Hava Language Technical Reference

April 25, 2009 *(draft)*
Steven T. Hackman, Loren K. Platzman
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Hava is a numerical calculator that evaluates collections of interrelated, recursively defined, mathematical rules. Like a spreadsheet, it can define a value in terms of neighboring values. Unlike a spreadsheet, it exposes definitions in a text-based source file, and it permits values to be arranged more generally than a rectangular grid.

Hava source format

A Hava program is a collection of *statements*. Each statement may extend over any number of lines, and ends in a mandatory semicolon. Hava supports four kinds of statements: import, variable definition, token definition, and structure definition.

The sequence in which statements appear in the source determines the sequence in which results will appear in the report, but it does not affect what values are ultimately obtained. In particular, it is *not* necessary to place an identifier definition ahead of statements that make use of that identifier.

Comments

Hava ignores whitespace and comments of the following form:

```
// This comment continues to the end of the line

/* This comment might
     occupy many lines */
```

A *documentation comment* takes the form `/**…*/` and is reproduced in the report. (Spacing, including tabs, is retained.) An empty documentation comment `/** */` generates a blank line in the report. Unlike other comments, which may appear anywhere in the source, a documentation comment may appear only between statements.

Import statement

The import statement merges one or more files into the Hava source.

```
import sample1, c:\hava\sample2;
```

More on imports at the end of this document.

Variable definition

A *variable definition* associates an identifier (the *variable name)* with a rule to compute its value(s). A *simple variable* represents a single value.

```
pi = 3.1415;
```

An *indexed variable* represents any number of values, classified by indexes:

```
f(i,j) = i+j;
```

An indexed variable may look like a function, but it actually represents a collection of stored values. Each value will be computed only if it is invoked, and will be retained if it is computed, so that it does not have to be computed a second time. In this sense, an indexed variable resembles an area of a spreadsheet where the cells share a single formula, and each indexed value resembles a cell in that area.

Variations

A variable definition prefixed by the keyword `function` will be computed as needed, without saving values for reuse. This reduces memory use at the expense of runtime. A variable definition prefixed by the keyword `private` will be omitted from the report. Additional prefixes are `table` (see List Values) and `final` (see More on Imports).

Token definition

A *token* is a unique ordinal value:

```
token RED, BLUE, GREEN;
```

It can be assigned to a variable, like this:

```
color = RED;
```

Structure definition

A *structure* permits many values to be gathered as one.

```
struct Point(x, y), State(inventory, price);
```

A structure value is created by invoking the structure name as a constructor:

```
s = State(1000, 2.17);
```

The dot operator is used to reference a structure's fields by name:

```
x = s.inventory;
```

Identifiers

An identifier is a name that is given meaning by a Hava statement. It may contain letters, digits and underscores, but may not start with a digit. Identifiers are case-sensitive.

Valid identifiers: `x, _Very_Long_Identifier, H2O`

Invalid identifiers: `007, James Bond, Yahoo!`

The following names are reserved by Hava and cannot be used as identifiers:

keywords: `else, final, function, if, import, in, struct, table, to, token.`

literals: `true, false`

tokens: `BLANK, BOOLEAN, ERROR, IGNORE, INTEGER, LIST, REAL, REPLICATING, STRUCTURE, TOKEN.`

functions: `ceiling, exp, floor, ln, random, round, sqrt.`

iterators: `argmax, argmin, collect, first, for, join, last, max, min, sum.`

fields: `listSize, structSize, structType, valueType.`

Naming conventions

The following naming conventions are highly recommended, but are not formally required:

upper case for tokens (`CONTINUE`),

upper leading camel case for structures (`MyState`) , and

lower leading camel case for local variables and fields (`myVariable`).

There is no naming convention for global variables and functions.

Definition scope

Identifiers defined in statements have *global scope*. That is, their definition is recognized throughout the program. Indexes and function arguments, on the other hand, have *statement scope*. That is, their definition is recognized only in the statement where they appear. Thus, it is legal to define indexed variables `f(i)` and `g(i)`, since each `i` is recognized in a separate statement. It is legal to define variables `i` and `f(i)`, but the variable `i` will not be accessible when evaluating `f(·)`.

A single identifier may be associated with more than one variable or function, provided each has a different number of indexes. Thus, it is legal to define `f(i)` and `f(i, j)`, and also `f`.

An identifier associated with a token or structure may be defined in only one way, but the definition may be repeated any number of times, without generating an error.

Value

The values assigned to variables may be of various types. The *primitive types* are *integer* (`42`), *boolean* (`true`), token (CONTINUE), and *real* (`3.14159`). The *compound types* are *list* and *structure*.

The type of any particular value depends on how it was obtained. For example, `2+2` is of type integer, but sqrt(`2`) is of type real. Mismatched types will result in a run-time error. For example, the three statements

```
a=42; b=true; c=a+b;
```

cause a run-time error.

The type of any value may be determined from the following built-in fields:

`x.valueType`	A token representing the type of this value.
`x.listSize`	The number of elements (if a list).
`x.structType`	The structure name (if a structure).
`x.structSize`	The number of fields (if a structure).

Literal values

A *literal* is a value given explicitly in the code. The booleans literals are `true` and `false`. The empty list literal is `()`. Integer literals are groups of digits without a decimal point. Real literals also include a decimal point, and/or a power of 10, like this:

```
1.23e-2
```

As a consequence of improper arithmetic, the values NAN (not a number) or INFINITY might be created. These are real values, not tokens. However, they cannot appear in Hava source as literals.

List values

A list is a compound value whose elements are enumerated or referenced by position. A list is created by enclosing its members in parentheses:

```
list = (1, 2, 3);
```

One exception to this form: (x) is simply x, not a set containing only x. To create a singleton, use collect (see below). However, ( ) does represent the empty list.
The elements of a list may themselves be lists. For example,

```
a = ((1, 2, 3), (4, 5, 6), (7, 8, 9));
```

is a list of lists that acts as a matrix.

Structure values

A structure value is created by following the structure name with a list of field values. For the structure defined as `State(inventory, price)`, we might construct a value `State(1000, 2.15)`.

The list may contain fewer values than are defined for that structure: `State(1000)`, or `State()` or, equivalently, `State`. This is not an error unless/until a missing field is referenced. Note that the structure name used without arguments is treated as a token. This token is also the value stored in the field `structType` of any structure value.

Table format

The prefix `table` in front of a variable definition causes its value(s) to be displayed in the solution as a table, suitable for copying to a spreadsheet. If the value(s) cannot be organized into a table, then the prefix `table` has no effect.

For example, suppose that the simple variable `x` has as its value a list containing the integers 1, 2 and 3. The default appearance in the solution is:

```
x     (1, 2, 3)
```

In table format, the elements are listed on separate rows:

```
x
      1
      2
      3
```

If you wish the elements to appear in a row, define x to be a list containing the single list (1, 2, 3):

```
table x = collect((1, 2, 3));
```

Now the solution will appear like this:

```
x
      1     2     3
```

Table format also recognizes and organizes nested structures (values of type structure whose fields may also be structures), lists of structures, and structures of lists.

Granulation

In large problems, it may be convenient to reduce the accuracy of indexes so that fewer indexed values need to be stored. Any index (of an index variable) or field (of a structure) may be designated as *granulated* by providing a *grain* specification in its declaration. Values assigned to a granulated index or field are rounded to the nearest multiple of the grain.

```
f(x, y:0.1) = x + y;
a = f(1, 2.345);          // f (1, 2.3) evaluates to 3.3
```

The value of a variable may likewise be rounded to the nearest multiple of a grain, by placing a grain specification at the end of the assignment statement:

```
f(x) = sqrt(x):0.1;
a = f(2);                 // f(2) evaluates to 1.4
```

Indeed, any numerical expression can be granulated:

```
f(x) = 1/(sqrt(x):0.1);
a = f(2);                 // f(2) evaluates to 0.714286.
```

Expressions

Operators

Hava supports arithmetic operations,

```
x = a+b;          // Addition
x = a-b;          // Subtraction
x = -a;           // Negation
x = a*b;          // Multiplication
x = a/b;          // Division
x = a^b;          // Raise to a power
```

relational operations,

```
b = x<y;          // x less than y?
b = x<=y;         // x less than or equal to y?
b = x>y;          // x greater than y?
b = x>=y;         // x greater than or equal to y?
b = x!=y;         // x unequal to y?
b = x==y;         // x equal to y? (note double equal sign!)logical operations,

b = b1 && b2;     // Logical and
b = b1 || b2;     // Logical or
b = !b1;          // Logical not
```

arithmetic functions,

```
y = sqrt(x);      // Square root
y = exp(x);       // Exponential function
y = ln(x);        // Natural logarithm
i = round(x);     // Round to nearest integer
i = floor(x);     // Round down
i = ceiling(x);// Round up
i = x:g;          // Round to nearest multiple of g
```

list accessors,

```
x = a[i];         // The i-th element of a.
x = a[i,j];       // The j-th element of the i-th list in list a.
b = i in a;       // true if i is an element of a, false otherwise.
j = i:a;          // The position where i first occurs in a.
```

and list generators:

```
s = a to b;       // The list of integers from a to b.
s = 2^a;          // The set of sublists of a, each in the same sequence as in a.
                  // 2^a includes the empty list as well as a itself.
```

Relational operators can be chained:

```
if  (0 < x < N) ...
```

Operator precedence

The following list shows operators, sequenced from highest to lowest precedence. Operators having the same precedence are evaluated from left to right.

Extraction	`., []`
Power	`^`
Unary	`!, -`
Multiplication	`*, /`
Addition	`+, -`
Granulate or index-in	`:`
Iterator index assignment	*identifier* = ...
Relational	`<, <=, >, >=, ==, !=, in, to`
Logical and	&&
Logical or	`||`
Filter	`|`

Random variables

The built-in function random returns a sample random value. It can be used in four ways:

1. When called without an argument, it returns a random real number, uniformly distributed between zero and one:

   ```
   r = random;
   ```

2. When called with an integer argument, say n, it returns a random integer equally likely to take any value from 1 to n (inclusive):

   ```
   r = random(12);
   ```

3. When called with an argument that is a list of structures, each of which contains a field called prob, it returns a random element in the list, each element being chosen according to its given probability. (If the probabilities in the list sum to less than one, then the value IGNORE will be generated with the remaining probability. If the probabilities in the list sum to more than one, then the later items will never be generated.)

   ```
   struct Event(prob, val);
   pmf = (Event(0.4, 13), Event(0.6, 42));
   r = random(pmf);
   ```

4. When called with an argument that is any other kind of list, it returns a random element in the list, each element being equally likely to be selected.

If REPLICATING is defined to be a token, then `random` will return the same sequence of values each time a program is executed.

Conditional expressions

A *conditional expression* takes the form:

```
if (boolean) {expression} else {expression}
```

Conditional expressions may be chained:

```
if (boolean) {expression}
else if (boolean) {expression}
else {expression}
```

Iterated expressions

An *iterated expression* takes the general form:

iterator-name (*iterator-directive*) {*expression*}

The *iterator name* may be one of the following: `argmax, argmin, collect, join, first, for, last, max, min` or `sum`. Alternatively, any iterator name may be used as a function:

```
x = max(a, b, c);
```

The *iterator directive* takes one of the following forms:

```
i=a to b   or
i in a     or simply
i=a
```

Iterator directives may be chained (except when the iterator name is `argmax` or `argmin`):

```
i in a, j in b...
```

Each iterator directive may also be filtered:

```
(i in a | i < 3)
```

Here is a more elaborate example:

```
(i in a | i<4, j=1 to i | j<i, k=i+j)
```

Iterator variables have *local scope*. That is, they are meaningful only in subsequent chained iterator directives, and in the iterator's managed expression.

Some iterator names (`max, min, argmax, argmin, sum`) are familiar and require no further description. The others gather the iterated values into a list. All ignore prospective elements that equal the token value `IGNORE`. If an iterated value is itself a list, `join` will break it up and gather its elements separately. Thus, if

```
a = (1, 2, 3);
b = (4, 5, 6);
```

then the following values will be computed:

```
c = collect(a, b);   // c will be assigned the value ((1, 2, 3), (4, 5, 6))
d = join(a, b);      // d will be assigned the value (1, 2, 3, 4, 5, 6)
```

The iterator `first` returns the first value it encounters. The iterator `last` returns the last. The iterator `for` returns the single value in the list (which is also first and last) if the list contains exactly one value, or `ERROR` otherwise.

When no iterated values are found (the generated list is empty), `max, min, argmax, argmin` and `first` and `last` return `IGNORE`, `sum` returns 0 (zero), and `collect` and `join` return the empty list, `()`.

The iterator `sum` can accept lists as summands:

```
u = sum(i=1 to 1000, r=random) {(1, r, r^2)};
sampleSize = u[1];
sampleMean = u[2] / u[1];
sampleVariance = (u[3]-u[2]^2/u[1]) / (u[1]-1);
```

In addition, `sum` treats booleans as indicator functions, that is, `true` is treated as `1`, and `false` is treated as `0`.

Note that the keyword `in` can be used to mean either "*is i in a*?" or "*for all i in a*". Thus one might encounter:

```
collect(i in a | i in b) {i}
```

This is perfectly correct!

More on Imports

Hava files may be imported from the local drive or from a web address (URL). If an imported file address does not include a local drive or http prefix, it is assumed to be located wherever the importing Hava file is located. If imported from the source editor, it is assumed to be located wherever the Hava interpreter itself is running. (Fine print: Due to security restrictions, the Hava applet cannot import from the local drive.) The file extension `.hava` is optional.

The importing Hava source may override variable definitions in the file(s) it imports. Example: File `sample.hava` solves a problem of size N and defines `N=10`. Now the following one-line source will cause `sample.hava` to be loaded and executed for a problem of size 10:

```
import sample;
```

But the following two-line source will run the sample for a problem of size 20:

```
import sample;
N = 20;
```

Difficulties will arise if two files are imported in which the same variable is defined. Since neither imported the other, neither can override the other.

The keyword `private` in front of an import statement causes all variables and documentation comments defined in the imported file to be omitted from the report.

The keyword `final` in front of any variable definition will prevent it from being overridden if this file is imported. The keyword `final` in front of the import statement designates all variables of the imported file as final.

Imported Hava files may import other Hava files. A Hava file may be imported any number of times, from any number of other Hava files; the later imports are simply ignored.

Overridden variables remain accessible in expressions, by prefixing them with the name of the imported file where they are defined, followed by a dot. (If the imported filename contains whitespace, it is interpreted as a single space when retrieving the file, and as a single underscore in the prefix.) Example: File `sample.hava` defines many variables, but we are only interested in seeing the single variable `solution`. We could do this:

```
private import sample;
solution = sample.solution;
```