RWTH Aachen

Department of Computer Science
Technical Report

Syntax-Directed Derivative Code (Part II: Intraprocedural Adjoint Code)

Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2005-17

RWTH Aachen · Department of Computer Science · August 2005

Syntax-Directed Derivative Code (Part II: Intraprocedural Adjoint Code)

Uwe Naumann

LuFG Software and Tools for Computational Engineering, Department of Computer Science
RWTH Aachen University, D-52056 Aachen Germany
WWW: `http://www.stce.rwth-aachen.de`, Email: `naumann@stce.rwth-aachen.de`

Abstract. This is the second instance in a series of papers on single-pass generation of derivative codes by syntax-directed translation. We consider the automatic generation of adjoint code by reverse mode automatic differentiation implemented as the bottom-up propagation of synthesized attributes on the abstract syntax tree. A proof-of-concept implementation is presented based on a simple LALR(1) parser generated by the parser generator `bison`. The approach offers all advantages of adjoint codes while exhibiting the highly desirable ease of implementation.

1 Motivation and Summary of Results

In this paper we present a method for generating *adjoint* versions of numerical simulation programs that implement vector functions

$$F : I\!R^n \rightarrow I\!R^m, \quad \mathbf{y} = F(\mathbf{x}), \quad \mathbf{x} = (x_k)_{k=1,\ldots,n}, \; \mathbf{y} = (y_l)_{l=1,\ldots,m}, \tag{1}$$

automatically by syntax-directed translation. The resulting adjoint programs $\bar{F} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}})$ compute adjoints $\bar{\mathbf{x}}$, that is, products of the transposed of the Jacobian matrix

$$F' = (f'_{l,k})_{k=1,\ldots,n}^{l=1,\ldots,m} \equiv \left(\frac{\partial y_l}{\partial x_k}\right)_{k=1,\ldots,n}^{l=1,\ldots,m} \in I\!R^{m\times n} \tag{2}$$

with a direction $\bar{\mathbf{y}}$ in the output space $I\!R^m$. Formally,

$$\bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) \equiv (F')^T \cdot \bar{\mathbf{y}} \quad . \tag{3}$$

To motivate the requirement for adjoint codes we discuss a simple parameter estimation problem as the solution of a non-linear least-squares optimization problem. Consider a numerical simulation program for a mathematical model $\mathbf{y} = F(\mathbf{x}, \mathbf{p})$ where $\mathbf{x} \in I\!R^{n_1}$, $\mathbf{p} \in I\!R^{n_2}$, and $\mathbf{y} \in I\!R^m$. For given measurements $(\mathbf{x}^j, \mathbf{y}^j), j = 1, \ldots, k$, we define the residual function

$$r_i(\mathbf{p}) \equiv \mathbf{y}^i - F(\mathbf{x}^i, \mathbf{p}), \quad i = 1, \ldots, k \quad .$$

Informally, our objective is to adapt the parameters $\mathbf{p}$ such that the data is represented by the model in the best possible way. This simple kind of parameter estimation can be performed by solving the nonlinear least-squares problem

$$\text{minimize} \quad g(\mathbf{p}) \equiv \frac{1}{2} r^T(\mathbf{p}) r(\mathbf{p}) \quad ,$$

for example, using steepest descent

$$\mathbf{p}^{j+1} = \mathbf{p}^j - \alpha_j \nabla g(\mathbf{p}^j)$$

to minimize some norm of the residual. Note, that

$$\nabla g(\mathbf{p}^j) = (r'(\mathbf{p}^j))^T r(\mathbf{p}^j)$$

can be computed by an adjoint model at a small constant multiple of the cost for evaluating r itself. A line-search for α_j, that is

$$\text{minimize} \quad g(\alpha_j) \quad \big(\text{for given } \mathbf{p}^j \text{ and } \nabla g(\mathbf{p}^j)\big)$$

is performed at each step j by applying Newton's method to $g'(\alpha_j) = 0$ as

$$\alpha_j^{k+1} = \alpha_j^k - \frac{g'(\alpha_j^k)}{g''(\alpha_j^k)} \quad .$$

The scalar first and second derivatives of the univariate scalar function $g(\alpha)$ can be computed via a univariate Taylor model [GUW00]. The latter can be generated easily by making simple modifications to the syntax-directed tangent-linear code generator from part one of this series of papers [Nau05].

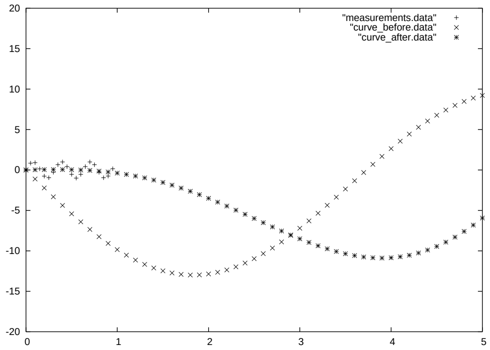
As an example we consider the very simple model[1]

$$y = \sum_{i=0}^{\nu-1} p_i \sin(p_{\nu+i} \cdot x) \tag{4}$$

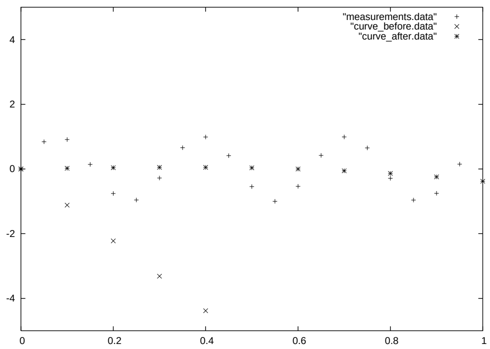
for $\mathbf{p} \in I\!R^n$, $n = 2\nu$, and $x, y \in I\!R$. We set $\nu = 10$, $p_i = i + 1$, and $p_{\nu+i} = \cos(i+1)$. Suppose that we have n measurements available for the interval $[0, 1)$ evenly distributed according to $\mathbf{y}_i = \sin(n \cdot i)$. The graph of the function and the measurements are plotted in Figure 1(a) using $\times$ and $+$, respectively. Running the optimization procedure leads to a new set of parameters resulting in the fitted curve that is shown as a sequence of $*$ symbols in Figure 1(a). Figure 1(b) shows all three curves over the interval of interest.

There are at least three different approaches to computing the gradient of g with respect to $\mathbf{p}$. One can approximate its values by finite difference quotients or use a tangent-linear model that can be generated by syntax-directed translation as described in [Nau05]. In both of these cases, the computational complexity is of the order of n as either each of the inputs needs to be perturbed separately or n directional derivatives need to be computed. Alternatively, an adjoint model can give us ∇g at a computational complexity of order $m = 1$ plus some overhead for reversing the control and data flow that heavily depends on the method used and on the computational platform. We ran a little experiment with $\nu = 10^4$ to test the actual run time differences. The result is summarized in the following table.

[1] We use this function as a model of a person attempting to walk along a "straight line" under the influence of too much alcohol during a motivational lecture on "Adjoints by Source Transformation." The example proved to be useful for getting students interested in the subject...



(a)



(b)

Fig. 1. Curve Fitting through Parameter Estimation

ν	FDC	TLC	ADC
10^4	45 sec	43 sec	$<$ 1 sec
10^5	$>$ 1 h	$>$ 1 h	$<$ 1 sec

The codes can be downloaded for verifying experiments from the project's website. The obvious conclusion from these simple experiments is that adjoint codes must play a crucial role in modern numerical analysis such as in optimization or in the context of numerical inverse problems in general. As numerical simulation programs for real-world applications are not as simple as Equation (4), it is highly desirable to generate adjoint codes automatically. Various tools for *automatic differentiation* (AD; see [Gri00]) including ADIFOR 3 [CF00], ADOL-C [GJU96], the differentiation-enabled NAGWare Fortran 95 compiler [NR05], OpenAD [WHH$^+$05], TAF [GK03], and TAPENADE [HP04] have been developed over the past decades that help scientists and engineers to handle much larger dimensions – both in terms of the code size and of the dimension of the parameter space – as used to be possible with classical numerical differentiation by finite differences or by hand-coding derivative codes.

In this paper we present a single-pass approach to the automatic generation of adjoint code by syntax-directed translation. The method is both elegant and easy to implement. A disadvantage is the inability to perform data flow analysis due to the missing intermediate representation. Nevertheless, a syntax-directed adjoint code generator may serve as a support tool in the context of semi-automatic development of adjoint codes, and may even represent a good starting point for the development of more sophisticated source transformation tools for AD. Moreover, it could represent a useful extension of the first pass of existing tools to generate an intermediate representation of the program that is easier to optimize by state-of-the-art compiler algorithms.

The structure of the paper is as follows. In Section (2) we summarize the theoretical concepts behind reverse mode AD in the context of adjoint code generation by source transformation. The syntax-directed translation algorithm for straight-line programs is introduced in Section (3) and generalized to subroutines with control-flow structures in Section (4). A simple proof-of-concept implementation is discussed in Section (5). We make detailed references to the source code that is appended in Section (A). We draw conclusions in Section (6) together with an outlook to syntax-directed adjoint code generation in the presence of interprocedural flow of control.

2 Fundamentals of Adjoint Codes

Let the subroutine / user-defined function $\mathbf{y} = F(\mathbf{x})$ implement a vector function as defined in Equation (1). The values of the m *dependent* variables y_j, $j = 1, \ldots, m$, are calculated as functions of the n *independent* variables x_i, $i = 1, \ldots, n$. The subroutine F represents an implementation of the mathematical model for some underlying real-world application and it will be referred to as the *forward code.* The forward code is expected to be written in some high-level imperative programming language such as C or Fortran.[2] More general, it should be possible to decompose F into a sequence of scalar assignments of the form

$$v_j = \varphi_j(v_k)_{k \prec j} \ , \quad j = 1, \ldots, q \ , \tag{5}$$

(referred to as the *code list* in [Nau05]) where $q = p + m$ and such that the result of every intrinsic function and elementary arithmetic operation is assigned to a unique intermediate variable v_j, $j = 1, \ldots, q$. Following the notation in [Gri00] we write $k \prec j$ whenever some variable v_j depends directly on another variable v_k. The code list induces a directed acyclic *computational graph* $G = (V, E)$ with integer vertices $V = \{1 - n, \ldots, q\}$ and edges $E = \{(i, j) : i \prec j\}$ as shown, for example, in [GR91]. It is assumed that the local partial derivatives

$$c_{ji} = \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \tag{6}$$

of the *elemental* functions φ_j, $j = 1, \ldots, q$, exist and that they are jointly continuous in some open neighborhood of the current argument $(v_k)_{k \prec j}$. In this case, an augmented version of the forward code can be implemented that computes F itself and the set of all local partial derivatives as defined in Equation (6). As in [Nau05] we refer to this augmented forward code as the *linearized code list.* The *linearized computational graph* is obtained by attaching the local partial derivatives to the corresponding edges.

Example

Consider

$$\begin{aligned} x &= x \cdot \sin(x \cdot y) \\ y &= x \cdot y \\ x &= \sin(x) \end{aligned} \tag{7}$$

The linearized computational graph is shown in Figure 2.

The reverse mode of AD (see [Gri00, Sect. 3.3] for a more complete coverage of the theoretical foundation of this method) uses these local partial derivatives to propagate adjoints $\bar{v}_j$ backwards for $j = q, \ldots, 1 - n$ with respect to the data flow of the forward code from the outputs to the inputs. Products of the

[2] As in [Nau05] and without loss of generality, our proof-of-concept implementation `sdac` (see Section (5)) focuses on a subset of C.

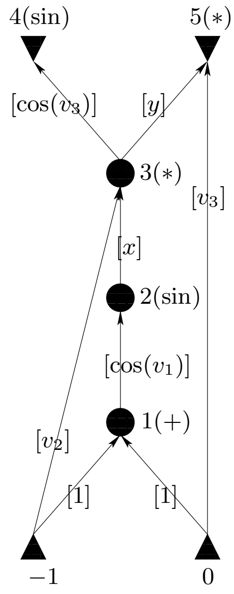


Fig. 2. Linearized Computational Graph: Intermediate and dependent vertices are marked with the corresponding elemental functions (in "(...)"). Expressions for the local partial derivatives are attached to the edges (in "[...]"). Initialization of $v_{-1} \equiv x$ and $v_0 \equiv y$ allows for the computation of all intermediate values $v_1, \ldots, v_5$ and the computation of values for the local partial derivatives.

transposed Jacobian matrix with a vector of adjoints of the outputs are computed by initializing the adjoints of the dependent variables $\bar{y}_j \equiv \bar{v}_{p+j}$, $j = 1, \ldots, m$.

In practical implementations we distinguish between two fundamental approaches to reverse mode. In its non-incremental version the local partial derivatives are computed during the augmented forward evaluation (first line in Equation (8)).

$$\begin{array}{l} v_j = \varphi_j(v_i)_{i \prec j};\ c_{j,i} = \dfrac{\partial \varphi_j}{\partial v_i} \quad \text{for } i \prec j \text{ and } j = 1, \ldots, q \\ \bar{v}_j = \displaystyle\sum_{k: j \prec k} c_{kj} \cdot \bar{v}_k \quad \text{for } j = q, \ldots, 1-n \end{array} . \tag{8}$$

In incremental reverse mode the local partial derivatives are computed during the adjoint evaluation (second line in Equation (9)).

$$\begin{array}{l} v_j = \varphi_j(v_i)_{i \prec j};\ \bar{v}_j = 0 \quad \text{for } j = 1, \ldots, q \\ c_{j,i} = \dfrac{\partial \varphi_j}{\partial v_i};\ \bar{v}_i = \bar{v}_i + c_{j,i} \cdot \bar{v}_j \quad \text{for } i \prec j \text{ and } j = q, \ldots, 1 \end{array} . \tag{9}$$

One can think of various combinations of both methods. The Jacobian $F'(\mathbf{x})$ is accumulated by reverse propagation of the Cartesian basis vectors in $I\!R^m$ at a complexity of $O(m)$. In particular, gradients of single dependent variables with respect to all independent variables can be obtained at a computational cost that is a small multiple of the cost of running the forward code.

In practice (from the viewpoint of a source transformation AD tool developer) the incremental form of reverse mode AD is preferred as it is better suited for state-of-the-art parsing algorithms and traversals of the abstract syntax tree (see Section (3) and established compiler literature such as [ASU86]). Moreover, the explicit construction of the code list is impossible because of control flow statements in the forward code. The theoretical concepts can be applied without change only to parts of the code whose data flow structure is statically known at compile time such as single assignments or sequences thereof (also known as *basic blocks*). In Section (3) we build code lists of single assignments to propagate adjoints. However, first a few general remarks should be made to facilitate a better understanding of the source transformation approach.

Consider an arbitrary assignment of the form

$$u_j = \varphi(u_i)_{i \prec j} \tag{10}$$

where, possibly, $\&u_j \in \{\&u_i : i \prec j\}$ as well as $\&u_{i_1} = \&u_{i_2}$ for $i_1 \prec j$ and $i_2 \prec j$. We use $\&u$ to denote the memory address referred to by a variable u. The first assignment in Equation (7) is a good example.

In general, it is undecidable at compile time if $\&u_{i_1} = \&u_{i_2}$ as this may depend on parameters that are only available at run time. Alias analysis [Muc97] may help. Lacking any program analysis one needs to assume conservatively that $\&u_j = \&u_i$ for all $i \prec j$ to ensure the correctness of the adjoint code.

As a consequence of overwriting a given location in memory may represent different code list variables. For example, x corresponds to v_{-1}, v_3, and v_4 in Equation (7) and Figure 2. Note that the adjoints of all code list variables need to be initialized with zero to ensure the correctness of the incremental reverse mode. The adjoint of u_j in Equation (10) dies (its value is no longer used) once it has been used to increment the adjoints of all arguments of φ. Referring to Figure 2 the value of $\bar{v}_3$ is dead once it has been used to increment $\bar{v}_2$ and $\bar{v}_{-1}$. However, the memory location $\&u_j$ may well be incremented by some succeeding adjoint statement for the reasons stated above. Hence, adjoint variables need to be reset to zero immediately after their death.

If we can prove that some $\&u_j$ is always read only once after its initialization and before getting overwritten, then its adjoint does not need to be initialized with zero and all adjoint statements that have $\bar{u}_j$ on the left-hand side simply overwrite its value. The restriction to code lists of single assignments ensures that the above requirement is satisfied. Hence, no assignment to an adjoint code list variable is in incremental form nor does an adjoint code list variable need to be reset to zero after its death.

Example

The simple syntax-directed adjoint code compiler `sdac` (see Section (5)) takes this approach. Listing 1.1 shows the adjoint code that is generated for the forward code in Equation (7). For example, the first statement is decomposed into a code list in lines 7 – 13. The code list variables are stored on a stack as they may potentially be overwritten by the code lists of succeeding assignment and at the same time they may be required to compute the partial derivatives of some preceding assignment. Only v1, v2, and v3 are overwritten and none of them is ever used by a preceding assignment. However, the data flow analysis that could detect such situations in general [HNP05] is not part of `sdac`. In any case the conservative approach ensures correctness.

In the adjoint section of the code (lines 21 – 34) all assignments to the adjoint code list variables (lines 21, 22, 24, 25, and 28–31) are non-incremental. The adjoint program variables x_ and y_ are always incremented when they occur on the left-hand side (lines 23, 26, 27, and 32 – 34). They are set to zero right after their values got assigned to an adjoint code list variable (lines 21, 24, 28). For example, setting x_=0 in line 21 ensures that the result of the incrementation in line 23 is numerically correct. The overall correctness of the code generated by `sdac` is verified against results obtained by running the tangent-linear code generated by `sdtlc` as described in Section (5).

Listing 1.1. Adjoint Code for Equation (7)

```
1  double v1, v1_;
2  double v2, v2_;
3  double v3, v3_;
4  double v4, v4_;
5  double v5, v5_;
6  double v6, v6_;
7  push(v1); v1=x;
8  push(v2); v2=x;
9  push(v3); v3=y;
10 push(v4); v4=v2*v3;
11 push(v5); v5=sin(v4);
12 push(v6); v6=v1*v5;
13 push(x); x=v6;
14 push(v1); v1=x;
15 push(v2); v2=y;
16 push(v3); v3=v1*v2;
17 push(y); y=v3;
18 push(v1); v1=x;
19 push(v2); v2=sin(v1);
20 push(x); x=v2;
21 pop(x); v2_=x_; x_=0;
22 pop(v2); v1_=cos(v1)*v2_;
23 pop(v1); x_+=v1_;
24 pop(y); v3_=y_; y_=0;
25 pop(v3); v1_=v3_*v2; v2_=v3_*v1;
26 pop(v2); y_+=v2_;
27 pop(v1); x_+=v1_;
28 pop(x); v6_=x_; x_=0;
29 pop(v6); v1_=v6_*v5; v5_=v6_*v1;
30 pop(v5); v4_=cos(v4)*v5_;
31 pop(v4); v2_=v4_*v3; v3_=v4_*v2;
32 pop(v3); y_+=v3_;
33 pop(v2); x_+=v2_;
34 pop(v1); x_+=v1_;
```

3 Adjoint Straight-Line Programs

As in [Nau05] we consider straight-line programs according to the following definition

Definition 1. *A* straight-line program (SLP) *is a sequence of scalar assignments described by the context-free grammar $G = (N, T, P, s)$ with nonterminal symbols*

$$N = \left\{ s \text{ (straight-line program)} \quad a \text{ (assignment)} \quad e \text{ (expression)} \right\}$$

terminal symbols

$$T = \left\{ \begin{array}{ll} V & (\textit{program variables; see line 14 in Appendix A.1, Listing 1.6}) \\ C & (\textit{constants; line 20}) \\ F & (\textit{unary intrinsic; line 13}) \\ O & (\textit{binary operator; line 26}) \\ \;,\;;\;)\;(& (\textit{remaining single character tokens; line 26}) \end{array} \right\}$$

start symbol s, and production rules

$$P = \left\{ \begin{array}{lll} (P1) & s :: a & (\textit{see line 28 in Appendix A.1, Listing 1.7}) \\ (P2) & s :: as & (\textit{line 29}) \\ (P3) & a :: V = e; & (\textit{line 40}) \\ (P4) & e :: V & (\textit{line 84}) \\ (P5) & e :: C & (\textit{line 95}) \\ (P6) & e :: F(e) & (\textit{line 73}) \\ (P7) & e :: eOe & (\textit{line 50 and line 62}) \end{array} \right\}$$

Any comments made in [Nau05] on the structure of this grammar, its use in the context of lexical and syntax analysis using `flex` and `bison`, and its sufficiency as a proof-of-concept implementation of the theoretical ideas presented in this paper apply in the current context as well. Again, we use an LALR(1)-parsing algorithm based on a push-down automaton with a characteristic finite automaton as in [Nau05, Equations (7) and (8)].

```
s (6)  →  a (5)
               |→  V("y")
               |→  =
               |→  e (4)
                        |→  F(" sin ")
                        |→  (
                        |→  e (3)
                                 |→  e (1)  →  V("x")
                                 |→  O(" * ")
                                 |→  e (2)  →  C("2")
                        |→  )
```

Fig. 3. Abstract Syntax tree of "$y = \sin(x * 2);$".

The production rules for syntax-directed compilation of adjoint SLPs are derived below. Both counters ν and c are initialized with one. Examples are provided based on the bottom-up derivation of the assignment "$y = \sin(x * 2);$" by performing the reductions **(P4)**, **(P5)**, **(P7)**, **(P6)**, **(P3)**, and **(P1)** to get the abstract syntax tree shown in Figure 3. The augmented forward section is synthesized in attribute $\mathbf{v}_\nu.a^f$ and the adjoint section in $\mathbf{v}_\nu.a^r$ for $\nu = 1, \ldots, 6$. Hence, the entire adjoint code consists of $\mathbf{v}_6.a^f$ followed by $\mathbf{v}_6.a^r$. Figure 4 shows the corresponding output as generated by `sdac`. Note that as a result of bottom-up parsing the examples provided with the extended reduction rules need to be read in the order of the reductions as performed by the parser, that is **(P4)**, **(P5)**, ..., **(P1)**.

We use syntax that is analogous to that used in [Nau05]. For example, $V.\bar{a}^f$ refers to the string that marks the adjoint of the variable marked by the string $V.a^f$, that is, if $V.a^f =$ "x", then $V.\bar{a}^f =$ "$\bar{x}$". In **(P6)** the symbolic transformation

$$\frac{\partial F.a}{\partial \text{ "}v_{[\mathbf{v}_{\nu-1}.j]}\text{"}}$$

is defined according to the differentiation rules of the elemental functions, for example, if $F.a =$ " cos " and "$v_{[\mathbf{v}_{\nu-1}.j]}$" $=$ "v_1", then

$$\frac{\partial F.a}{\partial \text{ "}v_{[\mathbf{v}_{\nu-1}.j]}\text{"}} = \frac{\partial \text{ " }\cos(v_1)\text{"}}{\partial \text{ "}v_1\text{"}} = \text{" } -\sin(v_1)\text{"} \quad .$$

(P1) $s :: a$

$$\begin{aligned} &\mathbf{v}_\nu.a^f = \mathbf{v}_{\nu-1}.a^f \\ &\mathbf{v}_\nu.a^r = \mathbf{v}_{\nu-1}.a^r \\ &\nu{+}{+} \end{aligned}$$

(See line 28 in Appendix A.1, Listing 1.7.)

Example: After the last reduction the entire augmented forward code has been synthesized in $\mathbf{v}_6.a^f$. The adjoint code is in $\mathbf{v}_6.a^r$.

$$\begin{aligned} \mathbf{v}_6.a^f = \text{"}&push(v_1);\ v_1 = x; \\ &push(v_2);\ v_2 = 2; \\ &push(v_3);\ v_3 = v_1 * v_2; \\ &push(v_4);\ v_4 = \sin(v_3); \\ &push(y);\ y = v_4;\text{"} \\ \mathbf{v}_6.a^r = \text{"}&pop(y);\ \bar{v}_4 = \bar{y};\ \bar{y} = 0; \\ &pop(v_4);\ \bar{v}_3 = \cos(v_3) * \bar{v}_4; \\ &pop(v_3);\ \bar{v}_1 = v_2 * \bar{v}_3;\ \bar{v}_2 = v_1 * \bar{v}_3; \\ &pop(v_2); \\ &pop(v_1);\ \bar{x}{+}{=}\bar{v}_1;\text{"} \end{aligned}$$

(P2) $s :: as$

$$\begin{aligned} &\mathbf{v}_\nu.a^f = \mathbf{v}_{\mu_1}.a^f + \mathbf{v}_{\mu_2}.a^f \\ &\mathbf{v}_\nu.a^r = \mathbf{v}_{\mu_2}.a^r + \mathbf{v}_{\mu_1}.a^r \\ &\qquad \text{where } \mathbf{v}_{\mu_1} \hat{=} a \text{ and } \mathbf{v}_{\mu_2} \hat{=} s \text{ in } as \\ &\nu{+}{+} \end{aligned}$$

(See lines 29–39.)

Example: This rule is not used as we are dealing with only one assignment. As in [Nau05] we use the notation "$\hat{=}$" in the sense of "corresponds to", that is, for example, $\mathbf{v}_{\mu_2}$ is the vertex in the abstract syntax tree that corresponds to the second non-terminal on the right-hand side of rule **(P2)**. The counter ν for the vertices in the abstract syntax tree is incremented by each reduction.

(P3) $a :: V = e;$

$$
\begin{aligned}
\mathbf{v}_\nu.a^f &= \mathbf{v}_{\nu-1}.a^f + \text{"}push\text{"} + \text{"(" } + V.a^f + \text{")"} + \text{";"} \\
&\quad + V.a^f + \text{" = "} + \text{"}v_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{";"} \\
\mathbf{v}_\nu.a^r &= \text{"}pop\text{"} + \text{"(" } + V.a^f + \text{")"} + \text{";"} \\
&\quad + \text{"}\bar{v}_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{" = "} + V.\bar{a}^f + \text{";"} \\
&\quad + V.\bar{a}^f + \text{" = 0;"} \\
&\quad + \mathbf{v}_{\nu-1}.a^r \\
c &= 1 \\
\nu&{+}{+}
\end{aligned}
$$

(See lines 40–49.)

Example: The augmented forward code is synthesized from the augmented forward code of the right-hand side ($\mathbf{v}_{\nu-1}.a^f$), the *push* statement that stores the overwritten value of the left-hand side (y), and the assignment of the value of the code list variable that holds the value of the right-hand side (v_4) to the variable on the left-hand side.

$$
\begin{aligned}
\mathbf{v}_5.a^f = \; & \text{"}push(v_1);\; v_1 = x; \\
& push(v_2);\; v_2 = 2; \\
& push(v_3);\; v_3 = v_1 * v_2; \\
& push(v_4);\; v_4 = \sin(v_3); \\
& push(y);\; y = v_4;\text{"}
\end{aligned}
$$

The adjoint code consists of a *pop* statement to restore the value of the variable on the left-hand side ($V.a^f$ = "y") followed by overwriting the adjoint of the code list variable that corresponds to the right-hand side ("$\bar{v}_{[\mathbf{v}_{\nu-1}.j]}$" = "$\bar{v}_4$") with the adjoint of the left-hand side ($V.\bar{a}^f$ = "$\bar{y}$"), the reinitialization of $V.\bar{a}^f$ with zero, and all this synthesized with the adjoint code of the right-hand side ($\mathbf{v}_{\nu-1}.a^r$). The data flow in the adjoint code is reversed with respect to that of the forward code. The code list counter c is reset to one in anticipation of the next statement-level code list to be built potentially (not in this simple example).

$$\begin{aligned}\mathbf{v}_5.a^r = \ & "pop(y);\ \bar{v}_4 = \bar{y};\ \bar{y} = 0;\\ & pop(v_4);\ \bar{v}_3 = \cos(v_3) * \bar{v}_4;\\ & pop(v_3);\ \bar{v}_1 = v_2 * \bar{v}_3;\ \bar{v}_2 = v_1 * \bar{v}_3;\\ & pop(v_2);\\ & pop(v_1);\ \bar{x}{+}{=}\bar{v}_1;"\\ \nu = 6 & \end{aligned}$$

(P4) $e :: V$

$$\begin{aligned}\mathbf{v}_\nu.j &= c{+}{+}\\ \mathbf{v}_\nu.a^f &= "push" + "(" + "v_{[\mathbf{v}_\nu.j]}" + ")" + ";"\\ &\quad + "v_{[\mathbf{v}_\nu.j]}" + " = " + V.a^f + ";"\\ \mathbf{v}_\nu.a^r &= "pop" + "(" + "v_{[\mathbf{v}_\nu.j]}" + ")" + ";"\\ &\quad + V.\bar{a}^f + "{+}{=}" + "\bar{v}_{[\mathbf{v}_\nu.j]}" + ";"\\ \nu{+}{+} & \end{aligned}$$

(See lines 84–94.)

Example: The value of the next code list variable "$v_{[\mathbf{v}_\nu.j]}$" is stored on the stack before the variable is overwritten with the value of the program variable $V.a^f$.

$$\begin{aligned}\mathbf{v}_1.j &= 1\\ c &= 2\\ \mathbf{v}_1.a^f &= "push(v_1);\ v_1 = x;"\end{aligned}$$

The value of the code list variable "$v_{[\mathbf{v}_\nu.j]}$" is restored and the adjoint $V.\bar{a}^f$ of the program variable is incremented with the adjoint of "$v_{[\mathbf{v}_\nu.j]}$", that is "$\bar{v}_{[\mathbf{v}_\nu.j]}$".

$$\begin{aligned}\mathbf{v}_1.a^r &= "pop(v_1);\ \bar{x}{+}{=}\bar{v}_1;"\\ \nu &= 2\end{aligned}$$

The value of a^r is the empty string for all terminal symbols, Hence, it can be omitted in the synthesis of $\mathbf{v}_\nu.a^r$.

```
double v1, v1_;
double v2, v2_;
double v3, v3_;
double v4, v4_;
push(v1); v1=x;
push(v2); v2=2;
push(v3); v3=v1*v2;
push(v4); v4=sin(v3);
push(y); y=v4;
pop(y); v4_=y_; y_=0;
pop(v4); v3_=cos(v3)*v4_;
pop(v3); v1_=v3_*v2; v2_=v3_*v1;
pop(v2);
pop(v1); x_+=v1_;
```

(a)

```
#include <stack>
using namespace std;

static stack<double> stack_v;

void push(double v) {
  stack_v.push(v);
}
void pop(double& v) {
  v=stack_v.top();
  stack_v.pop();
}
```

(b)

Fig. 4. (a) `sdac` output for "`y=sin(x*2);`" The correctness has been verified with `sdtlc` [Nau05] and finite difference approximation. (b) Implementation of stack.

(P5) $e :: C$

$$
\begin{aligned}
\mathbf{v}_\nu.j &= c{+}{+} \\
\mathbf{v}_\nu.a^f &= \text{"}push\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{"};\text{"} \\
&\quad + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"} = \text{"} + C.a^f + \text{"};\text{"} \\
\mathbf{v}_\nu.a^r &= \text{"}pop\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{"};\text{"} \\
\nu&{+}{+}
\end{aligned}
$$

(See lines 95–105.)

Example: The augmented forward code is analogous to that of **(P4)**.

$$
\begin{aligned}
\mathbf{v}_2.j &= 2 \\
c &= 3 \\
\mathbf{v}_2.a^f &= \text{"}push(v_2);\ v_2 = 2;\text{"}
\end{aligned}
$$

The adjoint code simply restores the value of the code list variable that was overwritten by the augmented forward code.

$$
\begin{aligned}
\mathbf{v}_2.a^r &= \text{"}pop(v_2);\text{"} \\
\nu &= 3
\end{aligned}
$$

(P6) $e :: F(e)$

$$
\begin{aligned}
\mathbf{v}_\nu.j &= c{+}{+}\\
\mathbf{v}_\nu.a^f &= \mathbf{v}_{\nu-1}.a^f\\
&\quad + \text{"}push\text{"} + \text{"(" } + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{")"} + \text{";"}\\
&\quad + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{" = "} + F.a^f + \text{"("} + \text{"}v_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{")"} + \text{";"}\\
\mathbf{v}_\nu.a^r &= \text{"}pop\text{"} + \text{"("} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{")"} + \text{";"}\\
&\quad + \text{"}\bar{v}_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{" = "} + \frac{\partial F.a^f}{\partial\text{"}v_{[\mathbf{v}_{\nu-1}.j]}\text{"}} + \text{" * "} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{";"}\\
&\quad + \mathbf{v}_{\nu-1}.a^r\\
\nu{+}{+}
\end{aligned}
$$

(See lines 73–83.)

Example: As before and succeeding the augmented forward code generated so far, the value of the expression that corresponds to the right-hand side of the production rule is assigned to the next code list variable "$v_{[\mathbf{v}_\nu.j]}$" whose old value needs to be stored before that.

$$
\begin{aligned}
\mathbf{v}_4.j &= 4\\
\mathbf{v}_4.a^f &= \text{"}push(v_1);\ v_1 = x;\\
&\qquad push(v_2);\ v_2 = 2;\\
&\qquad push(v_3);\ v_3 = v_1 * v_2;\\
&\qquad push(v_4);\ v_4 = \sin(v_3);\text{"}
\end{aligned}
$$

The adjoint code restores this value and it increments the adjoint of the code list variable that holds the value of the expression forming the argument of the unary intrinsic $F.a^f$ with the product of the corresponding local partial derivative and the adjoint of "$v_{[\mathbf{v}_\nu.j]}$".

$$
\begin{aligned}
\mathbf{v}_4.a^r &= \text{"}pop(v_4);\ \bar{v}_3 = \cos(v_3) * \bar{v}_4;\\
&\qquad pop(v_3);\ \bar{v}_1 = v_2 * \bar{v}_3;\ \bar{v}_2 = v_1 * \bar{v}_3;\\
&\qquad pop(v_2);\\
&\qquad pop(v_1);\ \bar{x} {+}{=} \bar{v}_1;\text{"}\\
\nu &= 5
\end{aligned}
$$

(P7) $e :: eOe$

$$
\begin{aligned}
\mathbf{v}_\nu.j &= c{+}{+}\\
\mathbf{v}_\nu.a^f &= \mathbf{v}_{\mu_1}.a^f + \mathbf{v}_{\mu_2}.a^f\\
&\quad + \text{"}push\text{"} + \text{"("} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{")"} + \text{";"}\\
&\quad + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{" = "} + \text{"}v_{[\mathbf{v}_{\mu_1}.j]}\text{"} + O.a^f + \text{"}v_{[\mathbf{v}_{\mu_2}.j]}\text{"} + \text{";"}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{v}_\nu.a^r = \text{"}pop\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{";"} \\
&\quad + \text{"}\bar{v}_{[\mathbf{v}_{\mu_1}.j]}\text{"} + \text{"} = \text{"} + \frac{\partial O.a^f}{\partial \text{"}v_{[\mathbf{v}_{\mu_1}.j]}\text{"}} + \text{"} * \text{"} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{";"} \\
&\quad + \text{"}\bar{v}_{[\mathbf{v}_{\mu_2}.j]}\text{"} + \text{"} = \text{"} + \frac{\partial O.a^f}{\partial \text{"}v_{[\mathbf{v}_{\mu_2}.j]}\text{"}} + \text{"} * \text{"} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{";"} \\
&\quad + \mathbf{v}_{\mu_1}.a^r + \mathbf{v}_{\mu_2}.a^r \\
&\qquad\qquad \text{where } \mathbf{v}_{\mu_1} \hat{=} e^1 \text{ and } \mathbf{v}_{\mu_2} \hat{=} e^2 \text{ in } e^1 O e^2 \\
&\nu{+}{+}
\end{aligned}
$$

(See lines 50–72.)

Example: The augmented forward code is analogous to that of **(P6)**.

$$
\begin{aligned}
\mathbf{v}_3.j &= 3 \\
\mathbf{v}_3.a^f &= \text{"}push(v_1);\ v_1 = x; \\
&\quad\ push(v_2);\ v_2 = 2; \\
&\quad\ push(v_3);\ v_3 = v_1 * v_2;\text{"}
\end{aligned}
$$

In the adjoint code the adjoints of both code list variables that store the values of the two arguments of the binary operator $O.a^f$ need to be incremented with the product of the corresponding local partial derivative and the adjoint of the code list variable that holds the value of the expression that is reduced according to the right-hand side of the production rule.

$$
\begin{aligned}
\mathbf{v}_3.a^r &= \text{"}pop(v_3);\ \bar{v}_1 = v_2 * \bar{v}_3;\ \bar{v}_2 = v_1 * \bar{v}_3; \\
&\quad\ pop(v_2); \\
&\quad\ pop(v_1);\ \bar{x} {+}{=} \bar{v}_1;\text{"} \\
\nu &= 4
\end{aligned}
$$

4 Adjoint Subroutines

As in [Nau05] we consider subroutines defined syntactically as follows.

Definition 2. *A* subroutine *is described by the following context-free grammar* $G = (N, T, P, r)$.

$$
N = \left\{ \begin{array}{ll} r \text{ (sequence of statements)} & s \text{ (statement)} \\ e \text{ (expression)} & c \text{ (condition)} \end{array} \right\}
$$

$$
T = \left\{ \begin{array}{rl} \vdots & \textbf{\textit{(see Definition 1)}} \\ IF & (\textit{unary intrinsic; see line 14 in Appendix A.2, Listing 1.9}) \\ WHILE & (\textit{binary operator; line 15}) \end{array} \right\}
$$

start symbol r*, and production rules*

$$P = \left\{ \begin{array}{lll} (P1) & r :: s & (\textit{see line 45 in Appendix A.2, Listing 1.10}) \\ (P2) & r :: sr & (\textit{line 46}) \\ (P3) & s :: V = e; & (\textit{line 122}) \\ & \vdots \ \textbf{\textit{(see Definition 1)}} & \\ (P8) & s :: IF(c)\{s\} & (\textit{see line 76 in Appendix A.2, Listing 1.10}) \\ (P9) & s :: WHILE(c)\{s\} & (\textit{line 96}) \\ (P10) & c :: V < V & (\textit{line 116}) \end{array} \right\}.$$

The presence of control-flow structures in a subroutine has a significant impact on the way the adjoint code is generated. In Section (3) we saw that the data-flow in the adjoint section of the code is reversed compared with that of the forward code (or, equivalently, the augmented forward section of the adjoint code). Hence, the flow of control needs to be reversed too as it defines the data flow between the basic blocks. Informally, loops need to be executed in reverse order and the same branches need to be executed both by the augmented forward and the adjoint section of the adjoint code. The obvious solution is to enumerate the basic blocks and to push their indexes onto a *control stack* during the evaluation of the augmented forward code. The adjoint code then simply restores the indexes of all basic blocks followed by the execution of the corresponding adjoint basic blocks. From Section (3) we know how to generate the latter. The stack that enables the reversal of the data flow by storing the values of overwritten variables is referred to as the *data stack*.

As for SLP's the first attribute a^f that is associated with all vertices in the AST is used to synthesize the augmented forward code. Due to the selected approach to the reversal of the flow of control the adjoint basic blocks need to be synthesized individually making the second attribute a^r a vector of length equal to the number of basic blocks in the subroutine.

In the following we present a set of extended shift and reduce actions that make the syntax-directed generation of adjoint code for entire subroutines as defined in Definition 2 work. We focus on the differences from the set of rules given in Section (3) by using " $\vdots$ " to avoid obvious duplication. We comment on the single rules without presenting examples as we did in Section (3). Instead an adjoint code generated automatically by `sdac` is discussed in Section (5).

(P1) $r :: s$

$$\begin{aligned} &\mathbf{v}_\nu.a^f = \mathbf{v}_{\nu-1}.a^f \\ &\mathbf{v}_\nu.a^r_i = \mathbf{v}_{\nu-1}.a^r_i \quad i = 0, \ldots, idxBB \\ &\nu{+}{+} \end{aligned}$$

The adjoints of all basic blocks that have been parsed so far need to be copied. Note that the requirement to synthesize all entries in a^r is restricted to vertices that may occur above assignments (see **(P3)**) in the AST, that is, AST vertices that are generated as the result of the reductions **(P1)**, **(P2)**, **(P8)**, and **(P9)**. The remaining reductions lead to AST vertices for which $\mathbf{v}_\nu.a^r_i$ is equal to the empty string for $i \neq idxBB$.

(P2) $r :: sr$

$$\begin{aligned}
\mathbf{v}_\nu.a^f &= \mathbf{v}_{\mu_1}.a^f + \mathbf{v}_{\mu_2}.a^f \\
\mathbf{v}_\nu.a_i^r &= \mathbf{v}_{\mu_2}.a_i^r + \mathbf{v}_{\mu_1}.a_i^r \quad i = 0, \ldots, idxBB \\
&\quad \text{where } \mathbf{v}_{\mu_1} \hat{=} s \text{ and } \mathbf{v}_{\mu_2} \hat{=} r \text{ in } sr \\
\nu&{++}
\end{aligned}$$

The adjoint basic blocks are synthesized by concatenating the values of the corresponding attributes of both successors of the AST vertex $\mathbf{v}_\nu$.

(P3) $s :: V = e;$

$$\begin{aligned}
\mathbf{v}_\nu.a^f &= \begin{cases} \text{"}push_c\text{"} + \text{"("} + idxBB + \text{")"} + \text{";"} & \text{if } newBB \vee \neg idxBB \\ \text{""} & \text{otherwise} \end{cases} \\
\mathbf{v}_\nu.a^f &= \mathbf{v}_{\nu-1}.a^f + \text{"}push\text{"} + \text{"("} + V.a^f + \text{")"} + \text{";"} \\
&\quad + V.a^f + \text{" = "} + \text{"}v_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{";"} \\
\mathbf{v}_\nu.a^r_{idxBB} &= \text{"}pop\text{"} + \text{"("} + V.a^f + \text{")"} + \text{";"} \\
&\quad + \text{"}\bar{v}_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{" = "} + V.\bar{a}^f + \text{";"} \\
&\quad + V.\bar{a}^f + \text{" = 0;"} \\
&\quad + \mathbf{v}_{\nu-1}.a^r_{idxBB} \\
c &= 1 \\
\nu&{++}
\end{aligned}$$

If the assignment is the first in the current basic block, that is, if $(newBB \vee \neg idxBB)$ returns TRUE, then the index of this basic block needs to be pushed onto the control stack. The value of the variable on the left-hand side ($V.a^f$) is stored on the data stack prior to getting overwritten with the value of the code list variable, that is the value of the expression on the right-hand side.

The adjoint of the assignment that is currently parsed is preceded by restoring the value of $V.a^f$, and it is followed by resetting the adjoint of $V.a^f$ to zero. The adjoint of the section of the current basic block ($idxBB$) that has been parsed so far is appended. As before, the code list variable counter c is reset to 1 to set the ground for correctly building the code list of the next assignment. All reduce actions end with the incrementation of the AST vertex counter ν.

(P4) $e :: V$

$$\begin{aligned}
&\vdots \\
\mathbf{v}_\nu.a^r_{idxBB} &= \text{"}pop\text{"} + \text{"("} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{")"} + \text{";"} \\
&\quad + V.\bar{a}^f + \text{"+="} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{";"} \\
\nu&{++}
\end{aligned}$$

The synthesis of the adjoint code is restricted to the current basic block as pointed out in the discussion of **(P1)**.

(P5) $e :: C$

$$\begin{aligned}
&\vdots \\
\mathbf{v}_\nu.a^r_{idxBB} &= \text{"}pop\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{"};\text{"} \\
\nu{+}{+}&
\end{aligned}$$

We simply restore the value of the code list variable that was overwritten with the constant $C.a^f$.

(P6) $e :: F(e)$

$$\begin{aligned}
&\vdots \\
\mathbf{v}_\nu.a^r_{idxBB} &= \text{"}pop\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{"};\text{"} \\
&+ \text{"}\bar{v}_{[\mathbf{v}_{\nu-1}.j]}\text{"} + \text{"} = \text{"} + \frac{\partial F.a^f}{\partial\text{"}v_{[\mathbf{v}_{\nu-1}.j]}\text{"}} + \text{"} * \text{"} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{"};\text{"} \\
&+ \mathbf{v}_{\nu-1}.a^r_{idxBB} \\
\nu{+}{+}&
\end{aligned}$$

The only difference from Section (3) is the restriction to the current basic block.

(P7) $e :: eOe$

$$\begin{aligned}
&\vdots \\
\mathbf{v}_\nu.a^r_{idxBB} &= \text{"}pop\text{"} + \text{"}(\text{"} + \text{"}v_{[\mathbf{v}_\nu.j]}\text{"} + \text{"})\text{"} + \text{"};\text{"} \\
&+ \text{"}\bar{v}_{[\mathbf{v}_{\mu_1}.j]}\text{"} + \text{"} = \text{"} + \frac{\partial O.a^f}{\partial\text{"}v_{[\mathbf{v}_{\mu_1}.j]}\text{"}} + \text{"} * \text{"} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{"};\text{"} \\
&+ \text{"}\bar{v}_{[\mathbf{v}_{\mu_2}.j]}\text{"} + \text{"} = \text{"} + \frac{\partial O.a^f}{\partial\text{"}v_{[\mathbf{v}_{\mu_2}.j]}\text{"}} + \text{"} * \text{"} + \text{"}\bar{v}_{[\mathbf{v}_\nu.j]}\text{"} + \text{"};\text{"} \\
&+ \mathbf{v}_{\mu_1}.a^r_{idxBB} + \mathbf{v}_{\mu_2}.a^r_{idxBB} \\
&\qquad \text{where } \mathbf{v}_{\mu_1} \hat{=} e^1 \text{ and } \mathbf{v}_{\mu_2} \hat{=} e^2 \text{ in } e^1 O e^2 \\
\nu{+}{+}&
\end{aligned}$$

The treatment is analogous to **(P6)**.

(P8) $s :: IF(c)\{r\}$

Shift Action:

$$newBB = 1$$

Reduce Action:

$$\mathbf{v}_\nu.a^f = "if" + "(" + \mathbf{v}_{\mu_1}.a^f + ")" + "\{" \mathbf{v}_{\mu_2}.a^f + "\}"$$
$$\text{where } \mathbf{v}_{\mu_1} \hat{=} c \text{ and } \mathbf{v}_{\mu_2} \hat{=} r \text{ in } IF(c)\{r\}$$
$$\mathbf{v}_\nu.a_i^r = \mathbf{v}_{\mu_2}.a_i^r \quad i = 0, \ldots, idxBB$$
$$newBB = 1$$
$$\nu{+}{+}$$

Prior to parsing the branch body r, that is, while shifting through the right-hand side of the production rule, we need to ensure that the next assignment is correctly recognized as the first entry of the next basic block. For the same reason, we need to set $newBB = 1$ after parsing the IF-statement.

(P9) $s :: WHILE(c)\{r\}$
The treatment is analogous to **(P8)**.

(P10) $c :: V < V$

$$\mathbf{v}_\nu.a^f = V^{(1)}.a^f + " < " + V^{(2)}.a^f$$
where $V^{(1)}$ and $V^{(2)}$ correspond to the tokens preceding and succeeding the $<$ token, respectively.
$$\nu{+}{+}$$

Conditions get simply unparsed.

5 Implementation

Our simple proof-of-concept implementation (called `sdac` for syntax-directed adjoint code compiler) uses the compiler tools `flex`[3] and `bison`[4] The source code is shown in Appendix A.2. Furthermore, we present a simplified version that generates adjoint SLP's in Appendix A.1. `sdac` is meant to serve as a starting point for further development of more complete syntax-directed adjoint code compilers that provide better coverage for the commonly used programming languages. The source code can be downloaded from the project website.

In the following we present a small case study that is supposed to illustrate the current functionality of `sdac`.

Listing 1.2 shows a small input file that needs to be transformed into adjoint code. The same example has been used in [Nau05], thus providing a point for comparison of the numerical results.

Listing 1.2. test.in

```
t=0;
while (x<t) {
  if (x<y) {
```

[3] `http://www.gnu.org/software/flex/`
[4] `http://www.gnu.org/software/bison/`

```
    x=y+1;
  }
  x=sin(x*y);
}
```

We call `sdac test.in > test.out` to obtain the output in Listing 1.3.

Listing 1.3. test.out

```
double v1, v1_;
double v2, v2_;
double v3, v3_;
double v4, v4_;
push_c(0);
push_v(v1); v1=0;
push_v(t); t=v1;
while (x<t) {
if (x<y) {
push_c(1);
push_v(v1); v1=y;
push_v(v2); v2=1;
push_v(v3); v3=v1+v2;
push_v(x); x=v3;
}
push_c(2);
push_v(v1); v1=x;
push_v(v2); v2=y;
push_v(v3); v3=v1*v2;
push_v(v4); v4=sin(v3);
push_v(x); x=v4;
}
int i_;
while (pop_c(i_)) {
if (i_==0) {
pop_v(t); v1_=t_; t_=0;
pop_v(v1);
}
else if (i_==1) {
pop_v(x); v3_=x_; x_=0;
pop_v(v3); v1_=v3_; v2_=v3_;
pop_v(v2);
pop_v(v1); y_+=v1_;
}
else if (i_==2) {
pop_v(x); v4_=x_; x_=0;
pop_v(v4); v3_=cos(v3)*v4_;
pop_v(v3); v1_=v3_*v2; v2_=v3_*v1;
pop_v(v2); y_+=v2_;
pop_v(v1); x_+=v1_;
}
}
```

To verify the correctness of the transformation we provide a driver that compares the values of the two gradient entries as computed by the adjoint code with an approximation obtained by applying forward finite differences. The driver contains wrappers for the original code (lines 29–32) and the adjoint code (lines 34–37) in the form of the subroutines `test` and `test_`. Furthermore it implements the data and control stack together with the corresponding storage and retrieval functions (lines 3, 7–27).

Note that only one call of the adjoint routine is required in Listing 1.4, line 55 as opposed to two calls for the finite difference approximation (or, similarly, of the tangent-linear routine as discussed in [Nau05]).

Listing 1.4. test.cpp

```
1 #include <cmath>
2 #include <iostream>
3 #include <stack>
4
5 using namespace std;
6
7 static stack<double> stack_v;
8 static stack<int> stack_c;
9
10 void push_v(double v) {
11   stack_v.push(v);
12 }
13 void pop_v(double& v) {
14   v=stack_v.top();
15   stack_v.pop();
16 }
17 void push_c(int c) {
18   stack_c.push(c);
19 }
20 int pop_c(int& c) {
21   if (!stack_c.empty()) {
22     c=stack_c.top();
23     stack_c.pop();
24     return 1;
25   }
26   return 0;
27 }
28
29 void test ( double &x, double y) {
30 double t;
31 #include "test.in"
32 }
33
34 void test_ ( double &x, double& x_, double y, double& y_) {
35 double t,t_;
36 #include "test.out"
37 }
38
39 int main() {
40 {
41   cout << "finite differences:" << endl;
42   double h=1e-6, x=-.5, y=-5., x_=x+h, y_=y;
43   test(x,y);
44   test(x_,y_);
45   cout << "dx/dx=" << (x_-x)/h << endl;
46
47   x=-.5, x_=x, y_=y+h;
48   test(x,y);
49   test(x_,y_);
50   cout << "dx/dy=" << (x_-x)/h << endl;
51 }
52 {
53   cout << "adjoint code:" << endl;
54   double x=-.5, y=-5, x_=1., y_=0.;
```

```
  test_(x,x_,y,y_);
  cout << "dx/dx=" << x_ << endl;
  cout << "dx/dy=" << y_ << endl;
}
  return 0;
}
```

The numerical results are identical with those in [Nau05].

```
finite differences:
dx/dx=4.00571
dx/dy=0.400572
adjoint code:
dx/dx=4.00572
dx/dy=0.400572
```

6 Conclusions and Outlook

The syntax-directed generation of adjoint code is elegant and relatively simple to implement. No internal representation of the forward code needs to be generated. A resulting disadvantage is the lack of data flow analysis which makes a full domain-specific optimization of the generated code impossible. Standard optimizations are performed potentially by the compiler that is used to translate the adjoint into object code.

The proposed approach represents a reasonable trade-off between the effort required for the tool development and the quality of the generated code. Moreover, the question about where the limits of the syntax-directed approach in the context of adjoint code generation are is still open and the subject of ongoing research.

In part III of this series of reports on syntax-directed generation of derivative code we will focus on adjoint code for numerical programs with interprocedural flow of control induced by subroutine calls. Work is underway to increase the syntactic richness of the input accepted by `sdac` with the objective to provide a tool that covers more and more practically relevant cases.

Bibliography

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BCH+05] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, volume 50 of *Lecture Notes in Computational Science and Engineering.* Springer, 2005.

[CF00] A. Carle and M. Fagan. ADIFOR 3.0. Technical Report CAAM-TR-00-02, Rice University, 2000.

[CG91] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.

[GJU96] A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Soft.*, 22:131–167, 1996.

[GK03] R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *Proceedings in Applied Mathematics and Mechanics*, 2(1):54–57, 2003.

[GR91] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[CG91]*, pages 126–135. SIAM, 1991.

[Gri00] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.

[GUW00] A. Griewank, J. Utke, and A. Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation*, 69:1117–1130, 2000.

[HNP05] L. Hascoët, U. Naumann, and V. Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

[HP04] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, 1997.

[Nau05] Uwe Naumann. Syntax-directed derivative code (Part I: Tangent-linear code). Preprint AIB-2005-16, RWTH Aachen, August 2005.

[NR05] U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In *[BCH+05].* 2005.

[WHH+05] C. Wunsch, C. Hill, P. Heimbach, U. Naumann, J. Utke, M. Fagan, and N. Tallent. OpenAD. Preprint ANL/MCS-P1230-0205, Argonne National Laboratory, February 2005.

A A Proof-of-Concept Implementation

A.1 Adjoint SLP's

Listing 1.5. ast.h

```
typedef struct {
  int j;
  char* af;
  char* ar;
} astNodeType;

#define YYSTYPE astNodeType
```

Listing 1.6. scanner.l

```
%{
#include "ast.h"
#include "parser.tab.h"
%}

whitespace        [ \t\n]+
symbol            [a-z]
const             [0-9]

%%

{whitespace} { }
"sin" { return SIN; }
{symbol} {
  yylval.af = (char*)malloc(2*sizeof(char));
  strcpy(yylval.af,yytext);
  yylval.ar=0; yylval.j=0;
  return SYMBOL;
}
{const} {
  yylval.af = (char*)malloc((strlen(yytext)+1)*sizeof(char));
  strcpy(yylval.af,yytext);
  yylval.ar=0; yylval.j=0;
  return CONSTANT;
}
. { return yytext[0]; }

%%

void lexinit(FILE *source)
{
  yyin=source;
}
```

Listing 1.7. parser.y

```
%{

#include <stdio.h>
#include "ast.h"

extern int yylex();
extern void lexinit(FILE*);
```

```

static int c=1,cmax=1;

%}

%token SYMBOL CONSTANT SIN IF WHILE

%left '+'
%left '*'

%%

code : sequence_of_assignments
  {
    for (c=1;c<cmax;c++) printf("double v%d, v%d_;\n",c,c);
    printf("%s%s",$1.af,$1.ar);
    free($1.af); free($1.ar);
  }

  ;
sequence_of_assignments : assignment { $$=$1; }
  | assignment sequence_of_assignments
  {
    $$.af=(char*)malloc((strlen($1.af)+strlen($2.af)+1)*sizeof(char));
    sprintf($$.af,"%s%s",$1.af,$2.af);
    free($2.af); free($1.af);

    $$.ar=(char*)malloc((strlen($1.ar)+strlen($2.ar)+1)*sizeof(char));
    sprintf($$.ar,"%s%s",$2.ar,$1.ar);
    free($2.ar); free($1.ar);
  }
  ;
assignment : SYMBOL '=' expression ';'
  {
    $$.af=(char*)malloc((strlen($3.af)+2*strlen($1.af)+$3.j%10+14)*sizeof(char));
    sprintf($$.af,"%spush(%s); %s=v%d;\n",$3.af,$1.af,$1.af,$3.j);
    $$.ar=(char*)malloc((3*strlen($1.af)+$3.j%10+strlen($3.ar)+20)*sizeof(char));
    sprintf($$.ar,"pop(%s); v%d_=%s_; %s_=0;\n%s",$1.af,$3.j,$1.af,$1.af,$3.ar);
    free($3.ar); free($1.af); free($3.af);
    c=1;
  }
  ;
expression : expression '*' expression
  {
    $$.af=(char*)malloc((strlen($1.af)+strlen($3.af)+2*c%10+$1.j%10+$3.j%10+21)*sizeof(char));
    $$.j=c++; if (c>cmax) cmax=c;
    sprintf($$.af,"%s%spush(v%d); v%d=v%d*v%d;\n",$1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
    free($1.af); free($3.af);

    $$.ar=(char*)malloc((2*$1.j%10+2*$3.j%10+3*$$.j%10+strlen($1.ar)+strlen($3.ar)+34)*sizeof(char));
```

```
      sprintf($$.ar,"pop(v%d); v%d_=v%d_*v%d; v%d_=v%d_*v%d;\n%s%s",$$.j,$1.j,$$.j,$3.j,$3.j,$$.j,$1.j,$3.ar,$1.ar);
      free($1.ar); free($3.ar);

    }
    | expression '+' expression
    {
      $$.af=(char*)malloc((strlen($1.af)+strlen($3.af)+2*c%10+$1.j%10+$3.j%10+21)*sizeof(char));
      $$.j=c++; if (c>cmax) cmax=c;
      sprintf($$.af,"%s%spush(v%d); v%d=v%d+v%d;\n",$1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
      free($1.af); free($3.af);

        $$.ar=(char*)malloc(($1.j%10+$3.j%10+3*$$.j%10+strlen($1.ar)+strlen($3.ar)+28)*sizeof(char));
        sprintf($$.ar,"pop(v%d); v%d_=v%d_; v%d_=v%d_;\n%s%s",$$.j,$1.j,$$.j,$3.j,$$.j,$3.ar,$1.ar);
        free($1.ar); free($3.ar);
    }
    | SIN '(' expression ')'
    {
      $$.af=(char*)malloc((strlen($3.af)+2*c%10+$3.j%10+23)*sizeof(char));
      $$.j=c++; if (c>cmax) cmax=c;
      sprintf($$.af,"%spush(v%d); v%d=sin(v%d);\n",$3.af,$$.j,$$.j,$3.j);
      free($3.af);

      $$.ar=(char*)malloc((2*$3.j%10+2*$$.j%10+strlen($3.ar)+27)*sizeof(char));
      sprintf($$.ar,"pop(v%d); v%d_=cos(v%d)*v%d_;\n%s",$$.j,$3.j,$3.j,$$.j,$3.ar);
      free($3.ar);
    }
    | SYMBOL
    {
      $$.af=(char*)malloc((2*c%10+strlen($1.af)+16)*sizeof(char));
      $$.j=c++; if (c>cmax) cmax=c;
      sprintf($$.af,"push(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

      $$.ar=(char*)malloc((strlen($1.af)+2*$$.j%10+18)*sizeof(char));
      sprintf($$.ar,"pop(v%d); %s_+=v%d_;\n",$$.j,$1.af,$$.j);

      free($1.af);
    }
    | CONSTANT
    {
      $$.af=(char*)malloc((2*c%10+strlen($1.af)+16)*sizeof(char));
      $$.j=c++; if (c>cmax) cmax=c;
      sprintf($$.af,"push(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

      $$.ar=(char*)malloc((2*$$.j%10+10)*sizeof(char));
      sprintf($$.ar,"pop(v%d);\n",$$.j);
      free($1.af);
    }
    ;

%%
```

```

int yyerror(char *msg) { printf("ERROR: %s \n",msg); return -1; }

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

A.2 Adjoint Subroutines

Listing 1.8. ast.h

```
#define maxBB 100

typedef struct {
  int j;
  char* af;
  char* ar[maxBB];
} astNodeType;

#define YYSTYPE astNodeType
```

Listing 1.9. scanner.l

```
%{
#include "ast.h"
#include "parser.tab.h"
%}

whitespace      [ \t\n]+
symbol          [a-z]
const           [0-9]

%%

{whitespace} { }
"if" { return IF; }
"while" { return WHILE; }
"sin" { return SIN; }
{symbol} {
  yylval.af = (char*)malloc(2*sizeof(char));
  strcpy(yylval.af,yytext);
  int i;
  for (i=0;i<maxBB;i++) yylval.ar[i]=0;
  yylval.j=0;
  return SYMBOL;
}
{const} {
  yylval.af = (char*)malloc((strlen(yytext)+1)*sizeof(char));
```

```
  strcpy(yylval.af,yytext);
  int i;
  for (i=0;i<maxBB;i++) yylval.ar[i]=0;
  yylval.j=0;
  return CONSTANT;
}

. { return yytext[0]; }

%%

void lexinit(FILE *source)
{
  yyin=source;
}
```

Listing 1.10. parser.y

```
%{

#include <stdio.h>
#include "ast.h"

extern int yylex();
extern void lexinit(FILE*);

static int c=1,cmax=1;
static int newBB=0;
static int idxBB=0;

%}

%token SYMBOL CONSTANT SIN IF WHILE

%left '+'
%left '*'

%%

code : sequence_of_statements
  {
    for (c=1;c<cmax;c++) printf("double v%d, v%d_;\n",c,c);
    printf("%s",$1.af);
    free($1.af);
    int i;
    printf("int i_;\n");
    printf("while (pop_c(i_)) {\n");
    for (i=0;i<=idxBB;i++) {
      if (i==0)
        printf("if");
      else
        printf("else if");
      if ($1.ar[i])
        printf(" (i_==%d) {\n%s}\n",i,$1.ar[i]);
      else
        printf(" (i_==%d) {\n}\n",i);
      free($1.ar[i]);
```

```
    }
    printf("}\n");
  }

  ;
sequence_of_statements : statement { $$=$1; }
  | statement sequence_of_statements
  {
    $$.af=(char*)malloc((strlen($1.af)+strlen($2.af)+1)*sizeof(char));
    sprintf($$.af,"%s%s",$1.af,$2.af);
    free($2.af); free($1.af);

    int i;
    for (i=0;i<=idxBB;i++) {
      if ($2.ar[i]&&$1.ar[i]) {
        $$.ar[i]=(char*)malloc((strlen($1.ar[i])+strlen($2.ar[i])+1)*sizeof(char));
        sprintf($$.ar[i],"%s%s",$2.ar[i],$1.ar[i]);
        free($2.ar[i]); free($1.ar[i]);
      }
      else if ($2.ar[i]) {
        $$.ar[i]=(char*)malloc((strlen($2.ar[i])+1)*sizeof(char));
        sprintf($$.ar[i],"%s",$2.ar[i]);
        free($2.ar[i]);
      }
      else if ($1.ar[i]) {
        $$.ar[i]=(char*)malloc((strlen($1.ar[i])+1)*sizeof(char));
        sprintf($$.ar[i],"%s",$1.ar[i]);
        free($1.ar[i]);
      }
    }
  }
  ;
statement : assignment { $$=$1; }
  | if_statement { $$=$1; }
  | while_statement { $$=$1; }
  ;
if_statement : IF '(' condition ')' '{'
  {
    newBB=1;
  }
  sequence_of_statements '}'
  {
    $$.af=(char*)malloc((strlen($3.af)+strlen($7.af)+12)*sizeof(char));
    sprintf($$.af,"if (%s) {\n%s}\n",$3.af,$7.af);
    free($3.af); free($7.af);
    int i;
    for (i=0;i<=idxBB;i++) {
      if ($7.ar[i]) {
        $$.ar[i]=(char*)malloc((strlen($7.ar[i])+1)*sizeof(char));
        sprintf($$.ar[i],"%s",$7.ar[i]);
        free($7.ar[i]);
      }
    }
    newBB=1;
  }
  ;
```

```
while_statement : WHILE '(' condition ')' '{'
  {
    newBB=1;
  }
  sequence_of_statements '}'
  {
    $$.af=(char*)malloc((strlen($3.af)+strlen($7.af)+15)*sizeof(char));
    sprintf($$.af,"while (%s) {\n%s}\n",$3.af,$7.af);
    free($3.af); free($7.af);
    int i;
    for (i=0;i<=idxBB;i++) {
      if ($7.ar[i]) {
        $$.ar[i]=(char*)malloc((strlen($7.ar[i])+1)*sizeof(char));
        sprintf($$.ar[i],"%s",$7.ar[i]);
        free($7.ar[i]);
      }
    }
    newBB=1;
  }
  ;
condition : SYMBOL '<' SYMBOL
  {
    $$.af=(char*)malloc((strlen($1.af)+strlen($3.af)+2)*sizeof(char));
    sprintf($$.af,"%s<%s",$1.af,$3.af);
    free($1.af); free($3.af);
  }
assignment : SYMBOL '='
  {
    if (newBB) idxBB++;
  }
  expression ';'
  {
    if (newBB||!idxBB) {
      $$.af=(char*)malloc((strlen($4.af)+idxBB%10+3*strlen($1.af)+2*$4.j%10+27)*sizeof(char));
      sprintf($$.af,"push_c(%d);\n%spush_v(%s); %s=v%d;\n",idxBB,$4.af,$1.af,$1.af,$4.j);
    }
    else {
      $$.af=(char*)malloc((strlen($4.af)+3*strlen($1.af)+2*$4.j%10+16)*sizeof(char));
      sprintf($$.af,"%spush_v(%s); %s=v%d;\n",$4.af,$1.af,$1.af,$4.j);
    }
    $$.ar[idxBB]=(char*)malloc((3*strlen($1.af)+$4.j%10+strlen($4.ar[idxBB])+22)*sizeof(char));
    sprintf($$.ar[idxBB],"pop_v(%s); v%d_=%s_; %s_=0;\n%s",$1.af,$4.j,$1.af,$1.af,$4.ar[idxBB]);
    free($4.ar[idxBB]);
    newBB=0;
    free($1.af); free($4.af);
    c=1;
  }
  ;
expression : expression '*' expression
  {
    $$.j=c++; if (c>cmax) cmax=c;
```

```
      $$.af=(char*)malloc((strlen($1.af)+strlen($3.af)+2*$$.j%10+$1.j%10+$3.j%10+20)*sizeof(char));
      sprintf($$.af,"%s%spush_v(v%d); v%d=v%d*v%d;\n",$1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
      free($1.af); free($3.af);

      $$.ar[idxBB]=(char*)malloc((2*$1.j%10+2*$3.j%10+3*$$.j%10+strlen($1.ar[idxBB])+strlen($3.ar[idxBB])+36)*sizeof(char));
      sprintf($$.ar[idxBB],"pop_v(v%d); v%d_=v%d_*v%d; v%d_=v%d_*v%d;\n%s%s",$$.j,$1.j,$$.j,$3.j,$3.j,$$.j,$1.j,$3.ar[idxBB],$1.ar[idxBB]);
      free($1.ar[idxBB]); free($3.ar[idxBB]);

    }
    | expression '+' expression
    {
      $$.j=c++; if (c>cmax) cmax=c;
      $$.af=(char*)malloc((strlen($1.af)+strlen($3.af)+2*$$.j%10+$1.j%10+$3.j%10+23)*sizeof(char));
      sprintf($$.af,"%s%spush_v(v%d); v%d=v%d+v%d;\n",$1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
      free($1.af); free($3.af);

        $$.ar[idxBB]=(char*)malloc((2*$1.j%10+2*$3.j%10+3*$$.j%10+strlen($1.ar[idxBB])+strlen($3.ar[idxBB])+30)*sizeof(char));
        sprintf($$.ar[idxBB],"pop_v(v%d); v%d_=v%d_; v%d_=v%d_;\n%s%s",$$.j,$1.j,$$.j,$3.j,$$.j,$3.ar[idxBB],$1.ar[idxBB]);
        free($1.ar[idxBB]); free($3.ar[idxBB]);
    }
    | SIN '(' expression ')'
    {
      $$.j=c++; if (c>cmax) cmax=c;
      $$.af=(char*)malloc((strlen($3.af)+2*$$.j%10+$3.j%10+25)*sizeof(char));
      sprintf($$.af,"%spush_v(v%d); v%d=sin(v%d);\n",$3.af,$$.j,$$.j,$3.j);
      free($3.af);

      $$.ar[idxBB]=(char*)malloc((2*$$.j%10+2*$3.j%10+strlen($3.ar[idxBB])+29)*sizeof(char));
      sprintf($$.ar[idxBB],"pop_v(v%d); v%d_=cos(v%d)*v%d_;\n%s",$$.j,$3.j,$3.j,$$.j,$3.ar[idxBB]);
      free($3.ar[idxBB]);
    }
    | SYMBOL
    {
      $$.j=c++; if (c>cmax) cmax=c;
      $$.af=(char*)malloc((2*$$.j%10+strlen($1.af)+18)*sizeof(char));
      sprintf($$.af,"push_v(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

      $$.ar[idxBB]=(char*)malloc((strlen($1.af)+2*$$.j%10+20)*sizeof(char));
      sprintf($$.ar[idxBB],"pop_v(v%d); %s_+=v%d_;\n",$$.j,$1.af,$$.j);
      free($1.af);
    }
    | CONSTANT
    {
```

```
      $$.j=c++; if (c>cmax) cmax=c;
      $$.af=(char*)malloc((2*$$.j%10+strlen($1.af)+18)*sizeof(char));
      sprintf($$.af,"push_v(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

      $$.ar[idxBB]=(char*)malloc(($$.j%10+12)*sizeof(char));
      sprintf($$.ar[idxBB],"pop_v(v%d);\n",$$.j);
      free($1.af);
    }
    ;

%%

int yyerror(char *msg) { printf("ERROR: %s \n",msg); return -1; }

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

1987-01 * Fachgruppe Informatik: Jahresbericht 1986
1987-02 * David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 * Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 * Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 * Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 * Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 * Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 * Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 * Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 * Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 * Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12 J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 * Gabriele Esser, Johannes Rückert, Frank Wagner: Gesellschaftliche Aspekte der Informatik
1988-02 * Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 * Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 * Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 * Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 * Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 * Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 * Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 * W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 * Kai Jakobs: Towards User-Friendly Networking
1988-11 * Kai Jakobs: The Directory - Evolution of a Standard
1988-12 * Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 * Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

1988-16 * Fachgruppe Informatik: Jahresbericht 1987

1988-17 * Wolfgang Thomas: Automata on Infinite Objects

1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs

1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment

1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers

1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies

1988-24 * Uschi Heuter: Generalized Definite Tree Languages

1989-01 * Fachgruppe Informatik: Jahresbericht 1988

1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)

1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management

1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines

1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking

1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD

1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars

1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)

1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik

1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual

1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems

1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages

1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)

1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine

1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus

1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions

1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code

1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine

1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus

1992-19-09 D. Howe, G. Burn: Experiments with strict STG code

1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes

1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine

1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)

1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer

1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine

1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell

1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation

1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages

1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)

1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)

1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers

1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)

1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)

1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting

1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)

1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models

1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures

1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)

1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language

1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language

1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing

1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free

1992-24 K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems
1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming
1996-01 * Jahresbericht 1995
1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees
1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability
1996-05 Klaus Pohl: Requirements Engineering: An Overview
1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement
1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools
1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases
1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future
1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 * Jahresbericht 1998
1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge, Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools
2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 Mareike Schoop: Cooperative Document Management
2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, Subbarayan Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Crash-Prone Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.