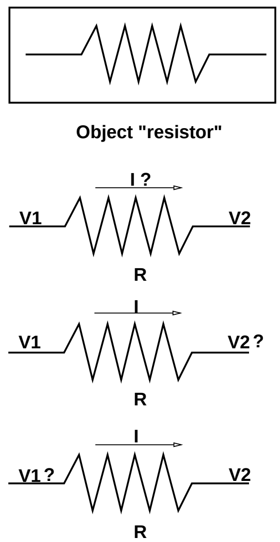
Objects, Re-use, and Causality



V1 − V2 = R*I

I = (V1−V2)/R

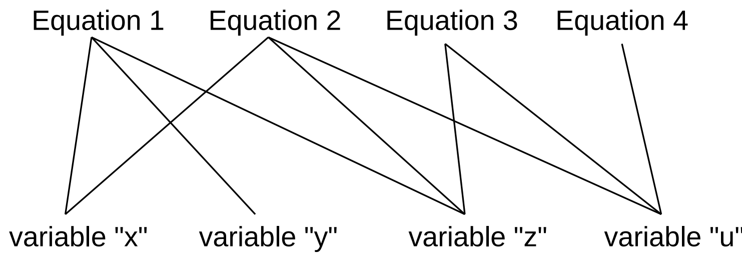
V2 = V1 − R*I

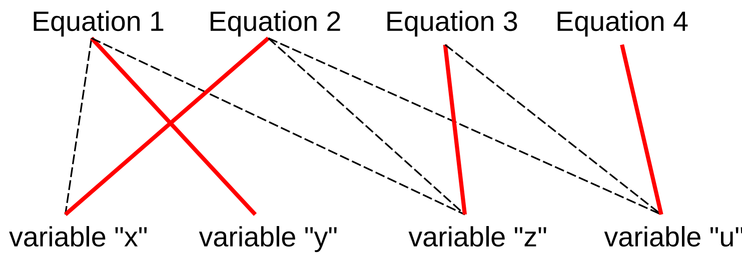
V1 = V2 + R*I

Causality Assignment

$$\left\{\begin{array}{rcll} x+y+z & = & 0 & \text{Equation 1} \\ x+3z+u^2 & = & 0 & \text{Equation 2} \\ z-u-16 & = & 0 & \text{Equation 4} \\ u-5 & = & 0 & \text{Equation 4} \end{array}\right.$$

Causality Assignment = bipartite (dependency) graph maximum cardinality matching



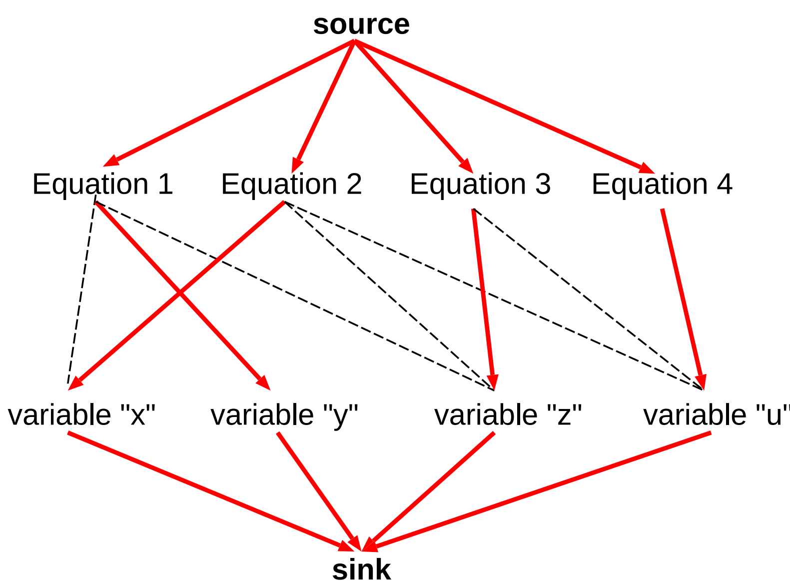


Causality Assignment: causality assigned

$$\left\{\begin{array}{rcll} x+\underline{y}+z & = & 0 & \text{Equation 1} \\ \underline{x}+3z+u^2 & = & 0 & \text{Equation 2} \\ \underline{z}-u-16 & = & 0 & \text{Equation 4} \\ \underline{u}-5 & = & 0 & \text{Equation 4} \end{array}\right.$$

$$\left\{\begin{array}{rcl} \underline{y} & = & -x-z \\ \underline{x} & = & -3z-u^2 \\ \underline{z} & = & u+16 \\ \underline{u} & = & 5 \end{array}\right.$$

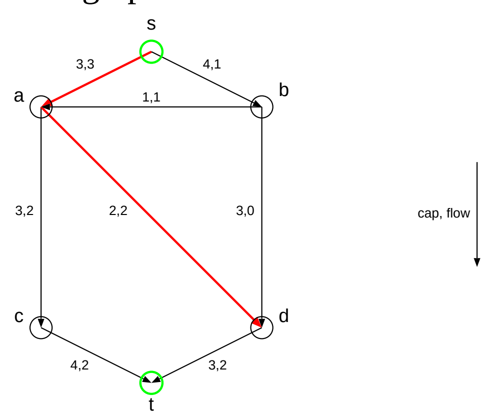
causality assignment: network flow



Network Flow Problems

$$G = [V, E]$$

Directed graph G with *source* s and *sink* t



Network Flow: definitions

Positive *capacity* $cap(v,w)$ on every edge $[v,w]$.
$cap(v,w) = 0$ if $[v,w]$ is not an edge.
A *flow* on G is any real-valued function f with properties:

1. *skew symmetry*. $f(v,w) = -f(w,v)$.
   $f(v,w) > 0$ is called a flow *from* v to w.
2. *capacity constraint*. $f(v,w) \leq cap(v,w)$. If $[v,w]$ is an edge such that $f(v,w) = cap(v,w)$, the flow is said to *saturate* $[v,w]$.
3. *flow conservation*. For every vertex v other than s and t

$$\sum_{w} f(v,w) = 0$$

Network Flow: *maximum flow*

The *value* $|f|$ of a flow f is the net flow out of the source

$$\sum_{v} f(s,v)$$

Maximum Flow Problem (Ford and Fulkerson).

Network Flow: *cut*

A *cut*: partition $X, \overline{X}$ of the vertex set V into two parts X and $\overline{X} = V - X$ such that X contains s and $\overline{X}$ contains t.
The *capacity* of a cut $X, \overline{X}$ is

$$cap(X, \overline{X}) = \sum_{v \in X, w \in \overline{X}} cap(v, w)$$

Flow across a cut is

$$f(X, \overline{X}) = \sum_{v \in X, w \in \overline{X}} f(v, w)$$

Network Flow: *max-flow min-cut theorem*

For any flow f, the flow across any cut $X, \overline{X}$ is equal to the flow value.
Capacity constraint $\rightarrow$ flow across cut cannot exceed capacity of the cut.
Maximum flow is not greater than the capacity of a *minimum cut*.

max-flow min-cut theorem: *maximum flow* = *minimum cut*

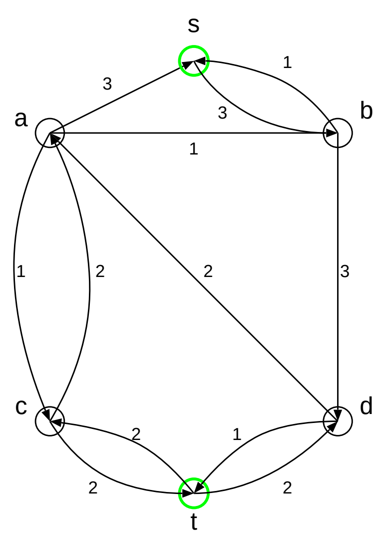
Network Flow: *residual graph*

Residual capacity for flow f

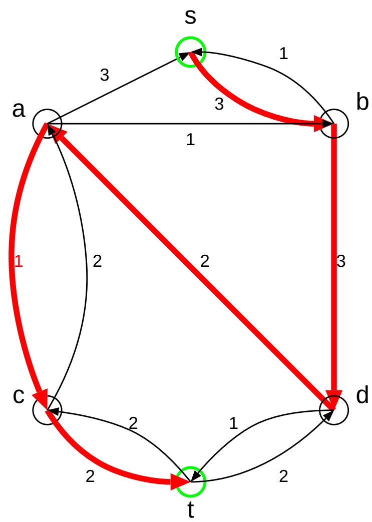
$$res(v,w) = cap(v,w) - f(v,w)$$

Up to $res(v,w)$ additional flow can be pushed along $[v,w]$.
Residual graph R is graph with edges $res(v,w)$.
Augmenting path from s to t.
Residual capacity is *minimum* $res(v,w)$.

Network Flow: *residual graph*



Network Flow: *augmenting path*



Ford Fulkerson

- Augmenting step
  1. Find an augmenting path p for the current flow.
  2. Increase the value of the flow
     by pushing $res(p)$ units of flow along p.
- Pathfinding step
  1. Find a path p_i from s to t in G^*.
  2. Let Δ_i be the minimum of $f^*(v,w)$ for $[v,w]$ an edge of p_i.
     For every edge $[v,w]$ on p_i, decrease $f^*(v,w)$ by Δ_i and delete $[v,w]$ from G^* if its flow is now zero.
  3. Increment i by one.

Path Finding: which path ?

- Edmonds and Karp:
  augmentation along path with maximum residual capacity.
- Dinic:
  augmentation along shortest augmenting path.
  Length: number of edges a path contains.

Dinic's algorithm: find *blocking flows* to saturate edges

1. Begin with zero flow.
2. Find a *blocking flow* f' on the *level graph* for the current flow f.
   Blocking flow: every path from the source s to the sink t contains a saturated edge.
3. Replace f by the flow $f+f'$ defined by:

$$(f+f')(v,w) = f(v,w) + f'(v,w).$$

4. Repeat until the sink t is not in the level graph for the current flow.

Level Graph

- R: the residual graph for a flow f.
- *level* of v = the length of the shortest path from s to any vertex v in R.
- *Level graph* L for f = the subgraph of R containing
  - only the vertices reachable from s
  - only the edges $[v,w]$ such that

$$level(w) = level(v) + 1.$$

  L contains every *shortest* augmenting path and can be constructed in $O(m)$ time by *breadth-first search*.

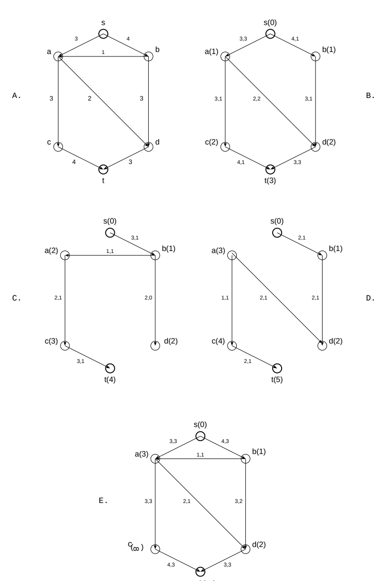
Finding a Blocking Flow (DFS)

- *Initialize*: Let $p = [s]$ and $v = s$. Go to *Advance*.
- *Advance*: If there is no edge out of v, go to *Retreat*. Otherwise, let $[v,w]$ be an edge out of v. Replace p by $p\&[w]$ and v by w. If $w \neq t$ repeat *Advance*; if $w = t$ go to *Augment*.
- *Augment*: Let Δ be the minimum of $(cap(v,w) - f(v,w))$ for $[v,w]$ an edge of p. Add Δ to the flow of every edge on p, delete from G all newly saturated edges, and go to *Initialize*.
- *Retreat*: If $v = s$ halt. Otherwise, let $[u,v]$ be the last edge on p. Delete v from p and $[u,v]$ from G, replace v by u, and go to *Advance*.

Dinic Performance
(m is number of nodes, n is number of edges)

- Finds a blocking flow in $O(nm)$ time, and a maximum flow in $O(n^2 m)$ time.
- On a unit network, Dinic's algorithm finds a blocking flow in $O(m)$ time, and a maximum flow in $O(n^{1/2} m)$ time. Unit network: edge capacities integer, each vertex v other than the source and the sink has either a single entering edge of capacity one, or a single outgoing edge of capacity one.
- On a network whose edge capacities are all one, Dinic's algorithm finds a maximum flow in $O(\min\{n^{2/3} m, m^{3/2}\})$ time.

Example



Symbolic Manipulation (Computer Algebra)

Simplification of expressions, re-writing of equations, symbolic solving, . . .

- Mathematica
- REDUCE
- AXIOM
- MACSYMA
- MuPAD (`http://www.mupad.de/`)

(muPAD) examples

```
>> 100!;

93326215443944152681699238856266700490715968264381621468592963895217599993\
22991560894146397615651828625369792082722375825118521091686400000000000000\
0000000000


>> (x+1)^4;

                                 4
                          (x + 1)

>> expand(%);

                          2      3    4
                 4 x + 6 x  + 4 x  + x  + 1

>>  x^2+2*x+1;

                               2
                        2 x + x  + 1
```

(muPAD) examples

```
>> factor(%);

                              [1, x + 1, 2]

>> diff(x^2+2*x+1,x);

                                 2 x + 2


>> int(x^2+2*x+1,x);

                                        3
                                   2   x
                              x + x  + --
                                       3

>> 2+3+4+x+4;

                                 x + 13

>> 2+ 3+x+y+x*y+x^2+y^3+4;

                                       2    3
                       x + y + x y + x  + y  + 9
```


(muPAD) examples

```
>> solve({x+a*y-2,x-b*y+4},{x,y});

                    { {     2 b - 4 a        6   } }
                    { { x = ---------, y = ----- } }
                    { {       a + b        a + b } }

>> subs(%,a=3,b=4);

                          {{x = -4/7, y = 6/7}}

>> generate::C(x^2+4-sin(y));

                    "    t4 = -sin(y) + x*x + 4.0 ;"

>> generate::TeX(x^2+4-sin(y));

                   "- \\sin\\left(y\\right) + x^2 + 4"
```

Canonical Form

```
>> (y+2)+3 + x;

                              x + y + 5

>> 5+y+x;

                              x + y + 5

>> 2+3+2*y+x-y;

                              x + y + 5

>> 2+x -y -x;

                                2 - y
```


Canonical Form

(Davenport)

A representation of a mathematical object (e.g., polynomial) is *canonical* if two different representations always correspond to two different objects.

A correspondence f between a class O of objects and a class R of representations is a *representation* of O by R if each element of O corresponds to one or more elements of R (otherwise it is not represented) and each element of R corresponds to one and only one element of O (otherwise we do not know which element of O is represented).

The representation is canonical if f is *bijective*. With a canonical representation it is possible to check *equality* of objects by verifying that their representations are equal.

Normal Form

If O has the structure of a monoid, a weaker concept may be defined. A representation is called *normal* if zero has only one representation. Every canonical representation is normal, but the converse is false.

Having a unique representation for zero is important to be able to test for division by zero.

A normal representation over a group also gives us an algorithm to determine whether two elements a and b of O are equal. It is sufficient to check whether $a - b = 0$. In a canonical representation, it suffices to check whether a's and b's representations are identical.

Regular and Natural form

A representation should be *regular*
$A = x^2 + x, A + 1 = (x^3 - 1)/(x - 1), A - x = x^2, \ldots$ is *not* regular.

Representations must be *natural*. Some form of simplification should occur. For polynomials in one variable, every power of x should appear at most once, and powers should be sorted in ascending or descending order.

Polynomials

Representations of polynomials: *dense* and *sparse*.

- Dense: vector of coefficients
- Sparse: list of (coeff, degree) tuples

$$(x^{1000}+1)(x^{1000}-1) = x^{2000}-1$$

Polynomial *in* a particular variable: $\sin(x) + 3 * \sin^2(x) - 2$
is polynomial in $\sin(x)$

Increasing or decreasing powers.

Polynomials in multiple variables

canonical, natural representation

Different types of ordering:

- *lexicographic*: alphabetically ordered. Within one variable name, ordered by powers. If the powers of that variable are the same, look at the next (lexicographic) variable. $x^2+2xy+x+y^2+y+1$
- *total degree, then lexicographic*: lexicographic distinction between same total degree, ordered by total degree. $x^2+2xy+y^2+x+y+1$
- *total degree, then inverse lexicographic*: $y^2+2xy+x^2+y+x+1$

Types, Domains, Algebraic Structures

Used to define *generic* operations

Definitions (Birkhoff & McLane)

1. Semigroup $S, +$
   - closure
   - associativity
2. Monoid
   - Semigroup with unit 0
3. Group
   - Identity 1
   - Inverse

4. Commutative (Abelian)
   - Semigroup
   - Monoid
   - Group
5. Ring $R, +, *$
   - $R, +$ Abelian Group
   - $R, *$ Monoid with unit 1
   - $*$ is distributive on both sides over $+$
6. Commutative Ring
   - Ring and $R, *$ is commutative
7. Field $=$ Commutative Ring
   - each non-zero element has multiplicative inverse

Computer Algebra $\sim$ Compilers

1. lexical analysis
2. syntactic analysis (grammar parsing)
3. intermediate representation: Abstract Syntax Tree (AST) and Symbol Table (ST)
4. operations (symbolic manipulation) on AST+ST
5. compiler compilers
   - Gentle `http://www.first.gmd.de/gentle`
   - TRAP `http://www.first.gmd.de/smile/trap`
   - ANTLR `http://www.antlr.org`
   - PCCTS `http://www.ocnus.com/pccts.html`

- Catalog of Compiler Construction Tools
  `http://www.first.gmd.de/cogent/catalog`

Gentle example


```
--
-- models.g
--
--  basic example of constant folding
--
-- HV 28/10/1999
--

--
-- Types
--

-- Opaque types and actions (defined externally)
--
'type' IDENT

'action' id_to_string(IDENT -> STRING)
'action' Put(STRING)
'action' printString(STRING)
'action' printInteger(INT)

-- the Node "type": expression nodes of the AST
--
'type' Node
  model(Node, IDENT)
  sequence(Node, Node)
  plus(Node, Node)
  minus(Node, Node)
  times(Node, Node)
  divide(Node, Node)
  unaryplus(Node)
```

```
    unaryminus(Node)
    pow(Node,Node)
    variable(IDENT)
    integer(INT)
    nil

  -- a list of Nodes
  --
  'type' NodeList
    list(Node,NodeList)
    nil

  -- process an input file
  --

  'root' process

  'nonterm' process
     'rule' process:
       -- parse the concrete syntax into AST/ST
       parse(-> MDLS)
       print(MDLS)

       -- write back the AST/ST in concrete syntax
       writeModel(MDLS)

       -- constant fold the AST/ST
       constFold(MDLS -> CMDLS)

       printString("-------------------------\n")
```



```
       -- write back the AST/ST in concrete syntax
       writeModel(CMDLS)

  --
  -- parse the concrete syntax
  -- generate an Abstract Syntax Tree (AST) and Symbol Table (ST)
  --

  'nonterm' parse(-> Node)
     'rule' parse(-> MDLS): models(-> MDLS)

  'nonterm' models(-> Node)
     'rule' models(-> nil):
     'rule' models(-> P): model(-> P)
     'rule' models(-> sequence(PS, P)): models(-> PS) ";" model(-> P)

  'nonterm' model(-> Node)
     'rule' model(-> model(ES, Name)): "MODEL" Ident(-> Name)
                                       "{" expressions(-> ES) "}"

  'nonterm' expressions(-> Node)
     'rule' expressions(-> nil):
     'rule' expressions(-> AE): addexpr(-> AE)
     'rule' expressions(-> sequence(ES, E)):
                     expressions(-> ES) ";" addexpr(-> E)

  'nonterm' addexpr(-> Node)
     'rule' addexpr(-> ME): multexpr(-> ME)
     'rule' addexpr(-> plus(XE, YE)): addexpr(-> XE) "+" multexpr(-> YE)
     'rule' addexpr(-> minus(XE, YE)): addexpr(-> XE) "-" multexpr(-> YE)
```

```
'nonterm' multexpr(-> Node)
   'rule' multexpr(-> UE): powexpr(-> UE)
   'rule' multexpr(-> times(XE, YE)): multexpr(-> XE) "*" powexpr(-> YE)
   'rule' multexpr(-> divide(XE, YE)): multexpr(-> XE) "/" powexpr(-> YE)

'nonterm' powexpr(-> Node)
   'rule' powexpr(-> UE): unaryexpr(-> UE)
   'rule' powexpr(-> pow(X,Y)): unaryexpr(-> X) "^" powexpr(->Y)

'nonterm' unaryexpr(-> Node)
   'rule' unaryexpr(-> integer(N)): Number(-> N)
   'rule' unaryexpr(-> variable(ID)): Ident(-> ID)
   'rule' unaryexpr(-> unaryminus(X)): "-" unaryexpr(-> X)
   'rule' unaryexpr(-> unaryplus(X)): "+" unaryexpr(-> X)
   'rule' unaryexpr(-> X): "(" addexpr(-> X) ")"

--
-- constant fold the AST
--

'action' constFold(Node -> Node)
  'rule' constFold(variable(ID) -> variable(ID)):
  'rule' constFold(integer(N) -> integer(N)):
  'rule' constFold(pow(E1, E2) -> EC):
         constFold(E1 -> ER1)
         constFold(E2 -> ER2)
         (|
           where(ER2 -> integer(N))
           (|
             eq(N,0)
             where(integer(1) -> EC)
```



```
           ||
             eq(N,1)
             where(ER1 -> EC)
           |)
         ||
           where(pow(ER1, ER2) -> EC)
         |)
  'rule' constFold(plus(E1, E2) -> EC):
         constFold(E1 -> ER1)
         constFold(E2 -> ER2)
         (|
           where(ER1 -> integer(N))
           where(ER2 -> integer(M))
           where(integer(N+M) -> EC)
         ||
           where(plus(ER1, ER2) -> EC)
         |)
  'rule' constFold(minus(E1, E2) -> EC):
         constFold(E1 -> ER1)
         constFold(E2 -> ER2)
         (|
           where(ER1 -> integer(N))
           where(ER2 -> integer(M))
           where(integer(N-M) -> EC)
         ||
           where(minus(ER1, ER2) -> EC)
         |)
  'rule' constFold(times(E1, E2) -> EC):
         constFold(E1 -> ER1)
         constFold(E2 -> ER2)
         (|
```

```
            where(ER1 -> integer(N))
            where(ER2 -> integer(M))
            where(integer(N*M) -> EC)
          ||
            where(times(ER1, ER2) -> EC)
          |)
  'rule' constFold(divide(E1, E2) -> EC):
          constFold(E1 -> ER1)
          constFold(E2 -> ER2)
          (|
            where(ER1 -> integer(N))
            where(ER2 -> integer(M))
            where(integer(N/M) -> EC)
          ||
            where(divide(ER1, ER2) -> EC)
          |)
  'rule' constFold(unaryplus(E) -> EC): constFold(E -> EC)
  'rule' constFold(unaryminus(E) -> ECC):
          constFold(E -> EC)
          (|
            where(EC -> integer(N))
            where(integer(-N) -> ECC)
          ||
            where(unaryminus(EC) -> ECC)
          |)
  'rule' constFold(model(N, I) -> model(NC, I)): constFold(N -> NC)
  'rule' constFold(sequence(N1, N2) -> sequence(NC1, NC2)):
          constFold(N1 -> NC1)
          constFold(N2 -> NC2)
  'rule' constFold(nil -> nil):
```



```
--
-- write back AST/ST in concrete syntax
--

'action' writeModel(Node)
  'rule' writeModel(model(Body, Name)):
         printString("MODEL ")
         id_to_string(Name -> NameString)
         printString(NameString)
         printString("\n")
         printString("{\n")
         writeModel(Body)
         printString("\n}\n")
  'rule' writeModel(sequence(P1, P2)):
         writeModel(P1)
         printString(";\n")
         writeModel(P2)
  'rule' writeModel(plus(A1, A2)):
         printString("(")
         writeModel(A1)
         printString("+")
         writeModel(A2)
         printString(")")
  'rule' writeModel(minus(A1, A2)):
         printString("(")
         writeModel(A1)
         printString("-")
         writeModel(A2)
         printString(")")
  'rule' writeModel(times(A1, A2)):
         writeModel(A1)
         printString("*")
```

```
        writeModel(A2)
  'rule' writeModel(divide(A1, A2)):
        writeModel(A1)
        printString("/")
        writeModel(A2)
  'rule' writeModel(pow(E1,E2)):
        printString("(")
        writeModel(E1)
        printString(")")
        printString("^")
        printString("(")
        writeModel(E2)
        printString(")")
  'rule' writeModel(unaryplus(U)):
        writeModel(U)
  'rule' writeModel(unaryminus(U)):
        printString("-")
        writeModel(U)
  'rule' writeModel(variable(ID)):
        id_to_string(ID -> NameString)
        printString(NameString)
  'rule' writeModel(integer(N)):
        printInteger(N)
  'rule' writeModel(nil):

-- print a list of nodes
--
'action' printList(NodeList)
  'rule' printList(NL):
  (|
    eq(NL, nil)
```



```
  ||
    where(NL -> list(H,T))
    writeModel(H)
    printList(T)
  |)

----
---- tokens
----
'token' Number(-> INT)
'token' Ident(-> IDENT)
```

Example Input

```
MODEL prog1
{
 a+b+c+d;
 1+2+3+4*3;
 xˆ5+3*7-3;
 5+y+4ˆ3+4*3;
 xˆ0+ yˆ3-7-4;
 zˆ1-a*b*c
}
```


Example Output

```
model(
   sequence(
      sequence(
         sequence(
            sequence(
               sequence(
                  plus(
                     plus(
                        plus(
                           variable(
                              <<134659968>>
                           ),
                           variable(
                              <<134659984>>
                           )
                        ),
                        variable(
                           <<134660000>>
                        )
                     ),
                     variable(
                        <<134660016>>
                     )
                  ),
                  plus(
                     plus(
                        plus(
                           integer(
                              1
                           ),
                           integer(
```



```
                              2
                           )
                        ),
                        integer(
                           3
                        )
                     ),
                     times(
                        integer(
                           4
                        ),
                        integer(
                           3
                        )
                     )
                  )
               ),
               minus(
                  plus(
                     pow(
                        variable(
                           <<134660032>>
                        ),
                        integer(
                           5
                        )
                     ),
                     times(
                        integer(
                           3
                        ),
```

```
                    integer(
                       7
                    )
                 )
              ),
              integer(
                 3
              )
           )
        ),
        plus(
           plus(
              plus(
                 integer(
                    5
                 ),
                 variable(
                    <<134660048>>
                 )
              ),
              pow(
                 integer(
                    4
                 ),
                 integer(
                    3
                 )
              )
           ),
           times(
              integer(
```



```
                 4
              ),
              integer(
                 3
              )
           )
        )
     ),
     minus(
        minus(
           plus(
              pow(
                 variable(
                    <<134660032>>
                 ),
                 integer(
                    0
                 )
              ),
              pow(
                 variable(
                    <<134660048>>
                 ),
                 integer(
                    3
                 )
              )
           ),
           integer(
              7
           )
```

```
            ),
            integer(
               4
            )
         )
      ),
      minus(
         pow(
            variable(
               <<134660064>>
            ),
            integer(
               1
            )
         ),
         times(
            times(
               variable(
                  <<134659968>>
               ),
               variable(
                  <<134659984>>
               )
            ),
            variable(
               <<134660000>>
            )
         )
      )
   ),
   <<134659952>>
```



```
)
MODEL prog1
{
(((a+b)+c)+d);
(((1+2)+3)+4*3);
(((x)^(5)+3*7)-3);
(((5+y)+(4)^(3))+4*3);
((((x)^(0)+(y)^(3))-7)-4);
((z)^(1)-a*b*c)
}
-------------------------
MODEL prog1
{
(((a+b)+c)+d);
18;
(((x)^(5)+21)-3);
(((5+y)+(4)^(3))+12);
(((1+(y)^(3))-7)-4);
(z-a*b*c)
}
```