| | |
|---|---|
| Group | : Computational Linguistics |
| Topic | : formalism |

| | |
|---|---|
| Title | : **Compiler Generator** |
| Author | : René Leermakers |

| | |
|---|---|
| Doc.Nr. | : 0167 |
| Date | : 17-12-86 |
| Status | : concept |
| Supersedes | : - |
| Distribution | : Software Group |
| Clearance | : Project |
| Keywords | : compiler, M-rules |


Institute for Perception Research

# Contents

# 1 Introduction

This document describes a compiler generator, i.e. a program that converts an attribute grammar that defines some language, into a compiler for this language. The ideas for this generator developed out of the fear and awe that crept into me, as I contemplated the task of building a compiler for Rosetta Mrules. The main advantage of a compiler generator is a superior maintainability of the compilers generated by it, in addition to a complete documentation. The main advantage of the present generator, compared to others, is probably the amount of freedom in writing the basis grammar. The idea is that in this way syntax can more easily be oriented towards semantics.

In section 2, I will describe the attribute grammar. It is a grammar in the spirit of the Rosetta Surface Grammar, enriched with inherited attributes and a new kind of synthesized attributes. Parameters are used as in the surface grammar but there are now two kinds of them. Section 3 is devoted to the subject of attribute evaluation, and section 4 contains some conclusions and comments. In a subsequent document I will present the current syntax for writing the attribute grammar and the actual implementation of the compiler generator.

# 2 Attribute Grammar

## 2.1 Some Remarks

The present attribute grammar could be called a procedural attribute grammar, as the concepts of a procedures and processes play a dominant role, rather than that of a function. Note that a procedure can be seen as a state space transformer. As a state is a function from the set of identifiers to their values, a transformation in state space is a function from a function space to itself. Thus, rather than a filthy concept, a procedure is a specification of higher-order functions.

Given two arbitrary sets of identifiers $A$ and $B$, I will write

$$pr : A \rightarrow B$$

to mean that pr is a state transformer that leaves invariant all values of the identifiers that are not in B, and that changes the values of the identifiers in B in terms of the values of the identifiers in A before the transformation.

Attributes are thought of as identifiers inside the processes that evaluate them. The specification of such a process for each node in a syntax tree, is part of the definition of attribute evaluation, and thus of the compiler generator, rather than being an implementation of it.

## 2.2 Basis Grammar

The basis of the attribute grammar is a simple rewrite system $G =< N, T, P >$, where $N$ is the set of non-terminals of which $UTT$ (utterance) is the start-symbol, $T$ the set of terminals, and $P$ the set of production rules $P_n = < n, R_n, S_n >$. A production rule is commonly written as $n \rightarrow R_n$ with neglect of $S_n$, which we call the set of categorial symbols. In $P_n$, $n \in N$, and for each $n \in N$ there is exactly one $P_n \in P$, and hence one $R_n$. $R_n$ is a regular expression of categorial symbols. Each symbol $s \in S_n$ occurs once and only once in it. To each symbol $s \in S_n$ corresponds a category, i.e. a terminal or a non-terminal, which we denote by $\hat{s}$. The regular expression $R_n$ is said to define $n$. $UTT$ does not correspond to any symbol, in order to avoid recursive definitions of $UTT$:

$$\exists_n s \in S_n \rightarrow \hat{s} \in N \cup T / \{UTT\}$$

Apart from the one-to-one correspondence between production rules and non-terminals, the rewrite system G is not restricted. Hence, any context-free, possibly ambiguous, language can be defined by a suitable G.

A sentence is part of the language if at least one syntax tree exists, with the terminals of the sentence as its leaves. A syntax tree t is defined as

- $t =< n, s, t_1, .., t_n >$, where
  1. the fields of t are often referred to as $t.n$, $t.s$, $t.t_i$
  2. $s$ is a symbol $s \in S_n$ and ($n = \hat{s}$ or $n = UTT$)
  3. $k > 0$ iff $n \in N$ (i.e. $t$ is a leaf if $t.n \in T$)
  4. $t_i$ are syntax trees.
  5. $t_1.s...t_n.s$ in L($R_{t.n}$), the language of symbol sequences defined by the regular expression $R_{t.n}$

## 2.3 Surface Attributes and Parameters

To each production rule $P_n$ corresponds a set of surface parameters $SP_n$ and to each $n \in N \cup T$ a set of surface attributes $SA_n$. To each symbol $s \in S_n$ for some n, also corresponds a set of surface attributes $SA_s$, which is a copy of $SA_{\hat{s}}$. Surface attributes are synthesized only.

To each symbol $s \in S_n$ corresponds a procedure

$$pre_{n,s} : SP_n \cup SA_s \rightarrow \{status\},$$

where *status* is a predefined identifier with boolean values.

To each symbol $s \in S_n$ also corresponds a procedure $act_{n,s}$ that transforms the values of the elements of $SP_n$, which transformation depends solely on the values of the

parameters in $SP_n$ before the transformation and on the values of the elements of $SA_s$:

$$act_{n,s} : SP_n \cup SA_s \rightarrow SP_n.$$

To each production rule $P_n$ and hence to each $n \in N$, as there is a one-to-one correspondence between the two, correspond a status changing procedure $pre_n$, a parameter initialization procedure $Sinit_n$, and a procedure $act_n$ that gives all values of $AS_n$ a value:

$$pre_n : SP_n \rightarrow \{status\}$$
$$Sinit_n : \emptyset \rightarrow SP_n$$
$$act_n : SP_n \rightarrow SA_n.$$

### 2.4 Procedural Attributes and Parameters

To each production rule $P_n$ corresponds a set of procedural parameters $PP_n$ and to each $n \in N \cup T$ two sets of procedural attributes $PI_n$ and $PS_n$, which are of the inherited and synthesized types, respectively. The sets $PI_n$ and $PS_n$ are disjunct, i.e. a procedural attribute is either inherited or synthesized. The set $PI_{UTT}$ is always empty. Again, to each categorial symbol $s$ also correspond sets of attributes $PI_s$ and $PS_s$, which are copies of $PI_{\hat{s}}$ and $PS_{\hat{s}}$, respectively.

To each symbol $s \in S_n$ also correspond two procedures $inh_{n,s}$ and $syn_{n,s}$:

$$inh_{n,s} : PP_n \rightarrow PI_s$$
$$syn_{n,s} : PP_n \cup PS_s \cup SA_s \rightarrow PP_n.$$

Note that the procedure $syn_{n,s}$ is very similar to $act_{n,s}$, and changes the value of procedural parameters, whereas $inh_{n,s}$ assigns values to all attributes in $PI_s$.

To each production rule $P_n$ and hence to each $n \in N$, correspond two procedures $Pinit_n$ and $syn_n$. The procedure $Pinit_n$ initializes the parameters $PP_n$ in terms of the elements of $PI_n$ and $SA_n$ whereas $syn_n$ assigns values to all elements of $PS_n$ in terms of the elements of $PP_n$:

$$Pinit_n : PI_n \cup SA_n \rightarrow PP_n$$
$$syn_n : PP_n \rightarrow PS_n$$

## 3 Atribute Evaluation

In this section we will specify the order in which the attributes are to be evaluated for a given syntax tree. As was mentioned before, this will be done by defining a process for each tree node. The concurrent execution of all such processes defines the correct attribute evaluation.

To each node in the syntax tree we assign a process, with communication channels to and from the processes of its neighbouring nodes, i.e. to its father (if present) and its sons (if present). These channels are named by the relevant categorial symbol. These names are unique as long as $S_n \cap S_m = \emptyset$ if $n \neq m$. The order of evaluation is such that for each node in a syntax tree firstly the surface attributes are evaluated, then the inherited procedural attributes, and lastly the synthesized procedural attributes.

The dependencies of the various attribute evaluations is such that the attribute evaluation for a given tree takes place in three passes across the tree, starting at the leaves of the syntax tree, going up to the top, down to the leaves again, and once more up to the top.

It would be straightforward to increase the number of passes arbitrarily, but in practise three passes is often enough. In fact, I think it might even be wise to force the writer of the attribute grammar to write a three-pass grammar. For one, it makes the grammar as a whole more readable, whereas also the efficiency of the attribute evaluation is increased.

A process at a node $t$ with $t.n = n$ looks like

BEGIN
IF #sons(t) $\geq 1$ THEN
  $Sinit_n(\mathcal{SP}_n)$;
  FOR i=1 TO #sons(t)
    DO
    $s := t.t_i.s$;
    $s?SA_s$;
    $pre_{n,s}(SA_s, SP_n, status)$;
    $act_{n,s}(SA_s, \mathcal{SP}_n)$
    OD;
  $pre_n(SP_n, status)$;
  $act_n(SP_n, \mathcal{SA}_n)$
ELSE $terminalattributes$(t,$\mathcal{SA}_n$)
FI;
$t.s!SA_n$;
$t.s?PI_n$;
$Pinit_n(SA_n, PI_n, \mathcal{PP}_n)$;
FOR i=1 TO #sons(t)
  DO
  $s := t.t_i.s$;
  $inh_{n,s}(PP_n, \mathcal{PI}_s)$;
  $s!PI_s$;
  $s?PS_s$;
  $syn_{n,s}(PP_n, PS_s, SA_s, \mathcal{PP}_n)$
  OD;

$syn_n(PP_n,\mathcal{PS}_n)$;
$t.s!PS_n$
END;

The out parameters of procedures have been written with calligraphic letters. The procedure *terminalattributes* gives values to the surface attributes of terminals. The exclamation mark signals the sending of a message through the channel mentioned before it, and the question mark signals the waiting for a message on the mentioned channel. For convenience, we have assumed that the process at the top of the tree has a parent process doing nothing but accepting the messages from the UTT process containing synthesized attributes and sending an empty message to the UTT process to give $PI_{UTT}(=\emptyset)$ a "value".

In the above process definition, whenever the variable *status* becomes false, the attribute evaluation is assumed to stop. In this way, syntax trees can be rejected during the first pass, the surface attribute evaluation. Surface attributes can thus be used to single out one syntax tree in cases where a sentence leads to more than one parse. It is assumed that only one syntax tree survived the first pass. Of course, this tree could still be rejected 'by hand' by introducing some boolean attribute in $PS_{UTT}$, signalling the correctness.

In the current implementation (in pascal), surface attributes are calculated in the scanner (terminal attributes) and during the syntax tree construction, and procedural attributes are evaluated by procedures rather than processes (hence their name), with communication channels implemented as in and out parameters of these procedures.

# 4 Discussion

We have presented a specific kind of attribute grammar, to be used to specify a compiler. The writing of the grammar comes down to:

- Define the basis grammar, i.e. the sets N, T, and the production rules, in particular its regular expressions.
- Define, together with their types, the sets of attributes $SA, PI, PS$, and the parameter sets $SP, PP$.
- Define the procedures $Sinit, Pinit, pre, act, inh, syn$.

To keep a grammar like this understandable, it is wise to separate clearly between syntax and semantics. Preferably the borderline between the two should be between the evaluation of surface attributes and procedural attributes. Thus, surface attribute evaluation is primarily meant for syntactic purposes, which is also the rationale for its power to filter out syntax trees.