# Language Specification Document

The language supports the following types

Integer type: The keyword used for representing integer data type is int and will be supported by the underlying architecture. A statically available number of the pattern [0-9][0-9]*
is of integer type.

Real type: The keyword used for representing real data type is real and will be supported by the underlying architecture. A statically available real number has the pattern [0-9][0-9]*.[0-9][0-9] and is of
type real.

Record type: This is the constructed data type of the form of the Cartesian product of types of its constituent fields. A record, for example, the following record is defined to be of type 'finance' and its

actual type is <int , real , int>
record #finance
type int: value;
type real:rate;
type int: interest;
endrecord

A record type must have at least two fields in it, while there can be more fields as well.
The type information is fetched at the semantic analysis phase. A variable identifier of type finance is

declared as follows:

type record #finance : d5bb45;

The names of fields start with any alphabet and can have names as words of English alphabet (only small

case). The fields are accessed using a dot in an expression as follows:

d5bb45.value <--- 30;

d5bb45.rate <--- 30.5;

Types of these constructed names using variable name and record fields are same as field

types defined in the record type definition correspondingly.

Test Case:

A test case handling addition operation on two records and use of record variables in parameters list is

depicted below. The record type #book declared in _main function is also visible in function _recordDemo1.

_recordDemo1 input parameter list [record #book d5cc34, record #book d2cd]

output parameter list[record #book d3];

record #new

type int : value;

type real: cost;

endrecord;

```
d3<--- d5cc34 + d2cd;
return [d3];
end

_main
record #book
type int : edition;
type real: price;
endrecord;

type record #book b2;
type record #book c2;
type record #book d2;
type record #new b7bc34;

b2.edition <--- 3;
b2.price <--- 24.95;
c2.edition <--- 2;
c2.price <--- 98.80;

% following is a valid statement as the types of d2, b2 and c2 are same
d2<--- b2 + c2;
% Note that d2<--- b2 + b7bc34; is not type correct as the language follows name equivalence.
% Note that the types of b2 and b7bc34 are not the same.

[d2]<--- call _function1 with parameters[b2,c2];
```

```
write(d2);
end
```

Name Equivalence:

The language supports name equivalence and not structural equivalence, meaning similar structured record definitions are treated as different. For example, #new and #book are the two record types
with similar structure (sequence and type of variables) but different names. Assignment statements with
variables from two different record types are not allowed. Also, once a record type is
defined, its re-occurrence anywhere in the program is not allowed and is treated as an error.

Operations on Records:

A variable of record type can only be multiplied or divided by a scalar (integer or real); two record type variables cannot be multiplied together nor can be divided by the other. Two operands of record type can be added, subtracted if the types of the operands match and both the operands are of record type. Semantically an addition/subtraction means addition/subtraction of corresponding field values, for example:

```
type record #finance : d5;
type record #finance : c4;
type record #finance : c3;
c3 <--- c4 + d5;
```

Global:

Defines the scope of the variable as global, visible anywhere in the code. Syntax for specifying

a variable of any type to be global:

type int: c5d2: global;

Functions:

There is a main function preceded by the keyword _main. The function definitions precede the function calls. Function names start with an underscore, e.g., _function1.

Example:

_function1

input parameter list [int c2, int d2cd]

output parameter list [int b5d, int d3];

b5d<---c2+234-d2cd;

d3<---b5d+20;

return [b5d, d3];

end

_main

type int: b4d333;

type int : c3ddd34; type int:c2d3; type int c2d4;

read(b4d333); read(c3ddd34);

[c2d3, c2d4]<--- call _function1 with parameters [b4d333, c3ddd34];

write(c2d3); write(c2d4);

end

The language does not support recursive function calls. Function overloading is not allowed in the language. Function's actual parameters' types must match with those of formal parameters. Even if

a

single actual parameter in a function call statement does not match with the formal parameter's type, it

is an error.

Statements:

1. Variable Assignment: Variables can be assigned values using `<---`.

   Example:

```
d5bb45.value <--- 30;
d5bb45.rate <--- 30.5;
```

2. Global Declaration: Variables can be made globally accessible throughout the code using `global`.

3. Function Definitions and Calls: Functions are defined with input/output parameter lists, prefixed with `_`.

4. Input and Output: `read()` for input, `write()` for output.