CAST

Language description and libraries

Rajit Manohar

This document describes the language used by the Caltech Asynchronous Synthesis Tools (CAST), and how to write new tools that understand this language using the standard CAST library.

1 A Tutorial Introduction

The Caltech Asynchronous Synthesis Tools (CAST) is a suite of programs that can be used to design, verify, and test asynchronous circuits. The tools share a common input language—CAST—which can be used to describe circuits at different levels of detail.

CAST is a hierarchical, lexically scoped circuit description language. A single CAST file can be used to describe the transistor implementation of a circuit as well as a high-level functional description of the same circuit.

There are two parts to CAST—a parsing phase, and an instantiation phase. In the parsing phase, the input is converted into internal data structures. In the instantiation phase, data structures are created for every circuit element specified in the CAST file. Errors may be reported in any of these stages. As a rule of thumb, errors that can be checked statically are reported during parsing.

1.1 A Simple Example

To get a feel for how a circuit is described in CAST, we begin with a simple example circuit. The purpose of this circuit is to create a dual rail channel and attach a bit-bucket to it.

```
/* my first cast program */
define dualrail() (node 0,1,a)
{
  spec {
    exclhi(0,1)  // exclusive high directive
  }
}

define bitbucket() (dualrail d)
{
  prs {
    d.0 | d.1  -> d.a+
   ~d.0 & ~d.1 -> d.a-
  }
}

bitbucket b;
dualrail c;

b.d = c;
```

`define dualrail` is used to define a new type `dualrail` with an empty first parameter list, and a second parameter list which consists of three nodes named `0`, `1`, and `a`. A `node` is an electrical node in the circuit, and is a built-in type. Observe that integer identifier names for nodes are permitted. The *body* of the type `dualrail` is enclosed in braces.

The type `dualrail` has a body which consists of a `spec` body. The construct *name* `{ ... }` is used to specify a *language-specific body* within a type definition. A type can have any number of these language-specific bodies. A program using the CAST library indicates which language-specific bodies it recognizes, and the CAST library will automagically ignore all other bodies. In this case, the word `spec` is recognized to be a *specification* body.

The specification body contains an assertion indicating that the nodes `0` and `1` are exclusive high, meaning that $\neg 0 \vee \neg 1$ is an invariant of the system. Whenever a circuit element of type `dualrail` is created, this specification will be automatically attached to it.

The type `bitbucket` is defined to take a `dualrail` variable `d` as its argument. The body of `bitbucket` contains a different language-specific body, namely a `prs` body. `prs` bodies correspond to *production rules*. In the example, the production rules for `bitbucket` corresponds to what is commonly referred to as a "bit-bucket" for a four phase dual rail channel.

The statement `bitbucket b` creates an *instance* of type `bitbucket` named `b`. The statement is said to be an *instantiation*. Execution of this statement creates variable `b` of type `bitbucket`, production rules corresponding to the body of a `bitbucket`, and the specification body for the `dualrail` variable `b.d`. Similarly, the statement `dualrail d` creates an instance of type `dualrail`. CAST uses the standard dot-notation to access the names that are in parameter lists, since they are analogous to the fields of types in conventional programming languages.

The final statement `b.d=c` *connects* the two `dualrail` types `b.d` and `c`. The effect of connecting two types is to connect all the fields specified in the parameter lists of the types. Connecting two nodes corresponds to making them the same electrical node. Therefore, the connect statement specifies that the nodes `b.d.0`, `b.d.1`, `b.d.a` are electrically connected to the nodes `d.0`, `d.1`, `d.a` respectively.

CAST recognizes both C and C++ style comments, and they are treated as white space along with spaces, tabs, and new lines. A C-style comment begins with the characters `/*` and ends with `*/`. Everything between the beginning and end of the comment is treated as whitespace. A C++-style comment begins with `//` and continues till the end of line.

1.2 Variables and Expressions

Variables in CAST fall into two basic categories: *parameters* and *circuit elements*. A parameter is a variable that is used to parameterize a circuit element in some way and must be of type integer (`int`), real (`float`), or boolean (`bool`). Circuit elements are either electrical nodes (`node`) or *user-defined types* created using the `define` construct. In the example above, both `dualrail` and `bitbucket` are user-defined types.

A variable identifier can be a sequence of digits, letters, and underscores. Integers can be used as identifier names for variables of type `node` and user-defined types. The following declarations are legal:

```
bitbucket b;
dualrail 1x;
dualrail 2;
node 5;
```

On the other hand, the following declaration is incorrect.

```
int 5;
⇒ error  FATAL: In INSTANTIATION, expected identifier, got ‘5’
```

CAST treats an arbitrary string of characters as an identifier. If you enclose the string in double quotes, CAST treats the string with the double quotes stripped off as the identifier name. Therefore,

```
int "5";
```

would constitute a valid instantiation.

The names in the parameter list of a user-defined type are the parts of the type that are visible externally. Other parts of the defined type cannot be accessed outside the body of the type itself. For example, consider the following definition of `bitbucket`.

```
define bitbucket() (dualrail d)
{
  node p;
  prs {
    d.0 | d.1  -> d.a+
   ~d.0 & ~d.1 -> d.a-
  }
}
```

If we had used this definition, then although `b.p` is a node within the bit-bucket `b`, we cannot access it by statements outside the body. Therefore, a statement such as `b.p=c.0` would result in the following message:

```
bitbucket b;
dualrail c;
b.p = c.0;
⇒ [error] FATAL: ‘b’ (type ’bitbucket’) does not have ‘p’ as a field
```

Expressions look very much like C expressions. Expressions can be of two types: numeric or logical. Numeric expressions can be constructed from identifiers, numeric constants, parentheses for grouping, and the arithmetic operators `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division respectively. The unary minus operator is also supported. The operator `%` is used for computing the remainder. Logical expressions can be constructed from logical variables, the logical constants `true` and `false`, and the logical operators `&`, `|`, and `~` denoting the and, or, and negation operations respectively. Numeric expressions can be compared using `<`, `<=`, `>`, `>=`, `==`, and `!=` for the operators less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to respectively.

1.3 Arrays

Since most circuits consist of a set of components that are replicated a number of times, CAST has a very flexible array mechanism. The simplest way to create an array is shown below.

```
node[10] x;
dualrail[10,10] d;
```

The first statement creates an array of ten nodes named `x`. whose elements are `x[0]`, …, `x[9]`. The second statement creates a two-dimensional array of `dualrail` variables named `d` whose elements are `d[0,0]`, …, `d[9,9]`. The array index ranges can also be specified directly within the square brackets, as shown below.

```
node[6..12] x; // create nodes x[6], ..., x[12]
```

The expression `6..12` is referred to as a range expression.

CAST provides a mechanism for creating *sparse arrays*. These are arrays where the list of valid indices is not a contiguous block. Consider the following instantiations:

```
node x[10];
node x[12];
```

The first instantiation creates `x`, an array which has only one valid index, namely `10`. The second instantiation adds another valid index `12` to `x`! Therefore, `x` now is a sparse one-dimensional array with elements at positions `10` and `12`. The following examples show some of the errors that can occur.

```
node x[10];
node x[12];
node[12] x;
⇒ error  Instantiation of ‘x[0..11]’ overlaps with previous
instantiation

node x[10];
int x[5];
⇒ error  Type mismatch in instantiation of ‘x’
```

Arrays can be connected to other arrays by using the `=` operation. If the two variables being connected are both arrays of the same type, an element-by-element connection is performed as long as the two arrays have the same size. The following examples show some of the array connections that can be performed.

```
node[4] x;
node[7..10] y;
x=y;       // successful
int z[10];
x=z;
⇒ error  In CONNECT, incompatible base types ‘node[]’ and ‘int[]’
```

1.4 Loops and Conditionals

Loops and conditionals can be used to create complex circuit structures. Loops are useful when creating sparse array structures, or for connecting two sets of arrays to one another. In the example shown below, an array of `10` nodes is created by using the sparse array instantiation mechanism within a loop.

```
< i : 0 .. 9 : node x[i]; >
```

The angle brackets are used to group the loop statement. `i` is the dummy variable of the loop, and ranges from `0` to `9`. If the range is specified as a single expression, the range is assumed to be zero to the value of the expression minus one.

The conditional statement in CAST is of the form "if expression is true, then execute the body." In the example below, node `x[i]` is created if `i>0` evaluates to true.

```
[ i > 0 -> node x[i]; ]
```

Conditionals are also extensively used when creating recursive structures. The following is an example where the odd-numbered indices of array `x` and `y` are connected together.

```
node[10] x, y;
< i : 10 :  [ i % 2 == 1 -> x[i] = y[i]; ] >
```

1.5 Scoping

In the second definition of `bitbucket`, the variable `p` was defined within the body of the type definition. Therefore, this variable is local to the type, and cannot be accessed by any construct outside the body of the type. Different instances of `bitbucket` get different copies of `p`, since it is a local variable. If we had created a dualrail channel `p` after the bitbucket, this `p` has no relation to the `p` in the body of `bitbucket`.

In a particular part of the CAST language, every variable that is defined in a scope outside the body can be used within the body itself. The braces `{ ... }` introduce a new scope. Parameters of types have the same scope as items defined within the body of the type. However, parameters are special in that they can also be accessed from outside the type using dot-notation.

CAST does not have a special "global" keyword. Global nodes can be created by simply defining them in the outer-most scope. For instance, CAST files will tend to begin with

```
node Reset,Reset_;
```

This permits the names `Reset` and `Reset_` to be used within all the bodies in the CAST file, effectively making the names global. It is possible to define a variable that hides (*shadows*) a definition in an outer scope. This practice should be avoided as far as possible.

```
node Reset,Reset_;
```

```
define test() (node m)
{
  node Reset;    // shadows earlier definition of Reset!
}
⇒ warning  Definition of variable 'Reset' shadows earlier definition
```

The definition of types can be nested within scopes as well. However, nested type definitions cannot be accessed from outside the scope in which they are visible.

```
define bitbucket() (dualrail d)
{
  define dr() (node[2] x)
  {
    ...
  }
}
bitbucket.dr x;
⇒ error  Can't access internal definitions!
```

As before, a warning is printed if a type name shadows a previous type definition.

```
define bitbucket() (dualrail d)
{
  define dualrail() (node[2] x)
  {
    ...
  }
}
⇒ warning  Name 'dualrail' shadows a previous declaration
```

1.6 Importing and Including Files

The keyword `include` can be used in a manner similar to the C `#include` directive to include other CAST files. The contents of the specified filename are textually substituted at the location of the `include` statement. These statements can be placed anywhere within a CAST file.

The keyword `import` is similar to `include`, and can occur anywhere in a CAST body. If the same file is imported twice within the same scope, chances are that some types would be multiply defined. To avoid such problems, imports of files which have already been imported within the same scope or an outer scope are ignored. Therefore, always use `import` to include type definitions defined elsewhere.

```
import "channel.cast";
...
```

Both `include` and `import` search in the current directory first, then in the colon-separated path specified by `CAST_PATH`, and finally in `CAST_HOME/cast`.

2 Types and Variables

Variables are the basic data objects in CAST. Instantiations specify which variables are created, and state what type they have. The type of an object completely specifies what the object is and how it can be used.

Types come in two flavors: parameters and circuit elements. Parameters are variables whose types are `int`, `float`, `bool`, or arrays of `ints`, `floats`, or `bools`. All other types refer to circuit elements. The basic circuit element is an electrical `node`.

There are some restrictions on variable names. Ordinarily a variable identifier can be constructed as an arbitrary sequence of underscores, letters, and digits. Unlike a number of programming languages, CAST accepts identifiers that begin with digits as well. Identifier names are case sensitive, so `case` and `Case` are different identifiers.

Integers are permitted to be used as identifier names for nodes and user-defined types. There is no ambiguity in this convention because nodes and user-defined types can never be part of integer expressions, and vice versa.

2.1 Basic Types

There are four basic types supported by CAST:

- `int`, for integer variables.
- `float`, for real-valued variables.
- `bool`, for boolean-valued variables.
- `node`, for electrical nodes.

The first three of the four types (and arrays of them) are referred to as *parameter* types or *meta-language* types.

Variables of these basic types can be created by specifying the type name followed by a comma-separated list of identifier names.

```
node a,b,c,1,1x2;
int x,y,z;
```

```
float w2,w_3;
```

The statements above are referred to as *instantiations*, since they create variables that are instances of the basic types. It is an error to have more than one instantiation of a variable in the same scope.

```
node a;
int a;
⇒ [error] Reinstantiation of ‘a’
```

An instantiation can be accompanied by a single *initializer* (see Section 3.1 [Simple Connections], page 17) which initializes the value of a variable.

```
int a=5, c=8;
float b=8.9;
```

The order of initialization of variables is unspecified; therefore, using constructs such as

```
int a=5, b=a;
⇒ [error] Identifier ‘a’ not instantiated
```

should be avoided, and the behavior of such a construct is unspecified. In the current implementation, the result would be the error shown above, indicating that CAST does not know about variable `a` in the initialization of `b`.

2.2 Array types

An array of a basic type or user-defined type can be created using CAST's array syntax. The types

```
int[4]
float[7]
bool[1..6]
```

are all examples of arrayed types. The number in square brackets specifies the range of the array. In the first two examples, valid array indices range from zero to three and zero to six respectively. The third example specifies the array indices to range from one to six. In general, if the array index range is specified by a single integer, the lower bound of the range is zero, and the upper bound is the specified integer minus one. Instead of simple integers, arbitrary integer expressions can also be used as array range specifications, as shown below.

```
int[5*3]
float[7*x+(y%2)-p]
```

Multidimensional arrays are specified by listing the ranges separated by commas within square brackets. Two and three-dimensional arrays of nodes are specified as shown in the example below.

```
node[5,3]
node[1..6,9,2..10]
```

Note that the range specification must be of integer type. For example, if y were a real-valued variable, then the following type would be invalid.

```
float[7*y] x;
⇒ error  In ARRAY-SPEC, array range expression is not of type INT
```

Variables of array types can be created by specifying the type followed by a list of identifiers as shown below.

```
node[4] a,b,c;
node[10..20] m;
node[5..10,8..10] p;
```

CAST provides a mechanism for constructing *sparse arrays*, i.e., those whose range need not be a single contiguous block. It is possible to create an array of nodes whose elements exist only at, say, positions 4 and 5 of the array. The syntax for creating the aforementioned array is shown below.

```
node n[4],n[5];
```

The fact that the array index occurs as part of the variable name and not part of the type specifies that the instantiation is a sparse array construction. These sparse array instantiations can be mixed with ordinary instantiations, permitting the definition of arrays which can be dynamically extended in CAST.

```
node n[5];
node[10..12] n; // n is now defined at positions 5, 10, 11, 12
```

The sparse array syntax also supports the definition of more than one element per instantiation. The definition below specifies an instantiation of elements of array `m` at positions `6,5`, `6,6`, ..., `6,10`.

```
node m[6,5..10]
```

Note that this is quite different from the statement

```
node[6,5..10] m;
```

which indicates that array `m` is to be instantiated at positions `0,5`, ..., `5,10`.

Array instantiations can also be followed by initializers (see Section 3.2 [Array Connections], page 18). Some examples are shown below.

```
node[10] n, m; n=m; // successful
node[11..20] m=n;   // unsuccessful
⇒ error    Attempt to connect two arrays ‘m’ and ‘n’ of
                     incompatible size (20 v/s 10)
```

The reason the second example fails is that after creating elements 11 through 20 of array `m`, the connection `m=n` is attempted. This connection operation fails because arrays `m` and `n` have differing sizes (see Section 3.2 [Array Connections], page 18). However, the following construction can be used for such cases:

```
node[10] n, m; n=m;
node m[11..20]=n; // successful
```

Initializers are short-hand for real connection statements. The example that failed is equivalent to

```
node[10] n, m; n=m;
node[11..20] m;
m=n;              // fails!
```

The example that was successful is equivalent to

```
node[10] n, m; n=m;
node m[11..20];
m[11..20]=n;   // successful
```

2.3 User-defined types

User-defined type can be used to create complex circuit structures. A new user-defined type name is introduced by using the `define` statement. The `define` keyword must be followed by a valid identifier name and two parameter lists enclosed in parentheses, as shown in the example below.

```
define test(int N) (node n)
{
...
}
```

The first parameter list can only contain meta-language types, and the second parameter list cannot contain any meta-language type (see Section 2.1 [Basic types], page 9). The part of the definition enclosed in braces is said to be the *body* of the user-defined type.

If the body of the user-defined type is replaced by a single semi-colon, the statement corresponds to a type *declaration*. Declarations are typically used when defining mutually recursive types. The declaration corresponding to type `test` is

```
define test(int N) (node n);
```

Parameter lists have a syntax similar to instantiations. A type specifier can be followed by a list of identifiers. Semicolons are used to separate parameters of differing types, as shown in the example below.

```
define test2(int N,M; float p,q) (node a,b,c; node[10] d) { }
```

Square brackets can also be used following the identifier names. The meaning of these square brackets is identical to the ordinary sparse array instantiation (see Section 2.2 [Array types], page 10). The following is a different way to define type `test2`.

```
define test2(int N,M; float p,q) (node a,b,c, d[0..9]) { }
```

Since the names `N` and `n` are the parts of the type that are visible to the environment of a type, the names of the parameters in the declaration must match those in the definition.

```
define test(int N) (node n) { ... }
define test(int M) (node n);
⇒ error  Type mismatch in declaration of type ‘test’
```

```
define test(int M) (node n);
define test(int N) (node n) { ... }
⇒ [error]  Type mismatch in definition of type ‘test’
```

A type can only have one definition in a given scope.

```
define test(int N) (node n) { ... }
define test(int N) (node n) { ... }
⇒ [error]  Redefinition of of type ‘test’
```

Type names and variable names share the same name space. Creating a type definition with the same name as an instance variable or vice versa is erroneous.

```
int test;
define test(int N) (node n) { ... }
⇒ [error]   Name ‘test’ has been previously instantiated
```

2.3.1 Instantiations

User-defined type variables can be instantiated in much the same manner as ordinary type variables.

```
test x;
// x.N and x.n refer to the parameters of x
```

creates an instance of type `test` named `x`. Creating an instance of a type creates instances of all the parameters listed as well as creating whatever is specified by the body of the type definition. The list of parameters of a user-defined type can be accessed from the scope outside the type definition by using dot-notation. These externally visible parameters are analogous to the *fields* of structures or record types.

2.3.2 Nested definitions

Like well-scoped languages such as Scheme, and unlike languages such as C, CAST supports nested definitions. This provides a very nice mechanism for modular descriptions of circuits. Suppose one wanted to describe a circuit element with an input and output channel of type `chan`, but which internally uses a number of different types. The proper way to define such a circuit element is as follows:

```
define circuit() (chan in, out)
{
   define buffer() (chan L, R)  { ... }
   define foo()  (...) { ... }
   ...
   buffer b;
   ...
}
```

The names `buffer` and `foo` are local to the definition of type `circuit`, and therefore can be used within the body without fear of being confused with other definitions bearing the same name. If the same names have been declared or defined before `circuit` is encountered, CAST reports a warning.

```
define buffer() (node x0,x1,xe,y0,y1,ye) { ... }
define circuit() (chan in, out)
{
   define buffer() (chan L, R) { ... }
   ...
}
⇒ warning  Name ‘buffer’ shadows a previous declaration
```

Another part of the circuit can use the `buffer` defined within the body of `circuit` by referring to it using the dot-notation as follows to resolve the ambiguity.

```
circuit.buffer b;
```

2.4 Parameterized types

Parameterized types give CAST considerable flexibility in type definitions. CAST guarantees that the order in which parameters are created and initialized is from left to right. Therefore, one can use the value of one parameter in the definition of another one!

```
define test(int N) (node[N] n) { ... }

test(5) x;  // creates test with N set to 5
test(10) y;  // creates test with N set to 10
```

As shown in the example above, the arguments in the first parameter list can be specified by listing them next to the type name (see Section 3.3 [User-defined Connections], page 19). Trailing arguments can be omitted from the parameter list attached to the type as shown in the example below.

```
define test(int N; float[N] w) (node[N] n) { ... }

test(5) x;
```

3 Connections

When two nodes are connected to each other by wires, they effectively become one electrical node. This connection operation is part of the CAST language, and is denoted by the = sign. The = operation is also overloaded for meta-language variables to denote assignment. Multiple connections can be specified in a single statement by repeatedly using the = operator. This section describes the different connection statements supported by CAST.

3.1 Simple Connections

The simplest possible connection statement is the connection of two variables of type `node`.

```
node x, y;
x=y;
```

The effect of this operation is to alias the two nodes. After this operation is performed, both `x` and `y` refer to the same value. Meta-language types can also be "connected" to expressions. The result of such a "connection" is that the right hand side of the = sign is evaluated, and assigned to the variable on the left. Such connections are only meant to initialize the values of parameters.

```
int x, y;
x=5;
y=x*1+2;          // success
```

Whereas connecting nodes is a symmetric operation, connecting meta-language variables is not symmetric, as illustrated below.

```
int x, y;
x=5;
x=y*1+2;
⇒ error  Uninitialized parameter 'y'
```

Meta-language parameter connections correspond to assignment statements. CAST permits assigning floating-point values to integer-valued variables, and vice versa. However, a meta-language variable can only be assigned one value.

```
int x;
x=5;
x=8;
```

```
⇒ $\boxed{\text{error}}$ Reinitializing ‘x’ with different value
```

3.2 Array and Subrange Connections

Array connections in CAST are extremely flexible. In general, if two arrays have the same basic type and have the same number of elements, they can be connected with a simple connect directive. In the example below, nodes `x[0]`, ..., `x[9]` are connected to nodes `y[10]`, ..., `y[19]` respectively.

```
node[10] x;
node y[10..19];
x=y;
```

Connecting two arrays of differing sizes is an error.

```
node[10] x;
node y[10..20];
x=y;
⇒ $\boxed{\text{error}}$ Attempt to connect two arrays ‘x’ and ‘y’ of incompatible
                 size (10 v/s 11)
```

CAST provides a *subrange* mechanism for connecting parts of arrays to one another. The example below shows a connection between elements `x[3]`, ..., `x[7]` to `y[12]`, ..., `y[16]`.

```
x[3..7] = y[12..16];
```

Connections between two arrays with differing numbers of dimensions is also permitted. When a connection between multidimensional arrays is specified, the ranges to be connected are sorted in lexicographic order (with indices closer to the variable having higher weight) and the corresponding array elements are connected. For example, in the example below `x[1]` would be connected to `y[0,1]`.

```
node[4] x;
node[2,2] y;
x = y;
```

When combined with the subrange mechanism, these array connections can be used in complex ways. Connecting rows and columns of a two-dimensional array to a one-dimensional one is equally simple.

```
node[4] row,col;
```

```
node[4,4] y;
row=y[1,0..3];   // connect row
col=y[0..3,1];   // connect column
```

A list of variables can be treated as a single array by enclosing it in braces.

```
node[3] x;
node 0,1,2;
x = {0,1,2};
```

3.3 User-defined Type Connections

The result of connecting two user-defined types is to connect all the variables listed in their parameter lists. A connection is only permitted if the two types match.

When instantiating a variable of a user-defined type, the variables in the parameter list can be connected to other variables by using a mechanism akin to parameter passing.

```
define dualrail() (node 0,1,a)
{
  spec {
    exclhi(0,1)  // exclusive high directive
  }
}

node d0,d1,da;
dualrail c(d0,d1,da);
```

In the example above, nodes `d0`, `d1`, and `da` are connected to `c.0`, `c.1`, and `c.a` respectively. Nodes can be omitted from the connection list and trailing commas can be omitted as well. The following statement connects `d1` to `c.1` after instantiating `c`.

```
dualrail c(,d1);
```

Since connections can be specified by parameter-passing, it is useful to permit anonymous instantiations, as shown below.

```
dualrail() (d0,d1,da);
```

Here, the name of the instance variable is omitted, but we can still refer to parts of the `dualrail` variable because we have access to all its parameters.

Since parameter passing is treated as a connection, all the varied connection mechanisms are supported in parameter lists as well.

4 Control Flow

Complex datapath elements are usually comprised of arrayed versions of simpler cells. Although arrays can be created directly using arrayed instantiations, CAST supports looping constructs which can be a convenient way to create arrayed structures. More complex structures such as trees can be easily created using recursive instantiations.

4.1 Loops

An example of the loop construct in CAST is shown below:

```
< i : 10 : node x[i]; >
```

The variable `i` is the loop index and its scope is delimited by the angle brackets. The colons separate the different parts of the loop construct. The number `10` is an abbreviation for the range `0..9`. The body of the loop is the statement `node x[i];`. The effect of this statement is the same as

```
node x[0];
node x[1];
...
node x[9];
```

The body of the loop can contain any CAST body, and therefore can have multiple statements in it. A more common use of the loop statement is shown below:

```
<i : 1..8 : register r[i](in[i],out[i],control); >
```

In the example above, registers numbered `1` through `8` are created. Their first two parameters are connected to the corresponding elements of arrays `in` and `out`. However, they have a shared control that is passed in as the third parameter.

Since loops are part of CAST bodies, they can occur in the body of another loop. Thus, nested loops are also supported. However, types cannot be defined in the body of a loop.

4.2 Selections

Conditional execution is supported by the selection statement. The syntax of a selection statement is:

```
[ boolean_expression -> body ]
```

The specified body is executed if the boolean expression is true, and is skipped otherwise. For instance, we can create 32 registers with something special for register 0 as follows:

```
<i : 32 :
    [ i == 0 -> zeroreg r0(in[i],out[i],control); ]
    [ i != 0 -> reg r[i](in[i],out[i],control); ]
>
```

Boolean expressions can be constructed from boolean variables, the constants `true` and `false`, and the boolean operators `&`, `|`, and `~` denoting the and, or, and negation operations respectively. Numeric expressions can be compared using `<`, `<=`, `>`, `>=`, `==`, and `!=` for the operators less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to respectively.

4.3 Recursion

Type definitions can be recursive. For instance, the following definition can be used to create a tree structure.

```
define tree(int N) (node[N] a)
{
  [ N == 1 -> leaf l(a[0]); ]
  [ N > 1  -> tree(N/2) t0(a[0..N/2-1]);
              tree(N-N/2) t1(a[N/2..N-1]);
  ]
}
```