RTL Reference

1. JVM

Record Transformation Language (RTL) runs on the JVM. Runtime support for operations on data types are all implemented in Java. This constrains the data types to be compatible to Java's data types.

2. Lexical Conventions

Only the ASCII character set is supported, characters outside of that character set in source files will cause a error.

Regular expressions as defined in flex are used to add clarity. These expression follow the rules of regular expression as used by the flex package. These expressions will be in `mono spaced font`.

2.1 Blanks

```
blank = [ \t\n\r]+
```

Unless blanks are inside a string literal, they are ignored and provide no meaning to the program except as separators for program elements.

2.2 Comments

```
comment = "#".*$
```

Outside of a string literal, all characters following a `#` character on the same line of code are considered comments and are ignored by the compiler.

2.3 Identifiers

```
letter     = [a-zA-Z]
digit      = [0-9]
identifier = ({letter}|"_")({letter}|"_"|{digit})*
```

All characters in an identifier are significant. Identifiers are limited 256 characters. Identifiers are case sensitive, upper and lower case characters are considered different.

2.4 Integer Literals

```
integer = ({digit})+("_"+{digit}+)*
```

For readability, all integers can have the underscore character embedded in the number, the underscore is ignored when evaluating the value of the literal. Note, the integer literal cannot have a trailing underscore.

2.5 Floating Point Literals

```
float = {integer}"."{integer}([eE][-+]?{digit}+)?
```

Just as in integer literals, underscore characters can be embedded in the number for readability. Floats in RTL conform to double precision 64 bit IEEE 754 floating point standard. Note, all floating point literals require a digit before and after the decimal point. `.34` and `124.` are not validate floating point numbers.

2.4 Date and Time Literals

```
bdate = {digit}{4}{digit}{2}{digit}{2}
edate = {digit}{4}{digit}{2}{digit}{2}
btime = {digit}{2}{digit}{2}{digit}{2}?{letter}?
etime = {digit}{2}:{digit}{2}:{digit}{2}?{letter}?
datetime = {bdate}T{btime}|{edate}T{etime}
```

2.5 Character Literals

```
character = [ -~]
```

RTL supports all ASCII printable characters as literals.

2.6 String Literals

```
String = "\"".*"\""
```

To embed a double qoute in a string, you can prefix the double quote with a backslash. `"This \"string\" is quoted."` is an example where the word string is surrounded by double quotes.

2.7 Boolean Literals

```
boolean = true|false
```

2.8 Set Literals

```
bliteral = {integer}|{float}|{date}|{character}|{string}|{boolean}
set = \{{blank}*({bliteral}{blank}*(,{blank}*|{bliteral}
{blank}*)*)*\}
```

A set is a collection of values with the same data type containing no duplicates. Set literals are separated by bracketing a comma separated list of values between the `{` and `}` brackets. `{}` is considered an empty set.

2.8.1 Examples

```
{ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" }
{ 1, 2, 3, 4 }
```

```
{ 34,8, 90.2, 891.19 }
```

2.9 Map Literals

```
kv = {bliteral}":"{bliteral}
map = "{"{blank}*({kv}{blank}*(,{blank}*|{kv}{blank}*)*)*]"}"
```

A maps key and value types can be different, but must the same type respectively. Map literals are created by bracketing a comma separated list of key value pairs between the `{` and `}` brackets. Key value pairs are any two values separated by a colon. `{:}` is considered an empty map.

2.9.1 Examples

```
{ "bob" : 34, "ted" : 27, "alice" : 37 }
{:}
{ 98 : ~2008-12-25~, 44 : ~2009-01-23~ }
```

2.10 Array Literals

```
array = "["{blank}*{bliteral}{blank}*(,{blank}*|{bliteral}
{blank}*)*"]"
```

An array is a list of values of the same data type. Array literals are created by bracketing a comma separated list of values between angle brackets `[` and `]`. `[]`, is considered an empty array.

2.10.1 Examples

```
[ 89, 90, 91, 92 ]
[~12:34:00~, ~12:34:00~, ~12:34:00~, ~12:34:00~, ~12:34:00~]
[true, false, false, true, false]
```

2.11 Keywords

These identifiers are keywords, and as such, cannot be used as program identifiers.

```
bool       break      char       check      date
datetime   delete     else       file       filter
float      for        foreach    function   if
int        log        map        process    print
rec        return     set        string     time
to         trans      transform  while
```

3. Objects and names

All objects must be declared, and there fore named, before they can be accessed. Static scoping is used to limit the scope of a name either the file level or to a block delimited by `{}`. Binding the same name

to a different object in an inner scope will hide the out binding.

4. Expressions

Expressions are evaluated in the order they are listed.

4.1 (*expression*)

Provides grouping for expression so that the expression inside the brackets must be evaluated before any expression before or after the grouped expressions.

4.2 $expression_1$ [$expression_2$]

If $expression_1$, is an array, then $expression_2$ must evaluate to an integer and provides access to the contents of the array at that position. If $expression_1$, is a map, then $expression_2$ must evaluate the same data type as the key for the map and provides access to the value of the key-value pair.

4.3 $expression_1$ { $expression_2$ }

Provides access to the contents of an array $expression_1$, $expression_2$ must evaluate to an integer.

4.4 ! *expression*

Negates the value of the expression, only valid for boolean expressions.

4.5 – *expression*

Reverses the numeric sign of the expression.

4.6 Binary expressions

$expression_1$ * $expression_2$ - multiplication
$expression_1$ / $expression_2$ - division
$expression_1$ + $expression_2$ - addition
$expression_1$ – $expression_2$ - subtract

Evaluates to numerical result of the normal mathematical operations. These operators are only valid for numerical data types int and float.

$expression_1$ < $expression_2$ - less than
$expression_1$ > $expression_2$ – greater than
$expression_1$ <= $expression_2$ - less than or equal
$expression_1$ >= $expression_2$ – greater than or equal
$expression_1$ == $expression_2$ - equal
$expression_1$!= $expression_2$ – not equal

expression$_1$ `&&` *expression*$_2$ – logical and

expression$_1$ `||` *expression*$_2$ – logical or

Evaluates to a boolean value, the data types of *expression*$_1$ and *expression*$_2$ must be the same.

expression$_1$ `=` *expression*$_2$ – assignment

Assign the values of *expression*$_2$ to the object in *expression*$_1$. The the data types of *expression*$_1$ and *expression*$_2$ must be the same.

4.4 Operator precedence

Operators are listed in order of precedence, lower precedence operators are listed below those with a higher precedence.

Operator Type	Operator	Associativity
Primary	`() [] {}`	Left to right
Unary	`! -`	Right to left
Binary	`* /`	Left to Right
	`+ -`	
	`< > <= >= == !=`	
	`&& ||`	
Assignment	`=`	Right to left

5. Declarations

A declaration creates an object and gives it a name. These objects are usually variables or functions. The object can then be referenced by name as long as the name is in scope. When the name goes out of scope, the object can no longer be referenced and for all purposes, does not exist any more.

Name scoping uses static scoping rules, and names declared in an inner block will hide names declared outside that block. Functions can only be declare at the file level, nested function declarations are not supported.

6. Statements

A *statement* is a single language construct, such as an expression. A *statement_list* is an grouping of statements that are evaluated in the order they are listed.

6.1 Control Statements

`if`, `while`, `for` and `foreach` provide a way to control the execution of a *statement_list*.

6.1.1 If statements

`if (` *expression* `) {` *statement_list*$_1$ `}`

`if` (*expression*) { *statement_list*$_1$ } `else` { *statement_list*$_2$ }

If the *expression* evaluates to true, then the *statement_list*$_1$ is executed, otherwise, *statement_list*$_2$ is executed if present. The open and closing brackets are always required around either *statement_list*.

6.1.2 While statement

`while` (*expression*) { *statement_list* }

The *expression* is evaluated first, and as long as the *expression* evaluates to true, the *statement_list* will be execute repeatedly. The open and closing brackets are always required around either *statement_list*.

6.1.3 For statement

`for` (*expression*$_1$`;` *expression*$_2$`;` *expression*$_3$) { *statement_list* }

expression$_1$ is evaluated once, before any other expressions are evaluated. Then the *statement_list* is executed. Then *expression*$_3$ and *expression*$_2$ are evaluated in that order. As long as *expression*$_2$ evaluates to true, the *statement_list* will be repeatedly executed. The open and closing brackets are always required around either *statement_list*.

6.1.4 Foreach statement

`foreach` (*expression*$_1$ `in` *container*) { *statement_list* }

The `foreach` statement iterates through the elements of the container, providing access to each element in turn. *expression*$_1$ needs to declare the variable that will hold the contents of the elements in the container. Modification of the data in the variable does not change the contents of the elements in the container, the variable contains a copy element. The open and closing brackets are always required around either *statement_list*.

6.2 Break statement

`break`

The `break` statement only has meaning in the body of a looping statements `while`, `for` and `foreach`. It will terminate the loop and continue execution after the loop body.

6.3 Return statements

`return` *expression*

This will cause any function to stop executing the statements in it's statement_list and return the value of the *expression* to the caller.

6.4 Log statements

`log` *file string*

Writes the *string* to the output data stream name file. Each *string* is written on its own line with the

current date and time written out before *string*.

7. Data types

These are the supported primitive data types.

Name	Type	Values
`char`	A single character.	Any printable ASCII character.
`string`	A sequence of characters	
`bool`	A binary value.	True or false.
`int`	An integral value.	-2^{63} to $2^{63} - 1$
`float`	A floating point number.	-2
`date`	A date.	1900-01-01 to 9999-12-31
`time`	A time of day.	00:00:00 to 23:59:59
`datetime`	A date and time combined.	

7.1 Built in data conversion functions

There are a set of supplied functions that take a single expression and convert the data from the expression into another type. These functions all have the name `to_`{*data_type*}. For example, to convert a string into an integer, you would call the function `to_int()`. `date`, `time` data types are an exception to this, they cannot only be converted to and from `string`, and `datetime` can only be converted to a `date` or `time` data type.

7.2 Data containers

arrays, sets and maps are supported data containers. They can contain any primitive data types.

7.2.1 Array interface

These functions will can be used on array containers.

`add( array, value ) bool` – adds value to the end of the array. If the value cannot be added, the function will return false.

`add_at( array, int, value ) bool` – adds value at the array position, expanding the array as needed to accommodate the new position. If the value cannot be added, the function will return false.

`remove_at( array, int ) bool` – removes the position from the array, reducing the number of positions in the array by one. If the position does not exist, the function will return false.

`count( array ) int` – Returns the size of the array

In addition to these functions, an array's contents can be accessed by using the `[]` operator. The `[]` operator can be used to update the value at a position or retrieve its value in an expression.

7.2.2 Set interface

These functions will can be used on set containers.

`add( set, value ) bool` – adds the value to the set, if the values already exists, the function returns false.

`remove( set, value ) bool` – removes the value from the set if it exists, otherwise it returns false.

`exists( set, value ) bool` – returns true if the value is in the set.

`count( set ) int` – returns the size of the set.

7.2.2 Map interface

These functions will can be used on map containers.

`add( map, key, value ) bool` – adds the key-value pair to the map, if the key already exists, the function returns false and does not add the key-value pair.

`remove( map, key ) bool` – removes the key-value pair from the map if it the key exists, otherwise it returns false.

`exists( map, key ) bool` – returns true if the key is in the map.

`count( set ) int` – returns the size of the map.

In addition to these functions, a map's contents can be accessed by using the `[]` operator. The `[]` operator can be used to update the value at a position or retrieve its value in an expression. The key for the key-value pair must be specified inside the operator's brackets.

7.3 Date and Time manipulations

+ and – are supported operations on date and time data types. These operators add and subtract days and seconds respectively.

7.3.1 Date inteface

These functions will can be used on a date data type.

`to_string( date ) string` – return the string representation of a date.

`year( date ) int` – return year part of the date.

`month( date ) int` – return month part of the date.

`day( date ) int` – return day part of the date.

7.3.2 Time interface

These functions will can be used on a time data type.

`to_string( time ) string` – return the string representation of a time.

`hour( time ) int` – return hour part of the time.

`minute( time ) int` – return minute part of the time.

`second( time ) int` – return seconds part of the time.

7.3.3. Datetime inteface

These functions will can be used on a datetime data type.

`to_string( datetime ) string` – return the string representation of a datetime.

`to_date( datetime ) date` – extract the date part.

`to_time( datetime ) time` – extract the time part.

`to_datetime( date, time ) datetime` – construct a datetime from a date and time.

8. Strings

Strings are objects just like any other data type. They are not arrays of characters. String manipulations are all performed using built in functions.

`substring( string, pos, len ) string` – returns the inner string of the specified `string` start at `pos` (string positions start at 0) for len characters. If the length requested extents past the end of the string, only those characters are returned.

`concat( string`$_1$`, string`$_2$ `) string` – concatenates `string`$_1$ and `string`$_2$ to form a new string which is returned.

9. Data streams

`file` *name* = "*path*" `noheader sep=`*ch* `output|log`

There are three types of data streams, input, output and logging. Input data streams are the default. The path can be an absolute path or relative to the current process's directory. Only the *name* and *path* of a `file` statement are required, `noheader`, `sep`, `output` and `log` are all optional attributes for a data stream.

All data stream except the log data stream operate on Character Seperated Value or CSV files. Each line in the file is treated as a record to be processed.

The `noheader` attributes for input streams informs the reader that the first line of the CSV file is data. For output streams, no header is written to the resulting CSV file.

The `sep=`*ch* attribute changes the default separation character from a `','` to *ch*. This can be specified for both input and output data streams, but has no meaning for a log data stream.

The `output` attribute changes the data stream from the default input to an output stream.

The `log` attribute changes the data stream to a logging output stream. Instead of writing records, each line of text is written with a data time stamp. The `output` and `log` attribute cannot be specified for the same data stream.

Any attempt to read or write to a data stream that fails, will result in the termination of the application with an OS appropriate error message written to standard error. The return code will be the OS error

number.

10. Functions

10.1 Generic functions

`function` *expression* `(` *param_list* `)` *return_type* `{` *statement_list* `}`

Functions create grouping of statements that can be executed by name. The parameter_list defines a list of variables that have local scope to the function whose values must be provided by the caller. The value of the *return_type* is either the value of the last expression in the *statement_list* or the value of the expression in a `return` statement. The data type of the returned value from the function must match the data type declared in the *return_type*.

10.2 Filter functions

`filter` *expression* `{` *statement_list* `}`

A filter function has an implicit return_type of bool. These functions are used to filter out records that should not be included in the result set from processing the input stream. The filter is called for each record in the input stream. It has a single implicit parameter `rec`, that is a map containing the value of the current record. The elements of the map can be accessed by name or column number. If the function returns false, the record is not included in the output stream.

Any changes to the values in the `rec` map will not be reflected in the output stream.

10.3 Transform functions

`transform` *expression* `{` *statement_list* `}`

A transform function has no return type, it does not return any value. These functions are used to transform a record before it is written to an output stream. The transform function is called for each record in the input stream. It has a single implicit parameter `rec`, that is a map containing the value of the current record. The elements of the map can be accessed by name or column number.

11. Processing

`process` *input check_trans_list* `to` *output*

This is the main part of the application. The `process` statement will run each record in the input stream through the list of data checks and transformations in the *check_trans_list*. If the record evaluates to true for all `check` statements it will be written to the output stream with any modifications made by the `trans` statements in the *check_trans_list.* The `check` and `trans` statements are executed in the order they are declared.

11.1 rec map

To access the contents of the current record, an implicit map is created. The keys for the map are the names of the columns in the header row of the input file. If the input file has a `noheader` attribute, then the keys are the column numbers starting with 0 for the first column. The values in the map are

the read in values in the CSV row.

11.2 Check statement

`check { ` *expression* ` }`

If the expression evaluates to true, then the record passes the check. The code block for the expression is the same as the code block for a `filter` function. A rec map is implicitly defined in the code block.

`check` *filter_function*

The *filter_function* must be a `filter` function. It will be called for each record in the input stream and if the `filter` function returns false, the record will not be included in the output stream.

`check "`*key*`"` *file*

The *file* should be an input stream with a single column. If there are more columns in the input stream then, columns after the first column are ignored. If the value of the rec map for the *key* is not found in the first column of the *file*, the check will fail and the record will not be included in the output stream.

11.3 Trans statement

`trans { ` *statement_list* ` }`

The code block for the statement_list is the same as the code block for a `transform` function. A rec map is implicitly defined in the code block. Any changes to the values of the rec map will be included in the output stream.

`trans` *transform_function*

The *transform_function* must be a `transform` function. It will be called for each record in the input stream. Any changes to the values of the rec map will be included in the output stream.

`trans "`*key*`"` *file*

The *file* should be an input stream with two columns. If there are more columns in the input stream then, columns after the first two columns are ignored. If the value of the rec map for the *key* is not found in the first column of the *file*, then no transformation will be made. If the key is found, then the values of the rec map for the key will be update to the value of the second column on the same row. Any changes to the values of the rec map will be included in the output stream.