Progress Report on Functions in AMPL

David M. Gay
AMPL Optimization, Inc.
Albuquerque, New Mexico, U.S.A.
dmg@ampl.com
http://www.ampl.com
Small nonlinear example using a function

ampl:  var x; s.t. c: sin(x) = .5;
ampl:   solve;
MINOS 5.51: optimal solution found.
...
ampl:   print x; print asin(.5);
0.5235987755982988
0.5235987755982989
ampl:   display x - asin(.5);
x - asin(0.5) = -1.11022e-16
Load Library

When the builtin functions do not suffice, AMPL’s `load` command can introduce libraries of “imported” functions that have been compiled from suitable programming languages, such as C, C++, and Fortran. For example, the GNU Scientific Library, compiled for use with `load`, is available from https://ampl.com/resources/extended-function-library.

Why should functions be expressed in AMPL?

- Many MIP solvers permit “callback” functions to influence their solution algorithms. Introducing functions expressed in AMPL would permit making better interfaces to such solvers.

- AMPL functions might simplify some scripts.

- AMPL functions might help express some nonlinear problems.
Recursive function example: factorial

function fact(n)
{
    if n <= 2 then {
        if n <= 1 then
            let n := 1;
            return n;
        return n;
    }
    return n * fact(n-1);
}

Running the *fact* function

```AMPL
ampl: display{i in 0..5} fact(i);

fact(i) [*] :=
0  1
1  1
2  2
3  6
4  24
5  120
;
```
“show” command for a function

ampl: show fact;

function fact(param n) {
    if n <= 2 then {
        if n <= 1 then
            let n := 1;
            return n;
        return n*fact(n - 1);
    }
    return n*fact(n - 1);
}
Parsing challenge with domains

AMPL function declarations have long been allowed to specify a domain for each argument, e.g.,

\[
\text{function hypot(Reals,Reals);}
\]

It is nice to allow

\[
\text{function foo}(a,b) \{ \text{return } a + 2*b; \}
\]

which is easy to handle if the arguments are unbound symbols. The argument list of an AMPL function should be a new context, but for imported functions we must allow set expressions for domains. Various remedies are possible; for now, new keyword “\textbf{new}” indicates that a parameter name has a new meaning.
Keyword \texttt{new}

File \texttt{foo}:

\begin{verbatim}
param a; let a := 4.2;
function foo(a,b) { return a + b; }
\end{verbatim}

Invoking \texttt{``ampl foo''} gives output
\texttt{foo1}, line 2 (offset 32):
syntax error
context: function >>>>>> foo(a, <<< b) { return a + b; }

But \texttt{``function foo(new a, b)''}
or \texttt{``function foo(param a, b)''} is OK.
Local context

```plaintext
param a; let a := 4.2;
function foo(new a,b) { return a + b; }
show foo;
display a;

...gives output

  function foo(param a, param b) {
      return a + b;
  }

a = 4.2
```
Declarations within contexts now allowed

```AMPL
param a := 1.2;
{
  param a := 3.4;
  print 'Inner context 1: a =', a;
  {
    param a := 5.6;
    print 'Inner context 2: a =', a;
  }
  print 'Back to inner context 1: a =', a;
}
print 'Outermost context: a =', a;
```
Output from Declarations within contexts

Inner context 1: a = 3.4
Inner context 2: a = 5.6
Back to inner context 1: a = 3.4
Outermost context: a = 1.2
option hidewarn

0 (default) ⇒ no warning
1 ⇒ warning
2 ⇒ error

option hidewarn 1;
function foo()
{
  param a := 1.2; display a;
  {
    param a := 3.4; display a;
  }
  display a;
}
call foo;
Output from $hidewarn example

a is already defined

context: { param a >>> := <<< 3.4; display a; }

a = 1.2

a = 3.4

a = 1.2
Declarations within looping contexts

Within a looping context, newly declared entities retain their values until the context ends.

```AMPL
param a := 1.2;
{ for{i in 1..3} {
    param a default 3.7;
    let a := a + 1;
    display i, a;
}
    display a;
}
print 'Outermost context: a =', a;
```
Output from Declarations within looping contexts

\[ i = 1 \]
\[ a = 4.7 \]

\[ i = 2 \]
\[ a = 5.7 \]

\[ i = 3 \]
\[ a = 6.7 \]

\[ a = 1.2 \]

Outermost context: \[ a = 1.2 \]
For AMPL functions used in AMPL scripts, all commands are allowed in function bodies. AMPL functions visible to solvers, whether in callbacks or nonlinear expressions, will not be allowed to declare variables or execute commands other than `let`, `return`, flow-of-control, `call` and printing commands. In addition, functions in nonlinear expressions will not be allowed to have OUT args.
Declaring arguments to functions

Arguments can be declared immediately:

    function foo(set S, param p{S})

or first listed, then declared:

    function foo(p, S; set S; param p{S})

either of which could restrict the indexing set of p in the body of foo, in which

    indx(p) = S.

S would have to be a subset of the indexing set of the var or param passed as p.
Implicit indexing sets

Subscripted params and sets can have their indexing sets passed implicitly. In declaring subscripted params and sets as arguments, it suffices to specify the arity (dimension) of the indexing set, e.g.,

```
function foo(param p{1})
```

specifies that `p` is indexed over a set of single values.
Example of implicit indexing sets

The silly example:

function foo(param p{1})
{
    display p;
}
set S; param p{S};
data; param :S: p := a 1 b 2.5;
call foo(p);

gives output

p [*] :=
a 1
b 2.5
;
Recursive functions versus recursive sets and params

AMPL has long allowed recursive set and parameter definitions, such as

\[
\text{param factorial}\{i \text{ in } \text{integer}[0, \text{Infinity})\}
= \text{if } i < 2 \text{ then } i \text{ else } i \times \text{factorial}[i-1];
\]

Recursive set and parameter definitions effectively cache their computed values, so are automatically efficient. A purely recursive function may be much less efficient.
Ackermann’s function

Famous example from recursive-function theory:

```python
# recursive Ackermann’s function
function Acker(m,n)
{
    if m == 0 then return n + 1;
    if n == 0 then return Acker(m-1,1);
    return Acker(m-1, Acker(m,n-1));
}
```
Recursive param variant of Ackermann’s function

```
param nM default 20;
param nN default 200000;
set M = 0 .. nM;
set N = 0 .. nN;
param Ackermann{m in M, n in N} =
  if m == 0 then n + 1
  else if n == 0 then Ackermann[m-1,1]
  else Ackermann[m-1, Ackermann[m,n-1]];

#Ackermann[4,1] = 65533
#Ackermann[4,2] runs out of 32-bit memory
```
Returning sets

set A; set B; set C;
function foo(set S, set T) returns set
{ return S union T; }
data; set A := a b c; set B := x y;
display A, B;
display foo(A, B);

gives output

set A := a b c;
set B := x y;
set foo(A, B) := a b c x y;
Soon... Returning tuples as well

Sometimes it is useful to return sets...

```plaintext
set S; set T;
param p{S};
...
let T := argmaxset(p);
# T = \{t in S: p[t] == max\{i in S\} p[i]\}
```

or tuples of values and sets

```plaintext
let (t,T) := argmax(p);
# T = \{s in S: p[s] == max\{i in S\} p[i]\}
# and t = p[s] for s in T
```
Closures

For use in callbacks, our plan is to convey “closures” with functions in .nl files. This will permit the functions to access values from outer contexts. Whether changes to these values are communicated back to the AMPL session will be governed by a new option, just as option send_suffixes determines whether suffix values in .sol files are returned to the AMPL session.
AD for AMPL functions

AMPL itself (aside from imported functions) has been a *primitive recursive* language. For example, `.nl` files do not contain loops — all loops have been expanded by the AMPL processor before it writes the `.nl` file. This allows the AMPL/solver interface library (ASL) to set up structures needed for automatic differentiation in the course of reading the `.nl` file. Imported functions participate by providing first and possibly second partial derivatives for their numeric arguments.
AD for AMPL functions (cont’d)

AMPL functions appearing in objectives and constraints could be fully (and mutually) recursive, which will require the ASL to dynamically set up new AD structures for each new signature (combination of differentiable arguments) of a call and to allocate memory for the AD values for each call. More generally, if tuple returns are involved, arranging for efficient evaluations will be a challenge.
Function arguments in AMPL scripts

Imported functions (made available with `load` commands) can presently only be called with numeric or string arguments. Given the declarations

```ampl
set S; param p{S};
function foo;
```

imported function foo could only be provided all of `param p` by a call of the form

```ampl
call foo(card(S), {i in S} p[i]);
```

(next slide)
Function arguments in AMPL scripts (cont’d)

But an AMPL function foo, declared with

```
function foo(param p{dimen 1});
```

or

```
function foo(param p{dimen 1}) { ... }
```

or

```
set S; # ...
function foo(param p{S}) { ... }
```

could simply be called by

```
call foo(p);
```
Function arguments in AMPL scripts (cont’d)

Within foo, p’s declared and actual indexing sets could be accessed by the (provisionally named) new builtin functions \texttt{dind} and \texttt{indx}, as in

\begin{verbatim}
    dind(p)
\end{verbatim}

and

\begin{verbatim}
    indx(p)
\end{verbatim}

so

\begin{verbatim}
    sum\{i in indx(p)\} p[i]
\end{verbatim}

would be the sum of all components of p.
Status

The initial slides in this talk show output on my laptop. Tuple assignments and returning tuples and functions should (I hope!) work soon. The “new” keyword may disappear, in favor of “param” or “set”. We intend to make 64-bit “beta” binaries for Linux, MacOSX, and MS Windows available from the AMPL web site, https://ampl.com. The “beta” binaries will use the usual AMPL licensing mechanism and will work with current AMPL licenses. Initially AMPL functions will only work in AMPL scripts. Extensions to the solver-interface library (ASL) are further in the future.
Some references

The AMPL web site

https://ampl.com

has more on AMPL, including pointers to papers on AMPL and on the AMPL/solver interface library (ASL). When available, pointers to beta copies of AMPL with function extensions will appear on the AMPL web site.

For more on AD (automatic differentiation), see

http://www.autodiff.org
Load library example

Example: \( \text{gsl\_log1p}(x) \) computes \( \log(1 + x) \), avoiding the roundoff error that would occur in computing \( 1 + x \) for small \( |x| \).

ampl:  load amplgsl.dll;
ampl:  function gsl\_log1p;
ampl:  display log(1 + 5e-16), gsl\_log1p(5e-16);
\[ \log(1 + 5e-16) = 4.44089e-16 \]
\[ \text{gsl\_log1p}(5e-16) = 5e-16 \]
ampl:  print log(1 + 1.2e-17), gsl\_log1p(1.2e-17);
\[ 0 \quad 1.2e-17 \]
Out args

An imported function can have “out args”, arguments to which the function provides values. For example, if file foo contains

```AMPL
load swap.dll;
param a; param b;
data; param a := 1.2; param b := 3.4;
function swap(INOUT, INOUT);
display a, b;
display swap(a,b); ## or "call swap(a,b);"
display a, b;
```

then invoking “ampl foo” produces the output...
Output from “AMPL foo”

\[ a = 1.2 \]
\[ b = 3.4 \]

\[ \text{swap}(a, b) = 1 \]

\[ a = 3.4 \]
\[ b = 1.2 \]