Additional ASC Programming Comments

- NOTE: These are additional notes to be added to "ASC Programming" slides by Michael Scherger.
- Comparison of *logical parallel* and *index parallel*
  - A *index parallel* variable selects a single scalar value from a parallel variable.
  - A *logical parallel* variable L is normally used to store the result of a search such as

    L[$] = A[$] .eq. B[$]
  - ASC implementation simplifies usage by not formally distinguishing between the two.
    - The correct type should be selected to improve readability.
- Mixed mode operations are supported and their result has the "natural" mode. For example, if

```
int scalar a, b, c;
int parallel p[$], q[$], r[$], t[$,4];
index parallel x[$], y[$];
```

then

```
c = a + b                scalar integer
q[$] = a + p[$]          parallel integer variable
a + p[x]                 integer value
r[$] = t[x,2]+3*p[$]     parallel integer variable
x[$] = p[$] .eq. r[$]    index parallel variable
```


- Array Dimensions
  - Int parallel can have up to 3 dimensions
    - First dimension is "$", the parallel dimension
  - The array numbering is zero-based, so the declaration

    int parallel A[$,2]

    creates the following 1dimensional variables:

    A[$,0], A[$,1], A[$,2]
- Dynamic Storage allocation
  - *allocate* is used to activate a cell to store a new association record
    - Creates a parallel index that points to the new cell
  - *release* is used to de-allocate storage of specified records in association
    - Can release multiple records simultaneously.
  - Example:

```
char parallel   node[$], parent[$];
logical parallel   tree[$];
index parallel   x[$];
associate node[$], level[$], parent[$] with tree[$];
.....
allocate x in tree[$]
   node[x] = 'B'
endallocate x;
release parent[$] .eq. 'A' from tree[$].
```


- Parallel IF-THEN-ELSE Example and Mask Trace

```
if A[$] .eq. 2
        then  A[$] = 5;
        else   A[$] = 0;
endif;
```

A[$] BEFORE	MASK BEFORE	A[$] AFTER	THEN MASK	ELSE MASK
2	1	5	1	0
5	1	0	0	1
3	0	3	0	0
2	1	5	1	0
1	1	0	0	1

- *any – elsenany* statement
  - All active cells execute statements inside the *any-block* if there is one responder.
  - If there are no responders, then all active cells execute the statements inside the *elsenany* block
  - *any* can be used alone (without the *elsenany)*
  - Example

```
any A[$] .eq. 10
    B[$] =11;
elsenany
    B[$] = 100;
endany;
```

- *if then – elsenany statement*

- *for* construct
  - Often used when a process must be repeated for each cell that satisfies a certain condition.
  - The index variable is available throughout the body of the *for* statement
  - The index value of *for* is only evalulated initially
  - Example:

```
sum = 0;
for x in A[$] .eq. 2
    sum = sum + B[$];
endfor x;
```

  - Trace for example:

A[$]	B[$]	x	loop	sum
1	1	0		0
2	2	1	1st	2
2	3	1	2nd	5
1	4	0		
2	5	1	3rd	10

- *Loop-Until* Construct for sequential repetitions
  - Used for sequential type repetitions
  - See earlier slide and primer for details

- *while* contruct
  - Unlike the *for* statement, this construct re-evaluates the logical conditional statement prior to each execution of the body of the *while*.
  - The bit array resulting from the evaluation of the conditional statement is assigned to the index parallel variable on each pass.
  - The index parallel array is available for use within the body each loop.
  - The body of the *while* construct will continue to be executed until there are no responders (i.e., all zeros) in the index parallel variable.
  - Study example and trace in ASC Primer carefully to make sure you understand *while*.
- *get* statement
  - Used to retrieve a value from a specific field in a parallel variable satisfying a specific conditional statement.
  - Example:

```
get x in tail[$] .eq. 1
    val[x] = 0;
endget x;
```

  - Read trace of this example in on page 24 of ASC Primer to make sure its action is clear.

- *next* statement
  - Similar to *get* except *next* updates the set of responders each time it is called.
  - Unlike *get,* two successive calls to *next* is expected to select two distinct cells (and two distinct association records).
  - Can be used in loops to sequentially process each responder.
  - See page 22-23 of ASC Primer for more details.
- The *maxval* and *minval* functions
  - *maxval* returns the maximum value of the specified items among the active responders.
  - Similarly, *minval* returns the minimum value.
  - Example:

```
if (tail[$] .neq. 1) then
    k = maxval( weight[$]);
endif;
```

  - See trace of example on pg 27 of Primer.
- The *maxdex* and *mindex* functions
  - They return the index of an (association) entry where a maximum or minimum occurs.
  - If maximum/minimum value occurs at more than one location, an arbitrary selection is made as to which index is returned.

- *setscope/endsetscope*
  - *setscope* jumps out of current mask setting to another mask setting.
  - One use is to reactivate currently inactive processors.
  - Also allow immediate return to a previously calculated mask, such as an association.
  - is an unstructured command such as *go-to* and jumps from current environment to a new environment.
    - Use sparingly
  - *endsetscope* resets mask to preceding setting.
- Restricted subroutine capability is currently available
  - See *call* and *include* on pg 25-6 of Primer.
- Use of personal pronouns and articles in ASC make code easier to read and shorter.
  - See page 29 of ASC Primer.
- The ASC Monitor is important for evaluation and comparison of various ASC algorithms and software. (See Pg 30-31 of ASC Primer)

- Scalar variable input
  - Static input can be handled in the code.
  - Also, *define* or *deflog* statements can be used to handle static input.
  - Dynamic input is currently not supported directly, but can be accomplished as follows:
    - Reserve a parallel variable *dummy* (of desired type) for input.
    - Reserve a parallel index variable *used*.
    - A value to be stored in scalar variables is first read into *dummy* using a parallel-read and then transferred using *get* or *next* to the appropriate scalar variable.
    - Example:

```
read dummy[$] in used[x];
get x in used[$]
    scalar-variable = dummy[x];
endget x;
```

    - NOTE: Don't need to use *associate* statement to associate dummy with *used*. Omission causes no problems as no check is currently made.