Universität Ulm
Fakultät für Informatik


# Again on Recognition and Parsing of Context-Free Grammars:

## Two Exercises in Transformational Programming

Helmuth Partsch
*Universität Ulm*

Nr. 94-17
Ulmer Informatik-Berichte
Dezember 1994

# Again on Recognition and Parsing of Context-Free Grammars:

## Two Exercises in Transformational Programmming

H. Partsch
Fakultät für Informatik
Universität Ulm

**abstract**

This paper deals with two aspects of transformational programming, viz. the formal derivation of logic programs from non-operational specifications and the construction of algorithms to solve problems formally specified by "inverse properties". Both aspects are illustrated by sample derivations of standard algorithms for recognition and parsing of context-free grammars.

## 0. Introduction

Transformational programming is short for a methodology of software development where, from a formal specification of a problem to be solved, programs correctly solving that problem are constructed by stepwise application of formal, semantics-preserving transformation rules. In the following the reader is assumed to be basically familiar with the idea of transformational programming and its basic principles. A brief introduction and an overview can be found, e.g., in [Bauer et al. 89] or [Boiten et al. 92]. For a comprehensive treatment of the subject, cf., e.g., [Partsch 90].

In this paper two examples of transformational programming are dealt with, both from the area of recognition and parsing of context-free grammars. Therefore, basic knowledge on context-free grammar theory is advantageous, but not mandatory to grasp the essence of the derivations.

Parsing and recognition of context-free grammars is a rather fruitful field for studying aspects of programming methodology, since a lot of generally applicable programming knowledge has accumulated in this area over many years. A comprehensive attempt to make this knowledge available for transformational programming is documented in [Partsch 86]. In front of this background, the present paper is to be seen as a kind of addendum to cover two further, not yet considered aspects.

The first case study we deal with gives a very simple derivation of Prolog programs for top-down recognition and parsing of context-free grammars. It is intended to illustrate that two paradigms of modern, formal software development, viz. transformational programming and

logic programming, fit nicely together and how they can benefit from each other. The second case study deals with recognition of context-free grammars as a typical representative of a class of problems, in the community known as "inverse problems".

As to the notation in our case studies, we will use fairly self-explanatory concepts from mathematics and functional programming, respectively, occasionally augmented by comments on less commonly known concepts or operators.

# 1. Preliminaries

In order to be able to specify the problem we want to deal with, we have to introduce some basic notions, notably in connection with sequences and context-free grammars.

## 1.1 Sequences

The primitive data type we are building on is the type of **sequences** (over some basic type $\alpha$) which we abbreviate by $\alpha^*$. As elementary, totally defined operations on sequences we assume to have available (for $s$, $t$ of type $\alpha^*$, $x$ of type $\alpha$):

| | |
|---|---|
| $\varepsilon$ | empty sequence |
| $\langle x \rangle$ | sequence former |
| $s +\!\!+ t$ | concatenation, and |
| $\lvert s \rvert$ | length. |

Whenever clear from the context, explicit sequence formers will be avoided, i.e., concatenation will also be used for adding (single) elements to a sequence.

In addition to the totally defined operations, we also assume to have available some partially defined ones:

| | |
|---|---|
| $s_i$ | indexed access (only defined for $1 \leq i \leq \lvert s \rvert$, otherwise undefined) |
| $s_1$ | first element (undefined for $s = \varepsilon$) |
| $s_\#$ | last element (undefined for $s = \varepsilon$) |
| $\overline{\lceil} s$ | rest (undefined for $s = \varepsilon$) |
| $s \overline{\rceil}$ | starter (undefined for $s = \varepsilon$). |

For all operations, characteristic properties such as $s \neq \varepsilon \Rightarrow s = s_1 +\!\!+ \overline{\lceil} s$, usually explicitly given by an algebraic type definition (cf. [Partsch 90]), are also assumed to be available. Furthermore, we will use $x \in D$ to denote "$x$ is of type D" (for arbitrary data type D).

## 1.2 Context-free grammars

As usual, a **context-free grammar** G is defined by

$G =_{def} (N, T, Z, Prods)$

where

N is a non-empty, finite set of symbols ("non-terminals");
T is a non-empty, finite set of symbols ("terminals") with $N \cap T = \varnothing$;

$Z \in N$ ("axiom"); and

Prods is a non-empty, finite subset of $N \times V^*$ ("productions"), where $V =_{def} N \cup T$.

In order to keep our treatment of the problem reasonably short, we further assume (w.l.o.g.) that

- G is *reduced* (i.e., has no unproductive or unreachable non-terminals);
- G has no *chain* or *ε-productions*

  (i.e., no productions of the forms $(X, Y)$ or $(X, \varepsilon)$ for $X, Y \in N$)

In order to be able to specify our sample problems, viz. recognition and parsing, we need some notions fom the theory of context-free grammars, notably the notions of "direct derivability",

$$\rightarrow \subseteq V^* \times V^*: x \rightarrow y =_{def} \exists\, l, r \in V^*, (a, b) \in \text{Prods}: x = lar \wedge y = lbr,$$

and "derivability"

$$\Rightarrow^* \subseteq V^* \times V^*,$$

the reflexive, transitive closure of $\rightarrow$.

As to the latter, rather than giving an explicit definition of derivability, we just state two of its characteristic properties (which hold analogously for arbitrary reflexive, transitive closures):

(0) $(x \Rightarrow^* y) \equiv (x = y \vee \exists\, z \in V^*: x \Rightarrow^* z \wedge z \rightarrow y)$.

(1) $(x \Rightarrow^* y) \equiv (x = y \vee \exists\, z \in V^*: x \rightarrow z \wedge z \Rightarrow^* y)$

Also, we will use the following characteristic property of context-free grammars (for arbitrary natural number $n \geq 1$):

(2) $(x \Rightarrow^* y) \equiv$

$$\exists\, x_1, \ldots, x_n \in V^*, y_1, \ldots, y_n \in T^*: x = x_1 \dashv\!\vdash \ldots \dashv\!\vdash x_n \wedge y = y_1 \dashv\!\vdash \ldots \dashv\!\vdash y_n \wedge (x_1 \Rightarrow^* y_1 \wedge \ldots \wedge x_n \Rightarrow^* y_n).$$

With these preliminaries, we are now able to state what is meant by "recognition problem" (for $w \in T^*$) in context-free grammar theory:

$$Z \Rightarrow^* w.$$

The "parsing problem" is closely related to the recognition problem: For $w \in T^*$ with $Z \Rightarrow^* w$ the "parse structure" of $w$ is asked for. This parse structure contains all information on how to derive $w$ from Z by successive applications of productions from the grammar. In its simplest form, the parse structure consists of a sequence of productions to be successively applied (to the leftmost occurrence of a nonterminal symbol of the string at hand) in order to finally arrive at $w$. More frequently, however, the parse structure is represented by a "parse tree" which has Z as its root, nonterminal symbols as its inner nodes, and the terminal symbols from $w$ as its leaves (from left to right). Additionally, it is required for a parse tree that every inner node together with the sequence of its son nodes (from left to right) is a production of the grammar.

## 2. Deriving prolog programs for top-down recognition and parsing

The goal of this case study is to illustrate that the idea of transformational programming profitably can be used in the synthesis of logic programs to solve problems given by a descriptive specification. As will be seen below, this is particularly advantageous for such kind of problems where a solution is not immediately obvious from the specification of the problem.

### 2.1 The formal specification

To start with, we assume

T, N, V, Prods

to denote given, basic types to represent the constituents of our given context-free grammar G = (N, T, Z, Prods). The definitions of the recognition and parsing problems then are simple transcriptions of the definitions as given in the previous section:

*rec*: T* → Bool,
*rec*(*w*) =$_{def}$ Z ⇒* *w*

and

*parse*: T* → Prods* | **dummy**,
*parse*(*w*) =$_{def}$ **if** *rec*(*w*) **then some** *p* ∈ Prods*: *isparse*(*p*, Z, *w*) **else dummy fi**,

(where a suitable definition of the predicate *isparse* still is to be given, see below).

### 2.2 The essence of the formal development

Our primary focus of interest is the body of the function *rec*

Z ⇒* *w*

from which, by a simple *generalization* (V for N; V* for V) we obtain

*rec*(*w*) =$_{def}$ *r*(<Z>, *w*) **where**
*r*: V* × T* → Bool,
*r*(*u*, *v*) =$_{def}$ *u* ⇒* *v*.

Thus, the function *r* becomes our new focus of interest and we aim at transforming *r* into an operational specification using the "generalized unfold-fold strategy" from [Partsch 90].

We start by introducing a case distinction ($v = \varepsilon \vee v \neq \varepsilon$) motivated by the structure of type T*:

$(v = \varepsilon \,\Delta\, u \Rightarrow^* v) \vee (v \neq \varepsilon \,\Delta\, u \Rightarrow^* v)$

(where Δ denotes sequential conjunction).

Next, we try to simplify both disjuncts. For the first disjunct we use

$v = \varepsilon \vdash (u \Rightarrow^* v) \equiv (u = \varepsilon)$

which is an immediate consequence from (1) and the definition of context-free grammar. For the second disjunct we use

$v \neq \varepsilon \vdash u \neq \varepsilon$

which is an obvious consequence from (1) and

$u \neq \varepsilon \vdash u \equiv u_1 \mathbin{+\!\!+} \lceil u \rceil$,

one of the characteristic properties of sequences. Together, we obtain

$(v = \varepsilon \,\Delta\, u = \varepsilon) \vee (v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, (u_1 \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v)$.

In order to proceed, we now concentrate on the last conjunct of the second disjunct and try to simplify further (under the premise $v \neq \varepsilon \,\Delta\, u \neq \varepsilon$). To this end, we first introduce another case distinction by adding the tautology $(u_1 \in \mathrm{T} \vee u_1 \in \mathrm{N})$, which simply exploits the definition of V, as an additional conjunct. Thus, by additionally using the distributivity of $\vee$ over $\Delta$ we get

$(u_1 \in \mathrm{T} \;\Delta\, (u_1 \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v) \vee$
$(u_1 \in \mathrm{N} \,\Delta\, (u_1 \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v)$.

Now, both disjuncts may be simplified individually:

$u_1 \in \mathrm{T} \vdash$
$(u_1 \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v$
$\equiv$ [ property (2)]
$\exists\, v_1, v_2 \in \mathrm{T}^*: v = v_1 \mathbin{+\!\!+} v_2 \wedge u_1 \Rightarrow^* v_1 \wedge \lceil u \rceil \Rightarrow^* v_2$
$\equiv$ [ $u_1 \in \mathrm{T} \Rightarrow u_1 = v_1$; $u_1 = v_1 \Rightarrow v_2 = \lceil v \rceil$; $\Delta$ for $\wedge$ ]
$u_1 = v_1 \,\Delta\, \lceil u \rceil \Rightarrow^* \lceil v \rceil$

$u_1 \in \mathrm{N} \vdash$
$(u_1 \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v$
$\equiv$ [ property (2)]
$\exists\, v_1, v_2 \in \mathrm{T}^*: v = v_1 \mathbin{+\!\!+} v_2 \wedge u_1 \Rightarrow^* v_1 \wedge \lceil u \rceil \Rightarrow^* v_2$
$\equiv$ [ property (1) applied to $u_1 \Rightarrow^* v_1$ ]
$\exists\, v_1, v_2 \in \mathrm{T}^*: v = v_1 \mathbin{+\!\!+} v_2 \wedge (\exists\, z \in \mathrm{V}^*: u_1 \to z \wedge z \Rightarrow^* v_1) \wedge \lceil u \rceil \Rightarrow^* v_2$
$\equiv$ [ $u_1 \in \mathrm{N} \Rightarrow (u_1 \to z) \equiv (u_1, z) \in \mathrm{Prods}$ ]
$\exists\, v_1, v_2 \in \mathrm{T}^*: v = v_1 \mathbin{+\!\!+} v_2 \wedge (\exists\, z \in \mathrm{V}^*: (u_1, z) \in \mathrm{Prods} \wedge z \Rightarrow^* v_1) \wedge \lceil u \rceil \Rightarrow^* v_2$
$\equiv$ [ rearrangement of quantifiers ]
$\exists\, z \in \mathrm{V}^*: (u_1, z) \in \mathrm{Prods} \wedge \exists\, v_1, v_2 \in \mathrm{T}^*: v = v_1 \mathbin{+\!\!+} v_2 \wedge z \Rightarrow^* v_1 \wedge \lceil u \rceil \Rightarrow^* v_2$
$\equiv$ [ property (2); $\Delta$ for $\wedge$ ]
$\exists\, z \in \mathrm{V}^*: (u_1, z) \in \mathrm{Prods} \,\Delta\, (z \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v$

Putting together all pieces, we have as an (intermediate) overall result for $r$:

$(v = \varepsilon \,\Delta\, u = \varepsilon) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 \in \mathrm{T} \;\Delta\, u_1 = v_1 \,\Delta\, \lceil u \rceil \Rightarrow^* \lceil v \rceil) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 \in \mathrm{N} \;\Delta\, \exists\, z \in \mathrm{V}^*: (u_1, z) \in \mathrm{Prods} \,\Delta\, (z \mathbin{+\!\!+} \lceil u \rceil) \Rightarrow^* v)$

This can be further simplified. For the second disjunct, we first use

$$u_1 = v_1 \Rightarrow u_1 \in T$$

and by the general property

$$(B \Rightarrow A) \vdash (A \,\Delta\, B) \equiv B)$$

result in

$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \,\Delta\, \overline{u} \Rightarrow^* \overline{v}\,).$$

For the third disjunct we first use

$$\exists\, z \in V^*: (u_1, z) \in \text{Prods} \Rightarrow u_1 \in N$$

and by the same general property obtain

$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \,\Delta\, (z +\!\!+ \overline{u}) \Rightarrow^* v).$$

Thus, altogether, we have

$$(v = \varepsilon \,\Delta\, u = \varepsilon) \vee$$
$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \,\Delta\, \overline{u} \Rightarrow^* \overline{v}) \vee$$
$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \,\Delta\, (z +\!\!+ \overline{u}) \Rightarrow^* v).$$

Finally, by folding with the definition of $r$ we get:

$$(v = \varepsilon \,\Delta\, u = \varepsilon) \vee$$
$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \,\Delta\, r(\overline{u}, \overline{v})) \vee$$
$$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \,\Delta\, r(z +\!\!+ \overline{u}, v)).$$

However, when trying to prove termination (in order to ensure correctness of folding), we realize that the absence of left-recursive productions as an additional constraint on our grammar G has to be required for a successful termination proof.

In order to deal with the parsing problem, we concentrate on its characterisitic predicate *isparse* which can be specified as follows:

$$\textit{isparse}: \text{Prods}^* \times V^* \times T^* \rightarrow \text{Bool},$$
$$\textit{isparse}(p, u, v) =_{\text{def}} r(u, v) \,\Delta\, \textit{apply}(p, u) = v$$

where *apply* is a partial function defined by

$$\textit{apply}: \text{Prods}^* \times V^* \rightarrow V^*,$$

$$\textit{apply}(p, u) =_{\text{def}} \begin{cases} u, \text{ if } p = \varepsilon \\ u_1 +\!\!+ \textit{apply}(p, \overline{u}), \text{ if } p \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 \in T \\ \textit{apply}(\overline{p}, z +\!\!+ \overline{u}) \\ \quad \text{if } p \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: p_1 = (u_1, z) \end{cases}$$

From its specification, a definition of *isparse* can be derived by simple transformation steps. First, we unfold the call of $r$ and distribute $\vee$ over $\Delta$:

$(v = \varepsilon \,\Delta\, u = \varepsilon \,\Delta\, apply(p, u) = v) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \,\Delta\, r(\overline{u}, \overline{v}) \,\Delta\, apply(p, u) = v) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \,\Delta\, r(z +\!\!+ \overline{u}, v) \,\Delta\, apply(p, u) = v).$

Now, all three disjuncts can be treated separately (under the respective premises):

- $v = \varepsilon \,\Delta\, u = \varepsilon \vdash$
  $apply(p, u) = v$
  $\equiv$ [ =-substitutivity ]
  $apply(p, \varepsilon) = \varepsilon$
  $\equiv$ [ def. of *apply* ]
  $p = \varepsilon$

- $v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \vdash$
  $r(\overline{u}, \overline{v}) \,\Delta\, apply(p, u) = v$
  $\equiv$ [ premise $v \neq \varepsilon \,\Delta\, u \neq \varepsilon$ ]
  $r(\overline{u}, \overline{v}) \,\Delta\, apply(p, u_1 +\!\!+ \overline{u}) = v_1 +\!\!+ \overline{v}$
  $\equiv$ [ $u_1 \in T$; def. of *apply* ]
  $r(\overline{u}, \overline{v}) \,\Delta\, u_1 +\!\!+ apply(p, \overline{u}) = v_1 +\!\!+ \overline{v}$
  $\equiv$ [ premise $u_1 = v_1$; = on sequences ]
  $r(\overline{u}, \overline{v}) \,\Delta\, apply(p, \overline{u}) = \overline{v}$
  $\equiv$ [ def. of *isparse* ]
  $isparse(p, \overline{u}, \overline{v})$

- $v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \vdash$
  $r(z +\!\!+ \overline{u}, v) \,\Delta\, apply(p, u) = v$
  $\equiv$ [ def. of *apply* ]
  $r(z +\!\!+ \overline{u}, v) \,\Delta\, p \neq \varepsilon \,\Delta\, p_1 = (u_1, z) \,\Delta\, apply(\overline{p}, z +\!\!+ \overline{u}) = v$
  $\equiv$ [ commutativity of independent disjuncts; def. of *isparse* ]
  $p \neq \varepsilon \,\Delta\, p_1 = (u_1, z) \,\Delta\, isparse(\overline{p}, z +\!\!+ \overline{u}, v).$

Summing up, we get the following definition for *isparse*:

$(v = \varepsilon \,\Delta\, u = \varepsilon \,\Delta\, p = \varepsilon) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, u_1 = v_1 \,\Delta\, isparse(p, \overline{u}, \overline{v})) \vee$
$(v \neq \varepsilon \,\Delta\, u \neq \varepsilon \,\Delta\, \exists\, z \in V^*: (u_1, z) \in \text{Prods} \,\Delta\, p \neq \varepsilon \,\Delta\, p_1 = (u_1, z) \,\Delta\, isparse(\overline{p}, z +\!\!+ \overline{u}, v)).$

## 2.3 Prolog programs

The result of our previous derivations can straightforwardly be rewritten into equational form with "pattern matching". Thus, for the recognition problem we get

$r(\varepsilon, \varepsilon) =_{def}$ **true**,
$r(u_1 +\!\!+ \overline{u}, v_1 +\!\!+ \overline{v}) =_{def} u_1 = v_1 \,\Delta\, r(\overline{u}, \overline{v}),$

$r(u_1 \dashv\!\vdash \ulcorner u, v) =_{def} \exists z \in V^*: (u_1, z) \in Prods \Delta r(z \dashv\!\vdash \ulcorner u, v))$,

and for the parsing problem

$isparse(\varepsilon, \varepsilon, \varepsilon) =_{def}$ **true**,

$isparse(p, u_1 \dashv\!\vdash \ulcorner u, v_1 \dashv\!\vdash \ulcorner v) =_{def} u_1 = v_1 \Delta isparse(p, \ulcorner u, \ulcorner v)$,

$isparse((u_1, z) \dashv\!\vdash \ulcorner p, u_1 \dashv\!\vdash \ulcorner u, v) =_{def} (u_1, z) \in Prods \Delta isparse(\ulcorner p, z \dashv\!\vdash \ulcorner u, v)$.

Both definitions can be immediately transcribed into Prolog programs, if we use lists to represent sequences of symbols, the facts

`isprod(x,y)` *(for all (x, y) ∈ Prods)*

to represent the production rules of our grammar G, and the constant `s` to denote its axiom. For the recognition problem, we thus get:

```
rec(s,W) :- r([s],W).

r([],[]).
r([U|Urest],[V|Vrest])
  :- U = V, r(Urest,Vrest).
r([U|Urest],V)
  :- isprod(U,Z), append(Z,Urest,X), r(X,V).
```

If we furthermore represent the initial terminal word *w* by a list `[w1,...,wn]` the desired result for the recognition problem will be computed by the query

```
?-  rec(s,[w1,...,wn]).
```

Likewise, we obtain immediately a solution to the parsing problem:

```
p([],[],[]).
p(P,[U|Urest],[V|Vrest])
  :- U = V, p(P,Urest,Vrest).
p([(U,Z)|Prest],[U|Urest],V)
  :- isprod(U,Z), append(Z,Urest,X), p(Prest,X,V).
```

If we again represent the initial terminal word *w* by a list `[w1,...,wn]` the desired result for the parsing problem will now be computed by the query

```
?-  p(P, [s], [w1,...,wn]).
```

## 3. Bottom-up recognition considered as an "inverse problem"

In this second case study, we deal with a typical representative of a class of problems frequently called "inverse problems". Intuitively, the problems in this class ask for computing a function

$f: \alpha \rightarrow \beta$

for some value $x$, where $f$ is initially specified in terms of its "inverse"

$g: \beta \rightarrow \alpha$,

e.g. by

$f(x) =_{\text{def}}$ **some** $y$: $g(y) = x$.

Specifying a problem in this way is particularly profitable in cases where $f$ is hard to be specified "directly", whereas a (constructive) specification of $g$ is already available or rather straightforwardly to formulate.

A simple instance of such an "inverse" problem is the specification of a (real) square root in terms of (real) multiplication:

**some** $y$: $y \times y = x$.

A somewhat larger class of "inverse problems" is covered, if we additionally allow further constraints on the result of the desired function $f$:

$f(x) =_{\text{def}}$ **some** $y$: $g(y) = x \wedge P(x, y)$

where $P$ is some additional predicate.

Again, an instance is provided by specifying the square root - this time constrained to non-negative results:

$sqr(x) =_{\text{def}}$ **some** $y$: $y \times y = x \wedge y \geq 0$.

In the following we deal with a more interesting instance of this latter form of "inverse problems", viz. recognition of context-free grammars, and show how a bottom-up algorithm can be derived by very simple transformation steps.

### 3.1 The formal specification

To start with, we assume, again,

T, N, V, Prods

to denote given, basic types for the constituents of a context-free grammar G = (N, T, Z, Prods).

Then the **parse structure** P of a terminal word with respect to grammar G can be defined by

$\text{P} =_{\text{def}} \text{T} \mid \text{Nont}$

where

$\text{Nont} =_{\text{def}} (\text{N } lhs \times \text{P}^* \; rhs\text{: } (lhs, symbs(rhs)) \in \text{Prods})$

and where

$symbs\text{: } \text{P}^* \rightarrow \text{V}^*$

$$symbs(p) =_{\text{def}} \begin{cases} \varepsilon, \text{ if } p = \varepsilon \\ p_1 \dashv\!\!+ symbs(\overline{p}), \text{ if } p \neq \varepsilon \;\Delta\; p_1 \in \text{T} \\ lhs(p_1) \dashv\!\!+ symbs(\overline{p}), \text{ otherwise} \end{cases}$$

According to this definition, a "parse" P is either a terminal symbol from T or an element from Nont. An element $q$ of Nont is in turn defined to be a pair consisting of a nonterminal symbol from N (its "root") to be obtained by $lhs(q)$ and a sequence of parses (to be obtained by $rhs(q)$) such that $lhs(q)$ and the "roots" of $rhs(q)$ constitute a production from Prods. In this context,

the auxiliary function *symbs*, given a sequence of parses, returns the sequence of symbols at the "roots" of the parses (where $\Delta$ denotes "sequential conjunction"). Thus, intuitively, parses are parse trees represented by "bracketed strings".

Subsequently, all functions will be uniformly defined on P* (rather than P), in order to avoid unnecessary distinctions between elements of P and P*, respectively.

Our ultimate goal are functions to solve the recognition and parsing problems. A suitable "inverse" of such functions is a function *unparse* that, given an element of P*, returns the sequence of terminal symbols composed of all leaves ("from left to right") of the parses in the sequence:

$$\textit{unparse}: P^* \rightarrow T^*$$

$$\textit{unparse}(p) =_{\text{def}} \begin{cases} \varepsilon, \text{ if } p = \varepsilon \\ \textit{unparse}(p_l) \dashv\!\vdash q \dashv\!\vdash \textit{unparse}(p_r), \\ \quad \text{if } p \neq \varepsilon \;\Delta\; p_l \dashv\!\vdash q \dashv\!\vdash p_r = p \;\Delta\; q \in T \\ \textit{unparse}(p_l) \dashv\!\vdash \textit{unparse}(\textit{rhs}(q)) \dashv\!\vdash \textit{unparse}(p_r), \\ \quad \text{if } p \neq \varepsilon \;\Delta\; p_l \dashv\!\vdash q \dashv\!\vdash p_r = p \;\Delta\; q \in Nont \end{cases}$$

Now we have all prerequisites to formally define the problem(s) we want to deal with. Using the above definitions, the recognition problem can be specified (as an "inverse problem") by

$$\textit{rec}: T^* \rightarrow \text{Bool}$$
$$\textit{rec}(w) =_{\text{def}} \exists\, p \in P: \textit{unparse}(\langle p \rangle) = w \wedge \textit{symbs}(\langle p \rangle) = \langle Z \rangle$$

and the parsing problem by

$$\textit{parse}: T^* \rightarrow (P \mid \textbf{dummy})$$
$$\textit{parse}(w) =_{\text{def}}$$
**if** $\textit{rec}(w)$ **then some** $p \in P$: $\textit{unparse}(\langle p \rangle) = w \wedge \textit{symbs}(\langle p \rangle) = \langle Z \rangle$
**else dummy fi.**

Although, maybe intuitively obvious, we should give a formal account of the adequacy of our specification, i.e., we should prove that

$$\textit{rec}(w) \equiv \langle Z \rangle \Rightarrow^* w$$

holds.

To this end, we introduce an auxiliary function

$$\textit{der}: P^* \times P^* \rightarrow \text{Bool}$$
$$\textit{der}(u, v) =_{\text{def}} \textit{symbs}(u) \Rightarrow^* \textit{symbs}(v)$$

and prove (see appendix) that the following property holds for arbitrary $w \in T^*$ and $p \in P^*$:

(2) $\quad \textit{unparse}(p) = w \equiv \textit{der}(p, w).$

With this property the proof for adequacy is a straightforward calculation:

$\textit{rec}(w)$
$\quad \equiv$ [ def. of *rec* ]
$\exists\, p \in P$: $\textit{unparse}(\langle p \rangle) = w \wedge \textit{symbs}(\langle p \rangle) = \langle Z \rangle$

$\equiv$ [ property (2) ]

$\exists\, p \in P^*: der(p, w) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ def. of *der* ]

$\exists\, p \in P^*: symbs(p) \Rightarrow^* symbs(w) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ substitutivity; def. of *symbs* ]

$\exists\, p \in P^*: \langle Z \rangle \Rightarrow^* w$

$\equiv$ [ $\exists$-simplification]

$\langle Z \rangle \Rightarrow^* w$

### 3.2 The formal development

Based on the formal problem specification as given in the previous section, we now aim at deriving an algorithm, basically by "inverting" the function *unparse*.

#### *3.2.1 Auxiliary operations*

The basic strategy to achieve a synthesis of an "inverse" of *unparse* tries to make use of "local inverses", i.e., the alternatives of the definition of *unparse* applied from right to left. For the first and second alternative, this is straightforward. For the third alternative, we introduce an auxiliary operation (as a kind of "inverse" corresponding to $unparse(p_l \dashv\!\vdash q \dashv\!\vdash p_r) = unparse(p_l) \dashv\!\vdash rhs(q) \dashv\!\vdash unparse(p_r)$)

$red$: ($P^*\ p \times P^*\ p'$: $isred(p)$) $\rightarrow$ Bool,

$red(p, p') =_{def} \exists\, n \in N, p_l, p_r, q \in P^*: p_l \dashv\!\vdash q \dashv\!\vdash p_r = p\ \Delta\ (n, symbs(q)) \in Prods\ \Delta$
$p' = p_l \dashv\!\vdash (n, q) \dashv\!\vdash p_r$

**where**

$isred$: $P^* \rightarrow$ Bool,

$isred(p) =_{def} \exists\, n \in N, p_l, p_r, q \in P^*: p_l \dashv\!\vdash q \dashv\!\vdash p_r = p\ \Delta\ (n, symbs(q)) \in Prods.$

For our subsequent development, some properties (of *der*, *red* and *isred* for all $p, u, u' \in P^*$) will be needed:

(3) $\neg isred(u) \vdash (der(p, u) \equiv (symbs(p) = symbs(u)))$
(proof: definition of *der*, property of $\Rightarrow^*$; assumption $\neg isred(u)$)

(4) $isred(u) \wedge der(p, u) \wedge p \neq u \Rightarrow \exists\, u' \in P^*: der(p, u') \wedge red(u, u')$)
(proof: transitivity of *der*)

(5) $(symbs(u) = \langle Z \rangle) \Rightarrow \neg isred(u)$
(proof: no chain productions in G)

(6) For $p, p'$ with $red(p, p') \equiv$ **true**, we have:
$(|p'| < |p|) \vee (|p'| = |p| \wedge \#_T(p') < \#_T(p))$
(where $\#_T(p)$ denotes the number of terminal symbols in $p$)
(proof: no chain and $\varepsilon$-productions in G)

*3.2.2 The formal development proper*

Our focus of interest is the function *rec*:

$rec(w) =_{def} \exists\, p \in \mathrm{P}: unparse(\langle p \rangle) = w \wedge symbs(\langle p \rangle) = \langle Z \rangle.$

By a simple *generalization* (replacing elements by singleton sequences) we obtain:

$rec(w) =_{def} \exists\, p \in \mathrm{P}^*: unparse(p) = w \wedge symbs(p) = \langle Z \rangle.$

Next, we use property (2):

$rec(w) =_{def} \exists\, p \in \mathrm{P}^*: der(p, w) \wedge symbs(p) = \langle Z \rangle.$

The next crucial step is to find an appropriate *embedding* (cf. [Partsch 90]). This step is not motivated in itself, but rather by the failure of an attempt to directly transform this specification into an algorithm using the "generalized unfold-fold strategy" from [Partsch 90]. Technically, this embedding makes use of the property $w \equiv \varepsilon \dashv\!\vdash w$ and turns the constant $\varepsilon$ into an additional parameter by means of an *abstraction* step. Moreover, it introduces an assertion to formalize the relationship between the old and the new parameters:

$rec(w) =_{def} r(\varepsilon, w)$ **where**

$r: (\mathrm{P}^* u \times \mathrm{T}^* s: unparse(u) \dashv\!\vdash s = w) \to \mathrm{Bool},$

$r(u, s) =_{def} \exists\, p \in \mathrm{P}^*: der(p, u \dashv\!\vdash s) \wedge symbs(p) = \langle Z \rangle.$

Now the function *r* becomes our new focus of interest and the remainder of the development proceeds along the "generalized unfold-fold strategy".

First, we introduce a *case distinction* which is guided by the type and the properties of the parameters. For the parameter *s* of type sequence we know that $(s = \varepsilon \vee s \neq \varepsilon)$ is a tautology. For the parameter *u*, $\varepsilon$ is the value we started from (in the embedding). Thus, a similar discrimination using $(u = \varepsilon \vee u \neq \varepsilon)$ is certainly doomed to fail. Therefore, we rather use the tautology $\neg isred(u) \vee isred(u)$. By combination of the two tautologies we come up with the following case distinction:

$(s = \varepsilon \wedge \neg isred(u)\ \Delta \exists\, p \in \mathrm{P}^*: der(p, u \dashv\!\vdash s) \wedge symbs(p) = \langle Z \rangle) \vee$

$(s \neq \varepsilon\ \Delta \exists\, p \in \mathrm{P}^*: der(p, u \dashv\!\vdash s) \wedge symbs(p) = \langle Z \rangle) \vee$

$(isred(u)\ \Delta \exists\, p \in \mathrm{P}^*: der(p, u \dashv\!\vdash s) \wedge symbs(p) = \langle Z \rangle)$

Our next efforts aim at simplifying the individual disjuncts. We reason as follows:

$s = \varepsilon\ \Delta \neg isred(u) \vdash$

$\exists\, p \in \mathrm{P}^*: der(p, u \dashv\!\vdash s) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ substitutivity using premise $s = \varepsilon$; neutrality of $\varepsilon$ w.r.t. $\dashv\!\vdash$ ]

$\exists\, p \in \mathrm{P}^*: der(p, u) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ property (3) ]

$\exists\, p \in \mathrm{P}^*: symbs(p) = symbs(u) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ commutativity and transitivity of = ]

$\exists\, p \in P^*$: $symbs(u) = \langle Z \rangle$

$\equiv$ [ $\exists$-simplification ]

$symbs(u) = \langle Z \rangle$.

Thus, for the first disjunct, we obtain as an intermediate result of the simplification

$s = \varepsilon \;\Delta\; \neg isred(u) \;\Delta\; symbs(u) = \langle Z \rangle$

which can be further simplified (using property (5) and $(B \Rightarrow A) \vdash (A \,\Delta\, B) \equiv B$) into:

$s = \varepsilon \;\Delta\; symbs(u) = \langle Z \rangle$

$s \neq \varepsilon \vdash$

$\exists\, p \in P^*$: $der(p, u \dashv\!\!\dashv s) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ $s \neq \varepsilon \vdash s = s_1 \dashv\!\!\dashv \overline{s}$; associativity of $\dashv\!\!\dashv$ ]

$\exists\, p \in P^*$: $der(p, (u \dashv\!\!\dashv s_1) \dashv\!\!\dashv \overline{s}) \wedge symbs(p) = \langle Z \rangle$

$isred(u) \vdash$

$\exists\, p \in P^*$: $der(p, u \dashv\!\!\dashv s) \wedge symbs(p) = \langle Z \rangle$

$\equiv$ [ property (4)]

$isred(u) \;\Delta\; \exists\, u' \in P^*$: $red(u, u') \;\Delta\; \exists\, p \in P^*$: $der(p, u' \dashv\!\!\dashv s) \wedge symbs(p) = \langle Z \rangle$

Hence, the overall result of the simplifications is:

$(s = \varepsilon \;\Delta\; symbs(u) = \langle Z \rangle) \vee$

$(s \neq \varepsilon \;\Delta\; \exists\, p \in P^*$: $der(p, (u \dashv\!\!\dashv s_1) \dashv\!\!\dashv \overline{s}) \wedge symbs(p) = \langle Z \rangle) \vee$

$(isred(u) \;\Delta\; \exists\, u' \in P^*$: $red(u, u') \;\Delta\; \exists\, p \in P^*$: $der(p, u' \dashv\!\!\dashv s) \wedge symbs(p) = \langle Z \rangle)$.

Now we realize that folding is possible in the second and third disjunct which results in

$(s = \varepsilon \;\Delta\; symbs(u) = \langle Z \rangle) \vee$

$(s \neq \varepsilon \;\Delta\; r(u \dashv\!\!\dashv s_1, \overline{s}\,) \vee$

$(isred(u) \;\Delta\; \exists\, u' \in P^*$: $red(u, u') \;\Delta\; r(u', s)$.

Since, however, $r$ was defined to be a partial function, we also have to prove that its assertion holds for the generated recursive calls. But this is simple, since the following properties obviously hold:

$unparse(u) \dashv\!\!\dashv s = w \wedge s \neq \varepsilon \vdash unparse(u \dashv\!\!\dashv s_1) \dashv\!\!\dashv \overline{s} = w$;

$unparse(u) \dashv\!\!\dashv s = w \wedge isred(u) \;\Delta\; \exists\, u' \in P^*$: $red(u, u') \vdash unparse(u') \dashv\!\!\dashv s = w$.

Moreover, we have to ensure the correctness of folding, e.g., by means of an additional proof of termination. But this again is simple, since either the length of $s$ is reduced or $u$ becomes "smaller" according to property (6).

### 3.3 Final remarks

Obviously, the result of our derivation can be specialized to the classical "shift-reduce"-recognition algorithm (with backtracking) by using particular versions ($p_2 = \varepsilon$) of *isred* and *red*:

$isred'$: P* $\rightarrow$ Bool,

$isred'(p) =_{def} \exists\, n \in \mathrm{N},\ p_l, q \in \mathrm{P}^*: p_l \dashv\!\vdash q = p\ \Delta\ (n, symbs(q)) \in Prods$

$red'$: (P* $p \times$ P* $p'$: $isred'(p)$) $\rightarrow$ Bool,

$red'(p, p') =_{def} \exists\, n \in \mathrm{N},\ p_l, q \in \mathrm{P}^*: p_l \dashv\!\vdash q = p\ \Delta\ (n, symbs(q)) \in Prods\ \Delta$
$p' = p_l \dashv\!\vdash (n, q)$

Also, a final transition to a functional or logic program (as in section 2) is straightforward for the recognition problem, since definition of logical disjunction implicitly takes care of backtracking. The same holds for a logic program for *parse* (where on termination, according to the assertion in $r$, $u$ yields the requested parse structure). For a functional program for *parse*, however, we have to explicitly take care of backtracking. Technically, this can be done by strengthening the assertion of $r$ to

$(s = \varepsilon\ \Delta\ symbs(u) = \langle Z \rangle) \Rightarrow unparse(u) = w$

and using the rule "argument on termination" from [Partsch 90].

## Acknowledgement

Many thanks to K. Achatz, A. Mößle, B. Rumpe, and T. Vullinghs for their critical remarks on earlier versions of these derivations.

## References

[Bauer et al. 89]
Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: Programming by formal reasoning - computer-aided intuition-guided programming. IEEE Transactions on Software Engineering **15**:2, 165-180 (1989)

[Boiten et al. 92]
Boiten, E.A., Partsch, H.A., Tuijnman, D., Völker, N.: How to produce correct software - an introduction to formal specification and program development by transformations. The Computer Journal **35**:6, 547-554 (1992)

[Partsch 86]
Partsch, H.: Transformational program development in a particular problem domain. Science of Computer Programming **7**:2, 99-241 (1986)

[Partsch 90]
Partsch, H.A.: Specification and transformation of programs - a formal approach to software development. Berlin: Springer 1990

# Appendix

## Property

(2) $unparse(p) = w \equiv der(p, w)$.

**proof** (by induction on the length of $p$):

a) base case. $p = \varepsilon$:

$unparse(\varepsilon) = w \equiv \varepsilon = w \equiv \varepsilon \Rightarrow^* w \equiv symbs(\varepsilon) \Rightarrow^* w \equiv der(\varepsilon, w)$

b) induction step: let $p = q \dashv\!\vdash p'$

b1) $q \in$ T:

$unparse(q \dashv\!\vdash p') = w$

$\equiv$ [ def. of *unparse*; $w = w_1 \dashv\!\vdash \overline{w}$ ]

$q \dashv\!\vdash unparse(p') = w_1 \dashv\!\vdash \overline{w}$

$\equiv$ [ def. of =]

$q = w_1 \wedge unparse(p') = \overline{w}$

$\equiv$ [ induction hypothesis ]

$q = w_1 \wedge der(p', \overline{w})$

$\equiv$ [ def. *der* ]

$q = w_1 \wedge symbs(p') \Rightarrow^* symbs(\overline{w})$

$\equiv$ [ def. *symbs*; $q = w_1 \equiv q \Rightarrow^* w_1$; property (2) ]

$symbs(q \dashv\!\vdash p') \Rightarrow^* symbs(w_1 \dashv\!\vdash \overline{w})$

$\equiv$ [ def. *der*; $w = w_1 \dashv\!\vdash \overline{w}$ ]

$der(q \dashv\!\vdash p', w)$

b2) $q \in$ Nont:

$unparse(q \dashv\!\vdash p') = w$

$\equiv$ [ def. of *unparse* ]

$unparse(rhs(q)) \dashv\!\vdash unparse(p') = w$

$\equiv$ [ def. of =; $q \in$ Nont $\vdash unparse(rhs(q)) \equiv unparse(q)$ ]

$\exists\, w', w'': w = w' \dashv\!\vdash w'' \wedge unparse(q) = w' \wedge unparse(p') = w''$

$\equiv$ [ induction hypothesis ]

$\exists\, w', w'': w = w' \dashv\!\vdash w'' \wedge der(q, w') \wedge der(p', w'')$

$\equiv$ [ def. *der* ]

$\exists\, w', w'': w = w' \dashv\!\vdash w'' \wedge symbs(q) \Rightarrow^* symbs(w') \wedge symbs(p') \Rightarrow^* symbs(w'')$

$\equiv$ [ def. *symbs* ]

$\exists\, w', w'': w = w' \dashv\!\vdash w'' \wedge lhs(q) \Rightarrow^* symbs(w') \wedge symbs(p') \Rightarrow^* symbs(w'')$

$\equiv$ [ property (2) ]

$\exists\, w', w'': w = w' \dashv\!\vdash w'' \wedge lhs(q) \dashv\!\vdash symbs(p') \Rightarrow^* symbs(w') \dashv\!\vdash symbs(w'')$

$\equiv$ [ def. *symbs*]

$\exists\, w', w'' : w = w' +\!\!+ w'' \wedge symbs(q +\!\!+ p') \Rightarrow^* symbs(w' +\!\!+ w'')$

$\equiv$ [ def. *der*; $\exists$-elimination ]

$der(q +\!\!+ p', w)$

(end of proof)

# Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

# List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03 *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06 *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07 *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluationof Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04 *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05 *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06 *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07 *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08 *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09 *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme

94-17 *Helmuth Partsch*
Again on Recognition and Parsin of Context-Free Grammars: Two Exercises in Transformational Programming

Ulmer Informatik-Berichte

ISSN 0939-5091

Herausgeber: Fakultät für Informatik

Universität Ulm, Oberer Eselsberg, D-89069 Ulm