Let's Accept Rejects, But Only After Repairs

Jan van Eijck

Abstract

In the quest for a useful syntax for language definition formalisms the SDF part of ASD+SDF of Klint and his co-workers — Visser has proposed to extend context free rules with reject productions. We propose to modify the definition to ensure modularity of reject grammars. Next, attention is drawn to the close link between reject grammars (under the modified definition), indexed least fixpoint grammars (Rounds) and simple literal movement grammars (Groenink). This link indicates that reject grammars are closely related to the 'mildly context-sensitive' grammar formalisms that have been developed for use in natural language analysis.

1 Reject Grammars

To increase the versatility of the ASD+SDF meta-environment for specification and syntax definition created by Paul Klint and his co-workers [6], Visser [9] proposes to add reject rules $A \rightarrow_r A_1 \cdots A_n$ to CF grammars. The recipe for handling these reject rules is, roughly, the following: if $A \rightarrow_r \alpha$ is a rule of the grammar, and if $\alpha \Rightarrow^* w$, then $A \not\Rightarrow^* w$.

The reason why this is only a rough indication of what is meant is the effect of the rejects rules on the notion of derivability. The familiar \Rightarrow^* , the reflexive transitive closure of one-step derivability, does no longer determine membership of the yield of a string of symbols of the grammar.

To precisely define the language generated by a CF reject grammar, a CF grammar with the peculiarity that some of its are marked as reject rules (indicated by $A \rightarrow_r \alpha$), we have to say the following. The derivable strings of a CF^r grammar G are defined as follows:

- 1. Every string of symbols derives itself.
- 2. If $A \to \alpha$ in G, and if α derives β , while for all reject rules $A \to_r \gamma$ in G it holds that γ does NOT derive β , then A derives β .
- 3. If $A \to_r \alpha$ in G, and α derives β , then A does NOT derive β .

Consider example grammar (1).

According to the recipe given above, grammar (1) generates the set of all strings over a, b except for the strings ending with a. In this particular case, the generated language is regular, but in general the generated language need not even be CF.

It should be noted that some care is needed to ensure that the format is well-defined. Observe that the presence of a reject rule of the form $A \rightarrow_r A$ will immediately lead to a contradiction: according to clause 1, A derives A, but then, according to clause 3, A does not derive A. We can remedy this by restricting clause 1 to strings of terminals, but it turns out that this is not enough, witness the following example.

Does example grammar (2) derive a? Suppose the answer is yes. i.e., suppose a is derivable from S. Then $A \to S$ and S derives a, and moreover, there are no reject rules with A as LHS. Therefore, according to clause 2 of the definition, A derives a. Since $S \to_r A$ is a reject rule of the grammar, according to clause 3, S does not derive a, and contradiction.

Suppose, on the other hand, that S does not derive a. Since $A \to S$ is the only rule with LHS A, it follows that A does not derive a either. According to clause 1 of the definition, a derives a. Since $S \to a$ in G, we can apply clause 2 to conclude that S derives a, and contradiction again.

We get a contradiction either way, even if we restrict clause 1 of the definition to terminal strings. In this example case, the problem is caused by the loop $S \rightarrow_r A \rightarrow S$. Visser proposes to rule out such loops in the formalism, by stipulating that a reject grammar should have no loops through a reject production. If I understand this correctly, Visser wants to rule out grammars where A derives $B, B \rightarrow_r \beta$ and β derives A. This seems to solve the problem, but at the (considerable) cost of destroying the modularity of the formalism. CF grammars are modular in the sense that adding extra rules to a CF grammar will always give a CF grammar. CF reject grammars satisfying the loop constraint are not, for the addition of extra rules may create a vicious loop and thereby destroy the constraint.

2 A Closer Look at the Definition

Let us see if we can find an alternative definition that preserves modularity. For that, we need to take a closer look at the definition of derivation paths for reject grammars. The relations \Rightarrow (for one step derivation), \Rightarrow_r (for one step rejection), and \Rightarrow^{\bullet} (for being on an accept path) are defined as follows:

Definitions of \Rightarrow , \Rightarrow _r

$$\frac{A \to \alpha}{\beta A \gamma \Rightarrow \beta \alpha \gamma} \qquad \qquad \frac{A \to r \alpha}{\beta A \gamma \Rightarrow_r \beta \alpha \gamma}$$

Definition of \Rightarrow^{\bullet} The following rules cover the three clauses from the definition of 'derivation' above. To be on the safe side, we restrict 1. to lists of terminals (strings from V_T^*).

1.

$$\frac{1}{\alpha \Rightarrow^{\bullet} \alpha} \alpha \in V_T^*$$

2.

$$\frac{\alpha \Rightarrow \beta}{\alpha \Rightarrow^{\bullet} \gamma} \qquad \begin{array}{c} \forall \delta : \text{ if } \alpha \Rightarrow_r \delta \text{ then } \delta \not\Rightarrow^{\bullet} \gamma \\ \alpha \Rightarrow^{\bullet} \gamma \end{array}$$

3.

$$\frac{\alpha \Rightarrow_r \beta \qquad \beta \Rightarrow^{\bullet} \gamma}{\alpha \not\Rightarrow^{\bullet} \gamma}$$

To be able to establish negative conclusions, we have to add: 4.

$$\frac{\forall \beta: \text{ if } \alpha \Rightarrow \beta \text{ then } \beta \not\Rightarrow^{\bullet} \gamma}{\alpha \not\Rightarrow^{\bullet} \gamma}$$

The following expresses that we have derived a contradiction: 5.

$$\frac{\alpha \Rightarrow^{\bullet} \beta \qquad \alpha \not\Rightarrow^{\bullet} \beta}{\perp}$$

We can now formalize our reasoning about grammar (2) as follows:

Looking at the rules 1–4, it seems that 3 and 4 deserve closer scrutiny, for they yield negative conclusions. But rule 4 is harmless. It just expresses the fact that \Rightarrow^{\bullet} is the *smallest* relation satisfying 2 and 3. It argues from a negative premise about \Rightarrow^{\bullet} to a negative conclusion about \Rightarrow^{\bullet} , so it does not involve a polarity switch. However, the other rule with a negative conclusion is vicious, for rule 3 switches polarity between premise and conclusion. In the next section we will see that this may destroy the monotonicity of the yield function of a grammar.

3 Denotational Semantics for Grammar Rules

To see the effect of rule 3, we consider the behaviour of the parallel single step yield function [G] associated with a grammar G. To define [G], we proceed as follows. Let V_N be the non-terminals of G, and V_T the terminals. Then an assignment for G is a function $i: V_N \to \mathcal{P}(V_T^*)$. Let I be the set of assignments for G. Associate a language with every member of V_T , by mapping every terminal to the singleton containing that terminal: $G_t := \{t\}$. To determine G_A , for $A \in V_N$, we proceed as follows. Every assignment i can be extended in a unique way to a function $i': (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$ by means of the stipulations:

- i'(x) := i(x) if $x \in V_N$,
- $i'(x) := G_x$ if $x \in V_T$,
- $i'(x\alpha) := i'(x)i'(\alpha)$.

The final step in the definition uses language concatenation.

Now if G is a standard CF grammar, then one parallel rewriting step according to G can be viewed as a function $[G] : I \to I$, given by [G](i) = j, where j is the assignment given by:

$$j(A) := \bigcup \{ i'(\alpha) \mid A \to \alpha \text{ a rule of } G \}$$

 $(3) \quad S \quad \rightarrow \quad ab \mid aSb$

For example, if G is grammar (3), and i is the assignment which maps S to \emptyset , then [G](i) is the assignment which maps S to $\{ab\}$.

The assignments have an obvious partial order on them: $i \sqsubseteq j$ iff for all $A \in V_N$, $i(A) \subseteq j(A)$. The minimal assignment in this ordering is the assignment that assigns \emptyset to any non-terminal. Call this assignment \bot . If [G] is monotonic, as it always is if G is a CF grammar, we can iterate the process of applying [G], starting with \bot , and we are bound to arrive at a fixpoint: an assignment j with the property that [G](j) = j. It is easy to see that [G] also is continuous, in the sense that it preserves limits of chains. Therefore the smallest fixpoint of [G] equals $\bigsqcup_{k \in \mathbb{N}} \{ [G]^k \bot \}$. If j is the smallest fixpoint of [G] and S is the start symbol of G, then j(S) is the language generated by G. (See Rounds [8] for more information on this perspective on grammar formalisms.)

In the example case of G equal to grammar (3), the fixpoint is only reached at infinity, and we have:

Let us apply this to the case of reject grammars. Here we have to slightly redefine the operation [G], in order to take the reject productions into account.

The function $[G]: I \to I$ is now given by [G](i) = j, where j is the assignment defined as:

$$\begin{aligned} j(A) &:= \bigcup \{ w \in i'(\alpha) \mid A \to \alpha \text{ a rule of } G, \\ \text{ and for all } A \to_r \beta \text{ rules of } G, w \notin i'(\beta) \}. \end{aligned}$$

If we take G to be grammar (1) we get the following:

Now let G be grammar (2). We get:

 $([G]\perp)(S)$ $\{a\} \quad ([G]\perp)(A)$ = Ø $([G]^2 \bot)(S) =$ Ø $([G]^2 \perp)(A)$ = $\{a\}$ $([G]^3\bot)(S) =$ $([G]^3 \perp)(A)$ $\{a\}$ = Ø

It turns out that the function [G] for this grammar is not monotonic, so it does not have a fixpoint, so G does not generate a well-defined language.

A Monotonic Semantics for Reject Grammars 4

The trouble with the definition of \Rightarrow_{\bullet} for reject grammars is that it involves reasoning from premises where \Rightarrow_{\bullet} occurs negatively to a conclusion where it occurs positively, and vice versa. We can avoid this by switching to a simultaneous definition of two relations \Rightarrow_a^{\bullet} ((for acceptance) and \Rightarrow_r^{\bullet} (for rejection), while taking care that the definition does not involve polarity switches between premises and conclusion.

1'.

$$\overline{\alpha \Rightarrow^{\bullet}_{a} \alpha} \ \alpha \in V_{T}^{*}$$

2'.

$$\frac{\alpha \Rightarrow \beta \qquad \beta \Rightarrow_a^{\bullet} \gamma \qquad \forall \delta: \text{ if } \alpha \Rightarrow_r \delta \text{ then } \delta \Rightarrow_r^{\bullet} \gamma \\ \alpha \Rightarrow_a^{\bullet} \gamma$$

3'.

$$\frac{\alpha \Rightarrow_r \beta \qquad \beta \Rightarrow_a^{\bullet} \gamma}{\alpha \Rightarrow_r^{\bullet} \gamma}$$

4'.

$$\frac{\forall \beta : \text{ if } \alpha \Rightarrow \beta \text{ then } \beta \Rightarrow^{\bullet}_{r} \gamma}{\alpha \Rightarrow^{\bullet}_{r} \gamma}$$

No further principles for handling negation are necessary, since there are no polarity switches in the rules.

To define a denotational semantics for this format, let an assignment for grammar G with nonterminal set V_N be a pair of functions $i_a : V_N \to V_T^*$, $i_r : V_N \to V_T^*$. Extension to $i'_a : (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$, $i'_r : (V_N \cup V_T)^* \to \mathcal{P}(V_T^*)$ as before. Let $[G] : I^2 \to I^2$ now be given by $[G](i_a, i_r) = (j_a, j_r)$, where j_a, j_r are the assignments given by:

$$j_{a}(A) := \bigcup \{ w \in i'_{a}(\alpha) \mid A \to \alpha \text{ a rule of } G, \\ \text{and for all } A \to_{r} \beta \text{ rules of } G, w \in i'_{r}(\beta) \}, \\ j_{r}(A) := \bigcup \{ w \in i'_{a}(\alpha) \mid A \to_{r} \alpha \text{ a rule of } G \} \\ \cup \bigcap \{ w \in i'_{r}(\beta) \mid A \to \beta \mid A \to \beta \text{ a rule of } G \}.$$

Defining \sqsubseteq on I^2 pointwise in terms of \sqsubseteq on I, we see that (\bot, \bot) is the bottom element of I^2 and that the function $[G] : I^2 \to I^2$ is monotonic and continuous. This means [G] has a fixpoint for every reject grammar G, and we can define the language generated by G as $i_a(S)$, where (i_a, i_r) is the least fixpoint of G. We may conclude that our alternative definition of the language works for any reject grammar. What this means is that we have succeeded in giving a modular definition of reject grammars. Moreover, it is not difficult to check that the new definition coincides with the original definition in all non-vicious cases.

5 A Logic Programming Perspective

A slightly different way to look at the rules of a context free grammar is as *constraints* on the interpretation of the grammar, where the interpretation of a (terminal or non-terminal) A is the language G_A . The rule $A \to A_1 \cdots A_n$ gives rise to the constraint $G_A \supseteq G_{A_1} \cdots G_{A_n}$. The interpretation of the grammar G is the 'smallest' assignment to the non-terminals for which all the constraints are satisfied.

To specify the constraints imposed by CF rules, we can use a translation into first order logic, by considering every non-terminal symbol A as a one-placed predicate over strings, and every terminal symbol a as a string. Then the rule $S \rightarrow ab$ gives rise to the first order formula S(ab), and the rule $S \rightarrow aSb$ to the first order formula $\forall x(S(x) \rightarrow S(axb))$ (read this as: 'if x is a string of type S, then axb is a string of type S as well'), and so on.

Leaving out the universal quantifiers and writing $A \to B$ as $B :\perp A$, we arrive at the following logic programming style notation for grammar (3):

$$\begin{array}{l} S(axb) :\perp S(x), \\ S(ab). \end{array}$$

Thus, CF rules correspond to definite clauses of a logic programming language with built-in concatenation. Now, how to extend this perspective to reject grammars? According to the original definition, we should translate every accept rule $A \to A_1 \cdots A_n$ as:

$$A(x_1\cdots x_n) :\perp A_1(x_1), \ldots, A_n(x_n), \neg A^r(x_1\cdots x_n),$$

and every reject rule $A \rightarrow_r B_1 \cdots B_m$ as:

$$A^r(x_1\cdots x_m) :\perp B_1(x_1),\ldots, B_m(x_m).$$

The logic programming interpretation of this translation will run into trouble in all cases where the negation as failure computation for negated atoms leads to floundering. (See Apt and Bol [1] or Doets [2].) Presumably, these are precisely the cases where the reject language definition gives rise to a vicious circle.

According to the modified definition, we should proceed as follows. First, for every terminal a of the grammar, put the following clauses:

a(x) : \bot x = a, $a^r(x)$: \bot $x \neq a.$

Next, add the following epsilon rules:

 $\begin{array}{ll} \epsilon(x) & :\perp & x = \epsilon, \\ \epsilon^r(x) & :\perp & x \neq \epsilon. \end{array}$

If A is a non-terminal, assume that the list of accept rules for A looks like this:

 $\begin{array}{rccc} A & \to & A_{11} \cdots A_{1n_1} \\ A & \to & A_{21} \cdots A_{2n_2} \\ & \vdots \\ A & \to & A_{p1} \cdots A_{pn_p}. \end{array}$

Pick new predicate symbols A_1^r, \ldots, A_p^r , and add the following clauses:

$$\begin{array}{rcl} A_{1}^{r}(x_{1}\cdots x_{n_{1}}) & :\perp & A_{11}^{r}(x_{1}), \ldots, A_{1n_{1}}^{r}(x_{n_{1}}), \\ A_{2}^{r}(x_{1}\cdots x_{n_{2}}) & :\perp & A_{21}^{r}(x_{1}), \ldots, A_{2n_{1}}^{r}(x_{n_{2}}), \\ & \vdots \\ A_{p}^{r}(x_{1}\cdots x_{n_{p}}) & :\perp & A_{p1}^{r}(x_{1}), \ldots, A_{pn_{1}}^{r}(x_{n_{p}}), \\ & A^{r}(w) & :\perp & A_{1}^{r}(w), \ldots, A_{p}^{r}(w). \end{array}$$

In case there are no reject rules for A, translate every accept rule $A \to A_1 \cdots A_n$ as:

 $A(x_1 \cdots x_n) :\perp A_1(x_1), \ldots, A_n(x_n).$

If there are reject rules for A, we may assume the set of reject rules for A to look like this:

$$\begin{array}{rrrr} A & \rightarrow_r & B_{11} \cdots B_{1m_1} \\ A & \rightarrow_r & B_{21} \cdots B_{2m_2} \\ & \vdots \\ A & \rightarrow_r & B_{k1} \cdots B_{km_k}, \end{array}$$

In that case, first add the following clauses:

$$\begin{array}{rcl} A^{r}(x_{1}\cdots x_{m_{1}}) & :\perp & B_{11}(x_{1}), \ldots, B_{1m_{1}}(x_{m_{1}}) \\ A^{r}(x_{1}\cdots x_{m_{2}}) & :\perp & B_{21}(x_{1}), \ldots, B_{2m_{2}}(x_{m_{2}}) \\ & \vdots \\ A^{r}(x_{1}\cdots x_{m_{k}}) & :\perp & B_{k1}(x_{1}), \ldots, B_{km_{k}}(x_{m_{k}}). \end{array}$$

Next, pick new predicate names B_1^r, \ldots, B_k^r , and add:

$$B_{1}^{r}(x_{1}\cdots x_{m_{1}}) :\perp B_{11}^{r}(x_{1}), \dots, B_{1m_{1}}^{r}(x_{m_{1}})$$
$$B_{2}^{r}(x_{1}\cdots x_{m_{2}}) :\perp B_{21}^{r}(x_{1}), \dots, B_{2m_{2}}^{r}(x_{m_{2}})$$
$$\vdots$$
$$B_{k}^{r}(x_{1}\cdots x_{m_{k}}) :\perp B_{k1}^{r}(x_{1}), \dots, B_{km_{1}}^{r}(x_{m_{k}})$$

Finally, translate every accept rule $A \to A_1 \cdots A_n$ as:

$$A(x_1\cdots x_n) :\perp A_1(x_1), \ldots, A_n(x_n), B_1^r(x_1\cdots x_n), \ldots, B_k^r(x_1\cdots x_n).$$

Let us check how this recipe works out for example grammars (1) and (2). The logic program corresponding to grammar (1), after removal of some redundancies:

$$S(\epsilon) :\perp a^{r}(\epsilon).$$

$$S(xy) :\perp A(x), S(y), a^{r}(xy).$$

$$S^{r}(a).$$

$$a^{r}(x) :\perp x \neq a.$$

$$A(a).$$

$$A(b).$$

The logic program corresponding to grammar (2):

$$S(x) :\perp a(x), A^{r}(x).$$

$$S^{r}(x) :\perp a^{r}(x).$$

$$A(x) :\perp S(x).$$

$$a(x) :\perp x = a.$$

$$a^{r}(x) :\perp x \neq a.$$

6 Rejects Versus Simple Movement

The clauses resulting from the translation instruction of the last section satisfy some special properties:

- Every variable occurring in the RHS of a clause occurs in the LHS,
- No variable occurs more than once in the LHS of a clause.

Moreover, the following property is easy to achieve by adding extra predicate symbols:

• Arguments in the RHSs of clauses are always single variables.

To impose this third constraint, the only thing we have to do is replace every clause of the form

 $A(x_1\cdots x_n) :\perp A_1(x_1), \ldots, A_n(x_n), B_1^r(x_1\cdots x_n), \ldots, B_k^r(x_1\cdots x_n)$

by a suitable clause set, employing new predicate symbols C_1, C_2 :

$$C_1(x_1\cdots x_n) \quad :\perp \quad A_1(x_1), \dots, A_n(x_n)$$
$$C_2(w) \quad :\perp \quad B_1^r(w), \dots, B_k^r(w),$$
$$A(w) \quad :\perp \quad C_1(w), C_2(w).$$

In Groenink [5] a grammar satisfying the three just mentioned constraints is called a *simple literal movement grammar* (SLMG). Groenink proves that the simple literal movement grammars correspond precisely with the index least fixpoint grammars (ILFPs) of Rounds [8], which in turn are precisely the grammars that can be parsed in polynomial time (in $O(n^c)$, where *n* is the size of the input string, and *c* is some constant).

Now it turns out that our definite clause translations of reject grammars can be massaged to satisfy a fourth constraint:

• all predicates are unary.

The only predicates that do not satisfy this constraint are the predicates for string equality and string inequality employed in the clauses for the terminal predicates $a, a^r, \epsilon, \epsilon^r$. These can be replaced by the following:

$$a(a),$$

$$a^{r}(\epsilon),$$

$$a^{r}(bx) \qquad \text{for all } b \in V_{T} \text{ with } b \neq a,$$

$$\epsilon(\epsilon),$$

$$\epsilon^{r}(ax) \qquad \text{for all } b \in V_{T}.$$

A final simplification is to replace every clause of the form

 $A(x_1 \cdots x_n) :\perp A_1(x_1), \ldots, A_n(x_n)$

by a set of clauses employing fresh predicate letters A'_2, \ldots, A'_{n-1} , as follows:

$$\begin{array}{rcl} A(xy) & : \bot & A_1(x), A_2'(y), \\ A_2'(xy) & : \bot & A_2(x), A_3'(y), \\ & \vdots \\ A_{n-1}'(xy) & : \bot & A_{n-1}(x), A_n(y), \end{array}$$

and every clause of the form

$$A(w) :\perp B_1(w), \ldots, B_m(w)$$

by a set of clauses

$$\begin{array}{rcl}
A(w) & : \bot & B_1(w), B_2'(w), \\
B_2'(w) & : \bot & B_2(w), B_3'(w), \\
& \vdots \\
B_{m-1}'(w) & : \bot & B_{m-1}(w), B_m(w).
\end{array}$$

As Groenink remarks, it is not difficult to translate SLMG clauses with monadic predicates into CF^r format, the key being that reject rules can be used to define intersections (Groenink [4]). Indeed, all clauses except for the intersection rules translate into CF rules. An intersection rule $S(w) :\perp A(w), B(w)$ translates into the following reject grammar (assuming a terminal set $\{a, b\}$, and taking care to employ fresh rewrite symbols where appropriate):

Here A' generates the complement of L_A , B' the complement of L_B . C generates the union of those complements, and S, finally, the complement of this union.

It should be noted that the correspondence between reject grammars and unary SLMGs only holds under the modified definition of reject grammars proposed above.

The correspondence between reject grammars and unary simple literal movement grammars shows that reject languages are among the tractable languages, i.e., the languages that can be parsed in polynomial time. Next on the agenda should be the attempt to put a reasonable bound on the constant c in $O(n^c)$ (see Groenink [5] for some hints), the definition of suitable partitions of the class of CF^r languages, and an attempt to get still more precise bounds for CF^r grammars satisfying appropriate extra constraints. For instance, if a reject grammar has the property that its reject rules cannot occur nested, what does this tell us about the value of c in $O(n^c)$? And what if the grammar is such that its reject chains have a bound k?

An example of a reject grammar generating the non-CF language $a^n b^n c^n$ was given in Van Eijck [3] (see Visser [9]). The example can easily be extended to show that all *n*-counting languages (languages of the form $a^n b^n \cdots k^n$, for an arbitrary list of terminals a, \ldots, k) are in CF^r.

An example of a very simple language that can be generated by a binary SLMG is a^{2^n} :

$$S(xy) \quad :\perp \quad S(x), \ x = y,$$

$$S(a).$$

This uses the binary equality predicate. It seems a reasonable conjecture that this language cannot be defined without the equality predicate (or another binary predicate that serves essentially the same purpose), but I have no formal proof of this. To prove such a negative claim one needs a pumping lemma for reject grammars (or equivalently, for unary SLMGs), and such a lemma is still lacking. So let's put that also on our research agenda.

At present, to my knowledge, we do not have examples of languages that are in binary SLML but not in unary SLML (or CF^r), nor do we have examples of languages in CS, but not in CF^r . A CS grammar for a^{2^n} is part of the formal language folklore (see Révész [7], exercise 1.5), but we have no proof that $a^{2^n} \notin CF^r$. The difficulty here is that it is not completely clear how the Chomsky hierarchy relates to the complexity hierarchy proposed in Rounds [8] and Groenink [5].

The following closure properties of CF^r languages are easy to establish:

$closure \ properties \ of \ CF^r \ languages$	
closed under union	yes
closed under concatenation	yes
closed under Kleene-star	yes
closed under ⁺	yes
closed under homomorphic image	no
closed under reversal	yes
closed under intersection with CF language	yes
closed under intersection	yes
closed under complement	yes.

The reason for the single 'no' in the table is the classical result that a formalism that can generate homomorphic images of intersections of CF languages can generate every r.e. language.

7 Reject Rules for Natural Language Analysis

Groenink shows in his thesis that SLMG (but with binary predicates) is a suitable formalism for tackling some hairy constructs in natural language syntax, such as cross-serial dependencies in Dutch. It remains an open question whether such natural language constructs can also be tackled with reject rules, as proposed by Visser. To settle that question, we have to know whether the inclusion between reject languages and binary SLM languages is proper or not. My conjecture is that it is, and that it will turn out that reject rules are not powerful enough for handling the full complexity of natural language syntax.

Still, I hope to have illustrated that the two PhD theses resulting from the dual SION project Visser/Groenink, the first supervised by Paul Klint and the

second by myself, are closely related. Paul and I were quite right, five years ago, when we told the Dutch computer science funding agency SION that parsing of natural languages and parsing of programming languages have a lot in common, and that money to explore those common aspects would be well spent. I am very glad we did get that grant.

References

- Krzysztof Apt and Roland Bol. Logic programming and negation: a survey. Journal of Logic Programming, 19-20:9–71, 1994.
- [2] Kees Doets. From Logic to Logic Programming. MIT Press, 1994.
- [3] Jan van Eijck. Email correspondence with Eelco Visser. CWI, July 1997.
- [4] Annius Groenink. Stellingen. PhD thesis, 1997.
- [5] Annius Groenink. Surface Without Structure. PhD thesis, Utrecht University, November 1997.
- [6] Paul Klint and co workers. The ASF+SDF Meta-environment User's Guide. CWI, 1995.
- [7] G. E. Révész. Introduction to Formal Languages. Dover, 1991. Reprint of 1983 publication by McGraw-Hill.
- [8] Bill Rounds. LFP: A logic for linguistic descriptions and an analysis of its complexity. *Computational Linguistics*, 14(4):1–9, 1988.
- [9] Eelco Visser. Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam, September 1997.