Range Definitions

integer range [1..5] one_five ;
integer range [1..5] one_five;

newType

range 2
lo = 1
hi = 5
bo = 4
base =

ConstExp 5
ConstExp 1
type

int 2

C1
... int 1
... int 5

DoMakeRange
Range Definitions

integer range [1..5] one_five;

newType

range 2
lo=1
hi = 5
bo = 4
base =

int 2

type
Array Definitions

integer array [one_five] [one_three] *a, b;

Assumes we have previously defined
integer range [1..5] one_five;
integer range [1..3] one_three;

DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

in DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

currentType

finalType

in DoMakeArray
integer array [one_five] [one_three] *a, b;

At end of loop on first pass

currentType

finalType

array 6
subs=
comp=
range 1 3...

int 2

in DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

currentType

finalType

array 30
subs=
comp=

range 1
5...

array 6
subs=
comp=

range 1
3...

int 2

mark

type

type

in DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

currentType

finalType

End of 2nd pass

array 30
subs=
comp=
range 1
5...

array 6
subs=
comp=
range 1
3...

int 2

in DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

currentType

finalType

array 30
subs=
comp=
range 1 5...

array 6
subs=
comp=
range 1 3...

int 2

in DoMakeArray
Array Definitions

integer array [one_five] [one_three] *a, b;

DoDeclVars exactly as before
Array Definitions

Notes:
Type pointers represent linked lists (if we have records then trees actually)

We may need to “walk” the list to process an array type.

The “head” pointer represents the type as a whole.
Subscript reduction

\[ x := a[3] \cdot [m] ; \]
Subscript reduction

\[ x := a[3] \cdot [m] ; \]

LD R0, #3
TRNG R0, +..(R14)
IS R0, #1
IM R0, #6
LDA R1, +4(R15)
IA R1, R0
Subscript reduction

\( x := a[3] \cdot [m] ; \)

\[
\begin{align*}
LD & \quad R0, \ #3 \\
TRNG & \quad R0, \ +(R14) \\
IC & \quad R0, \ #1 \\
IM & \quad R0, \ #6 \\
LDA & \quad R1, \ +4(R15) \\
IA & \quad R1, \ R0
\end{align*}
\]

This represents \( a[3] \)

\[
\begin{align*}
\text{array } 30 & \quad \text{range } 1, 5... \\
\text{subs=} & \quad \text{comp=} \\
\text{array } 6 & \quad \text{range } 1, 3... \\
\text{subs=} & \quad \text{comp=} \\
\text{int } 2 & \quad \text{in Subscript}
\end{align*}
\]
Subscript reduction

\[ x := a[3][m] \cdot; \]

LD R0, #3
TRNG R0, +..(R14)
IS R0, #1
IM R0, #6
LDA R1, +4(R15)
IA R1, R0

expr "m" (1,6,false)
expr var (0,1,true)
expr "x"

array 30
subs=
comp=
range 1 5...

array 6
subs=
comp=
range 1 3...

int 2

Subscript(second call)
Subscript reduction

\[ x := a[3][m] \cdot; \]

- LD R0, #3
- TRNG R0, +..(R14)
- IS R0, #1
- IM R0, #6
- LDA R1, +4(R15)
- IA R1, R0
- LD R0, +6(R15)
- TRNG R0, +..R14)
- IS R0 #1
- IM R0, #2
- IA R1, R0

**expr “m”**
(1,6,false)

**expr var**
(0,1,true)

**expr “x”**

```
array 30
subs=
comp=
range 1
5...

array 6
subs=
comp=
range 1
3...
```

int 2

in Subscript(second call)
**Subscript reduction**

\[
x := a[3][m] ;
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R0</td>
<td>#3</td>
</tr>
<tr>
<td>TRNG</td>
<td>R0</td>
<td>+(R14)</td>
</tr>
<tr>
<td>IS</td>
<td>R0</td>
<td>#1</td>
</tr>
<tr>
<td>IM</td>
<td>R0</td>
<td>#6</td>
</tr>
<tr>
<td>LD</td>
<td>R1</td>
<td>+4(R15)</td>
</tr>
<tr>
<td>IA</td>
<td>R1</td>
<td>R0</td>
</tr>
<tr>
<td>LD</td>
<td>R0</td>
<td>+6(R15)</td>
</tr>
<tr>
<td>TRNG</td>
<td>R0</td>
<td>+..R14</td>
</tr>
<tr>
<td>IS</td>
<td>R0</td>
<td>#1</td>
</tr>
<tr>
<td>IM</td>
<td>R0</td>
<td>#2</td>
</tr>
<tr>
<td>IA</td>
<td>R1</td>
<td>R0</td>
</tr>
</tbody>
</table>

This represents

\[
\text{array 30}
\]

\[
\text{subs} = \text{range 1 3...}
\]

\[
\text{comp} = \text{range 5...}
\]

\[
\text{array 6}
\]

\[
\text{subs} = \text{range 1 3...}
\]

\[
\text{comp} = \text{range 5...}
\]

\[
\text{int 2}
\]

**Assign:** exactly as before
Subscript reduction

Notes:
ReduceSubscript handles only one subscript. It is called multiple times if there are many.

It needs to check its two entries for legality--top = integer type, next has array type.
Range Checking 

Bounds offset for this array would be +0(R14)

newType

Assuming no constants had yet been allocated in C1 block
Bounds offset for this array would be $+4(R14)$.

In actuality, you enter two constants into the constant table, lowbound and highbound.

```cpp
int 2
range 2
lo = 1
hi = 5
bo = 4
base =

newType

ConstExp 5
ConstExp 1
type

C1
HALT
INT 1
INT 3
INT 1
INT 5

integer range [1..5] one_five ;
```