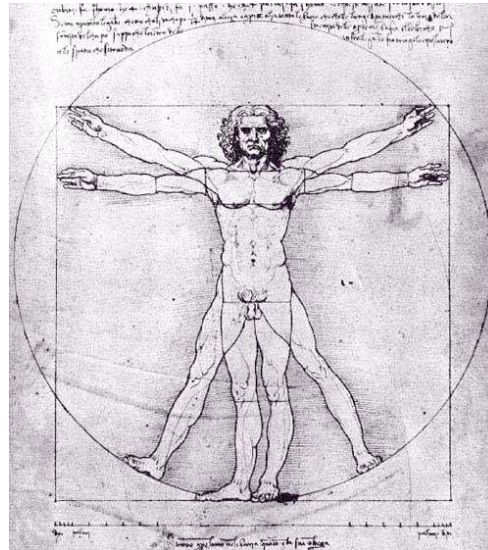


# Anatomy of a Small Compiler

COMS W4115



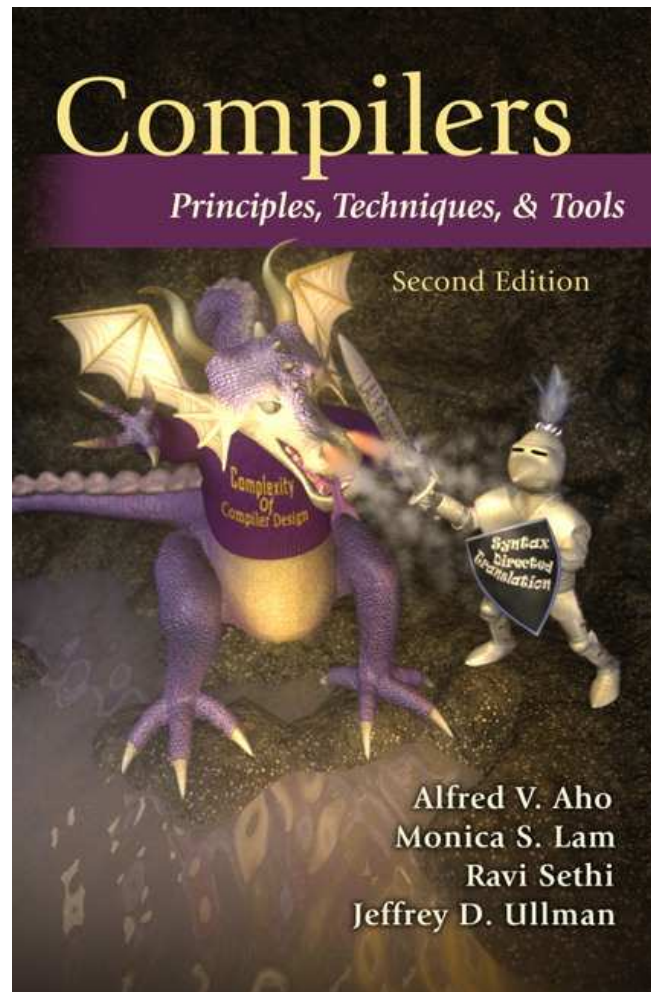
Prof. Stephen A. Edwards

Fall 2006

Columbia University

Department of Computer Science

# Appendix A of the Dragon Book



# A simple C-like language

```
{
  int i; int j;
  float[10][10] a;
  i = 0;
  while ( i < 10 ) {
    j = 0;
    while ( j < 10 ) {
      a[i][j] = 0;
      j = j+1;
    }
    i = i+1;
  }
  i = 0;
  while ( i < 10 ) {
    a[i][i] = 1;
    i = i+1;
  }
}
```

```
L1:  i = 0
L3:  iffalse i < 10 goto L4
L5:  j = 0
L6:  iffalse j < 10 goto L7
L8:  t1 = i * 80
      t2 = j * 8
      t3 = t1 + t2
      a [ t3 ] = 0
L9:  j = j + 1
      goto L6
L7:  i = i + 1
      goto L3
L4:  i = 0
L10: iffalse i < 10 goto L2
L11: t4 = i * 80
      t5 = i * 8
      t6 = t4 + t5
      a [ t6 ] = 1
L12: i = i + 1
      goto L10
L2:
```

# The Scanner

```
class MyLexer extends Lexer;  
options { k = 2; }
```

```
WHITESPACE : ( ' ' | '\t' | '\n' { newline(); } )+  
            { $setType(Token.SKIP); } ;
```

```
protected DIGITS : ('0'..'9')+ ;
```

```
NUM : DIGITS ( '.' DIGITS { $setType(REAL); } )? ;
```

```
AND :    "&&" ;    LE :    "<=" ;    SEMI :    ';' ;  
OR  :    "||" ;    GT :    '>' ;    LPAREN : '(' ;  
ASSIGN : '=' ;    GE :    ">=" ;    RPAREN : ')' ;  
EQ  :    "==" ;    LBRACE : '{' ;    PLUS :    '+' ;  
NOT :    '!' ;    RBRACE : '}' ;    MINUS : '-' ;  
NE  :    "!=" ;    LBRACK : '[' ;    MUL :    '*' ;  
LT  :    '<' ;    RBRACK : ']' ;    DIV :    '/' ;
```

```
ID : ( '_' | 'a'..'z' | 'A'..'Z' )  
     ( '_' | 'a'..'z' | 'A'..'Z' | '0'..'9' )*
```

# The Parser: Statements

```
class MyParser extends Parser;
options { buildAST = true; }
tokens { NEGATE; DECLS; }
```

```
program : LBRACE^ decls (stmt} RBRACE! ;
```

```
decls : (decl)* { #decls = #([DECLS, "DECLS"], #decls); } ;
```

```
decl : ("int" | "char" | "bool" | "float")
      (LBRACK! NUM RBRACK!)* ID SEMI! ;
```

```
stmt : loc ASSIGN^ bool SEMI!
      | "if"^ LPAREN! bool RPAREN! stmt
      (options {greedy=true;}: "else"! stmt)?
      | "while"^ LPAREN! bool RPAREN! stmt
      | "do"^ stmt "while"! LPAREN! bool RPAREN! SEMI!
      | "break" SEMI!
      | program
      | SEMI
      ;
```

# The Parser: Expressions

```
bool      : join (OR^ join)* ;

join      : equality (AND^ equality)* ;

equality  : rel ((EQ^ | NE^ ) rel)* ;

rel       : expr ((LT^ | LE^ | GT^ | GE^ ) expr)* ;

expr      : term ((PLUS^ | MINUS^ ) term)* ;

term      : unary ((MUL^ | DIV^ ) unary)* ;

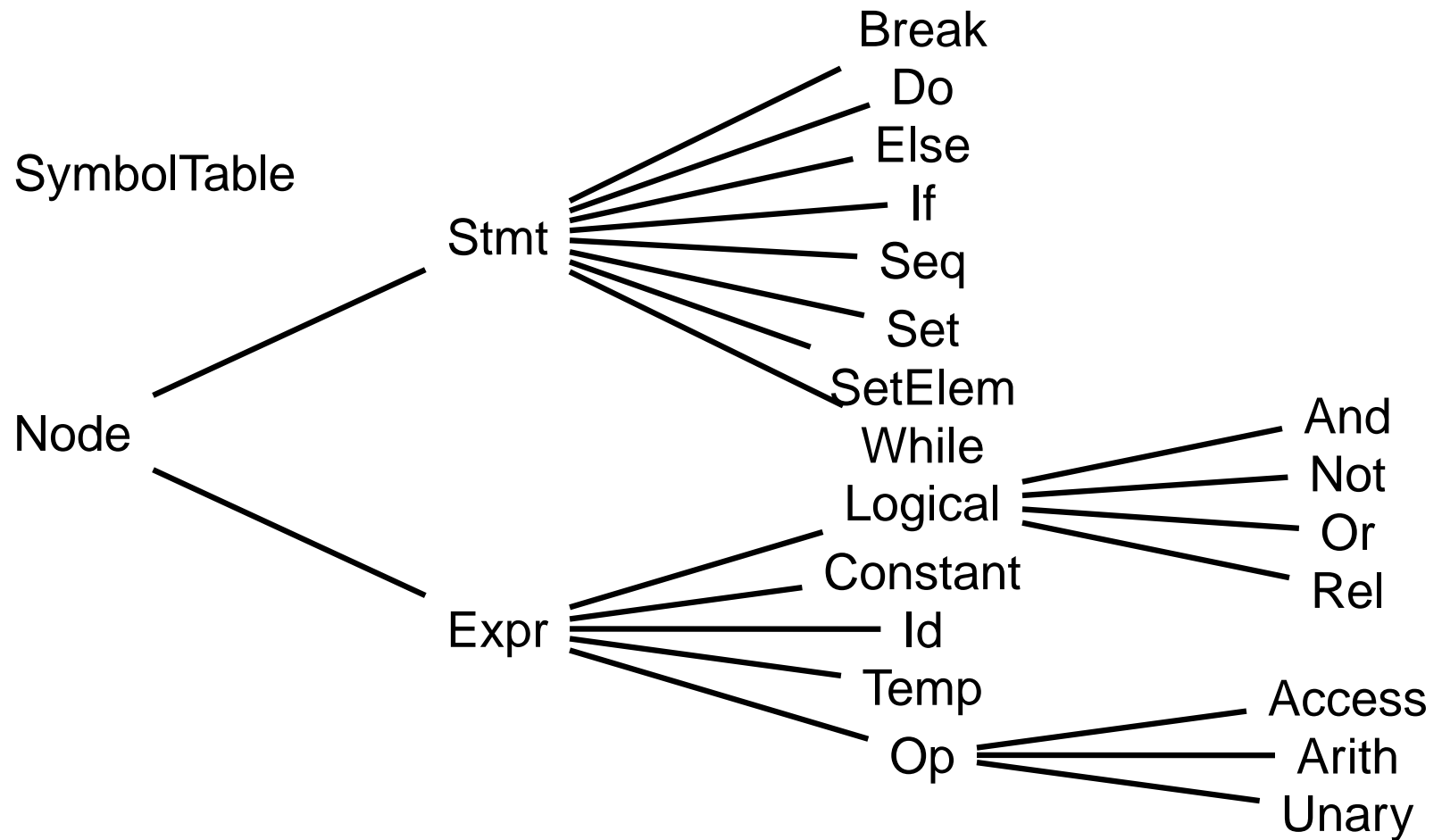
unary     : MINUS^ unary { #unary.setType(NEGATE); }
           | NOT^ unary | factor ;

factor    : LPAREN! bool RPAREN! | loc
           | NUM | REAL | "true" | "false" ;

loc       : ID^ (LBRACK! bool RBRACK!)* ;
```

# The IR Classes

Type ——— Array



# SymbolTable.java

```
public class SymbolTable {
    private Hashtable table;
    protected SymbolTable outer;
    public SymbolTable(SymbolTable st) {
        table = new Hashtable();
        outer = st;
    }
    public void put(String token, Type t, int b) {
        table.put(token, new Id(token, t, b));
    }
    public Id get(String token) {
        for (SymbolTable tab = this ; tab != null ;
            tab = tab.outer) {
            Id id = (Id)(tab.table.get(token));
            if ( id != null ) return id;
        }
        return null;
    }
}
```



# Type.java (Basic types)

```
public class Type {
    public int width = 0;
    public String name = "";
    public Type(String s, int w) { name = s; width = w; }
    public static final Type
        Int = new Type("int", 4), Float = new Type("float", 8),
        Char = new Type("char", 1), Bool = new Type("bool", 1);
    public static boolean numeric(Type p) {
        return p == Type.Char || p == Type.Int ||
            p == Type.Float; }
    public static Type max(Type p1, Type p2) {
        if (!numeric(p1) || !numeric(p2)) return null;
        else if (p1 == Type.Float || p2 == Type.Float)
            return Type.Float;
        else if (p1 == Type.Int || p2 == Type.Int)
            return Type.Int;
        else return Type.Char;
    }
}
```

# Node.java (Stmts and Exprs)

```
public class Node {
    void error(String s) { throw new Error(s); }
    static int labels = 0;
    public static int newlabel() { return ++labels; }
    public static void emitlabel(int i) {
        System.out.print("L" + i + ":");
    }
    public static void emit(String s) {
        System.out.println("\t" + s);
    }
}
```

# Expr.java (has a type)

```
public class Expr extends Node {
    public String s;
    public Type type;
    Expr(String tok, Type p) { s = tok; type = p; }
    public Expr gen() { return this; }
    public Expr reduce() { return this; }
    public void jumping(int t, int f) {
        emitjumps(toString(), t, f); }
    public void emitjumps(String test, int t, int f) {
        if (t != 0 && f != 0) {
            emit("if " + test + " goto L" + t);
            emit("goto L" + f);
        } else if (t != 0) emit("if " + test + " goto L" + t);
        else if (f != 0) emit("iffalse " + test + " goto L" + f);
    }
    public String toString() { return s; }
}
```

# Op.java (operator)

```
public class Op extends Expr {
    public Op(String tok, Type p) { super(tok,p); }
    public Expr reduce() {
        Expr x = gen();
        Temp t = new Temp(type);
        emit(t.toString() + " = " + x.toString());
        return t;
    }
}
```

# Arith.java (binary arithmetic)

```
public class Arith extends Op {
    public Expr expr1, expr2;
    public Arith(String op, Expr x1, Expr x2) {
        super(op, null); expr1 = x1;  expr2 = x2;
        type = Type.max(expr1.type, expr2.type);
        if (type == null) error("type error");
    }
    public Expr gen() { return new Arith(s, expr1.reduce(),
                                          expr2.reduce()); }
    public String toString() {
        return expr1.toString() + " " + s + " " +
               expr2.toString();
    }
}
```

# Logical.java (logical operator)

```
public class Logical extends Expr {
    public Expr expr1, expr2;
    Logical(String tok, Expr x1, Expr x2) {
        super(tok, null); expr1 = x1; expr2 = x2;
        type = check(expr1.type, expr2.type);
        if (type == null) error("type error");
    }
    public Type check(Type p1, Type p2) {
        if (p1 == Type.Bool && p2 == Type.Bool) return Type.Bool;
        else return null;
    }
    public Expr gen() {
        int f = newlabel(); int a = newlabel();
        Temp temp = new Temp(type);
        this.jumping(0, f);
        emit(temp.toString() + " = true");
        emit("goto L" + a); emitlabel(f);
        emit(temp.toString() + " = false");
        emitlabel(a);
        return temp;
    }
    public String toString(){
        return expr1.toString() + " " + s + " " + expr2.toString();
    }
}
```

# And.java (logical AND)

```
public class And extends Logical{
    public And(Expr x1, Expr x2) { super("&&", x1, x2); }
    public void jumping(int t, int f) {
        int label = f != 0 ? f : newlabel();
        expr1.jumping(0, label);
        expr2.jumping(t, f);
        if (f == 0) emitlabel(label);
    }
}
```

# Stmt.java (statements)

```
public class Stmt extends Node {  
    public Stmt() {}  
    public static Stmt Null = new Stmt();  
    public void gen(int b, int a) {}  
    int after = 0;  
    public static Stmt Enclosing = Stmt.Null;  
}
```



# While.java (while loop)

```
public class While extends Stmt{
    Expr expr;
    Stmt stmt;
    public While() { expr = null; stmt = null; }
    public void init(Expr x, Stmt s){
        expr = x;
        stmt = s;
        if (expr.type != Type.Bool)
            expr.error("boolean required in while");
    }
    public void gen(int b, int a){
        after = a;
        expr.jumping(0, a);
        int label = newlabel();
        emitlabel(label);
        stmt.gen(label, b);
        emit("goto L" + b);
    }
}
```

# Tree Walker (Program)

```
class MyWalker extends TreeParser;
{
    SymbolTable top = null;
    int used = 0; // Number of bytes in local declarations
}

program returns [Stmt s]
{ s = null; Stmt s1; }
: #(LBRACE
    { SymbolTable saved_environment = top;
      top = new SymbolTable(top); }
  decls
  s=stmts
  { top = saved_environment; }
)
;
```

# Tree Walker (Declarations)

```
decls
{ Type t = null; }
: #(DECLS
    (t=type ID { top.put(#ID.getText(), t, used);
                used += t.width; } *) )
;

type returns [Type t]
{ t = null; }
: ( "bool"   { t = Type.Bool;   }
  | "char"   { t = Type.Char;   }
  | "int"    { t = Type.Int;    }
  | "float"  { t = Type.Float;  } )
(t=dims[t])?
;

dims[Type t1] returns [Type t]
{ t = t1; }
: NUM (t=dims[t])?
  { t = new Array(Integer.parseInt(#NUM.getText()), t); }
;
```

# Tree Walker (Statements)

```
stmts returns [Stmt s]
{ s = null; Stmt s1; }
: s=stmt (s1=stmts { s = new Seq(s, s1); } )?
;
```

```
stmt returns [Stmt s]
{ Expr e1, e2;
  s = null;
  Stmt s1, s2;
}
: #(ASSIGN e1=expr e2=expr
  { if (e1 instanceof Id) s = new Set((Id) e1, e2);
    else s = new SetElem((Access) e1, e2);
  }
)
| #("if" e1=expr s1=stmt
  ( s2=stmt { s = new Else(e1, s1, s2); }
  | /* nothing */ { s = new If(e1, s1); } ))
```

```

| #("while"
  { While whilenode = new While();
    s2 = Stmt.Enclosing;
    Stmt.Enclosing = whilenode; }
  e1=expr
  s1=stmt
  { whilenode.init(e1, s1);
    Stmt.Enclosing = s2;
    s = whilenode; } )
| #("do"
  { Do donode = new Do();
    s2 = Stmt.Enclosing;
    Stmt.Enclosing = donode; }
  s1=stmt
  e1=expr
  { donode.init(s1, e1);
    Stmt.Enclosing = s2;
    s = donode; } )
| "break" { s = new Break(); }
| s=program
| SEMI { s = Stmt.Null; }
;

```

# Tree Walker (Expressions)

expr returns [Expr e]

```
{
  Expr a, b;
  e = null;
}
: #(OR      a=expr b=expr { e = new Or(a, b); } )
| #(AND     a=expr b=expr { e = new And(a, b); } )
| #(EQ      a=expr b=expr { e = new Rel("==", a, b); } )
| #(NE      a=expr b=expr { e = new Rel("!=", a, b); } )
| #(LT      a=expr b=expr { e = new Rel("<", a, b); } )
| #(LE      a=expr b=expr { e = new Rel("<=", a, b); } )
| #(GT      a=expr b=expr { e = new Rel(">", a, b); } )
| #(GE      a=expr b=expr { e = new Rel(">=", a, b); } )
| #(PLUS    a=expr b=expr { e = new Arith("+", a, b); } )
| #(MINUS   a=expr b=expr { e = new Arith("-", a, b); } )
| #(MUL     a=expr b=expr { e = new Arith("#", a, b); } )
| #(DIV     a=expr b=expr { e = new Arith("/", a, b); } )
| #(NOT     a=expr      { e = new Not(a); } )
| #(NEGATE  a=expr      { e = new Unary("-", a); } )
| NUM      { e = new Constant(#NUM.getText(), Type.Int); }
```

```

| REAL { e = new Constant(#REAL.getText(), Type.Float); }
| "true"          { e = Constant.True; }
| "false"         { e = Constant.False; }
| #(ID
  { Id i = top.get(#ID.getText());
    if (i == null)
      System.out.println(#ID.getText() + " undeclared");
    e = i;
  }
  ( a=expr
    { Type type = e.type;
      type = ((Array)type).of;
      Expr w = new Constant(type.width);
      Expr loc = new Arith("?", a, w);
    }
    ( a=expr
      { type = ((Array)type).of;
        w = new Constant(type.width);
        loc = new Arith("?", loc, new Arith("#", a, w));
      }
    )?
    { e = new Access(i, loc, type); }
  )?
)
;

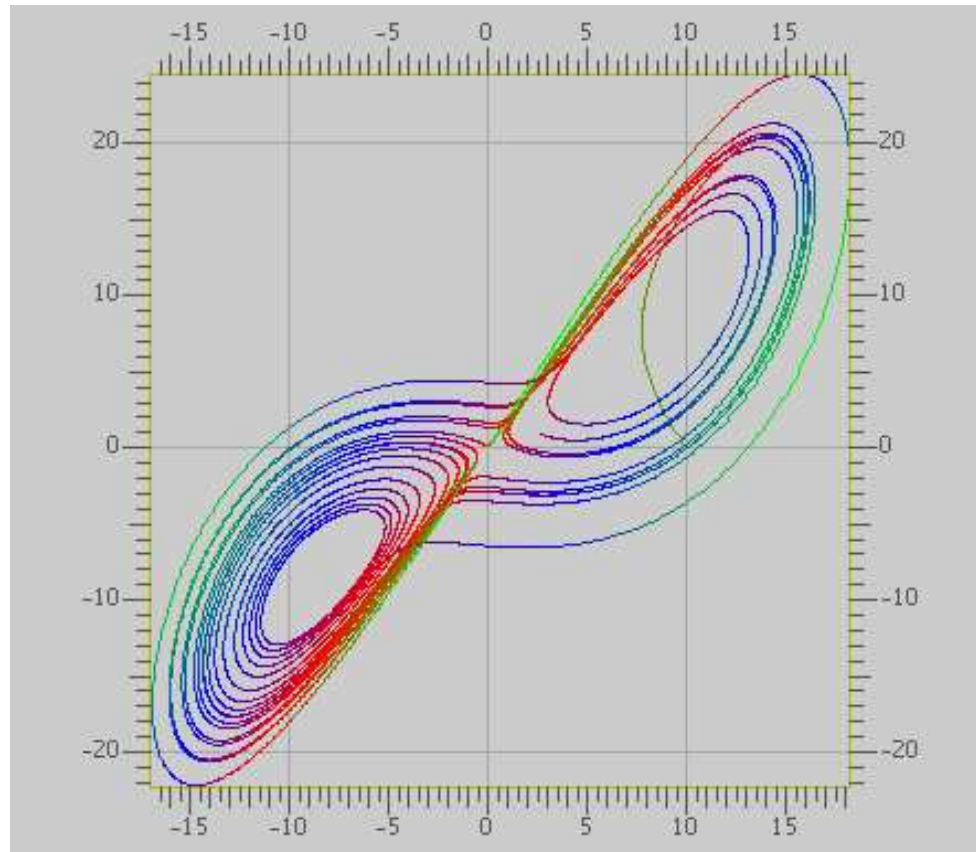
```

# Statistics

<b>File</b>	<b>Role</b>	<b># lines</b>
grammar.g	Scanner/Parser/Walker	190
Main.java	main() procedure	27
SymbolTable.java	Symbol table	20
Type.java	Basic types	19
Array.java	Array type	10
Node.java	Statements and Expressions	7
Stmt.java	A node	7
Break.java	break statement	10
Do.java	do-while statement	17
Else.java	if-else statement	17
If.java	if statement	14
Seq.java	statement sequences	15
SetElem.java	assign to array	22
Set.java	assign to scalar	19
While.java	while statement	18
Expr.java	A node	16
Constant.java	constant expression	11
Id.java	variable identifier	4
Temp.java	temporary variable	6
Op.java	operator (expression)	9
Access.java	array index	10
Arith.java	arithmetic expression	12
Unary.java	unary negation	10
Logical.java	logical operator (expression)	27
And.java	logical AND	9
Not.java	logical NOT	5
Or.java	logical OR	9
Rel.java	<, =, etc.	14
<b>total</b>		<b>550</b>



# Mx



# Mx

A Programming Language for Scientific Computation

Resembles Matlab, Octave, Mathematica, etc.

Project from Spring 2003

Authors:

Tiantian Zhou

Hanhua Feng

Yong Man Ra

Chang Woo Lee

# Example

Plotting the Lorenz equations

$$\frac{dy_0}{dt} = \alpha(y_1 - y_0)$$

$$\frac{dy_1}{dt} = y_0(r - y_2) - y_1$$

$$\frac{dy_2}{dt} = y_0 y_1 - by_2$$

# Mx source part 1

```
/* Lorenz equation parameters*/

a = 10;
b = 8/3.0;
r = 28;

/* Two-argument function returning a vector*/
func Lorenz ( y, t ) = [ a*(y[1]-y[0]);
                        -y[0]*y[2] + r*y[0] - y[1];
                        y[0]*y[1] - b*y[2] ];

/* Runge-Kutta numerical integration procedure*/
func RungeKutta( f, y, t, h ) {
    k1 = h * f( y, t );
    k2 = h * f( y+0.5*k1, t+0.5*h );
    k3 = h * f( y+0.5*k2, t+0.5*h );
    k4 = h * f( y+k3, t+h );
    return y + (k1+k4)/6.0 + (k2+k3)/3.0;
}
```

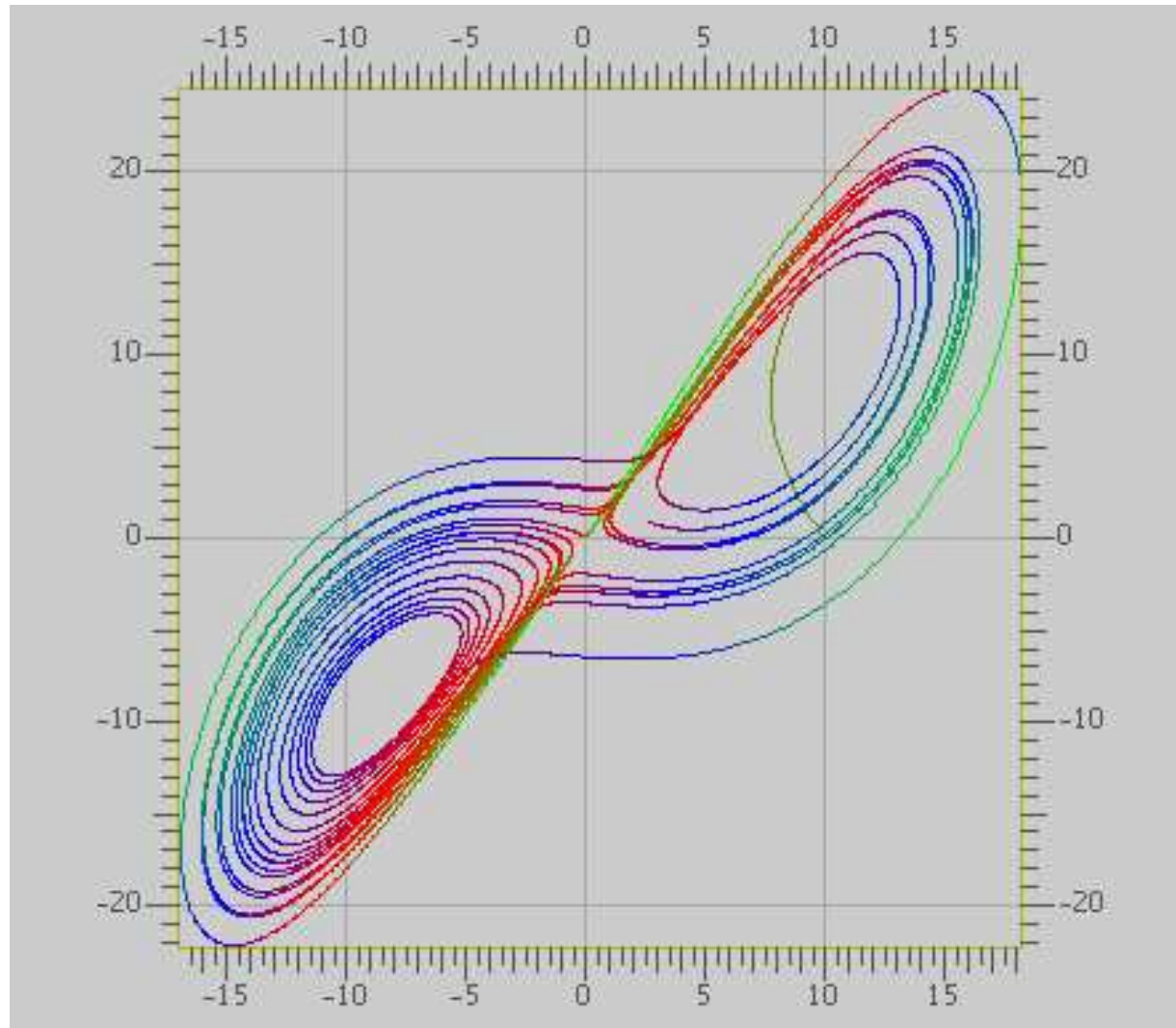
# Mx source part 2

```
/* Parameters for the procedure*/
N = 20000;
p = zeros(N+1,3);
t = 0.0;
h = 0.001;
x = [ 10; 0; 10 ];
p[0,:] = x'; /* matrix transpose */

for ( i = 1:N ) {
    x = RungeKutta( Lorenz, x, t, h );
    p[i,:] = x';
    t += h;
}

colormap(3);
plot(p);
return 0;
```

# Result



<b>file</b>	<b>lines</b>	<b>role</b>
		<b>Scanner and Parser: Builds the tree</b>
grammar.g	314	Lexer/Parser (ANTLR source)

**Interpreter: Walks the tree, invokes objects' methods**

walker.g	170	Tree Walker (ANTLR source)
MxInterpreter.java	359	Function invocation, etc.
MxSymbolTable.java	109	Name-to-object mapping

**Top-level: Invokes the interpreter**

MxMain.java	153	Command-line interface
MxException.java	13	Error reporting

**Runtime system: Represents data, performs operations**

MxDataType.java	169	Base class
MxBool.java	63	Booleans
MxInt.java	152	Integers
MxDouble.java	142	Floating-point
MxString.java	47	String
MxVariable.java	26	Undefined variable
MxFunction.java	81	User-defined functions
MxInternalFunction.m4	410	sin, cos, etc. (macro processed)
jamaica/Matrix.java	1387	Matrices
MxMatrix.java	354	Wrapper
jamaica/Range.java	163	e.g., 1:10
MxRange.java	67	Wrapper
jamaica/BitArray.java	226	Matrix masks
MxBitArray.java	47	Wrapper
jamaica/Painter.java	339	Bitmaps
jamaica/Plotter.java	580	2-D plotting

---

total 5371

# The Scanner

```
class MxAntlrLexer extends Lexer;

options {
    k = 2;
    charVocabulary = '\3'..'377';
    testLiterals = false;
    exportVocab = MxAntlr;
}

protected ALPHA : 'a'..'z' | 'A'..'Z' | '_';

protected DIGIT : '0'..'9';

WS : (' ' | '\t')+ { $setType(Token.SKIP); } ;

NL : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
    { $setType(Token.SKIP); newline(); } ;
```



# The Scanner

```
COMMENT : ( "/*" ( options {greedy=false;} :  
            NL  
            | ~( '\n' | '\r' )  
            )* "*/"  
        | "//" ( ~( '\n' | '\r' ))* NL  
    ) { setType(Token.SKIP); } ;
```

```
LDV_LDVEQ : "/" (  
                ('=' ) => '=' { setType(LDVEQ); }  
                | { setType(LDV); }  
            );
```

# The Scanner

LPAREN : '(' ;

RPAREN : ')' ;

/\* ... \*/

TRSP : '\\'' ;

COLON : ':' ;

DCOLON : '::' ;

ID options { testLiterals = true; }

: ALPHA (ALPHA|DIGIT)\* ;

NUMBER : (DIGIT)+ ('.' (DIGIT)\*)?

(( 'E' | 'e' ) ('+' | '-')? (DIGIT)+)? ;

STRING : '\"'!

( ~('\"' | '\\n') | ('\"'! '\"') \*

'\"'! ;

# The Parser: Top-level

```
class MxAntlrParser extends Parser;

options {
    k = 2;
    buildAST = true;
    exportVocab = MxAntlr;
}

tokens {
    STATEMENT;
    FOR_CON;
    /* ... */
}

program : ( statement | func_def ↯ EOF!
          { #program = #([STATEMENT,"PROG"], program); }
          ;
```

# The Parser: Statements

```
statement
: for_stmt
| if_stmt
| loop_stmt
| break_stmt
| continue_stmt
| return_stmt
| load_stmt
| assignment
| func_call_stmt
| LBRACE! (statement)* RBRACE!
  {#statement = #([STATEMENT,"STATEMENT"], statement); }
;
```

# The Parser: Statements 1

```
for_stmt : "for" ^ LPAREN! for_con RPAREN! statement ;
```

```
for_con : ID ASGN! range (COMMA! ID ASGN! range)*  
        { #for_con = #([FOR_CON,"FOR_CON"], for_con); }  
        ;
```

```
if_stmt : "if" ^ LPAREN! expression RPAREN! statement  
        (options {greedy = true;}: "else"! statement )?  
        ;
```

```
loop_stmt! : "loop" ( LPAREN! id:ID RPAREN! )? stmt:statement  
           { if ( null == #id )  
             #loop_stmt = #([LOOP,"loop"], #stmt);  
           else  
             #loop_stmt = #([LOOP,"loop"], #stmt, #id);  
           } ;
```

# The Parser: Statements 2

```
break_stmt : "break" ^ (ID)? SEMI! ;
continue_stmt : "continue" ^ (ID)? SEMI! ;
return_stmt : "return" ^ (expression)? SEMI! ;
load_stmt : "include" ^ STRING SEMI! ;
```

assignment

```
    : l_value ( ASGN ^ | PLUSEQ ^ | MINUSEQ ^ | MULTEQ ^
                | LDVEQ ^ | MODEQ ^ | RDVEQ ^
                ) expression SEMI!
```

;

```
func_call_stmt : func_call SEMI! ;
```

func\_call

```
    : ID LPAREN! expr_list RPAREN!
      { #func_call = #([FUNC_CALL,"FUNC_CALL"], func_call); }
```

;

# The Parser: Function Definitions

`func_def`

```
: "func" ^ ID LPAREN! var_list RPAREN! func_body  
;
```

`var_list`

```
: ID ( COMMA! ID )*  
  { #var_list = #([VAR_LIST,"VAR_LIST"], var_list); }  
| { #var_list = #([VAR_LIST,"VAR_LIST"], var_list); }  
;
```

`func_body`

```
: ASGN! a:expression SEMI!  
  { #func_body = #a; }  
| LBRACE! (statement)* RBRACE!  
  { #func_body = #([STATEMENT,"FUNC_BODY"], func_body); }  
;
```

# The Parser: Expressions

```
expression : logic_term ( "or" ^ logic_term * ) ;
logic_term : logic_factor ( "and" ^ logic_factor * ) ;
logic_factor : ("not" ^ )? relat_expr ;
relat_expr : arith_expr ( (GE ^ | LE ^ | GT ^
                          | LT ^ | EQ ^ | NEQ ^ ) arith_expr )? ;
arith_expr : arith_term ( (PLUS ^ | MINUS ^ ) arith_term * ) ;
arith_term : arith_factor
            ( (MULT ^ | LDV ^ | MOD ^ | RDV ^ ) arith_factor * ) ;
arith_factor
: PLUS! r_value
  { #arith_factor = #([UPLUS,"UPLUS"], arith_factor); }
| MINUS! r_value
  { #arith_factor = #([UMINUS,"UMINUS"], arith_factor); }
| r_value (TRSP ^)*;
r_value
: l_value | func_call | NUMBER | STRING | "true" | "false"
| array | LPAREN! expression RPAREN! ;
l_value : ID ^ ( LBRK! index RBRK! † ) ;
```



# The Walker: Top-level

```
{
    import java.io.*;
    import java.util.*;
}

class MxAntlrWalker extends TreeParser;
options{
    importVocab = MxAntlr;
}

{
    static MxDataType null_data = new MxDataType( "<NULL>" );
    MxInterpreter ipt = new MxInterpreter();
}
```

# The Walker: Expressions

```
expr returns [ MxDataType r ]
{
    MxDataType a, b;
    Vector v;
    MxDataType[] x;
    String s = null;
    String[] sx;
    r = null_data;
}
: #("or" a=expr right_or:.)
  { if ( a instanceof MxBool )
    r = ( ((MxBool)a).var ? a : expr(#right_or) );
    else
    r = a.or( expr(#right_or) );
  }
| #("and" a=expr right_and:.)
  { if ( a instanceof MxBool )
    r = ( ((MxBool)a).var ? expr(#right_and) : a );
    else
    r = a.and( expr(#right_and) );
  }
}
```

# The Walker: Simple operators

```
| #("not" a=expr)           { r = a.not(); }
| #(GE a=expr b=expr)      { r = a.ge( b ); }
| #(LE a=expr b=expr)      { r = a.le( b ); }
| #(GT a=expr b=expr)      { r = a.gt( b ); }
| #(LT a=expr b=expr)      { r = a.lt( b ); }
| #(EQ a=expr b=expr)      { r = a.eq( b ); }
| #(NEQ a=expr b=expr)     { r = a.ne( b ); }
| #(PLUS a=expr b=expr)    { r = a.plus( b ); }
| #(MINUS a=expr b=expr)   { r = a.minus( b ); }
| #(MULT a=expr b=expr)    { r = a.times( b ); }
| #(LDV a=expr b=expr)     { r = a.lfracts( b ); }
| #(RDV a=expr b=expr)     { r = a.rfracts( b ); }
| #(MOD a=expr b=expr)     { r = a.modulus( b ); }
| #(COLON (c1:. (c2:.)?)?)
| {
|     r = MxRange.create( (null==#c1) ? null : expr(#c1),
|                         (null==#c2) ? null : expr(#c2) );
| }
| #(ASGN a=expr b=expr)    { r = ipt.assign( a, b ); }
| #(FUNC_CALL a=expr x=mexpr){ r = ipt.funcInvoke(this, x); }
```

# The Walker: Literals, Variables, and Functions

```
| #(ARRAY                                { v = new Vector(); }
    (a=expr                               { v.add( a ); }
    )*
    ) { r = MxMatrix.joinVert( ipt.convertExprList( v ) ); }
| #(ARRAY_ROW                            { v = new Vector(); }
    (a=expr                               { v.add( a ); }
    )+
    ) { r = MxMatrix.joinHori( ipt.convertExprList( v ) ); }
| num:NUMBER                             { r = ipt.getNumber( num.getText() ); }
| str:STRING                             { r = new MxString( str.getText() ); }
| "true"                                 { r = new MxBool( true ); }
| "false"                                { r = new MxBool( false ); }
| #(id:ID                                 { r = ipt.getVariable( id.getText() ); }
    ( x=mexpr { r = ipt.subMatrix( r, x ); } *)
    )
| #("func" fname:ID sx=vlist fbody:.)
  { ipt.funcRegister( fname.getText(), sx, #fbody ); }
```

# The Walker: For and If statements

```
| #("for" x=mexpr forbody:.)
  {
    MxInt[] values = ipt.forInit( x );
    while ( ipt.forCanProceed( x, values ) ) {
      r = expr( #forbody );
      ipt.forNext( x, values );
    }
    ipt.forEnd( x );
  }
| #("if" a=expr thenp:. (elsep:.)?)
  {
    if ( !( a instanceof MxBool ) )
      return a.error( "if: expression should be bool" )
    if ( ((MxBool)a).var )
      r = expr( #thenp );
    else if ( null != elsep )
      r = expr( #elsep );
  }
```

# The Walker: Multiple expressions

```
mexpr returns [ MxDataType[] rv ]
{
    MxDataType a;
    rv = null;
    Vector v;
}
: #(EXPR_LIST
    ( a=expr
      )*
    )
  { rv = ipt.convertExprList( v ); }
| a=expr
  { rv = new MxDataType[1]; rv[0] = a; }
| #(FOR_CON
    ( s:ID a=expr
      )+
    )
  { v = new Vector();
    { a.setName( s.getText() ); v.add(a); }
    { rv = ipt.convertExprList( v ); }
;

```

# The Walker: Variable list

```
vlist returns [ String[] sv ]
{
    Vector v;
    sv = null;
}
: #(VAR_LIST      { v = new Vector(); }
   (s:ID         { v.add( s.getText() ); }
   )*)
)                { sv = ipt.convertVarList( v ); }
;
```