison of subtrees. In this case a data structure must be built that represents the relevant abstract tree. However, many interesting transductions can be described without introducing such an intermediate data structure.

More complicated transductions can be achieved by composing two-pass transducers using piping.

4.1 Adding context conditions ...

We now want to introduce checks on the context conditions, for which the information is available on completion of the first pass. To this end, some of the procedures are extended with a second-pass affix `ENV` denoting an environment composed of ranges. second-pass elements.

```
ENV:: conc ENV RANGE; only RANGE.
```

The second pass elements are separated from the first-pass elements by the sign `/`. The additions are shown in bold face.

```
ACTION program:
  block(0 / empty).
```

The outermost block stands an empty environment.

```
ACTION block(>BNO / >ENV):
  should be token("begin"),
    units(BNO, 0|OFFSET, empty|RANGE / conc ENV RANGE),
      should be token("end").
```

Notice that the units of a block stand in an environment composed in the second pass out of the surrounding environment and the declarations belonging to that block.

```
ACTION units(>BNO, >OFFSET>, >R> / >ENV):
  application (/ ENV), units tail;
  definition(BNO+1, OFFSET, R), units tail;
  block(BNO+1 / ENV), units tail
WHERE units tail:
  is token(";"), units(BNO, OFFSET, R / ENV); + .
PRED application( / >ENV):
  is token("apply"),
    ( is identifier(IDF),
        ( found(IDF, ENV, OFFSET, BNO);
          error("missing definition for "+IDF);
      error("identifier expected")).
PRED definition(>BNO, >OFFSET|OFFSET+1, >R>):
  is token("define"),
    ( is identifier(IDF) ,
        ( found(IDF, R, OFFSET1, BNO1),
            error("multiple definition for "+IDF);
          [R where IDF has OFFSET in BNO =:R] );
      error("identifier expected")).
TEST found(>IDF, >conc ENV RANGE, OFFSET>, BNO>):
  found(IDF, RANGE, OFFSET, BNO);
  found(IDF, ENV, OFFSET, BNO).
TEST found(>IDF, >only RANGE, OFFSET>, BNO>):
  found(IDF, RANGE, OFFSET, BNO).
TEST found(>IDF, >RANGE DEF, OFFSET>, BNO>):
  [DEF =: where IDF has OFFSET in BNO];
  found(IDF, RANGE, OFFSET, BNO) ).
TEST found(>IDF, >empty, OFFSET>, BNO>): - .
```

4.2 ... and code generation

Finally we shall introduce code generation, in the form of a transduction of the input to a sequence of abstract instructions of the form

reserve bn, size	reserve size words for block no bn
apply bn, offs	apply the identifier with offset offs and block no bn
unreserve bn, size	free the storage of this block.

To that end we introduce metarules describing the output

```
CODE:: empty; CODE INSTR.

INSTR:: reserve BNO OFFSET; apply BNO OFFSET; unreserve BNO OFFSET.
```

We extend some of the previous procedures.

```
ACTION program:
  block(0/ empty , empty| CODE).

ACTION block(>BNO/ >ENV , >CODE| CODE unreserve BNO OFFSET>):
  should be token("begin"),
    units(BNO, 0|OFFSET, empty|RANGE/ conc ENV RANGE ,
       CODE reserve BNO OFFSET| CODE),
     should be token("end").

ACTION units(>BNO, >OFFSET>, >R>/ >ENV , >CODE>):
  application(/ ENV , CODE), units tail;
  definition(BNO+1, OFFSET, R), units tail;
  block(BNO+1/ ENV , CODE), units tail
WHERE units tail:
  is token(";"), units(BNO, OFFSET, R/ ENV , CODE); + .

PRED application(/ >ENV , >CODE>):
  is token("apply"),
    ( is identifier(IDF),
        ( found(IDF, R, OFFSET, BNO) ,
           [CODE apply BNO OFFSET =: CODE];
          error("missing definition for "+IDF);
     error("identifier expected")).
```

In a number of simple steps we have achieved the required transduction, performing the syntax analysis, the checking of static semantics and the generation of a nontrivial translation in two passes, without explicitly constructing an abstract tree as an intermediary.

5 Modules

A major concern in writing large grammars (as in writing large programs) is the large number of attributes to be passed explicitly from one part of the grammar to others. This communication overhead, resulting from the explicit applicative character of grammars, has to be overcome in order to obtain a practical notation for programming. Uncritically introducing a notation for global variables on top of the grammatical formalism is unsatisfactory, both from a formal viewpoint and from Software Engineering considerations.

In CDL3, global variables can only be introduced in separate modules. A *module* is considered as an abstract data type that has precisely one instance. Since there is only one instance of it, the module need never be passed explicitly as a parameter.

A module consists of one structured object (an affix expression comprizing all its global variables) and the definitions of procedures to handle this object. The global variables are invisible transient parameters to all the procedures exported by the module and recursively of the procedures that call those procedures. Their introduction again serves to get rid of stereotypical parameters, and gain some security in the bargain.

5.1 Preludes and postludes

A module may have two special procedures, the *prelude* and the *postlude* of the module, which will be called respectively at the begining and at the end of the execution of the program.

The introduction of preludes and postludes allows to factor out the initializations and finalizations from the program. The programmer need not worry globally about propagating and calling initializations and finalizations, and can concentrate on the main algorithm (*separation of concerns*).

5.2 Example of a module

As a simple example of a module, just to show the notation used, we will provide lexical analysis procedures for the previous program.

```
MODULE lexico = buffered LINE.
```

The module has one global variable, named `LINE`.

```
DEFINES IDF,
        PRED is token(>TEXT),
        PRED should be token(>TEXT),
        PRED is identifier(IDF>),
        TEST is end of source.
```

The interface specifies all exported procedures and types, as well as the modules which will be used (none, in this case).

```
LINE, IDF:: TEXT.

PRELUDE read buffer.
ACTION read buffer:
  read line(LINE).
TEST is end of source:
  [LINE =: ""] .
PRED is token(>TEXT):
  is prefix(TEXT, LINE), layout.
```

The test `is prefix (>TEXT1, >TEXT2>)` is a built-in text operation.

```
ACTION should be token(>TEXT):
  is prefix(TEXT, LINE), layout;
  error(TEXT + " expected).
ACTION layout:
  is prefix(" ", LINE), layout;
  is prefix("\n", LINE), read buffer, layout;
  +.

L, D, T:: TEXT.
PRED is identifier(L+T>):
  is letter(L), rest identifier(T), layout.
PRED is letter(L>):
    split(LINE, L, LINE1), before("a", L), before(L, "z"),
      [LINE1 =: LINE].
```

The predefined `FUNCTION split(>TEXT, CHAR>, TEXT1>)` splits the first character off a text.

```
  PRED is digit(D>):
      split(LINE, D, LINE1), before("0", D), before(D, "9"),
        [LINE1 =: LINE].
```

6 Implementation status

A compiler for CDL3 has been implemented by J. Beney, generating C code which is compatible with most C compilers on PC's and UNIX workstations. Work still has to be invested in optimization and support tools. In the present state, the compiler is efficient enough to compile itself in a reasonable time on a PC.

7 Conclusion

Two principles underlie the design of CDL3: the exploitation of the borderline between grammars and programs, and the exploitation of the benefits of voluntary restrictions.

By the first principle, the advantages of the interpretation of programs as grammars (static semantic checks, the defect-philosophy) are combined with Software Engineering considerations (support of structured programming, modular programming, preludes and postludes, as well as the use of special tools and environments).

According to the second principle, we strive neither for maximal efficiency nor for maximal generality, but aim for expressiveness of notation and checkability of properties of programs combined with quite good efficiency, at the price of small and well-understood restrictions in generality (LL(1) base grammar, second level in Greibach Normal Form). Disregarding these restrictions, a CDL3 program can easily be embedded in a richer formalism (such as PROLOG or EAGs), mainly by turning `|` and `/` into comma's and eliminating some syntactic sugar. But then we loose the advantages afforded by the restrictions, in particular checkability.

We hope to have struck the right balance, resulting in a compact notation for the efficient development of secure syntax-based transducers, such as compilers, interpreters and man-machine interfaces.

References

[1] J. G. Beney, L. Frecon, *Langage et Systeme d'Ecriture de Traducteurs*, RAIRO Informatique Vol. 14, No. 4, pp 379-394, 1980.

[2] J. G. Beney, J. F. Boulicaut, *An Affix-Based Compiler Compiler designed as a Logic Programming System*, Proceedings of the CC'90, Schwerin, October 1990.

[3] P. Y. Cunin, M. Griffiths, J. Voiron *Comprendre la Compilation*, Lecture Notes in Computer Science, Springer-Verlag, 1980.

[4] P. Deransart, M. Jourdan, B. Lohro, *A survey on Attribute Grammars*, Lecture Notes in Computer Science 323, Springer-Verlag, 1988.

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

[6] H. Feuerhahn, C. H. A. Koster, *Static Semantic Checks in an Openended Language*, In: P. G. Hibbard and S. A. Schuman (Eds.), *Constructing Quality Software*, North Holland Publ. Cy., 1978.

[7] M. Fokkinga, J. Jeuring, L. Meertens, E. Meijer, *Translation of Attribute Grammars into Catamorphisms*, to appear in The Squiggolist, 1991.

[8] E. C. R. Hehner, S. A. Silverberg, *Programming with Grammars: an Exercise in Methodology-directed Language Design*, the Computer Journal, Vol 26 no 3, 1983.

[9] M. Jourdan, D. Parigot, *The FNC-2 System: Advances in Attribute Grammar Technology*, RR-834 INRIA April 1988.

[10] D. E, Knuth, *Semantics of Context-Free Languages*, Mathematical System Theory, Vol. 2, No. 2, pp 127-145, june 1968 and Vol. 5, pp 95-96, may 1971.

[11] C. H. A. Koster, *A Compiler Compiler*, Report MR127/71, Mathematical Centre, Amsterdam, 1971.

[12] C. H. A. Koster, *Affix Grammars*, In: J. E. L. Peck(Ed.), ALGOL 68 Implementation, North-Holland Publishing Co., Amsterdam 1971.

[13] C. H. A. Koster, *Affix Grammars for Natural Languages*, Proceedings Summer School on Attribute Grammars and their Applications, Prague, June 1991.

[14] J. Lewi, K. De Vlaminck, J. Huens, M. Huybrecht, *Project LILA User's Manual*, Report CW7, Applied Mathematics and Programming Division, Katholieke Universiteit Leuven, 1977.

[15] H. Meijer, *PROGRAMMAR: a Translator Generator*, Ph.D. Thesis, University of Nijmegen, 1986.

[16] H. Meijer, *The Project on Extended Affix Grammars at Nijmegen*, In: [5].

[17] H. H. Metcalfe, *A Parameterized Compiler based on Mechanical Linguistics*, Planning Research Corporation, Los Angeles, Calif. 1963.

[18] M. P. G. Moritz, *Description and Analysis of Static Semantics by Fixed-Point Equations*, Ph.d. Thesis, University of Nijmegen, 1989.

[19] D. A. Watt, *Analysis-oriented two-level Grammars*, Ph.D. thesis, University of Glasgow, 1974.

[20] A. van Wijngaarden et al. (Eds.), *Revised Report on the Algorithmic Language ALGOL68*, Acta Informatica 5, pp. 1–236, 1975.