AMPL Implementation Techniques

David M. Gay

AMPL Optimization, Inc.

dmg@ampl.com

Outline

- AMPL design features
- Processing phases
- Representations: linear, nonlinear
- Expression evaluation: functions versus big switch
- Object data versus scratch array
- Change propagation
- Expression rewrites
- Sets and set members

AMPL design features

- Explicit indexing (*no hidden magic*)
- Declare before use (*one-pass reading*)
- Separate model, data, commands (*orthogonality*)
- Separate solvers (*open solver interface*)
- Update entities as needed (*lazy evaluation*)
- Builtin math. prog. stuff (*presolve, red. costs, ...*)
- Aim for large scale nonlinear (*sparsity, generality*)

Processing phases (1980's)

Originally (as given in *Management Science* paper):

- parse (*lex*, *yacc*)
- read data
- compile
- generate
- collect
- presolve
- output

Processing phases with commands

Commands may modify data (*let*, *call*, *read*, *read table*) and problem state (*drop*, *restore*, *fix*, *unfix*), report results (*display*, *print*, *printf*), interact with databases (*read table*, *write table*), access libraries of functions (*load*, *unload*, *reload*), execute external commands (*shell*), and make other changes (*cd*, *delete*, *purge*, *remove*).

Now parsing proceeds until a command is complete. Simple commands are processed immediately; compound commands (*if-then-else* or *for* or *repeat* loops) are treasured up until complete, and then are executed.

Representing linear expressions

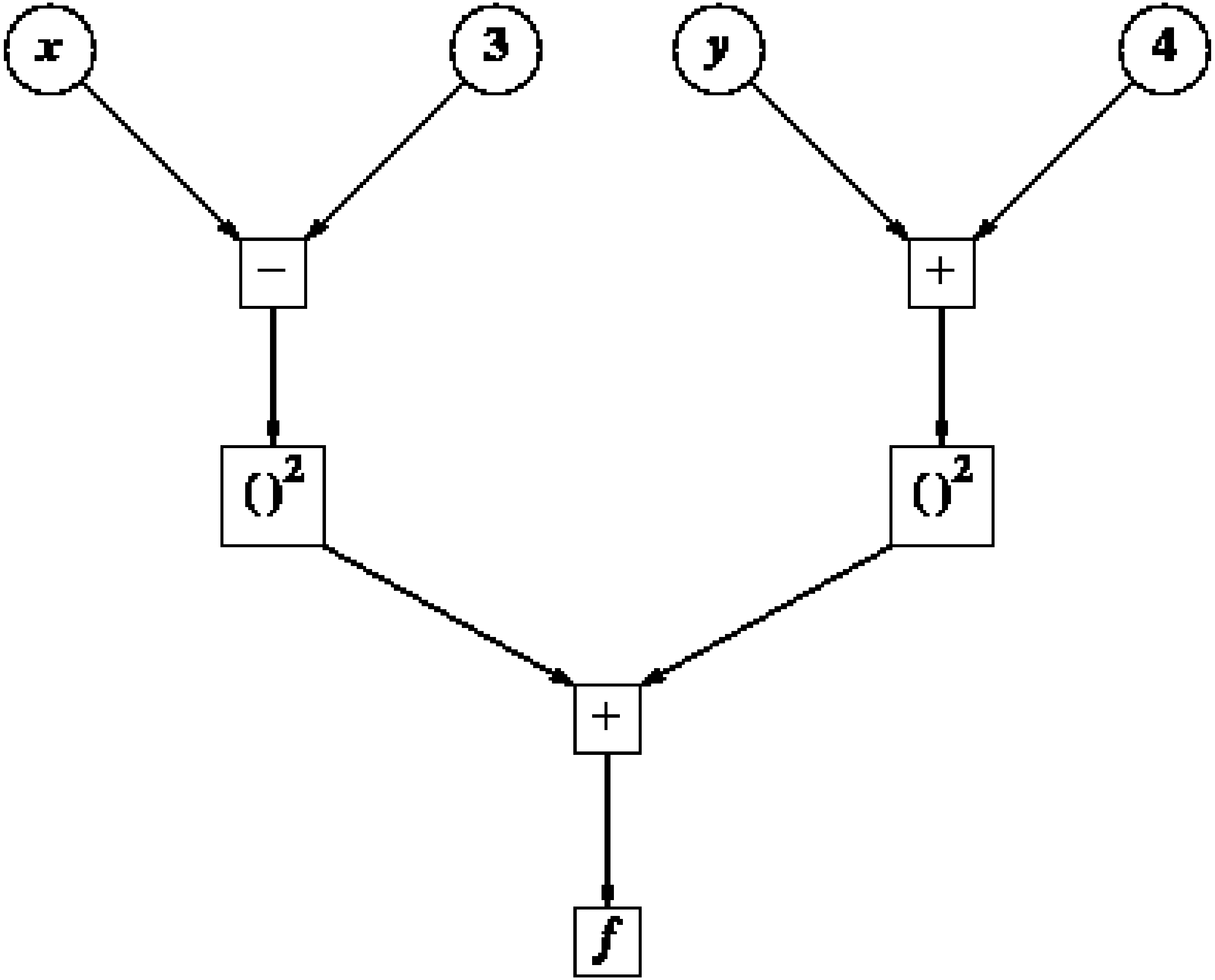
Currently:

```
struct varref {
        varref  *next;
        int     conno;
        real    coef; };
```

Plan for constraints and objectives:

```
struct varrefg {
        varrefg *next;
        int     yno, conno;
        real    coef; };
```

Expression graph example: $f = (x - 3)^2 + (y + 4)^2$



Evaluation via functions with object data

Example evaluation of `OPMULT`:

```
typedef real (*efunc)(struct expr*);
struct expr { efunc *op;
              expr *L, *R; real dL, dR; };

real f_OPMULT(expr *e)
{  expr *e1, *e2;
   e1 = e->L;
   e2 = e->R;
   return   (e->dR = (*e1->op)(e1))
          * (e->dL = (*e2->op)(e2)); }
```

Evaluation via functions with scratch array

Example evaluation of OPMULT:

```
typedef real (*efunc)(struct expr*, real*);
struct expr { efunc *op;
              expr *L, *R; size_t dL, dR; };

real f_OPMULT(expr *e, real *T)
{  expr *e1, *e2;
   e1 = e->L;
   e2 = e->R;
   return   (T[e->dR] = (*e1->op)(e1, T))
          * (T[e->dL] = (*e2->op)(e2, T)); }
```

Evaluation via big switch

```
real Eval(size_t *op, real *T)
{
  for(;;) switch(*op) {
     case OPMULT:
            T[op[1]] = T[op[2]] * T[op[3]];
            op += 4;
            break;
     case OPRET: return T[op[1]];
     ...
     }
}
```

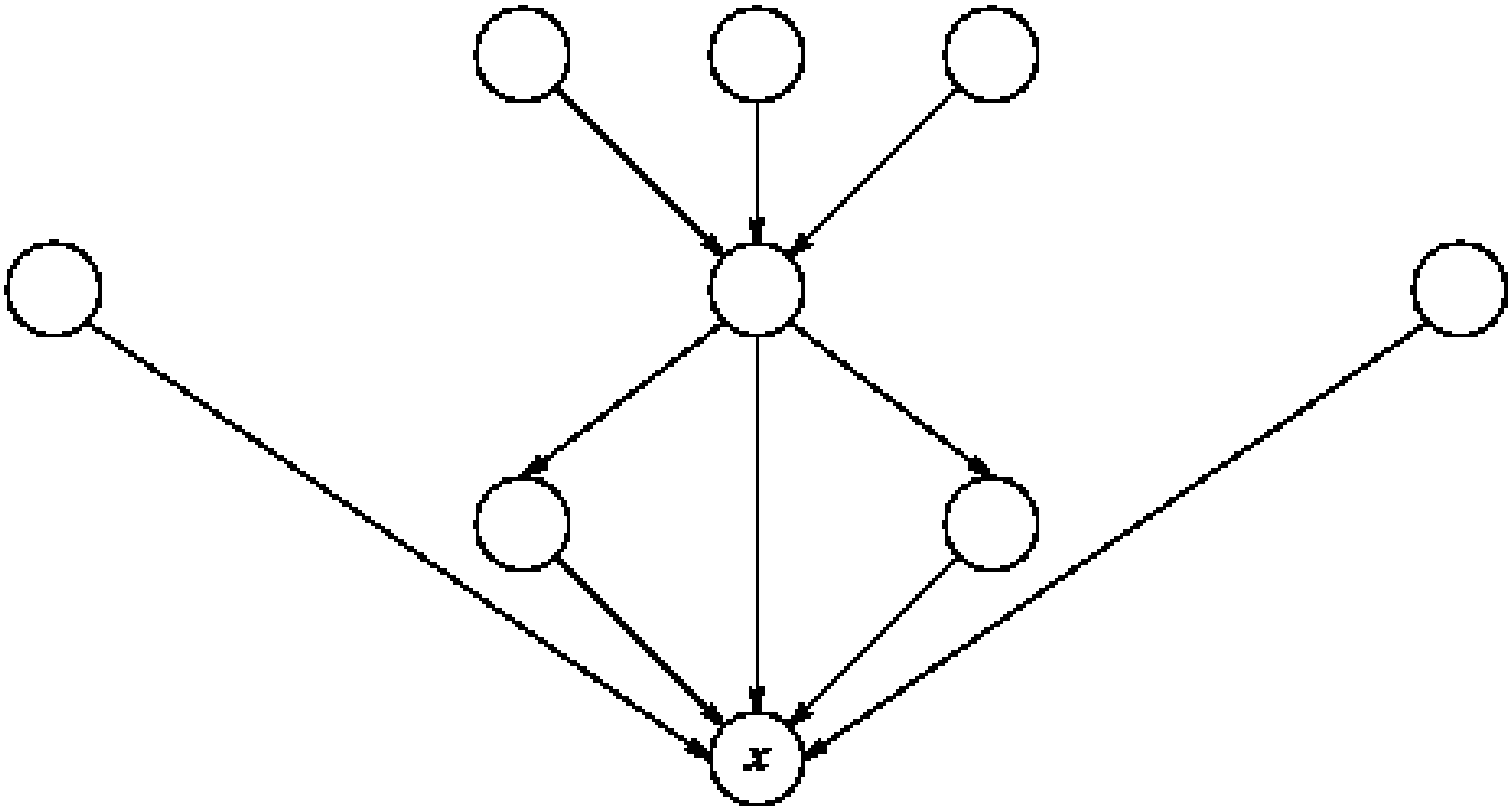
Current derivative propagation

```
struct derp { derp *next; real *a, *b, *c; }

 void
derprop(derp *d) {
    if (d) {
            *d->b.rp = 1.;
            do *d->a += *d->b * *d->c;
               while(d = d->next);
            }
    }
```

Alternate derivative propagation

```
struct derp { size_t a, b, c; }
struct dblock { dblock *next;
   size_t tnext; derp *d, *d0; };

void derprop(dblock *B, real *T) {
    derp *d, *d0; dblock *B1;
    for(; B; B = B1) {
        for(d = b->d, d0 = b->d0; --d >= d0; )
                T[d->a] += T[d->b] * T[d->c];
        if (!(B1 = B->next))
                B1 = *(dblock*)&T[B->tnext]; } }
```

Change propagation: dependencies



Currently: to check if x is up to date, trace all dependencies. Plan: notify dependents of change.

Kinds of changes

- Value change.
- Append to index set.
- Add to and reorder index set.
- Independently: note recompilation needed.

E.g., in cut generation, we add to index sets.

Issue: given

set A; set B; node c{A, B};

if we append to both `A` and `B`, should we reorder the constraints `c`?

“Need to recompile” notification

If a change only affects the **`default`** value of a set or param, the set or param may be marked as needing recompilation, which only needs to happen if and when the **`default`** expression needs to be evaluated.

Commands can also receive “need to recompile” notifications. Only recompiling when necessary should speed some command sequences. Although

for{i in S} let p[i] := ...;

will always be slower than

let{i in S} p[i] := ...;

the difference in speed should decrease.

Expression rewrites

Currently we use *object data* and may rewrite expressions during compilation, recording the rewrites so they may be undone should we need to recompile.

During execution, common expressions may also rewrite themselves to avoid re-evaluations. These rewrites are undone when the containing context ends.

Disadvantages: recursive evaluations require dynamic expression copying, and using parallel threads would be hard.

Alternate approach to expression rewrites

Alternative (in progress): compiling an expression gives a reference-counted compiled expression that differs from the original if any rewrites occur. During execution, common expressions record their values in a temporary-values array, still avoiding re-evaluations.

This alternative simplifies recursive evaluations (e.g., computation of values of recursive parameters and sets) and facilitates using parallel threads.

Parallel threads will require use of pointers to thread-specific data.

Sets and set members

AMPL sets currently contain *Symbols* or tuples thereof; a *Symbol* is a string-valued entity that may point to an associated object (e.g., a variable), with a special REAL object for *Symbols* corresponding to numbers. For any possible string value, there is at most one *Symbol*.

Plan: sets contain *Atoms* or tuples thereof. Initially no change will be apparent, but we can easily allow functions, tuples, and other numeric types (e.g., rational) to be *Atoms*. Functions expressed in AMPL may be useful, e.g., as solver call-backs.