E177, Reference/Assignment for Classes: Intro

February 12, 2008
http://jagger.me.berkeley.edu/~pack/e177

References

If \( M \) is a Matlab object, then

\[
\text{LHS = expr involving } M.\text{NameReference}
\]

first calls a method named \( \text{subsref} \) in the methods folder for \( \text{class}(M) \) to evaluate

\( M.\text{NameReference} \)

If no such method exists, then the built-in \( \text{subsref} \) is called, which

– works as expected if \( M \) is a struct
– generates an error if \( M \) is a user-defined class, a double, a cell, a char, etc...

Generally, for user-defined classes, a \( \text{subsref} \) method must be written.

In the case above, the execution is actually

\[
\begin{align*}
L.\text{type} &= \\ 
L.\text{subs} &= \text{'}\text{NameReference}\text{'}
\end{align*}
\]

Deeper Reference

If \( M \) is a Matlab object, then

\[
\begin{align*}
\text{LHS = expr involving } M.\text{Name1.Name2}
\end{align*}
\]

generates a call to \( \text{subsref} \) equivalent to

\[
\begin{align*}
\text{L(1).type} &= \text{'}\text{.}\text{'}, \\
\text{L(1).subs} &= \text{'}\text{Name1}\text{'}, \\
\text{L(2).type} &= \text{'}\text{.}\text{'}, \\
\text{L(2).subs} &= \text{'}\text{Name2}\text{'}, \\
\text{LHS = expr involving } \text{subsref}(M,L)
\end{align*}
\]

\textbf{Note:} The second argument is, in general, a non-scalar \text{struct} array. The length of the array indicates the depth of the reference.

Psuedo-code for 1st \text{subsref}

\[
\begin{align*}
\text{function } B = \text{subsref}(A,L) \\
\text{switch } L(1).\text{type} \\
\text{case } \text{'.'} \\
\quad B = \text{get}(A,L(1).\text{subs}); \ % \text{like get} \\
\text{otherwise} \\
\quad \% \text{some other cases we will learn about} \\
\text{end} \\
\text{end} \\
\% \text{Now do the remaining references recursively} \\
\text{if } \text{length}(L)>1 \\
\quad B = \text{subsref}(B,L(2:end));
\end{align*}
\]

Conventions for reference behavior

If \( M \) is a Matlab object, then a reference of the form

\[
\begin{align*}
M.\text{Name1}
\end{align*}
\]

should be equivalent to breaking it into 2 steps, first

\[
\begin{align*}
tmp = M.\text{Name1}
\end{align*}
\]

followed by a reference of the form

\[
\begin{align*}
tmp.\text{Name2}
\end{align*}
\]

More generally, a reference of the form

\[
\begin{align*}
M.\text{Name1.Name2.Name3.Name4.Name5}
\end{align*}
\]

should be equivalent to breaking it into 2 steps, first

\[
\begin{align*}
tmp = M.\text{Name1}
\end{align*}
\]

followed by a reference of the form

\[
\begin{align*}
tmp.\text{Name2.Name3.Name4.Name5}
\end{align*}
\]

\textbf{type:} Other references

If \( M \) is a Matlab object, then

\[
\begin{align*}
\text{LHS = expr involving } M.\text{IdxA,IdxB}.\text{Name2(IdxC,IdxD)}
\end{align*}
\]

The complex reference of \( M \) generates a call to \( \text{subsref} \) equivalent to

\[
\begin{align*}
\text{L(1).type} &= \text{'}\text{()}' \\
\text{L(1).subs} &= [[\text{IdxA} [\text{IdxB}]]] \ % \text{1x2 cell} \\
\text{L(2).type} &= \text{'}\text{.}\text{'}, \\
\text{L(2).subs} &= \text{'}\text{Name2}' \\
\text{L(3).type} &= \text{'}\text{()}' \\
\text{L(3).subs} &= [[\text{IdxC} [\text{IdxD}]]] \ % \text{1x2 cell} \\
\text{subsref}(M,L)
\end{align*}
\]
Psuedo-code for 1st subsref

```matlab
function B = subsref(A,L)
    switch L(1).type
    case '.
        B = get(A,L(1).subs);
    case '()'
        % code to handle ()
    case '{}'
        % code to handle {}
    end
    if length(L)>1
        B = subsref(B,L(2:end));
    end
end
```

1st Exception/rule

Consider a class @userclass. Suppose the methods folder is as shown.

Inside `method1.m`, the behavior for subsref-like operations is as follows:

Suppose `A` is an object of class `userclass`. Then any expression of the form

```
A.name, A(idx), etc
```

will treat `A` as a struct (calling the `@struct/subsref` method which we know about). Note that the existing `userclass/subsref` will not be called. The only manner in which to call the existing `userclass/subsref` is a direct call, for example

```
L.type = '{1}'; Lsubs = {{[1 3 5]} {[2 3 6]}};
refA = subsref(A,L)
```

Concatenation

If `A` and `B` are Matlab variables, then `[A B]`

- calls the method `horzcat(A,B)` in the methods folder for the higher precedence class of `A` and `B`
- If no such method exists, then the built-in `horzcat` is called.

In the 2nd case, if `A` and `B` are objects of the same user-defined class, then the result is of the same class, with the underlying struct having grown via the concatenation.

```matlab
A = e177poly([2 4 6]); size(A)
B = e177poly([8 10]); size(B)
C = [A B]; size(C), class(C)
struct(C), size(struct(C))
```

If `subsref` has not been defined for the class, then the built-in `subsref` does array referencing as expected

```matlab
isequal(C(1),A)
```

Psuedo-code for 2nd subsref

```matlab
function B = subsref(A,L)
    switch L(1).type
    case '.
        B = get(A,L(1).subs);
    case '()'
        B = builtin('subsref',A,L(1));
    otherwise
        % must still handle '{},' case
    end
    if length(L)>1
        B = subsref(B,L(2:end));
    end
end
```

Let `builtin` handle array reference

Concatenation/Reference

For some classes,

- the built-in array concatenation and
- the built-in array referencing

will not be the desired behavior.

In those cases, there needs to be tight coordination between the

- `horzcat`, `vertcat`, and `cat`
- `subsref` and `subsasgn`
- `get` and `set`

methods of the class. We will address this in next few lecture and homeworks.

At this point, note that most of the methods don’t properly account for arrays of polynomials…
### Assignments

If $M$ is a Matlab object, then
\[
M.NameReference = RHS
\]
calls a method named \texttt{subsasgn} in the methods folder for class (M). If no such method exists, then the built-in \texttt{subsasgn} is called, which
- works as expected if $M$ is a struct
- generates an error if $M$ is a user-defined class, a double, a cell, a char, etc...

Generally, for user-defined classes, a \texttt{subsasgn} method must be written.

In the case above, the call to \texttt{subsasgn} is equivalent to
\[
\begin{align*}
L.type &= '.,' & \text{ % scalar char} \\
L.subs &= 'NameReference' \\
M &= \texttt{subsasgn}(M,L,RHS)
\end{align*}
\]

### Deeper Assignment

A deeper assignment, for example
\[
M.Name1.Name2 = RHS
\]
generates a call to \texttt{subsasgn} equivalent to
\[
\begin{align*}
L(1).type &= '()' \\
L(1).subs &= \{IdxA\} \{IdxB\} & \text{ % 1x2 cell} \\
L(2).type &= '.' \\
L(2).subs &= 'Name2' \\
L(3).type &= '()' \\
L(3).subs &= \{IdxC\} \{IdxD\} & \text{ % 1x2 cell} \\
M &= \texttt{subsasgn}(M,L,RHS)
\end{align*}
\]

The second argument is, in general, a non-scalar \texttt{struct} array. The length of the array indicates the depth of the assignment.

### Conventions for assignment behavior

If $M$ is a Matlab variable, then an assignment of the form
\[
M.Name1 = RS
\]
is the same as
\[
\texttt{set}(M,'Name1',RS)
\]

More generally, an assignment of the form
\[
M.Name1.Name2.Name3.Name4.Name5 = RS
\]
should be equivalent to breaking it into 2 steps, first
\[
\text{tmp} = M.Name1 & \text{ % subsref}
\]
followed by two assignments
\[
\text{tmp.Name2.Name3.Name4.Name5} = RS; & \text{ % subsasgn}
\]
\[
M.Name1 = \text{tmp}; & \text{ % asgn/set}
\]

### Psuedo-code for 1st \texttt{subsasgn}

```plaintext
function B = subsasgn(A,L,RHS) 
switch L(1).type 
    case '.'
        if length(L)==1
            B = set(A,L(1).subs,RHS);
        else
            tmp = subsref(A,L(1));
            tmp = subsasgn(tmp,L(2:end),RHS);
            B = set(A,L(1).subs,tmp);
        end
    otherwise
        % some other cases still to do
end
```

### type: Other assignments

If $M$ is a Matlab object, then
\[
M(IdxA,IdxB).Name2(IdxC,IdxD) = RHS
\]
generates a call to \texttt{subsasgn} equivalent to
\[
\begin{align*}
L(1).type &= '()' \\
L(1).subs &= \{IdxA\} \{IdxB\} & \text{ % 1x2 cell} \\
L(2).type &= '.' \\
L(2).subs &= 'Name2' \\
L(3).type &= '()' \\
L(3).subs &= \{IdxC\} \{IdxD\} & \text{ % 1x2 cell} \\
M &= \texttt{subsasgn}(M,L,RHS)
\end{align*}
\]

### Additional Exceptions

If $A$ is a cell, then
\[
A(Idx) = RHS
\]
always calls the cell version of \texttt{subsasgn} regardless of the class of $RHS$.

However, if $RHS$ is superior to cell, then a direct call to \texttt{subsasgn}, of the form
\[
L.Type = '()'; L.subs = \{Idx\}; \\
A = subsasgn(A,L,RHS)
\]
will dispatch following the precedence rules directly.

Similarly, if $B$ is a struct, then
\[
B.fieldname = RHS
\]
always calls the built-in version of \texttt{subsref} (ie., for struct array) regardless of the class of $RHS$.
Psuedo-code for 1st `subsasgn`

function B = subsref(A,L)
switch L(1).type
case '.'
  if as we had two slides ago
  % code to handle ()
case '{}'
  % code to handle {}
end

Concatenation (recall)

If A and B are Matlab variables, then [A B]
- calls the method `horzcat(A,B)` in the methods folder for the
  higher precedence class of A and B
- If no such method exists, then the built-in `horzcat` is called.

In the 2nd case, if A and B are objects of a user-defined
class, then the result is of the same class, with the
underlying `struct` having grown via the concatenation.

A = el77poly([2 4 6]); size(A)
B = el77poly([8 10]); size(B)
C = [A B]; size(C), class(C)
struct(C), size(struct(C))

If `subsasgn` has not been defined for the class, then the
built-in version does array assignment as expected

C(2) = A*B; isequal(C(2),A*B)

Psuedo-code for 2nd `subsasgn`

function B = subsasgn(A,L,RHS)
switch L(1).type
case '.'
  % code as we had before
case '{}'
  if length(L)==1
    B = builtin('subsasgn',A,L(1),RHS);
  else
    tmp = builtin('subsref',A,L(1));
    tmp = subsasgn(tmp,L(2:end),RHS);
    B = builtin('subsasgn',A,L(1),tmp);
  end
otherwise
  % must still handle '{}' case
end