The SA-C Language – Version 1.0

J. P. Hammes and A. P. W. Böhm
Colorado State University

June 21, 2001

SA-C is a single-assignment, expression-oriented language designed for applications that do computationally intensive work with large arrays, targeting reconfigurable computing modules. The language mixes ideas from C, Sisal and Fortran 90.

The appendix contains SA-C's grammar, as BNF rules that are output by the parser generator.

1 Tokens

Identifier names in SA-C are case sensitive. Identifiers must start with an alphabetic character, which can be followed by any number of alphabetic, numeric, and '_' characters. Figure 1 shows SA-C's reserved words. Also, the words **bits**, **int** and **uint** followed by one or more digits, and **fix** and **ufix** followed by one or more digits, a period, and one or more digits, are reserved.

Constants are denoted as in C, with the extension of boolean constants **true** and **false**, and bits constants **0bBBB** and **0xXXX** where BBB stands for any sequence of binary digits and XXX stands for any sequence of hexadecimal digits. Some example constants are:

```
true
5
5.5
0x8A
0b0100
```

There are two kinds of comments. The first is C's conventional `/*–*/` style, which do not nest. The second is signaled by `//` which extends to the end of the current line. Otherwise, SA-C is not line oriented; line breaks represent white space.

2 Type specifications

SA-C has scalar base types, complex types, and arrays composed of scalars or complex numbers.

accum	asin	exp	lookup	sinh
acos	asinh	expm1	matrix	sqrt
acosh	assert	export	max	step
and	at	extents	mean	stripmine
array	atan	fabs	median	st_dev
array_accum	atanh	false	min	sum
array_and	atan2	final	mode	switch
array_concat	bool	float	modf	tan
array_conperim	cbrt	floor	next	tile
array_histogram	ceil	fmod	no_dfg	true
array_max	complex	for	no_fuse	vals_at_first_max
array_max_indices	concat	frexp	no_inline	vals_at_first_min
array_mean	copysign	if	no_unroll	vals_at_last_max
array_median	cos	imag	or	vals_at_last_min
array_min	cosh	in	pow	vals_at_maxs
array_min_indices	cross	histogram	print	vals_at_mins
array_mode	cube	hypot	product	vector
array_or	dot	ldexp	real	while
array_product	double	log	return	window
array_st_dev	elif	log10	rint	
array_sum	else	log1p	sin	

Figure 1: SA-C's reserved words.

2.1 Scalars

There are eight base types in SA-C: booleans, bits, signed integers, unsigned integers, signed fixed point numbers, unsigned fixed points, floats, and doubles. SA-C targets reconfigurable computing systems, hence some parts of a SA-C program will run on the host, and some parts will run on the FPGA array. The state of the art in FPGAs is such that code involving floats and doubles will currently run on the host. The SA-C compiler will issue a warning if floats or doubles are used in parts of the program running on the FPGA array.

The integer, bits, and fixed types require size or precision specifications. For ints the size specifies the total number of bits used in their representation. For fixed point numbers the first size specifier gives the total number of bits, and the second size specifier gives the number of bits used for the fraction part. An int of size one is disallowed, no size can be larger than 32, and no fraction can be larger than the total size. Some example base types are:

```
bool
bits5
uint6
fix16.4
ufix8.8
float
```

2.2 Complex Numbers

The complex types in SA-C have signed numeric subtypes int, fix, float, or double. Example complex types are:

```
complex int8
```

```
complex fix8.8
complex double
```

Complex numbers are written as "(" **real part, imaginary part** ")" such as **(1.0, 0.0)**. If x is a complex number, then **real(x)** is its real part, and **imag(x)** is its imaginary part.

2.3 Arrays

SA-C has multidimensional arrays with components limited to scalars and complex numbers. The language emphasizes array operations and includes mechanisms for taking array *slices* (or *sections*) and creating arrays through special loop constructs. All arrays are rectangular, and the array elements are stored at equal distances in memory, so that array element addresses can be calculated using simple linear expressions. The elements are not necessarily stored contiguously, as an array can be a (e.g. column) slice of another array.

The following array-related terms are defined, drawn from Fortran 90 terminology. The *rank* of an array is the number of dimensions it has. Thus, what we commonly call a 2D-array, or matrix, is an array of rank two. An array has an *extent* for each of its dimensions. Multiplying all the extents of an array together yields the number of elements, or the *size* of the array. An array's rank and extents together comprise its *shape*. One or more of an array's extents can be zero, which would imply that the array's size is zero.

An array type in SA-C describes two characteristics of an array: its component type and its rank. The type does *not* define an array's extents. Instead, during program execution each array carries its extents with it, and these extents can be accessed explicitly through the **extents** operator, and implicitly through the loop generators. As in C, SA-C array index ranges always start at zero, and multi-dimensional arrays have a storage sequence order that varies the right-most indices the fastest. The rank-two array examples in this document are shown with rows referenced by the left index and columns by the right index. An array type specifier consists of a base type followed by a comma-separated list of either colons or integer constants enclosed in square brackets. The colon indicates that the array extent is defined dynamically, an integer constant statically declares the extent. Static extents allow the compiler to generate more efficient code. Examples of array types are:

```
int8[:]                 // 1D array of 8-bit signed integers
uint4[8,:]              // 2D array (8 rows) of 4-bit unsigned integers
bool[2,2,2]             // 3D ``hypercube'' of booleans
```

Because an array's extents are not part of its type, arrays of different sizes can have the same type.

3 Statements

SA-C has three statements: assignment, **print** and **assert**. The latter two are used primarily in debugging, whereas assignment statements form the bulk of statement blocks. Statements are terminated by semicolons.

3.1 Assignment statements

SA-C assignments differ from those in C in three ways. First, most assignments declare and define a variable simultaneously. Second, multiple values can be assigned concurrently. Third, the assignment is a statement, not an expression.

Except for variables with loop-carried dependencies (discussed later), each variable on the left-hand side of an '=' is declared and given a type. Here are examples of simple assignments:

```
uint8 a = b + 3;        // create and define scalar 'a'
int8 A[8] = f(c,8);     // create and define rank-1 array 'A'
bool M[:,:] = g(A,q);   // create and define rank-2 array 'M'
```

The left-hand side of an assignment can have a comma-separated list of identifiers, and the right-hand side must supply the appropriate number of values. For example,

```
int8 a, uint12 b, int2 c = 12, 44, 2;
```

assigns the three values to the variables. Assigning to multiple values is useful especially with functions, loops and conditionals that may return multiple values. All right-hand sides of an assignment are computed before the values are stored to the left-hand side variables, allowing the expressions to be computed concurrently. Since there are times when not all values returned from a multiple-value computation are needed, an underscore (don't care) can occur in place of an identifier on the left side of an assignment statement. Since arrays are created only monolithically (see section 4.2), it is not possible to assign to an individual array element, so the *only* entities that can occur on the left-hand side of an assignment are identifiers and underscores.

Unlike C, an assignment is not an expression, i.e. it does not return a value. This eliminates questions involving evaluation order that would arise if assignments were embedded within expressions.

3.2 Print and assert statements

To help in debugging, the **print** and **assert** statements allow printing values during program execution, and aborting under specified conditions.

The **print** statement needs two items: a boolean expression and a comma-separated sequence of strings and identifiers. During execution, the statement will print only if its boolean expression evaluates to **true**. If so, the strings and identifier values will be printed. Four examples of print statements are shown:

```
print (v>42, "x=", x, " z=", z);
print (true, "test point two", m);
print (w==3, qq);
print (v>42, "found a 'v' greater than 42");
```

These demonstrate that the constant **true** can be used if the statement is to print unconditionally. The **assert** statement differs from **print** in two ways: it acts only if the boolean expression evaluates to **false**, and it aborts execution after the information has been printed.

3.3 Scope of variables

Each variable that is declared and defined on the left-hand side of an assignment statement is in scope beginning with the semicolon that terminates that statement and ending with the closing parentheses that ends the statement block containing the statement, except that over parts of that lexical range it may be shadowed by newly declared variables with the same name. Such newly declared variables may be at the same level as the previously declared variable, or they may be in statement blocks that are nested within the current statement block. Figure 2 shows a variable a that is shadowed by two other variables of that name.

```
{
...
...
uint8 a = ...
uint8 b = ...
uint8 c = if (...) {
            uint8 d = ...
            uint8 a = ...
            uint8 k = ...
            } --> (...)
          else {
            uint8 e = ...
            uint8 f = ...
            } --> (...);
uint8 g = ...
uint8 h = ...
uint8 a = ...
uint8 i = ...
uint8 j = ...
} --> (...);
```

Figure 2: Example of scoping and shadowing of variable a.

One consequence of the declaration and scoping rules is that, other than the printing and aborting that can take place through **print** and **assert** statements, a statement block has no external side-effects. In other words, the values of variables after exiting a statement block are the same as when entering the block. Additional variable scoping issues arise in the context of **for** loops, and will be dealt with after these loops have been described.

4 Expressions

Expressions play a dominant role in SA-C programs. The language's integer scalar arithmetic and bit operators are the same as those in C, with the same associativities and precedences. The infix binary operators are shown in figure 3; all but the condition expression's ':' and '?' are left-associative. The two unary operators, '!' for boolean "not" and '-' for integer "negate", have higher precedence than the infix binary operators. Scalar expressions can be type cast in a manner similar to that of C: a scalar type in parentheses preceding the expression.

*	multiply
/	divide
%	mod
+	add
-	subtract
<<	bit-shift left
>>	bit-shift right
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
!=	not equal
&	bit-wise and
^	bit-wise xor
|	bit-wise or
&&	boolean and
||	boolean or
? :	conditional operator

Figure 3: Infix binary operators, in precedence groupings, highest to lowest

The numeric types are: **uint, int, ufix, fix, float, double**. Signed integer and fixed point arithmetic is performed in two's complement. The arithmetic operators $*$ / $+$ and $-$ apply to the numeric types only. The mod operator % applies to **uints** and **ints**. Integers consist of an optional sign and an integral part. Fixed point number consists of an optional sign, an integral part, and a fraction part. An integer can be considered a fixed point number with zero fraction size. The result type of an arithmetic expression is defined as follows.

- If either of the operands is signed, the result is signed. An operation on integer or fixed operands yields a result with the maximum integral and fraction operand sizes. Neither the integral, nor the fractional size can exceed 32. If the total size of the result exceeds 32, the size of the fraction will be reduced so as to make the total size 32. As an example, a fix16.10 combined with a fix16.14 will result in a fix20.14, whereas a fix32.10 combined with a fix32.20 will result in a fix32.10.

  An operation on a float and an integer or fixed point results in a float. An operation on a double and a float or integer or fixed point results in a double.

- If both operands are unsigned integers, the result of an operation is an unsigned integer congruent modulo 2^n to the true mathematical result of the operation, where n is the maximum size of the operands (e.g., (uint8)0xFF + 1 = 0x00).

The boolean operators apply to **boolean** operands only, the bit operators apply to **bits** operands only.

The shift operators have a left operand of type **bits** and a right operand of type **uint** or **int**. The result is of the same type as the left operand. A negative right operand gives rise to an unspecified result, and possibly (on the host, NOT on the FPGA) a run time warning.

The comparison operators $== \; ! = \; < \; > \; <= $ and $>=$ apply to all scalar types, where **bits** are interpreted as **uints**, and $false < true$. The equality comparison operators $==$ and $! =$ also apply to complex numbers.

The cast operation is at the same precedence level as the other unary operators. Unless subexpressions are explicitly cast, a cast expression defines the maximum precision of all its subexpressions. An assignment **ltype lhs = rhs** has the same meaning as **ltype lhs = (ltype) rhs**, i.e. the right hand side is cast to the type of the left hand side.

Casting a value of some type to another type, either explicitly or through assignment, can have two effects: the value can be **converted**, i.e. its bit representation can be changed, or the value can be **interpreted** in terms of the new type without change in its bit representation (apart from truncating leftmost bits or adding zero bits on the left to adjust size.) Interpretation occurs when casting any type to and from **bits**. Conversion occurs when any **non bits** is cast to any **non bits**. Casting an unsigned number to a signed number first adds a sign bit to the representation, and then either pads or truncates according to size specifications. Casting a signed negative number to an unsigned number has an unspecified result and causes a runtime error on the host.

4.1 Statement blocks

A block of statements can occur as an expression. SA-C blocks differ from C blocks in that they always produce return values, signaled by the keyword **return** which is followed by a parenthesized, comma-separated list of expressions. For example,

```
{ uint4 a = 3;
   int6 b = 4;
   int8 c = a*a - b*b;
} return (c, a+b)
```

creates and assigns values to three scalar variables and returns two values computed with them. Since SA-C is a *single-assignment* language, no variable can be given more than one value.

When a statement block occurs on the right hand side of an expression, it must contain at least one statement. For example:

```
int8 x, int6 y =  return (c, a+b);
```

is incorrect, and should be written as

```
int8 x, int6 y =  c, a+b;
```

Statement blocks also occur as bodies of functions, conditionals and loops. In these cases, where the return value can be computed directly with no statements or declarations within the curly braces, the braces may be omitted.

4.2 Array operations

Arrays can be created in a number of ways. First, an array can be explicitly typed, sized by integer constants, and given an expression for each of its elements in an assignment. For instance:

```
uint8 A[3,4] = {{3,6,7,2}
               ,{4,3,2,1}
               ,{7,1,0,8}};
```

The size of the array must be specified on the left hand side of the assignment, and the patterns within the curly braces must match the size specifiers. Since array extents can be zero, it is possible for a set of curly braces to enclose no values. A zero extent makes an array's size zero, but the curly braces still must match the specified extents. For example, both of the following array expressions create size-zero arrays, but note how the arrays' extents require different patterns of curly braces:

```
int8 Z1[0,3] = {};
int8 Z2[3,0] = { {}, {}, {} };
```

The second way of creating an array is through a return value of a loop. These are dealt with separately in section 4.4.

The **extents** operator returns the extents of an array as multiple values. For example, assuming the above definition of A,

```
uint8 m, uint8 n = extents (A);
```

will read the extents of A and put them into the scalar variables m and n. Individual array elements can be referenced using a single set of square brackets, with comma-separated integer expressions. Array slices can be taken using colon notation, similar to that of Fortran 90. A lone colon in a given dimension signifies taking the entire index range of the dimension. A subrange can be specified with integer expressions on either or both sides of the colon, and an optional step as a third parameter. Here are some examples, assuming the above definition of array A:

```
A[1,2]              // returns scalar value '2'
A[:,1]              // returns vector {6,3,1}
A[0,:]              // returns vector {3,6,7,2}
A[2,1:3]            // returns vector {1,0,8}
A[0:1,0:1]          // returns matrix {{3,6},{4,3}}
A[0,2:]             // returns vector {7,2}
A[0,:2:2]           // returns vector {3,7}
```

The **array_concat(A,B)** operator creates an array by concatenating the arrays A and B, which have to be of equal type, and of equal extents in all but the rightmost dimension. For example, if A is as above and B is defined as

```
uint8 B[3,2] = {{3,2},{4,1},{7,8}}
```

then **array_concat(A,B)** will create the array:

$$\begin{bmatrix} 3 & 6 & 7 & 2 & 3 & 2 \\ 4 & 3 & 2 & 1 & 4 & 1 \\ 7 & 1 & 0 & 8 & 7 & 8 \end{bmatrix}$$

The **array_conperim(A,w,v)** operator creates an array by surrounding an existing array **A** with a **perimeter** of a certain width **w**. All perimeter elements get the value **v**. For example,

```
uint8[:,:] C = array_conperim(A,1,0);
```

creates the following array C:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 6 & 7 & 2 & 0 \\ 0 & 4 & 3 & 2 & 1 & 0 \\ 0 & 7 & 1 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

An array also can be referenced through the reduction operators described in figure 4. These have the appearance of function calls, but they are actually operators. All but the **array_histogram** operator take one mandatory argument: the array being reduced. The **array_histogram** operator requires two arguments: the source array and a range value for the histogram array being produced. An optional final argument can be given, specifying a boolean mask array whose shape must be identical to the value array being reduced.

Unless their operands are cast to a more appropriate type, the results of **array_sum, array_product, array_min, array_max, array_median, array_mean, array_st_dev, array_and** and **array_or** have the same type as the array components; **array_min_indices**, **array_max_indices** and **array_histogram** produce arrays of **uint32**s.

Most of these array reduction operators can be used with the **array_accum** operator, which allows separate reductions to be performed on regions of the source array, specified by an array of labels. (The **array_mode**, **array_min_indices** and **array_max_indices** cannot be used in an accumulation since they cannot guarantee that the resulting array would be rectangular. This is because different regions of the source array may have different numbers of modes, mins or maxes, which would lead to ragged arrays.) The **array_accum** operator always takes three arguments: a reduction, a range value for the resulting array, and a label array whose shape must be identical to the array being reduced. A run-time error will occur if a label in the label array exceeds the extent set by the range specification.

Figure 5 shows examples of the integer reduction operators and their return values, and figure 6 shows examples of the **array_accum** operator.

4.3 Conditional expressions

Conditionals in SA-C return values, so every **if** must have a matching **else**. Additional **elif** branches between the **if** and **else** branches are allowed. Each branch of a conditional has an optional block of statements, and must return the same number and types of values. Recall that for arrays, type matching means that they must have the same scalar types for their values, and must have the same rank. They are *not* required to have the same extents. Here is an example using a conditional expression:

```
int8 x =   if (m > 8) { int8 z = q*3 } return (z%4)
         elif (m < 5) { int8 s = q*q-p } return (m+s)
         else return(0);
```

Multiple value returns are very useful with conditionals when more than one value needs to be set based on a condition. For example:

array_sum	Sum the array elements. If the array is empty, the return value is zero.
array_product	Multiply the array elements. If the array is empty, the return value is one.
array_min	Find the minimum of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_max	Find the max of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_and	'And' the boolean or bits array elements. If the array is empty, a 'true' value or all ones value is returned.
array_or	'Or' the boolean or bits array elements. If the array is empty, a 'false' value or all zeroes value is returned.
array_median	Find the median of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_mean	Find the mean of the array elements. If the array is empty, a run-time error occurs.
array_st_dev	Find the standard deviation of the array elements, returning a scalar integer. If the array size is less than two, a run-time error occurs.
array_mode	Find the mode of the array elements, returning a rank-one integer array of the values which occur most. If the array is empty, an empty array is returned.
array_min_indices	Find the index locations of the minimal array elements, returning a rank-two integer array. This reduction does not work on arrays of complex numbers. If the array is empty, an empty array is returned.
array_max_indices	Find the index locations of the maximal array elements, returning a rank-two integer array. This reduction does not work on arrays of complex numbers. If the array is empty, an empty array is returned.
array_histogram	Create a histogram of the array elements, returning a rank-one integer array. This requires a range specifier as its second argument to set the size of the returned array.

Figure 4: Array reduction operators

$$A = \begin{bmatrix} 2 & 1 & 9 & 2 \\ 6 & 9 & 2 & 9 \\ 1 & 9 & 1 & 1 \end{bmatrix}, M = \begin{bmatrix} true & false & true & true \\ true & false & false & true \\ false & true & false & true \end{bmatrix}$$

array_sum (A)	52
array_sum (A, M)	38
array_product (A)	314928
array_product (A, M)	17496
array_max (A)	9
array_max (A, M)	9
array_min (A)	1
array_min (A, M)	1
array_max_indices (A)	$\begin{bmatrix} 0 & 2 \\ 1 & 1 \\ 1 & 3 \\ 2 & 1 \end{bmatrix}$
array_max_indices (A, M)	$\begin{bmatrix} 0 & 2 \\ 1 & 3 \\ 2 & 1 \end{bmatrix}$
array_min_indices (A)	$\begin{bmatrix} 0 & 1 \\ 2 & 0 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$
array_min_indices (A, M)	$\begin{bmatrix} 2 & 3 \end{bmatrix}$
array_histogram (A, 10)	$\begin{bmatrix} 0 & 4 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 4 \end{bmatrix}$
array_histogram (A, 10, M)	$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \end{bmatrix}$
array_mean (A)	4
array_mean (A, M)	5
array_mode (A)	$\begin{bmatrix} 1 & 9 \end{bmatrix}$
array_mode (A, M)	$\begin{bmatrix} 9 \end{bmatrix}$
array_st_dev (A)	4
array_st_dev (A, M)	4
array_median (A)	2
array_median (A, M)	6

Figure 5: Example of some array reduction operations.

$$A = \begin{bmatrix} 2 & 1 & 9 & 2 \\ 6 & 9 & 2 & 9 \\ 1 & 9 & 1 & 1 \end{bmatrix}, M = \begin{bmatrix} true & false & true & true \\ true & false & false & true \\ false & true & false & true \end{bmatrix}, L = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 1 & 3 & 2 & 0 \\ 3 & 3 & 2 & 2 \end{bmatrix}$$

array_accum (array_sum (A), 4, L)	$\begin{bmatrix} 11 & 9 & 13 & 19 \end{bmatrix}$
array_accum (array_sum (A, M), 4, L)	$\begin{bmatrix} 11 & 8 & 10 & 9 \end{bmatrix}$
array_accum (array_product (A), 4, L)	$\begin{bmatrix} 18 & 12 & 18 & 81 \end{bmatrix}$
array_accum (array_product (A, M), 4, L)	$\begin{bmatrix} 18 & 12 & 9 & 9 \end{bmatrix}$
array_accum (array_max (A), 4, L)	$\begin{bmatrix} 9 & 6 & 9 & 9 \end{bmatrix}$
array_accum (array_max (A, M), 4, L)	$\begin{bmatrix} 9 & 6 & 9 & 9 \end{bmatrix}$
array_accum (array_min (A), 4, L)	$\begin{bmatrix} 2 & 1 & 1 & 1 \end{bmatrix}$
array_accum (array_min (A, M), 4, L)	$\begin{bmatrix} 2 & 2 & 1 & 9 \end{bmatrix}$
array_accum (array_histogram (A, 10), 4, L)	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$
array_accum (array_histogram (A, 10, M), 4, L)	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
array_accum (array_mean (A), 4, L)	$\begin{bmatrix} 6 & 3 & 3 & 6 \end{bmatrix}$
array_accum (array_mean (A, M), 4, L)	$\begin{bmatrix} 6 & 4 & 5 & 9 \end{bmatrix}$
array_accum (array_st_dev (A), 4, L)	$\begin{bmatrix} 5 & 3 & 4 & 5 \end{bmatrix}$
array_accum (array_st_dev (A, M), 4, L)	run-time error (st_dev of size-one array)
array_accum (array_median (A), 4, L)	$\begin{bmatrix} 9 & 2 & 2 & 9 \end{bmatrix}$
array_accum (array_median (A, M), 4, L)	$\begin{bmatrix} 9 & 6 & 9 & 9 \end{bmatrix}$

Figure 6: Examples of some **array_accum** operations.

```
uint8 new_xa, uint8 new_xb, uint8 val =
    if (A[xa] < B[xb])
        return (xa+1, xb, A[xa])
    else
        return (xa, xb+1, B[xb]);
```

embodies an operation that could occur during the merging of two arrays. If the current A value is less than the current B value, the A value is "taken" by returning an incremented index xa, an unchanged index xb, and the value from A. Otherwise A's index is unchanged, B's index is incremented, and the value from B is taken.

The **switch** expression takes an **int** or **uint** argument to determine which case branch is to be taken. The values in the cases are constants.

Here is an example:

```
uint8 x = switch(a-64) {
           case 0,1,2: { int8 z = q*3 }   return(z%4)
           case 3,4:   { int8 s = q*q-p } return(m+s)
           default:    return(0)
          }
```

If the default branch is absent and none of the cases apply, the result of the switch expression is unspecified (and will give rise to a run time error in host implementations.) As in C, overlapping cases are not allowed.

4.4 Loops

Loops in SA-C have return values. There are two kinds of loops, denoted by the keywords **while** and **for**. The **while** loop is the more general, and is designed for situations where, upon encountering the loop, the executing program cannot determine how many iterations will take place. The **for** loop, in contrast, determines exactly how many times the loop will iterate. It is not possible to break out of a **for** loop early.

4.4.1 While loops

The **while** loop looks similar to its C counterpart, except for the return value(s) following its closing curly brace. Because the single-assignment principle is in direct conflict with a loop's need to update an iteration variable, SA-C views a variable as if it has a separate instantiation for each iteration of the loop. The new value for an iteration can be assigned by using the keyword **next** in place of the type declaration that would otherwise occur. SA-C allows a **next** assignment to each variable at most once *within the body of a loop*, in addition to an assignment that takes place prior to the loop.

As an example, consider the following SA-C code:

```
uint8 n=5; uint8 ac=0;
uint8 s = while (n>0) {
      next ac = ac+n;
      next n = n-1;
      } return(final(ac));
```

There are loop-carried dependencies in this example, in variables ac and n. Each of the variables assigned in the loop can be thought of as having a separate instantiation for each iteration, and the semantics of the loop can be seen as if it is fully unrolled:

```
n=5; ac=0;
ac1 = ac+n;
n1 = n-1;
ac2 = ac1+n1;
n2 = n1-1;
ac3 = ac2+n2;
n3 = n2-1;
ac4 = ac3+n3;
n4 = n3-1;
ac5 = ac4+n4;
n5 = n4-1;
s = ac5;
```

Note that any variable with a loop-carried dependency *must* have been defined before the loop is entered; otherwise the use of that variable in iteration one would not have a value. Simple static analysis by the compiler can detect the failure to give an initial value and report it as an error. Note also that the loop returns a value for s but leaves the initial values given to n and ac unchanged (because of the side-effect-free nature of statement blocks.) Thus any reads of either of these variables in statements following the loop will get a value of 5 for n and a value of 0 for ac.

As with all statement bodies in SA-C, the **while** loop's body can return multiple values, and if there are no statements the curly braces may be omitted (though this would cause nontermination if the loop is entered.) Return values of loops are fully dealt with in section 4.4.2.

4.4.2 For loops

A For loop is used when the number of iterations is known before the loop is executed. There are two types of for loops: for loops with loop carried dependencies, expressed using **next** assignments, and "forall" loops without loop carried dependencies.

For loops have three parts: a *generator*, a *body*, and a *return expression*.

Loop generators **For** loops use generators to create iterations, and to create values for those iterations. There are three *simple* forms: *scalar*, *array-component* and *window* generators. Simple generators can be combined into compound generators using **dot** and **cross** loop product operators, discussed later in this section.

Scalar generators SA-C uses a '~' notation to generate scalar values. For example,

```
for uint8 i in [5~10]
```

will produce six iterations, with i having the values 5, 6, 7, 8, 9, and 10. Multiple values can be generated, using comma-separated lists. This example shows a SA-C generator of two target variables, and its C equivalent:

```
/* SA-C: */                                  /* C: */
for int8 i, int8 j in [5~10,2~4]             for (i=5; i<=10; i++)
                                               for (j=2; j<=4; j++)
```

Because a loop's iteration variables and ranges are often associated with array extents, a single expression (i.e., without a '~') automatically generates the appropriate index range for an array of that extent:

```
/* SA-C: */                                  /* C: */
for int8 i, int8 j in [m,n]                  for (i=0; i<m; i++)
                                               for (j=0; j<n; j++)
```

If non-unit steps are needed, an optional **step** specification can be used:

```
/* SA-C: */                                  /* C: */
for int8 i, int8 j in [m,n] step (a,b)       for (i=0; i<m; i+=a)
                                               for (j=0; j<n; j+=b)
```

Occasionally scalar generators may be used to create a specific set of iterations without needing to reference an iteration variable. An underscore ('_') can take the place of such a variable:

```
for _ in [n]
```

will produce n iterations without having to declare an iteration variable.

Array-component generators An array component generator takes components (elements, vectors, planes, etc.) out of an array and sends these into the iterations of a loop. Here is an example of its simplest form: `for val in A` . Array A can be of any shape. The iterations will take element values from A, in storage sequence order, and assign them to *val*, one value per iteration. The variable *val* can be used inside the loop's body and in its return value expressions. Other array access patterns are possible. The general form of an array component generator is:

```
for val (access-pattern) in array-expression
```

For each dimension in the array to be accessed, the access pattern has either a '~' specifier to indicate that individual elements are to be accessed, or a ':' to indicate that a whole slice in the array is to be accessed. The default access pattern takes individual elements out of the array specified by the array-expression, so `for val in A` has the same meaning as `for val (~,~) in A` and `for vec (~,:) in A` accesses all rows in A, whereas `for vec (:,~) in A` accesses all columns in A. An access pattern with all colons is not allowed, as this does not generate a loop.

The **step** specification allows one to stride through an array: `for val in A step(2,2)` only accesses the even rows and columns of A. An underscore must occur in positions where a slice is accessed, and a value must occur in positions where individual elements are accessed.

The **at** specification is used in cases where one needs the array indices, as in `for val in A at (uint8 i, uint8 j)` . Here the variables i and j capture the array indices, and can be referenced both in the body of the loop and in the return expressions. Underscores are used in **at** specifications, both in positions where there is no iterating index, because a whole slice is accessed, and in positions where the user does not need to capture an index value. Example: `for vec (:,~) in A at (_,uint8 j)` .

Window generators Another method of generating sub-arrays is the **window** generator. This allows a window to "slide" over the source array, producing sub-arrays of the same rank as the source array. The **window** keyword signals such a generator, with a size specification. For example: `for window W[3,3] in A` will access all 3 by 3 sub-matrices W from A.

Windows work on arrays of any rank, but the ranks of the window size specification and the source array must be equal. A window generator can be given an optional step specifier, for example: `for window W[3,3] in A step (3,3)` will produce non-overlapping 3 by 3 sub-matrices W.

Dot and cross products Compound generators, made up of simple generators, can be formed as **dot** and **cross** products.

The simple generators that are combined in **dot** products must be of identical shapes, i.e. dimensionality and extents, so that corresponding elements from each component can be sent into a loop body. Here is an example of a **dot** product, where two arrays are added together on an element-by-element basis:

```
for a in A  dot  b in B {
  int10 s = a + b;
```

The **cross** product generates all combinations of values from a set of simple generators; each simple generator can be of any shape, e.g., `for int8 i in [1~3]  cross  int8 j in [2~3]` will produce the (i, j) value pairs **(1,2)**, **(1,3)**, **(2,2)**,**(2,3)**, **(3,2)**, and **(3,3)**.

Generator shapes The *shape* of an array has already been defined as its rank and extents. In SA-C a loop's generator is considered to have a *shape* as well; it is used in defining how some of the loop's return operators construct their values.

The rank of a scalar generator is equal to the number of comma-separated entries in the generator's square brackets. The rank of an array-component generator is equal to the number of '~' specifiers in its access pattern. The rank of a window generator is equal to the rank of the window itself. Each extent is equal to the number of produced iterations. Here are some examples:

```
uint4 A[6,8,9] =  {{... ;
for v in A                                  // generator shape [6,8,9]
for v in A[2:4,0,5:8]                       // generator shape [3,4]
for v (~,:,:) in A                          // generator shape [6]
for int8 i, int8 j in [5,6]                 // generator shape [5,6]
for int8 i in [7~11]                        // generator shape [5]
```

In a valid **dot** product, the shapes of the components must be the same; the shape of a **dot** product is the same as the shape of each of its components. The shape of a **cross** product is obtained by concatenating the shapes of its components. For example:

```
uint4 A[6,8,9] =  {{... ;
for v in A cross int8 i in [7]              // generator shape [6,8,9,7]
...
for v (:,~,:) in A cross w (~,:,~) in A     // generator shape [8,6,9]
...
```

Loop-indices can be assigned to local loop body variables, as in

```
for e in A step(2,2)
 uint8 i, uint8 j = loop_indices();
 ...
```

produces consecutive values for i and j starting from 0.

Loop return values There are three kinds of return values that can follow the closed curly brace of a loop: an expression, a reduction, and a structuring operation.

Final values A loop can return the final value of a nextified variable. For example, here is one way of calculating n-factorial:

```
uint8 ac = 1;
uint16 fact =
    for uint8 v in [2~n] {
         next ac = ac * v;
         } return(final(ac));
```

The variable ac is being updated in each iteration, and its final value is returned.

Reductions All of the array reduction operators have exact counterparts as loop reductions; the loop keywords are formed by removing the **array_** prefixes from the array reduction operator names. Loop reductions can be performed on scalar and complex values, unless specified otherwise. Unless they are cast to a more appropriate type, the result types of **sum, product, min, max, median, mean, st_dev, and** and **or** are the same as the operand types, and **histogram** produces an array of **uint32**s. The array element type of **vals_at_mins** and **vals_at_maxs** is determined in the same way as expression results (see section 4). Figure 7 defines the loop reductions.

Just like array reductions, the loop reduction forms can be given an optional mask value as a scalar boolean expression. The **accum** operator uses loop reduction operators in a manner similar to that of the **array_accum** operator; the label specification is a integer expression.

As an example, here is a loop that produces the sum and mean of the odd numbers from one to one hundred:

```
uint16 s, ufix16.4 m = for uint8 i in [1~100] step (2) return (sum(i), mean(i));
```

The logical reductions **and** and **or** are useful for determining whether some condition occurred during loop execution. For example, a loop to determine whether array B contains any values greater than 42 could look like this:

```
bool s = for b in B return (or (b>42));
```

Structure-building operators There are two kinds of structure-building operators that can be used in creating a loop return value: **array** and **concat**.

sum	Sum the values. If the loop zero trips, the return value is zero.
product	Multiply the values. If the loop zero trips, the return value is one.
min	Find the min of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
max	Find the max of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
and	'And' the boolean or bits values. If the loop zero trips, a 'true' value or all ones value is returned.
or	'Or' the boolean or bits values. If the loop zero trips, a 'false' value or all zeroes is returned.
median	Find the median of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
mean	Find the mean of the values. If the loop zero trips, a run-time error occurs.
st_dev	Find the standard deviation of the values. In the case of a zero or one trip loop, a run-time error occurs.
mode	Find the mode of the values, returning a rank-one array of the values which occur most. If the loop zero trips, an empty array is returned.
vals_at_mins(x,{p,q,r,..})	Find the specified values p,q,r,... in the loop instances where x is minimal. This returns a rank-two array, each row containing a set of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_maxs(x,{p,q,r,..})	Find the specified values p,q,r,... in the loop instances where x is maximal. This returns a rank-two array, each row containing a set of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_first_min(x,{p,q,r,..})	Find the specified values p,q,r,... in the first loop instance where x is minimal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_first_max(x,{p,q,r,..})	Find the specified values p,q,r,... in the first loop instance where x is maximal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_last_min(x,{p,q,r,..})	Find the specified values p,q,r,... in the last loop instance where x is minimal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_last_max(x,{p,q,r,..})	Find the specified values p,q,r,... in the last loop instance where x is maximal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
histogram	Create a histogram of the array elements, returning a rank-one **uint32** array. This requires a range specifier as its second argument to set the size of the returned array.

Figure 7: Loop reduction operators

The **array** operator builds an array out of the specified component in its parentheses. For example, the following loop returns an array of the even numbers from two to one hundred:

```
uint8 nums[:] = for uint8 v in [2~100] step (2) return (array (v));
```

The specified component of an **array** operator can itself be an array, and since SA-C does not have arrays-of-arrays, this returns a result array whose rank is the sum of the ranks of the generator and the component. Thus, if a loop's generator has shape [3,5] and the **array** operator is used on components of shape [8,2], the return value will be a rank-four array of shape [3,5,8,2]. A SA-C programmer can use the **vector**, **matrix** and **cube** keywords in place of **array** for return values of rank one, two and three respectively. As an example, assuming that the array A = {1,3,5}, then

```
for a in A {
 uint8 V[2] = for _ in [2] return(vector(a));
} return(matrix(V))
```

will create a two dimensional array with extents [3,2]:

$$\begin{bmatrix} 1 & 1 \\ 3 & 3 \\ 5 & 5 \end{bmatrix}$$

The **concat** and **tile** operators have meaning only where the components being combined are arrays. Concat works only in one dimensional loops and allows the component arrays to be concatenated to have different sizes in the last dimension, as in the **array_concat** operator. Tile works in loops of any dimension, and hence all component arrays must be of exactly the same shape. The components are tiled together in each dimension of the loop. The rank of the value from a **tile** operator is the max of the ranks of the generator and the component. The extents are obtained by right-aligning the two extents and multiplying them element-wise. For example, if the loop generator has shape [3,9,5] and the **tile** component has extents [7,11], the return value of the operator will be a rank-three array of extents [3,63,55].

For example, assuming again that A = {1,3,5}, then

```
for a in A {
 uint8 V[:] = for _ in [a] return(array(a));
} return(concat(V))
```

will create a one dimensional array with extents [9]:

$$\begin{bmatrix} 1 & 3 & 3 & 3 & 5 & 5 & 5 & 5 & 5 \end{bmatrix}$$

As an other example, assuming that B is:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

then `for _ in [3] return(tile(B))` will create a matrix with extents [2,6]:

$$\begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$$

and `for _,_ in [2,3] return(tile(B))` will create a matrix with extents [4,6]:

$$\begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$$

The **accum** operator takes three arguments: a reduction, an integer range value for the resulting array, and a label value in the loop body. For each label value, the reduction is applied to each loop body with that label value. Label values should be in the range defined by the range specification, otherwise a run-time error will occur. As an example:

```
for v in V dot l in L
return(accum(sum(v),256,l));
```

produces an array of extents [256] with sums of values v for each label value l from 0 to 255.

Reductions **mode**, **vals_at_mins**, and **vals_at_maxs** cannot be used in an accumulation since they cannot guarantee that the resulting array would be rectangular.

Scope of variables in 'for' loops The scope rules for a variable created in a loop generator are similar to the rules for other variables. A generator target variable's scope is the lexical range beginning with the variable's declaration and ending with the closing parentheses of the loop's return values. However, there is a semantic restriction on the use of generator target variables: they may not be referenced elsewhere in that loop's generator. Also, the range expressions in a loop's return specification may not reference any variable that has been declared in that loop's generators or body.

5 Function definitions

SA-C has both user defined functions and intrinsic functions.

5.1 User Defined Functions

A SA-C function definition has a form similar to that used in C, but allowing for SA-C types and multiple return values. For example, the line

```
int8, uint4[:,:] f1 (uint8 A[:,:], int8 z, bool M[:]) {
```

is the beginning of function **f1**'s definition. It returns two values: a scalar 8-bit signed integer and a 2D-array of 4-bit unsigned integers. It takes three arguments. Here is a complete example of a SA-C transpose function:

```
uint8[:,:] transpose (uint8 A[:,:]) {
    uint8 res[:,:] = for V (:,~) in A return (matrix (V));
    } return (res);
```

It can be written more simply:

```
uint8[:,:] transpose (uint8 A[:,:])
    return (for V (:,~) in A return (matrix (V)));
```

Function calls look the same as they do in C: the function name is followed by a parenthesized, comma-separated list of argument values.

A SA-C program consists of one or more function definitions. The function named "main" is considered the top-level function. All function names are in scope throughout the entire program. SA-C functions cannot be recursive, as a SA-C program is to be laid out on an FPGA.

Functions defined in other modules need to have a function prototype. For example,

```
int8 fi8 (int8);
```

declares **fi8** to have been defined elsewhere. Function prototypes of the same function can occur multiple times in the same file as long as they are consistent, and they can occur with the actual function definition.

5.2 Intrinsic Functions

SA-C recognizes the C math library functions in figure 8 as intrinsic functions. These functions will invoke the C math library on host implementations. In figure 8, **NUM** stands for any numerical non-complex type. Some functions return more than one result. The first result is defined as the result of the corresponding C math function, the second result is what the corresponding C function returns in a reference parameter.

Return Type(s)	Function	Parameter Type(s)
double	sin	NUM
double	cos	NUM
double	tan	NUM
double	asin	NUM
double	acos	NUM
double	atan	NUM
double	atan2	NUM, NUM
double	sinh	NUM
double	cosh	NUM
double	asinh	NUM
double	acosh	NUM
double	atanh	NUM
double	sqrt	NUM
double	cbrt	NUM
double	pow	NUM
double, double	modf	NUM
double	exp	NUM
double, int32	frexp	NUM
double	ldexp	NUM
double	log	NUM
double	log10	NUM
double	expm1	NUM
double	log1p	NUM
double	ceil	NUM
double	fabs	NUM
double	floor	NUM
double	fmod	NUM, NUM
double	copysign	NUM, NUM
double	hypot	NUM, NUM
double	rint	NUM

Figure 8: Return Type(s), Intrinsic Function, Parameter Type(s)

A Program examples

The following small programs may help to illustrate SA-C's language principles.

A.1 Matrix multiply

```
// matrix multiply of 'A' and 'B':
uint8[:,:] main (uint8 A[:,:], uint8 B[:,:]) {
    _, uint8 n = extents (A);
    uint8 m, _ = extents (B);
    assert (n==m, "matrices cannot be multiplied");
    uint8 res[:,:] =
      for VA (~,:) in A cross VB (:,~) in B {
        uint8 Ele = for a in VA dot b in VB  return (sum (a * b)); // inproduct(VA,VB)
      } return (matrix (Ele));
    } return (res);
```

A.2 Flattening a matrix to a vector

```
int8[:] main (int8 A[:,:]) return(
    for V (~,:) in A return (tile(V))
    );
```

A.3 Expanding an Image by doubling both extents

```
// Each pixel is replicated in both dimensions
uint8[:] main (uint8 A[:,:]) return(
     for e in A {
      uint8 E[2,2] = {{e,e},{e,e}};
     } return(tile(E))
     );
```

A.4 Shrinking an Image

```
// Each 2*2 submatrix is shrunk to one value (the maximum)
uint8[:] main (uint8 A[:,:]) return(
   for window W[2,2] in A step (2,2) return(matrix(array_max(W)));
  );
```

A.5 Restructuring a vector to a matrix

```
int8[:,:] main (int8 A[:], int8 n) {
    int8 s = extents (A);
    assert (s%n==0, "result array not rectangular", s, n);
    int8 res[:,:] = for window V[n] in A step (n) return (array (V));
    } return (res);
```

A.6 $S \times S$ Median filter

```
// s*s Median filter of an image, padded with an s/2 perimeter
uint8[:,:] main (int8 A[:,:], uint8 s) {
// s should be odd
  assert (( (bits8)(s) & 0b1 ) == 0b1,  "s should be odd\n");
  uint8 res[:,:] = for window W[s,s] in  array_conperim(A,s/2,0)
                  return(array(array_median(W)));
  } return(res);
```

A.7 Prewitt Horizontal Edge Detection

```
// Convolution with constant Mask
int16[:,:] main (uint8 A[:,:]) {
  int8 Mask[3,3] = {  {-1,-1,-1}  ,
                      { 0, 0, 0}  ,
                      { 1, 1, 1}  } ;

  int16 res[:,:]  = for window W[3,3] in A {
                      int16 ip = for w in W dot m in Mask
                                  return( sum (w*m) );
                    } return( array(ip) );
  } return( res );
```

A.8 Morphology: Conway's game of *Life*

```
uint1[:,:] main (uint1 A[:,:], uint16 n) {

    bool M[3,3] = { {true, true, true}  ,
                    {true,false, true}  ,
                    {true, true, true}  } ;
    uint1 res[:,:] =
      for _ in [n] {
        next A = for window W[3,3] in array_conperim (A, 1, 0) {
                  uint3 c = array_sum (W, M);
                  uint1 v = (c==3 || c==2 && W[1,1]==1) ? 1 : 0;
                } return (array (v));
        print (true, A);
       } return (A);
    } return (res);
```

A.9 Parallel Prefix

```
// Produce an array with the sum of input array elements 0 .. i in position i
uint8[:] main(uint8 vec[:]) {
   uint8 sz = extents(vec);
   uint8 dist = 1;
   uint16 cumvec[:] =
     while (dist < sz) {
      next vec =  for e in vec at (uint8 i) {
                    uint8 elem = ((i < dist) ? e : e + vec[i-dist]);
                  } return(array(elem));
      next dist = 2 * dist;
     } return(vec);
} return (cumvec);
```

A.10 Region Statistics

```
// For each region in A, indicated by a label in LabelPlane L
// with R different labels, produce the mean value.
uint8[:] main (uint8 A[:,:], uint8 L[:,:], uint8 R ) return(
   for pix in A dot lab in L return( accum(mean(pix),R,lab))
   );
```

A.11 Region Statistics: array operators

```
// Same as above
uint8[:] main (uint8 A[:,:], uint8 L[:,:], uint8 R ) return(
   array_accum(array_mean(A),R,L)
   );
```

B BNF rules

```
rule 1    program -> exports funcs
rule 2    exports -> /* empty */
rule 3    exports -> TOK_EXPORT export_list ';'
rule 4    export_list -> TOK_IDENT
rule 5    export_list -> export_list ',' TOK_IDENT
rule 6    funcs -> func
rule 7    funcs -> funcs func
rule 8    func -> opt_func_prag header '(' types_w_ids ')' body_w_return ';'
rule 9    func -> opt_func_prag header '(' ')' body_w_return ';'
rule 10   func -> opt_func_prag header '(' types_wo_ids ')' ';'
rule 11   func -> opt_func_prag header '(' types_w_ids ')' ';'
rule 12   func -> opt_func_prag header '(' ')' ';'
rule 13   opt_func_prag -> /* empty */
rule 14   opt_func_prag -> TOK_PRAGMA '(' func_pragmas ')'
rule 15   func_pragmas -> func_pragma
rule 16   func_pragmas -> func_pragmas ',' func_pragma
rule 17   func_pragma -> TOK_NO_INLINE
rule 18   func_pragma -> TOK_LOOKUP
rule 19   header -> types_wo_ids TOK_IDENT
rule 20   types_wo_ids -> type_wo_id
rule 21   types_wo_ids -> types_wo_ids ',' type_wo_id
rule 22   type_wo_id -> type optbrackets
rule 23   types_w_ids -> type_w_id
rule 24   types_w_ids -> types_w_ids ',' type_w_id
rule 25   type_w_id -> type TOK_IDENT optbrackets
rule 26   base_type -> TOK_INT
rule 27   base_type -> TOK_FIX
rule 28   base_type -> TOK_BITS
rule 29   base_type -> TOK_BOOL
rule 30   base_type -> TOK_FLOAT
rule 31   base_type -> TOK_DOUBLE
rule 32   type -> base_type
rule 33   type -> TOK_COMPLEX base_type
rule 34   optbrackets -> /* empty */
rule 35   optbrackets -> '[' optsizes ']'
rule 36   optsizes -> optsize
rule 37   optsizes -> optsizes ',' optsize
rule 38   optsize -> ':'
rule 39   optsize -> TOK_INTNUM
rule 40   for_loop -> opt_loop_prag TOK_FOR generators '{' stmts '}' TOK_RETURN '(' ret_exprs ')'
rule 41   for_loop -> opt_loop_prag TOK_FOR generators TOK_RETURN '(' ret_exprs ')'
rule 42   opt_loop_prag -> /* empty */
rule 43   opt_loop_prag -> TOK_PRAGMA '(' loop_pragmas ')'
rule 44   loop_pragmas -> loop_pragma
rule 45   loop_pragmas -> loop_pragmas ',' loop_pragma
rule 46   loop_pragma -> TOK_NO_UNROLL
rule 47   loop_pragma -> TOK_NO_DFG
rule 48   loop_pragma -> TOK_STRIPMINE '(' const_list ')'
rule 49   loop_pragma -> TOK_NO_FUSE
rule 50   generators -> simple_gen opt_cmpnds
rule 51   opt_cmpnds -> /* empty */
```

```
rule 52   opt_cmpnds -> more_dots
rule 53   opt_cmpnds -> more_crosses
rule 54   more_dots -> TOK_DOT simple_gen
rule 55   more_dots -> more_dots TOK_DOT simple_gen
rule 56   more_crosses -> TOK_CROSS simple_gen
rule 57   more_crosses -> more_crosses TOK_CROSS simple_gen
rule 58   simple_gen -> TOK_IDENT opt_extract TOK_GEN expr opt_elegen_step opt_at
rule 59   simple_gen -> cl4 TOK_GEN_SC '[' cl2 ']' opt_expr_step
rule 60   simple_gen -> TOK_WINDOW TOK_IDENT '[' cl7 ']' TOK_GEN expr opt_expr_step opt_at
rule 61   opt_extract -> /* empty */
rule 62   opt_extract -> '(' colon_twids ')'
rule 63   colon_twids -> colon_twid
rule 64   colon_twids -> colon_twids ',' colon_twid
rule 65   colon_twid -> ':'
rule 66   colon_twid -> '~'
rule 67   opt_elegen_step -> /* empty */
rule 68   opt_elegen_step -> TOK_STEP '(' opt_exprs ')'
rule 69   opt_exprs -> emp_expr
rule 70   opt_exprs -> opt_exprs ',' emp_expr
rule 71   emp_expr -> '_'
rule 72   emp_expr -> expr
rule 73   opt_expr_step -> /* empty */
rule 74   opt_expr_step -> TOK_STEP '(' exprs ')'
rule 75   opt_at -> /* empty */
rule 76   opt_at -> at_spec
rule 77   at_spec -> TOK_AT '(' cl4 ')'
rule 78   cl2 -> ele2
rule 79   cl2 -> cl2 ',' ele2
rule 80   ele2 -> expr '~' expr
rule 81   ele2 -> expr
rule 82   cl9 -> ele9
rule 83   cl9 -> cl9 ',' ele9
rule 84   ele9 -> type_w_id
rule 85   ele9 -> TOK_NEXT TOK_IDENT
rule 86   ele9 -> '_'
rule 87   cl4 -> ele4
rule 88   cl4 -> cl4 ',' ele4
rule 89   ele4 -> type_w_id
rule 90   ele4 -> '_'
rule 91   cl7 -> expr
rule 92   cl7 -> cl7 ',' expr
rule 93   ret_exprs -> ret_expr
rule 94   ret_exprs -> ret_exprs ',' ret_expr
rule 95   ret_expr -> TOK_FINAL '(' TOK_IDENT ')'
rule 96   ret_expr -> loop_reduction
rule 97   ret_expr -> struct_op '(' expr ')'
rule 98   ret_expr -> TOK_ACCUM '(' loop_reduction ',' expr ',' expr ')'
rule 99   loop_reduction -> reduce_op '(' expr ')'
rule 100  loop_reduction -> reduce_op '(' expr ',' expr ')'
rule 101  loop_reduction -> reduce_op '(' expr ',' expr ',' expr ')'
rule 102  loop_reduction -> TOK_VAL_AT_MINS '(' expr ',' value_cluster ',' expr ')'
rule 103  loop_reduction -> TOK_VAL_AT_MINS '(' expr ',' value_cluster ')'
rule 104  loop_reduction -> TOK_VAL_AT_MAXS '(' expr ',' value_cluster ',' expr ')'
```

```
rule 105  loop_reduction -> TOK_VAL_AT_MAXS '(' expr ',' value_cluster ')'
rule 106  value_cluster -> '{' exprs '}'
rule 107  arr_reduce_op -> opt_loop_prag TOK_ARR_SUM
rule 108  arr_reduce_op -> opt_loop_prag TOK_ARR_MIN
rule 109  arr_reduce_op -> opt_loop_prag TOK_ARR_MAX
rule 110  arr_reduce_op -> opt_loop_prag TOK_ARR_RED_AND
rule 111  arr_reduce_op -> opt_loop_prag TOK_ARR_RED_OR
rule 112  arr_reduce_op -> opt_loop_prag TOK_ARR_MEDIAN
rule 113  arr_reduce_op -> opt_loop_prag TOK_ARR_MAX_INDICES
rule 114  arr_reduce_op -> opt_loop_prag TOK_ARR_MIN_INDICES
rule 115  arr_reduce_op -> opt_loop_prag TOK_ARR_PRODUCT
rule 116  arr_reduce_op -> opt_loop_prag TOK_ARR_MEAN
rule 117  arr_reduce_op -> opt_loop_prag TOK_ARR_ST_DEV
rule 118  arr_reduce_op -> opt_loop_prag TOK_ARR_MODE
rule 119  arr_reduce_op -> opt_loop_prag TOK_ARR_HIST
rule 120  arr_reduce_op -> opt_loop_prag TOK_ARR_CONCAT
rule 121  reduce_op -> TOK_SUM
rule 122  reduce_op -> TOK_MIN
rule 123  reduce_op -> TOK_MAX
rule 124  reduce_op -> TOK_RED_AND
rule 125  reduce_op -> TOK_RED_OR
rule 126  reduce_op -> TOK_MEDIAN
rule 127  reduce_op -> TOK_PRODUCT
rule 128  reduce_op -> TOK_MEAN
rule 129  reduce_op -> TOK_ST_DEV
rule 130  reduce_op -> TOK_MODE
rule 131  reduce_op -> TOK_HIST
rule 132  struct_op -> TOK_ARRAY
rule 133  struct_op -> TOK_VECTOR
rule 134  struct_op -> TOK_MATRIX
rule 135  struct_op -> TOK_CUBE
rule 136  struct_op -> TOK_CONCAT
rule 137  struct_op -> TOK_TILE
rule 138  stmts -> /* empty */
rule 139  stmts -> stmts stmt
rule 140  stmt -> cl9 '=' exprs ';'
rule 141  stmt -> cl9 '=' TOK_LOOP_INDICES '(' ')' ';'
rule 142  stmt -> cl9 '=' arr_slice ';'
rule 143  stmt -> TOK_PRINT '(' expr ',' pr_entities ')' ';'
rule 144  stmt -> TOK_ASSERT '(' expr ',' pr_entities ')' ';'
rule 145  stmt -> error ';'
rule 146  pr_entities -> pr_entity
rule 147  pr_entities -> pr_entities ',' pr_entity
rule 148  pr_entity -> TOK_IDENT
rule 149  pr_entity -> TOK_STRING
rule 150  exprs -> expr
rule 151  exprs -> exprs ',' expr
rule 152  expr -> TOK_IDENT '(' exprs ')'
rule 153  expr -> TOK_INTRINSIC '(' exprs ')'
rule 154  expr -> TOK_ARR_CONPERIM '(' expr ',' expr ',' expr ')'
rule 155  expr -> TOK_EXTENTS '(' TOK_IDENT ')'
rule 156  expr -> '{' stmt stmts '}' TOK_RETURN '(' exprs ')'
rule 157  expr -> TOK_IDENT '(' ')'
```

```
rule 158  expr -> TOK_IDENT
rule 159  expr -> TOK_IDENT '[' array_indices ']'
rule 160  expr -> TOK_INTNUM
rule 161  expr -> TOK_FLOATNUM
rule 162  expr -> TOK_TRUE
rule 163  expr -> TOK_FALSE
rule 164  expr -> arr_reduce_expr
rule 165  expr -> switch
rule 166  expr -> TOK_ARR_ACCUM '(' arr_reduce_expr ',' expr ',' expr ')'
rule 167  expr -> '(' expr ',' expr ')'
rule 168  expr -> TOK_REAL '(' expr ')'
rule 169  expr -> TOK_IMAG '(' expr ')'
rule 170  expr -> expr '+' expr
rule 171  expr -> expr '-' expr
rule 172  expr -> expr '*' expr
rule 173  expr -> expr '/' expr
rule 174  expr -> expr '%' expr
rule 175  expr -> expr '<' expr
rule 176  expr -> expr '>' expr
rule 177  expr -> expr TOK_LE expr
rule 178  expr -> expr TOK_GE expr
rule 179  expr -> expr TOK_NE expr
rule 180  expr -> expr TOK_EQUAL expr
rule 181  expr -> expr TOK_AND expr
rule 182  expr -> expr TOK_OR expr
rule 183  expr -> expr '&' expr
rule 184  expr -> expr '|' expr
rule 185  expr -> expr '^' expr
rule 186  expr -> expr TOK_LEFT_SHIFT expr
rule 187  expr -> expr TOK_RIGHT_SHIFT expr
rule 188  expr -> '!' expr
rule 189  expr -> '-' expr
rule 190  expr -> '(' type_wo_id ')' expr
rule 191  expr -> '(' expr ')'
rule 192  expr -> for_loop
rule 193  expr -> while_loop
rule 194  expr -> conditional
rule 195  switch -> TOK_SWITCH '(' expr ')' '{' cases opt_default '}'
rule 196  cases -> /* empty */
rule 197  cases -> cases case
rule 198  case -> TOK_CASE const_list ':' body_w_return
rule 199  const_list -> constval
rule 200  const_list -> const_list ',' constval
rule 201  constval -> TOK_INTNUM
rule 202  constval -> '-' TOK_INTNUM
rule 203  opt_default -> /* empty */
rule 204  opt_default -> TOK_DEFAULT ':' body_w_return
rule 205  arr_reduce_expr -> arr_reduce_op '(' expr ')'
rule 206  arr_reduce_expr -> arr_reduce_op '(' expr ',' expr ')'
rule 207  arr_reduce_expr -> arr_reduce_op '(' expr ',' expr ',' expr ')'
rule 208  arr_slice -> '{' '}'
rule 209  arr_slice -> '{' exprs '}'
rule 210  arr_slice -> '{' slices '}'
```

```
rule 211  slices -> arr_slice
rule 212  slices -> slices ',' arr_slice
rule 213  while_loop -> TOK_WHILE '(' expr ')' '{' stmts '}' TOK_RETURN '(' ret_exprs ')'
rule 214  conditional -> TOK_IF '(' expr ')' body_w_return elifs TOK_ELSE body_w_return
rule 215  conditional -> expr '?' expr ':' expr
rule 216  elifs -> /* empty */
rule 217  elifs -> elifs elif
rule 218  elif -> TOK_ELIF '(' expr ')' body_w_return
rule 219  body_w_return -> '{' stmts '}' TOK_RETURN '(' exprs ')'
rule 220  body_w_return -> TOK_RETURN '(' exprs ')'
rule 221  array_indices -> array_index
rule 222  array_indices -> array_indices ',' array_index
rule 223  array_index -> expr
rule 224  array_index -> triple
rule 225  triple -> colon_bounds
rule 226  triple -> colon_bounds ':' expr
rule 227  colon_bounds -> opt_expr ':' opt_expr
rule 228  opt_expr -> /* empty */
rule 229  opt_expr -> expr
```