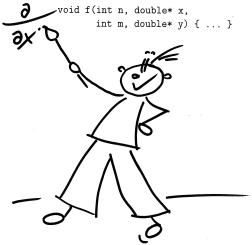
STCE RWTH AACHEN UNIVERSITY

The Art of Differentiating Computer Programs

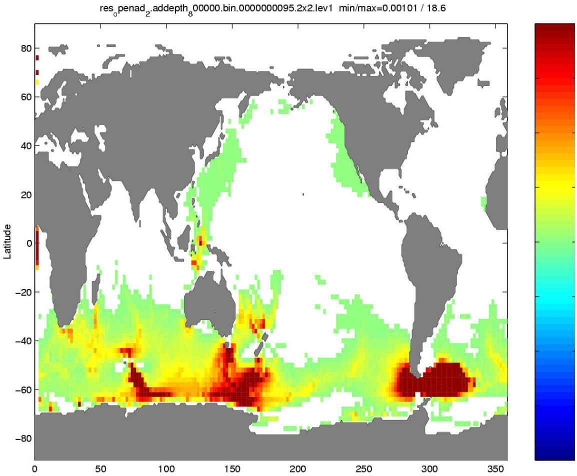
using Algorithmic Differentiation (AD)

Uwe Naumann

LuFG Informatik 12
Software and **T**ools for **C**omputational **E**ngineering
RWTH Aachen
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de



Motivation: Numerical Simulation
e.g., MITgcm



Formally

Given: Numerical simulation program

$$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x})$$

Wanted: Sensitivity of outputs $\mathbf{y} \in \mathbb{R}^m$ on inputs $\mathbf{x} \in \mathbb{R}^n$ (Jacobian $\nabla F(\mathbf{x})$)

$$\nabla F(\mathbf{x}) \equiv \left(\frac{\partial y_j}{\partial x_i} \right)_{i=0,\ldots,n-1}^{j=0,\ldots,m-1}$$

Sensitivities / Derivatives

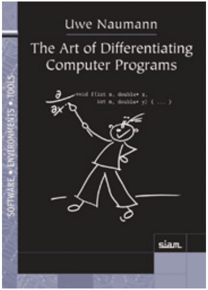
$y = f(x)$; effect of perturbation in x depending on $y' \equiv \frac{\partial y}{\partial x}$

- for large $y' > 0 \rightarrow$ large increase in y (instable)
- for small $y' > 0 \rightarrow$ small increase in y (stable)
- for large $y' < 0 \rightarrow$ large decrease in y (instable)
- for small $y' < 0 \rightarrow$ small decrease in y (stable)
- for $y' = 0 \rightarrow$ invariant

Outline

- 3 lectures
- 2 tutorials
- see tutorial sheet for contents

Literature



U. Naumann:
The Art of Differentiating Computer Programs.
An Introduction to Algorithmic Differentiation
Number 24 of Software, Environments, and Tools Series, SIAM, 2012.

naumann@stce.rwth-aachen.de

More ...

- Computational Differentiation (V3Ü1, Ba, WS)
- Combinatorial Problems in Scientific Computing (V2Ü2, Ma, SS)

Our Objective

For a given numerical simulation program, e.g.,

```
void f(int n, double *x, int m, double *y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}
```

build a program that can be used to compute the sensitivites of all outputs with respect to all inputs (Jacobian matrix).

STCE RWTH AACHEN UNIVERSITY

Specific Objective: Tangent-Linear Code

$$\dot{\mathbf{y}} = \nabla F(\mathbf{x}) \cdot \dot{\mathbf{x}}$$
$$\mathbf{y} = F(\mathbf{x})$$

```
void f(int n, double *x, int m, double *y)
```

becomes

```
void df(int n, double *x, double *dx,
        int m, double *y, double *dy)
```

and computes product of the Jacobian with $\dot{\mathbf{x}}$ in $\dot{\mathbf{y}}$; not the Jacobian itself.

Example

```
void f(int n, double *x, int m, double *y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}
```

- Single Assignment Code (SAC)
- Directed Acyclic Graph (AST, DAG)
- Linearized SAC
- Linearized AST, DAG
- Local Tangent-Linear Code
- Chain Rule

Jacobian
Computation and Validation

```
void f(int n, double *x, int m, double *y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}
```

- driver for tangent-linear code
- validation by finited differences

Validation through Approximation
Forward Finite Differences

Let $D \subseteq \mathbb{R}^n$ be an open domain and $F : D \to \mathbb{R}^m$ such that

$$F = \begin{pmatrix} F_0 \\ \vdots \\ F_{m-1} \end{pmatrix} \quad .$$

A *forward finite difference* approximation of the ith column of the Jacobian ∇F at point $\mathbf{x}^0$ is computed as

$$\frac{\partial F}{\partial x_i}(\mathbf{x}^0) \equiv \begin{pmatrix} \frac{\partial F_0}{\partial x_i}(\mathbf{x}^0) \\ \vdots \\ \frac{\partial F_{m-1}}{\partial x_i}(\mathbf{x}^0) \end{pmatrix} \approx_1 \frac{F(\mathbf{x}^0 + \mathbf{e}_i \cdot h) - F(\mathbf{x})}{h} \tag{1}$$

The *ith Cartesian basis vector in* $\mathbb{R}^n$ is denoted by $\mathbf{e}_i$.

However ...

Consider the approximation of the first derivative of $y = f(x) = x$ in single precision IEEE floating-point arithmetic on a 64 bit machine at $x = 10^6$ by the forward finite quotient

$$\nabla f(x) \approx \frac{f(x+h) - f(x)}{h}$$

with $h = 0.1$. Obviously, $\nabla f(x) = 1$ independent of x. The code

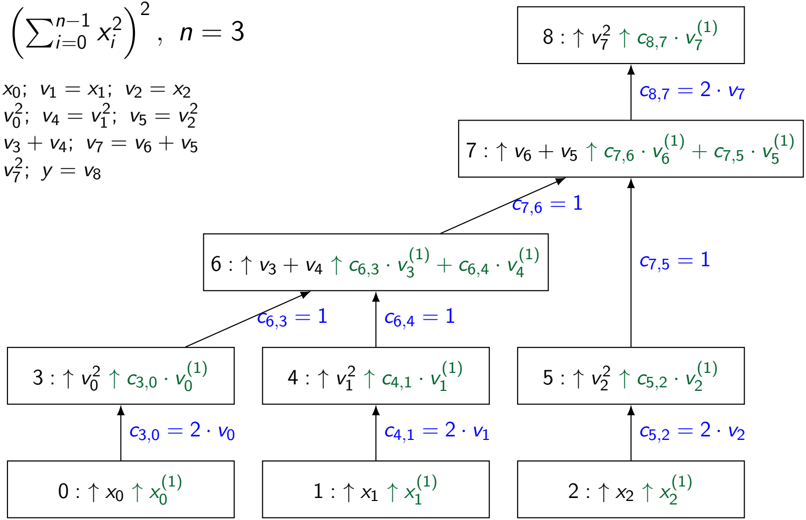
```
...
float x=1e6, h=1e-1;
cout << "(f(x+h)-f(x))/h=" << (x+h-x)/h << endl;
...
```

returns 1.25.

Tangent-Linear Code by Forward Mode AD

$$y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2, \quad n = 3$$

$v_0 = x_0;\ v_1 = x_1;\ v_2 = x_2$
$v_3 = v_0^2;\ v_4 = v_1^2;\ v_5 = v_2^2$
$v_6 = v_3 + v_4;\ v_7 = v_6 + v_5$
$v_8 = v_7^2;\ y = v_8$



Consider the nonlinear programming problem (NLP)

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x})$$

where, for example,

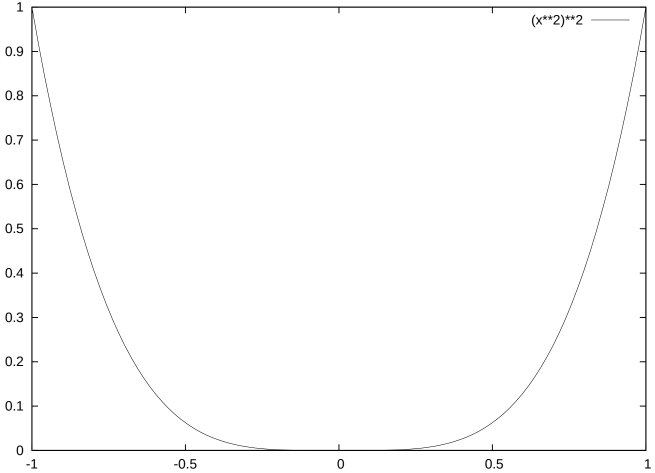
$$f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2 \quad (2)$$

is implemented as

```
void f(int n, double *x, double &y)
{
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}
```

The function has a global minimum at $\mathbf{x} = 0$.

Nonlinear Optimization



Steepest Descent

... computes

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \cdot \nabla f(\mathbf{x}^k) \quad .$$

for some suitable starting value $\mathbf{x}^0 = (x_i^0)_{i=0,\dots,n-1}$ and with a step length $\alpha_k > 0$, where $\nabla f(\mathbf{x}^k)$ denotes the gradient of f at the current iterate.

The step length $\alpha_k > 0$ is, for example, chosen by recursive bisection on α_k starting from $\alpha_k = 1$ $(0.5, 0.25, \dots)$ and such that a decrease in the objective value is ensured

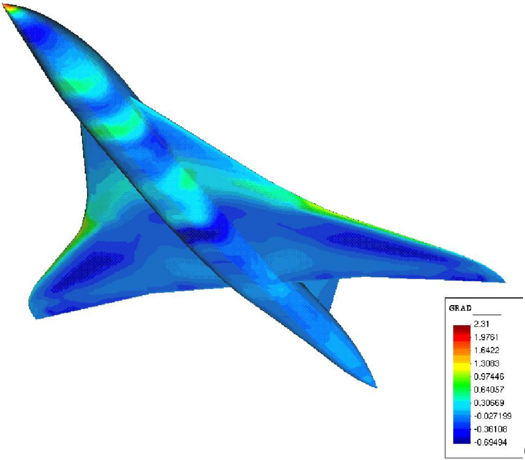
More Applications

...


6:24 AM EDT
1d 5d 1m 3m 6m 1y 2y 5y 10y Full
7.35
5.22
3.09
0.96
wikinvest
Nov 2008
Dec
Jan
Feb

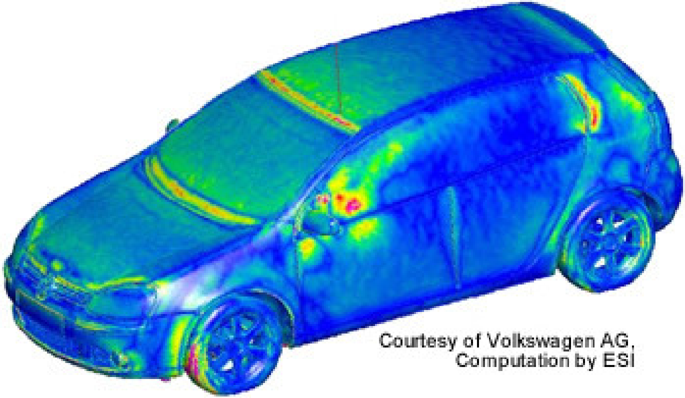
©BBA

Aeroacoustics



©INRIA

Automative Engineering



©VW / ESI

dcc (version 0.9)

See

`www.siam.org/books/se24`

for

- source
- use guide
- Windows version

By The Way ...

It also works for, e.g.

```
void f(int n, double *x, double &y) {
  int i=0;
  while (i<n) {
    if (i==0) { y=x[i]*x[i]; }
    else { y=y+x[i]*x[i]; }
    i=i+1;
  }
  y=y*y;
}
```

as well as for interprocedural code etc.

Parsing SL^2

A *Straight-Line Simple Language* program is a sequence of statements described by the grammar $G = (V_n, V_t, P, s)$ with nonterminal symbols

$$V_n = \left\{ \begin{array}{ll} s & \text{(sequence of statements)} \\ a & \text{(assignment)} \\ e & \text{(expression)} \end{array} \right\}$$

terminal symbols

$$V_t = \left\{ \begin{array}{rl} V & \text{(program variables)} \\ C & \text{(constants)} \\ F & \text{(unary intrinsic)} \\ L & \text{(linear binary arithmetic operator)} \\ N & \text{(nonlinear binary arithmetic operator)} \\ ;\ )\ (\ = & \text{(remaining single character tokens)} \end{array} \right\}$$

... start symbol s, and production rules

$$P = \left\{ \begin{array}{llllll} (P1) & s : a & (P2) & s : as & (P3) & a : V = e; \\ (P4) & e : eLe & (P5) & e : eNe & (P6) & e : F(e) \\ (P7) & e : V & (P8) & e : C & & \end{array} \right\} \quad .$$

Ambiguity

- Associativity?

$$a * b * c \stackrel{?}{=} (a * b) * c \stackrel{?}{=} a * (b * c)$$

- Operator Precedence?

$$a + b * c \stackrel{?}{=} (a + b) * c \stackrel{?}{=} a + (b * c)$$

... to be resolved by `bison`.

Parser → Parse Tree → Unparser

For example,

```
void f(int n, double *x, int m, double *y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}
```

- function body is word in G
- parser yields parse tree
- linearization annotates parse tree
- DFS tangent-linear code unparser

Unambiguous SL^2

An SL^2 *program* is a sequence of statements described by the grammar $G = (V_n, V_t, P, s)$ with nonterminal symbols

$$V_n = \left\{ \begin{array}{llllll} s & \text{(sequence of assignments)} & a & \text{(assignment)} & & \\ e & \text{(expression)} & t & \text{(term)} & f & \text{(factor)} \end{array} \right\}$$

terminal symbols

$$V_t = \left\{ \begin{array}{rl} V & \text{(program variables)} \\ C & \text{(constants)} \\ F & \text{(unary intrinsic)} \\ L & \text{(linear binary arithmetic operator)} \\ N & \text{(nonlinear binary arithmetic operator)} \\ ;\ )\ (\ = & \text{(remaining single character tokens)} \end{array} \right\}$$

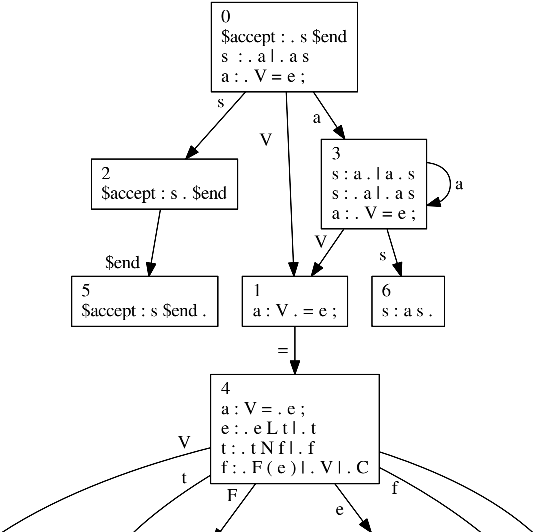
... start symbol s, and production rules

$$P = \left\{ \begin{array}{llllll} (P1) & s : a & (P2) & s : as & (P3) & a : V = e; \\ (P4) & e : eLt & (P5) & e : t & (P6) & t : tNf \\ (P7) & t : f & (P8) & f : F(e) & (P9) & f : V \\ (P10) & f : C & & & & \end{array} \right\} \quad .$$

implies

- left-most bracketing for chained binary operations
- precedence of nonlinear over linear operations

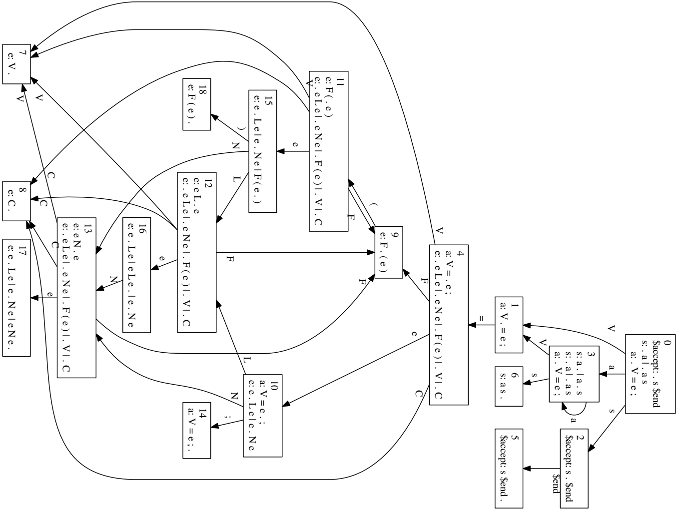
LR(0) Automaton



SLR Parsing
$V = F(VNC);$

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
$\vdots$					

Operator Precedence LR(0) Automaton



Operator Precedence Parsing
$V = F(VNC);$

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
5	0,1,4,9	11	$V = F(V$	$NC);$	S
6	0,1,4,9,11	7		$NC);$	R($P6$)
7	0,1,4,9	11	$V = F(e$	$NC);$	S
8	0,1,4,9,11	15	$V = F(eN$	$C);$	S
9	0,1,4,9,11,15	13	$V = F(eNC$	$);$	S
10	0,1,4,9,11,15,13	8		$);$	R($P7$)
11	0,1,4,9,11,15	13	$V = F(eNe$	$);$	S
12	0,1,4,9,11,15,13	17		$);$	R($P4$)
13	0,1,4,9	11	$V = F(e$	$);$	S

	STACK	STATE	PARSED	INPUT	ACTION
14	0,1,4,9,11	15	$V = F(e)$	;	S
15	0,1,4,9,11,15	18		;	R($P5$)
16	0,1	4	$V = e$	;	S
17	0,1,4	10	$V = e;$		S
18	0,1,4,10	14			R($P3$)
19		0	a		S
20	0	3			R($P1$)
21		0	s		S
22	0	2	$\$end$		S
23	0,2	5			R($P0$)
24		0	$\$accept$		ACCEPT

Implementation with flex and bison
scanner.l

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

whitespace        [ \t\n]+
variable          [a-z]
constant          [0-9]

%%

{whitespace}      { }
"sin"             { return F; }
"+"               { return L; }
"*"               { return N; }
{variable}        { return V; }
{constant}        { return C; }
.                 { return yytext[0]; }

%%

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

Implementation with flex and bison

parser.y

```
%token V C F L N

%left L
%left N

%%

s : a
  | a s
  ;
a : V '=' e ';' ;
e : e L e
  | e N e
  | F '(' e ')'
  | V
  | C
  ;

%%
...
```

Implementation with `flex` and `bison`
`parser.y` (Code Section)

```
...

#include<stdio.h>

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;
}
```

Toward Single-Pass Compilation

- Parse Tree Printer
  - data associated with parse tree nodes (**unsigned int** by default)
  - access to data corresponding to left-hand side of production rule by $$
  - access to data corresponding to symbols on right-hand side of production rule by postion, i.e. $1, $2, ...
  - see code
- Parser/Unparser
  - association of user-defined data (e.g. **char**$*$) through redefinition of preprocessor macro YYSTYPE
  - see code

Toward Single-Pass Compilation of Tangent-Linear Code

... attribute grammars:

- assignment-level SAC requires unique auxilliary variable names $\rightarrow$ enumeration of subexpression
- enumeration of subexpressions requires number of nodes (subexpressions) in subtree
- implementation of inherited attributes through global variables in `bison`

S-Attributed Counting of Subexpressions

$$
\begin{array}{rll}
a : & V = e; & \\
e^l : & F(e^r) & \{e^l.s := e^r.s + 1 \ \} \\
: & e^{r_1} L e^{r_2} & \{e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \ \} \\
: & e^{r_1} N e^{r_2} & \{e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \ \} \\
: & V & \{ \ e^l.s := 1 \ \} \\
: & C & \{ \ e^l.s := 1 \ \}
\end{array}
$$

Implementation

```
a : V '=' e ';' { printf("%d\n",$3); } ;
e : e L e { $$=$1+$3+1; }
  | e N e { $$=$1+$3+1; }
  | F '(' e ')' { $$=$3+1; }
  | V { $$=1; }
  | C { $$=1; }
  ;
```

L-Attributed Enumeration of Subexpressions

$$
\begin{array}{rll}
a : V = e; & \{\ e.i := 0\ \} \\
e^l : F(e^r) & \{\ e^l.s := e^r.s + 1;\ e^r.i := e^l.i + 1\ \} \\
: e^{r_1} L e^{r_2} & \{\ e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \\
 & \quad e^{r_1}.i := e^l.i + 1;\ e^{r_2}.i := e^{r_1}.i + e^{r_1}.s\ \} \\
: e^{r_1} N e^{r_2} & \{\ e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \\
 & \quad e^{r_1}.i := e^l.i + 1;\ e^{r_2}.i := e^{r_1}.i + e^{r_1}.s\ \} \\
: V & \{\ e^l.s := 1\ \} \\
: C & \{\ e^l.s := 1\ \}
\end{array}
$$

```
%{ unsigned int sacvc; %}

%token V C F L N

%left L
%left N

%%

a : V '=' { sacvc=0; } e ';'
e : e L e { $$=sacvc++; }
  | e N e { $$=sacvc++; }
  | F '(' e ')' { $$=sacvc++; }
  | V { $$=sacvc++; }
  | C { $$=sacvc++; }
  ;

%%
...
```

Single-Pass Assignment-Level SAC Attribute Grammar

$$
\begin{aligned}
&(P3) \quad a : V = e; \quad && e.i = 0 \\
& && a.c = e.c \\
& && \quad + V.c + \text{"=v0;"} \\
&(P4) \quad e^l : e^{r_1} L e^{r_2} \quad && e^l.s = e^{r_1}.s + e^{r_2}.s + 1 \\
& && e^{r_1}.i = e^l.i + 1 \\
& && e^{r_2}.i = e^{r_1}.i + e^{r_1}.s \\
& && e^l.c = e^{r_1}.c + e^{r_2}.c \\
& && \quad + \text{"v"} + e^l.i + \text{"=v"} + e^{r_1}.i + L.c \\
& && \quad + \text{"v"} + e^{r_2}.i + \text{";"}
\end{aligned}
$$

$$
\begin{aligned}
(P5) e^l : e^{r_1} N e^{r_2} \quad e^l.s &= e^{r_1}.s + e^{r_2}.s + 1 \\
e^{r_1}.i &= e^l.i + 1 \\
e^{r_2}.i &= e^{r_1}.i + e^{r_1}.s \\
e^l.c &= e^{r_1}.c + e^{r_2}.c \\
&\quad + \text{"v"} + e^l.i + \text{"=v"} + e^{r_1}.i + N.c \\
&\quad + \text{"v"} + e^{r_2}.i + \text{";"}
\end{aligned}
$$

Shift-reduce conflicts are resolved by specifying the order of evaluation for associativity and operator precedence.

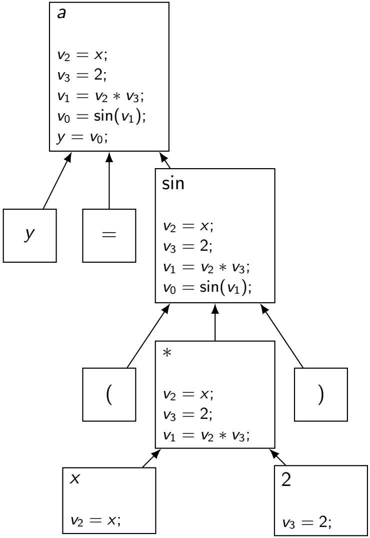
$$
\begin{aligned}
(P6) e^l : F(e^r) \quad & e^l.s = e^r.s + 1 \\
& e^r.i = e^l.i + 1 \\
& e^l.c = e^r.c \\
& \qquad + \text{"v"} + e^l.i + \text{"="} + F.c + \text{"(v"} + e^r.i + \text{");"} \\
(P7)\ e : V \quad & e.s = 1 \\
& e.c = \text{"v"} + e.i + \text{"="} + V.c + \text{";"} \\
(P8)\ e : C \quad & e.s = 1 \\
& e.c = \text{"v"} + e.i + \text{"="} + C.c + \text{";"}
\end{aligned}
$$

Single-Pass Assignment-Level SAC
Parsing $y = \sin(x * 2);$

i	PARSED	ACTION	$\$\$.i$	$\$\$.c$	Comment
0	V	S			
	...				
11	$V = F(V$	S			
7		R($P7$)	2	$\boxed{v_2 = x;}$	
	...				
13	V=F(eNC	S			
8		R($P8$)	3	$\boxed{v_3 = 2;}$	
13	V=F(eNe	S			
14		R($P5$)		$v_2 = x;$	$< \ldots = e^{r_1}.c$
				$v_3 = 2;$	$< \ldots = e^{r_2}.c$
			1	$\boxed{v_1 = v_2 * v_3;}$	$N.c = " * "$
					$e^{r_1}.i = 2,\ e^{r_2}.i = 3$
11	V=F(e	S			

15	V=F(e)	S			
18		R($P6$)		$v_2 = x;$	$<$
				$v_3 = 2;$	$<$
				$v_1 = v_2 * v_3;$	$< \ldots = e^r.c$
			0	$\boxed{v_0 = \sin(v_1);}$	$F.c = "\sin", e^r.i = 1$
4	V=e	S			
10	V=e;	S			
14		R($P3$)		$v_2 = x;$	$<$
				$v_3 = 2;$	$<$
				$v_1 = v_2 * v_3;$	$<$
				$v_0 = \sin(v_1);$	$< \ldots = e.c$
				$\boxed{y = v_0;}$	$V.c = "y", e.i = 0$
	...				
0	$accept	ACCEPT			

Single-Pass Assignment-Level SAC
Parse Tree



Single-Pass Assignment-Level SAC Implementation

See `codes/SD_SAC` and have fun!

Application of the syntax-directed assignment-level SAC compiler to the SL^2 program

```
x=x*y;
x=sin(x*y+3);
```

yields

```
v0=x; v1=y; v2=v0*v1; x=v2;
v0=x; v1=y; v2=v0*v1; v3=3; v4=v2+v3; v5=sin(v4); x=v5;
```

Single-Pass TLC

- attribute grammar
- implementation with `flex` and `bison`
- case studies (trafo + drivers)

Single-Pass Tangent-Linear Code
Attribute Grammar

$$
\begin{aligned}
(P3) \quad a : V = e; \quad e.i &= 0 \\
a.c &= e.c \\
&\quad + V.c + \text{"\_=v0\_;"} \\
&\quad + V.c + \text{"=v0;"}
\end{aligned}
$$

For $y = v_0$, we get $y^{(1)} = \frac{\partial y}{\partial v_0} \cdot v_0^{(1)} = v_0$.

Linear $(P4)$ and nonlinear $(P5)$ operators can be described by a single rule $P4/5$. The differences are restricted to the expressions for the local partial derivatives.

$$
\begin{aligned}
(P4/5) \quad e^l : e^{r_1} O e^{r_2} \quad & e^l.s = e^{r_1}.s + e^{r_2}.s + 1 \\
& e^{r_1}.i = e^l.i + 1 \\
& e^{r_2}.i = e^{r_1}.i + e^{r_1}.s \\
& e^l.c = e^{r_1}.c + e^{r_2}.c \\
& \qquad + \text{"v"} + e^l.i + \text{"\textvisiblespace="} + \partial_{e^{r_1}.i} O + \text{"}{*}\text{v"} + e^{r_2}.i \\
& \qquad + \text{"\textvisiblespace+"} + \partial_{e^{r_2}.i} O + \text{"}{*}\text{v"} + e^{r_1}.i + \text{"\textvisiblespace;"} \\
& \qquad + \text{"v"} + e^l.i + \text{"=v"} + e^{r_1}.i + O.c + \text{"v"} \\
& \qquad + e^{r_2}.i + \text{";"}
\end{aligned}
$$

where ...

... $O \in \{L, N\}$ and the local partial derivatives are

$$\partial_{e^{r_1}.i} L := 1 \quad ,$$

$$\partial_{e^{r_2}.i} L := \begin{cases} "1" & \text{if } L.c = "+" \\ "-1" & \text{if } L.c = "-" \end{cases} \quad ,$$

$$\partial_{e^{r_1}.i} N := \begin{cases} "\text{v}" + e^{r_2}.i & \text{if } N.c = "*" \\ "1/\text{v}" + e^{r_2}.i & \text{if } N.c = "/" \end{cases} \quad ,$$

and

$$\partial_{e^{r_2}.i} N := \begin{cases} "\text{v}" + e^{r_1}.i & \text{if } N.c = "*" \\ "-" + \partial_{e^{r_1}.i} N + " * " + \partial_{e^{r_1}.i} N + " * v" + e^{r_1}.i & \text{if } N.c = "/" \end{cases} \quad .$$

As before, shift-reduce conflicts are resolved by specifying the order of evaluation for associativity and operator precedence.

$$
\begin{aligned}
(P6) \quad e^l : F(e^r) \quad & e^l.s = e^r.s + 1 \\
& e^r.i = e^l.i + 1 \\
& e^l.c = e^r.c \\
& \qquad + \text{"v"} + e^l.i + \text{"\_="} + \partial_{e^r.i}F + \text{"* v"} + e^r.i + \text{"\_;"} \\
& \qquad + \text{"v"} + e^l.i + \text{"="} + F.c + \text{"(v"} + e^r.i + \text{");"}
\end{aligned}
$$

where

$$
\partial_{e^r.i}F := \begin{cases} \text{"cos(v"} + e^r.i + \text{")"} & \text{if } F.c = \text{"sin"} \\ \text{"}-\text{sin(v"} + e^r.i + \text{")"} & \text{if } F.c = \text{"cos"} \\ \text{"exp(v"} + e^r.i + \text{")"} & \text{if } F.c = \text{"exp"} \\ \vdots \quad \text{etc.} & \end{cases}
$$

$$
\begin{aligned}
(P7) \quad & e : V \quad & e.s &= 1 \\
& & e.c &= \text{"v"} + e.i + \text{"␣="} + V.c + \text{"␣;"} \\
& & & \quad + \text{"v"} + e.i + \text{"="} + V.c + \text{";"} \\
(P8) \quad & e : C \quad & e.s &= 1 \\
& & e.c &= \text{"v"} + e.i + \text{"␣=0;"} \\
& & & \quad + \text{"v"} + e.i + \text{"="} + C.c + \text{";"}
\end{aligned}
$$

Single-Pass Tangent-Linear Code
Parsing $y = \sin(x * 2);$

i	PARSED	ACTION	\$\$.i	\$\$.c
0	V	S		
	...			
11	$V = F(V$	S		
7		R($P7$)	1	$\boxed{v_2^{(1)} = x^{(1)};\ v_2 = x;}$
	...			
13	V=F(eNC	S		
8		R($P8$)	2	$\boxed{v_3^{(1)} = 0;\ v_3 = 2;}$
13	V=F(eNe	S		
14		R($P5$)	3	$v_2^{(1)} = x^{(1)};\ v_2 = x;$ $v_3^{(1)} = 0;\ v_3 = 2;$ $\boxed{v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ v_1 = v_2 * v_3;}$
11	V=F(e	S		
15	V=F(e)	S		

18		R($P6$)		$v_2^{(1)} = x^{(1)};\ \ v_2 = x;$
				$v_3^{(1)} = 0;\ \ v_3 = 2;$
				$v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ \ v_1 = v_2 * v_3;$
			4	$\boxed{v_0^{(1)} = \cos(v_1) * v_1^{(1)};\ \ v_0 = \sin(v_1);}$
4	V=e	S		
10	V=e;	S		
14		R($P3$)		$v_2^{(1)} = x^{(1)};\ \ v_2 = x;$
				$v_3^{(1)} = 0;\ \ v_3 = 2;$
				$v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ \ v_1 = v_2 * v_3;$
				$v_0^{(1)} = \cos(v_1) * v_1^{(1)};\ \ v_0 = \sin(v_1);$
				$\boxed{y^{(1)} = v_0^{(1)};\ \ y = v_0;}$
	...			
0	\$accept	ACCEPT		

Single-Pass Tangent-Linear Code Implementation

See `codes/SD_TLC` and have fun!

Application of the syntax-directed tangent-linear code compiler to the SL program

```
if(x<y) {
  x=sin(x);
  while (y<x) { x=sin(x*3); }
  y=4*x+y;
}
```

yields

```
if (x<y) {
  v0_=x_; v0=x;
  v1_=cos(v0)*v0_; v1=sin(v0);
  x_=v1_; x=v1;
  while (y<x) {
    v0_=x_; v0=x;
    v1_=0; v1=3;
    v2_=v0_*v1+v0*v1_; v2=v0*v1;
    v3_=cos(v2)*v2_; v3=sin(v2);
    x_=v3_; x=v3;
  }
  v0_=0; v0=4;
  v1_=x_; v1=x;
  v2_=v0_*v1+v0*v1_; v2=v0*v1;
  v3_=y_; v3=y;
  v4_=v2_+v3_; v4=v2+v3;
  y_=v4_; y=v4;
}
```

Outlook

- lectures
- theses
- jobs

Race

Gradient of

$$y = \left(\sum_{i=0}^{n-1} x_i \right)^2$$

in

- tangent-linear mode
- adjoint mode

for $n = 100, 500, 1000, \ldots$.

Race – Results