Gedae Idea Language Reference Manual

September 2013

Address:	Gedae, Inc. 1247 N Church St, STE 5 Moorestown, NJ 08057
Telephone:	(856) 231-4458
FAX:	(856) 231-1403
Internet:	www.gedae.com

Copyright

Table of Contents

Algebraic Expressions

Variables can be declared with an expression that sets their value.

```
C = 2 * PI * R;
```

Data Types

The following data types are supported: float, double, complex, and dcomplex (double complex), char, short, int, INT (32 bit on 32 bit OS, 64 on 64), dint (64 bit), unsigned char, unsigned short, unsigned int, unsigned INT, unsigned dint.

When declaring a variable, the data type can be declared.

When declaring a variable, the data type can be left off and inferred from the expression that sets it.

If a variable is typed, the result of the expression is cast to the data type.

```
int A = 1;
float B = A;
```

Ranges

Ranges are positive finite values.

```
range i = 10;
```

All ranges can be used in expressions.

Dimensions

There are three dimension types

- Families – preindices using square braces, for example, [f]x
- Arrays – postindices using square braces, for example, x[r][c]
- Time – postindices using parentheses, for example, x(t)

On the LHS of equations, dimensions can be expressed only as simple expressions (that is, single range variables).

If a variable is declared without the stream modifier or a time index and its expression does not include any streams, then it is a parameter. Parameters do not have a rate and cannot be dereferenced by a time index. Parameters can be declared without an expression to set their value.

```
x = 1;          // parameter
int y;          // unset parameter
z = x + y;      // parameter
```

If a variable is declared with the stream modifier or a time index, then it is a stream. If a variable's expression includes a stream, then it is a stream. All streams are infinite. All streams must be declared with an expression to set their value.

```
stream x = 1; // stream
y = x;        // stream
```

On the RHS of equations, dimensions can be any expression.

Operators

All C binary operators are supported.

The C unary operators ! and ~ are supported.

The C tertiary operator ?: is supported.

The # operator is a unary operator that returns the size of the range.

The ` operator is a binary operator for expressing powers.

A collapsing operator performs an operator across a dimension.

```
Sum += M[i][j];               // matrix sum
Dot += X[i] * Y[i];           // dot product
Prod += M[i][j] * N[j][k];    // matrix multiply
Dil[i][j] ||= Im[i+i1][j+j1]; // dilution image kernel
```

The dimension(s) of a collapsing operator are all ranges that appear in the RHS of the expression that do not appear on the LHS. If an index appears on the RHS and not the LHS, then the assignment operator must be a collapsing operator.

```
range i = 10;
a[i] = i;
x = a[i]; // illegal
```

The following collapsing operators are provided.

Function	Description	Initialization
+=	Increment and assign	0
-=	Decrement and assign	0
*=	Multiply and assign	1
/=	Divide and assign	1
&=	Bitwise AND and assign	0xFFFFFFFF
^=	Bitwise exclusive OR and assign	0
|=	Bitwise inclusive OR and assign	0
&&=	AND and assign	1
||=	Inclusive OR and assign	0
^^=	Exclusive OR and assign	0
>?=	Maximum and assign	Lowest negative number
<?=	Minimum and assign	Highest positive number

Functions

Expressions can be grouped together into functions. Functions can have any number of outputs and any number of inputs, including zero. Function outputs are not typed. The type is determined from the expressions that set the output.

```
Y foo(float X) {
  Y = (X+1)/2; // implicitly declared as float
}
```

Invocation

Functions can be invoked in expressions.

```
X = 2;
Y = foo(1);
Z = Y + foo(X);
```

All inputs from the function's argument list must be specified in the invocation.

If a multiple output function with N outputs is used, then up to N variables can be listed on the LHS. If M<N variables are listed, then the first M return values are used. When a multiple output function is used in an expression, only the first output is used.

```
Y,Z bar(float X) {
  Y = (X+1)/2;
  Z = (X-1)/2;
}
...
Y = bar(1)+1;      // sets Y to 2
Y2, Z2 = bar(Y);   // sets Y2 to 1 and Z2 to 0
```

Overloading

Functions can be used for both parameters and streams.

```
Y foo(float X) {
  Y = (X+1)/2;
}
...
int x = 2;
p = foo(x); // parameter
stream float y = x;
s = foo(y); // stream
```

Implementation

Functions can be implemented by expressions or by kernels. Kernels use the Equation Field to define the expression or function that they implement. If a kernel equation defines a function, then the family and token dimensions of the outputs must be specified.

```
Name: mf_qr
Equation: q[i][j], r[j][k] = qr(in);
```

Data Structures

Data structures allow tokens to be organized into a single container and referenced by field name.

Definition

The C struct syntax is used to define a structure.

```
struct cart {
  float x;
  float y;
  float z;
}
```

The data type of each element is defined in the structure definition. Fields cannot be arrays, but arrays of structures can be created.

Constructor

A data structure is constructed by providing an array of values.

```
float x = 2.9;
cart pnt = {x, 3.0, -1.3}; // parameter
```

Expressions can be used inside the constructor, but the constructor cannot be used inside expressions.

Structures can contain all streams or all parameters.

Reference

The C . (period) infix operator is used in expressions to reference the fields in a structure.

```
out cartDistance(cart in) {
  out = sqrt(in.x`2 + in.y`2 + in.z`2);
}
```

Dynamic Ranges

Dimensions are defined using ranges. Ranges can be a fixed length by setting their value to a constant parameter. When arrays are declared using those ranges, the arrays have a fixed size defined at compile time. A dynamic range is created by setting the range to a stream or nonconstant parameter. Variable-sized arrays are created by using constraints on dynamic ranges. These constraints define the maximum size of the arrays at compile time allowing the actual size to be specified at runtime.

Variable Arrays

Constraints can be used to define the limits of a range. If a constrained range is used in the dimension of an array, then the array is a variable array. Array dimensions must be constrained to have a maximum less than infinity.

```
t = t(-1) + 1;
N = (t%6)+5;             // 5 to 10
R = 1<<N;
range r = constrain(R,1024); // dynamic range, max 1024
complex x[r] = { (r+3.0)/2.0, r/2.0 };
z[r] = fft(x[r]);
```

Data Flow

The introduction of time into a programming language offers a new method of program structure and introduces issues with data flow that must be considered.

Static Rates

The relationship between rates of different streams is determined by their use in expressions. Expressions can be written with relative time indices, stating the number of tokens relative to the current time. When using relative time indices, decreases in rate can be accomplished by referencing multiple tokens at or ahead of the 0 index. Delay and overlap can be accomplished by referencing negative time indices.

```
range i = 10;
a = a(-1)+1;
b2 = (a + a(1))/2;              // decimation by 2 by averaging
c = b(-1);                      // delay by 1
c2 = b + b(-1);                 // overlap of size 1
d(i) = c;                       // hold for 10 samples
```

The combination of static rates must follow data flow rules. Multiple static rates on the same stream cannot be used in the same construct. The verification of static rates is done by the data flow graph (DFG) compiler.

Tiling

Relative time indices can also be used to specify tiling operations and the interpolation and decimation needed to perform this token decomposition. The interpolation from creating tiles from a matrix can be expressed using arithmetic on smaller ranges. The decimation from reconstituting a larger array from tiles can be expressed using the % and / operators on the ranges. Alternatively, the comma notation can be used.

```
range i = 10;                       // matrix size
range j = 10;
range i1 = 5;                       // tile size
range i2 = 2;
stream a[i][j] = i+10*j;
b[i2][j](i1) = a[i1*#i2+i2][j];   // create tiles
c[i2][j] = foo(b[i2][j]);
d[i][j] = c[i/#i1][j](i%#i1);     // reform matrix
```

Dynamic Rates

Dynamic ranges can create dynamic rates. The combination of dynamic rates must follow data flow rules. Streams of different rates cannot be used in the same expression. The verification of dynamic rates is done by the DFG compiler.

```
range r = 10;
a[r] = r;
b(r) = a[r];
d[r] = b + a[r];    // illegal combination of rates
```

Conditionals

Conditionals can be used to set values based on a parameter or stream value. They can also be used to reduce data rate.

If Statement

The C syntax is used for an if statement.

An if statement can introduce a dynamic stream.

```
Threshold = 0;
```

```
if (a > Threshold)
  b = a; // b is a slower rate than a
```

An if statement with else if and/or else clauses can be used to perform a select.

```
Threshold = 0;
if (a > Threshold)
  b = a;
else
  b = Threshold; // b is at the same rate as a
```

An if statement with else if and/or else clauses can be used to perform branching,

```
if (c == 0)
  x = foo(a);
else
  y = bar(a);
```

A complete branch and merge where for each input token, a token is produced on the output stream,

```
if (c == 0)
  b = foo(a);
else
  b = bar(a); // b is at the same rate as a
```

And an incomplete branch and merge where some branches do not produce tokens on the stream,

```
if (c == 0)
  b = foo(a);
else if (c == 1)
  b = bar(a); // b is not produced if c is not 0 or 1
```

Note a is consumed regardless of the condition. A token must be available on a regardless of the value of c, and that token is consumed regardless of the value of c.

Switch Statement

The C syntax for a switch statement is used with the change that each case statement is assumed to end in a break. Any overlap in case statements can be implemented by duplicating code in each case. As in C, the switch statement does not require a default clause, and thus it can introduce dynamic streams.

```
switch (c) {
  case 0:  e=c+1; d = a;
  case 1:  e=c+1; d = b;
  default: d = 0;
}
```

Iteration

Loops are used to iterate on tokens, i.e., all tokens are held during the duration of the loop. Loops must iterate on at least one token.

A for, while, or do-while loop specifies iteration. For for loops, the C syntax is used. For while and do-while loops, the syntax is very similar to the C syntax.

For Loop

Variables created by the for loop, called iteration variables, are initialized in the initialization section of the for construct. If these iteration variables are used on the RHS of expressions in the body of the loop, the variable refers to the current value of the token, not the new value computed during the current iteration.

```
for (c=1.0, k=0; k<10; k=k+1) {
  c = c * exp(-1); // iteration on held token
}
```

The new value of the token can be reused inside the body of the loop by using an intermediate variable inside the loop. Intermediate variables are variables defined inside the body of the loop.

```
for (c=1.0, d=1.0, k=0; k<10; k=k+1) {
  temp = c * exp(-1); // intermediate variable for “new” c
  c = temp;
  d = d + temp;
}
```

While Loop

While loops and do-while loops are also supported, but their syntax differs from the C syntax by adding a section to initialize iteration variables. This initialization section makes the syntax very similar to for loops.

```
while (c=1.0; c>0.1) {
  c = c * exp(-1); // iteration on held token
}
do (d=c) {
  d = d * exp(1);
} while (d<1.0);
```

State Reset

Conditionals and loops all include the resetting of state. Iteration constructs reset state at the beginning of the loop. Conditional constructs reset state when the branch changes.

The following example implements a conditional low pass filter. The low pass filter uses a delay to store the previous tokens. This state is reset whenever valid(-1) == 0 and valid == 1.

```
if (valid) {
```

```
  y = y(-1) + alpha * ( x – y(-1) );
}
```

State variables can be used in the data flow graph language to retain information across conditional or loop boundaries.

State Machines

State machines can be used to control the transitions between software modes.

Statedef Statement

A statedef defines the list of states in a state machine.

```
statedef { Search, Track } myState;
```

Statemachine Control Structure

A state machine is declared as a collection of states where the body of each state uses conditionals to define the transition to other states. The state machine creates a variable that defines the current state. The begin section is used to define the initial state of the machine.

```
statemachine (myState s) {
  begin  { s = Search; }
  Search { when (event0) s = Track; }
  Track  { when (event1) s = Search; }
}
```

Any code can be placed into the body of a state to define the behavior both when inside the state and on the transitions. The begin section can also be used inside each state to define the entry behavior such as the reset of a software mode.

```
statemachine (myState s) {
  begin  { s = Search; }
  Search {
    begin {
      resetSearch();
    }
    Targets[n],targetsFound = doSearch(data);
    if (targetsFound == 1) {
      endSearch();
      s = Track;
    }
  }
  Track  {
    begin {
      resetTrack();
    }
    doTrack(data,Targets[n]);
    when (event1) {
```

```
      endTrack();
      s = Search;
    }
  }
}
```

In the use of a when statement to specify a state transition, each branch must specify a destination state, and the transition conditions for a state must be fully contained in a single conditional. This rule prevents cases where events that cause transitions to be dropped. The only way to remain in the state is if no branch is taken.

```
when (event1) {        // illegal, possible drop of event2 if !c
  if (c) s = A;
} else when (event2) { // legal
  if (c) s = B;
  else   s = C;
} else when (event3) { // legal
  s = C;
}
```

In Idea conditionals, input data used in one branch of the conditional is consumed in all branches, regardless of whether it is referenced in that branch. State machines handle variables the same way; all data is consumed even if the variable is not referenced in the current state. In the Search and Track examples above, event0 and event1 are only referenced in one state each, but the data is consumed regardless. This draining of tokens can be circumvented by explicitly referencing the variable in the other state to retain the value in an output queue that is fed back to the other states.

```
statemachine (myState s) {
  begin  { s = Search; }
  Search {
    S_event1 = event1;
    when (event0 || T_event0) s = Track;
  }
  Track  {
    T_event0 = event0;
    when (event1 || S_event1) s = Search;
  }
}
```

Alternatively, the user can create their own error conditions or error state based on unexpected encountering tokens in a state.

```
statemachine (myState s) {
  begin  { s = Search; }
  Search {
    when (event0) s = Track;
    else when (event1) s = Error;
  }
  Track  {
    when (event1) s = Search;
```

```
    else when (event0) s = Error;
  }
  Error  { print("Illegal condition"); }
}
```

From Section

In addition to begin sections, from sections can be used to conditionally reset the software based on the previous state.

```
statedef { A, B, C } myState;

statemachine (myState s) {
  begin { s = A; }
  A     {
    when (event)       s = C;
    else when (event1) s = B;
  }
  B     {
    begin    { resetSoftware(); } // always reset at beginning
    when (event)       s = C;
  }
  C     {
    from (A) { resetSoftware(); } // don't need to reset from B
    when (event2)      s = A;
  }
}
```

Conditionals in from statements can only use the operators ! (not from the state) and || (from one of the states in the expression). From statements can have else and else-from branches.

End Section

Each state can have an end section to clean up the mode regardless of exit condition. Each state machine can also have an end section to cleanup resources created in other sections. Setting the state variable to end will cause the state machine to terminate and the end section to execute.

Hierarchical State Machines

Hierarchical state machines can be created by putting the code for a state machine inside the body of a state (or calling a function that contains a state machine). This hierarchy must be mirrored in the state definition.

```
statedef { TrackA, TrackB } subState;
statedef { Search, Track.subState } myState;
```

A state machine is self-contained; it can only transition between states in the current level of hierarchy. The child state machine is exited by transitioning to the end state or when the parent gets an event that causes a transition in the parent state machine.

```
s, done subStateMachine {
  statemachine (subState s) {
```

```
    begin  { s = TrackA; } // start Track in TrackA
    TrackA { when (eventA) s = TrackB; }
    TrackB { when (eventB) s = end; }
    end    { done = 1; }   // pass token to parent when done
  }
}
main() {
  event0, eventA, eventB, eventX = gui();
  statemachine (myState s) {
    begin  { s = Search; }
    Search {
      when (event0)      s = Track;
    }
    Track  {
      s_sub, done = subStateMachine(eventA,eventB);
      when (done)        s = Search; // go to Search when done
      else when (eventX) s = end;    // exit regardless of substate
    }
  }
}
```

Concurrent State Machines and Sequence Diagrams

Multiple (disjoint) state machines can operate concurrently. As Idea code is not explicitly sequenced, all state machines specified in the same application are concurrent. Concurrent state machines can interact with each other through the production of data by one state machine and consumption by another.

Consider an example of two two-state state machines. A token is produced by each state machine which tells the other state machine to change state. The A state machine leaves the Bar state when B is done with the Baz state, and the B state machine leaves the Qux state when A is done with the Foo state.

```
statedef { Foo, Bar } Astates;
statedef { Baz, Qux } Bstates;

main() {
  statemachine (Astates A) {
    begin { A = Foo; }
    Foo {
      doneA = doFoo();
      when (doneA) A = Bar;
    }
    Bar {
      doBar();
      when (doneB) A = Foo;
    }
  }
  statemachine (Bstates B) {
    begin { B = Qux; }
    Qux {
      doQux();
```

```
      when (doneA) B = Baz;
    }
    Baz {
      doneB = doBaz();
      when (doneB) B = Qux;
    }
  }
}
```

Concurrent state machines can also be used to specify sequence diagrams. A sequence diagram is a set of items with timelines that describe the behavior of the items and the interactions between them. In Idea, each item is a state machine, the timeline is the sequence of states in the state machine, and the interactions (or messages between the items) are data produced by one state machine and consumed by another. In the example above, A has the timeline of doing Foo and then Bar, and B has the timeline of doing Qux and then Baz, with the production of doneA and doneB representing the interactions between A and B.

Timing events can also be used to invoke these interactions between sequence diagrams (or any other data messages passed between state machines). The built-in function wait() takes the specified number of seconds before producing a token. The built-in function wallclock() returns the time in seconds and can be used to calculate elapsed time. The B state machine from the above example is modified to do nothing in the Baz state except wait for a timer to expire.

```
statemachine (Bstates B) {
  begin { B = Qux; }
  Qux {
    doQux();
    when (doneA) B = Baz;
  }
  Baz {
    doneB = wait(0.1);
    when (doneB) B = Qux;
  }
}
```