Harmonica Language Reference Manual

Guihao Liang (gl2520), Jincheng Li (jl4569),
Xue Wang (xw2409), Zizhang Hu (CVN, zh2208)

1. **Introduction**
   Harmonica is a language based on C and borrows ideas from Python / Go. It natively supports concurrency features and utilities commonly found in scripting languages.

2. **Lexical Elements**
   a. Identifiers
      Identifiers are arbitrary strings starting with [a-z A-Z] and followed by [a-z A-Z 0-9] (not underscores). We recommend camelCase.
   b. Key words
      `if, else, elseif, bool, int, float, string, list, struct, # (comment), return, parallel, channel, import` and various operators defined below.
   c. Constants
      A constant is a literal numeric or character value, such as 5 or 'm'
      i. Integer: decimal integers, [1-9]+[0-9]*
      ii. Float: real numbers following the C standard. Examples are 1.4, -0.3, 5e2, etc.
      iii. String: a string instance enclosed with double quotation marks.

3. **Data Types**
   a. Int
      Integer type. We only support 32-bit integers, ranging from -2^31 to 2^31-1.
   b. Float
      Floats are 8-byte double-precision floating point numbers.
   c. Boolean
      Boolean is just a boolean, true or false.
   d. Byte
      Byte represents 8-bits of raw data.
   e. Stringd
      String is a sequence of ascii characters. Its length is limited by maximum integer value, which is 2^31-1.
   f. Tuple[]
      A tuple is a sequence of elements (of non-uniform types), with maximum length 2^31-1.
   g. List[]
      List is a sequence of elements of the same type, with maximum length 2^31-1.

h. Struct
   Struct is a composite data structure supporting member variables and methods.
i. Function Types
   In Harmonica, function types are defined by the types of their input parameters and their return type. The notation for function types are simply a list of types separated by space, where the last type in the sequence is assumed to be the function's return type. For example, the type `Int Int List[Int]` is the type of a function that takes two integers and returns a list of integers.

4. Expressions and Operators

a. Expressions
   The definition of expression is same with C. An *expression* consists of at least one operand and zero or more operators.

   Operands are typed objects such as constants, variables, and function calls that return values. *Operators* specify an operation to be applied on its operands.

b. Assignment Expressions
   Assignment expression stores value to the variable.

   The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand (commonly referred to as the "lvalue") cannot be a literal or constant value, it should should be a variable can be modified..

```
int x = 10;
float y = 45.12 + 2.0;
int z = (2 * (3 + function () ));

struct foo {
int bar;
int baz;
} quux = {3, 4};
```

   You can also use the plain assignment expression to store values of a structure type.

   Also, we support compound assignments, such as +=, -=, *=, /=, %=, <<=, >>=, >>=, &=, ^=, |=.
   `a += b` is equal to `a = a + b;`

c. Arithmetic operators
   Unary operators are - +, with highest priority, followed by binary + - % operators, followed by binary + - operations.

d. Comparison operators
   Binary logical and relational operators: ==, , >, >=, !=, <=, <. These operators do shallow comparison, meaning the lvalue and rvalue.

   For structure, the equality comparison will compare each field by value, here's an instance:

```
struct float_string {
      string str = “str”;
      float   flt = 1.0;
};
float_string a, b;
```

   `a == b` is equal to `a.str == b.str && a.fl t == b.flt.`
   Like java, we don't propagate other non-zero type into bool. That is, in C or C++, non-zero can represent `true`, while in our language, you should use `a.flt > 0` explicitly.

e. Bit Shifting
   The left shift operator should be same to C, that is, new bits added on the right side will be 0.

```
fst_operand << snd_operand;
2 << 1;
```

   The right shift >> will be kind of complicated. If the value is signed value, then the bits added on the left will be 1, otherwise 0.

```
-2 >> 1;
```

   The zero right shift >>> is borrowed from Java, where the right added value are restricted to 0.

```
-2 >>> 1;
```

   If the second operand is greater than the bit-width of the first operand, the behaviour is undefined.

f. Bitwise logical
   Binary operators:
   Conjunction &: `11001001 & 10011011 = 10001001`
   inclusive disjunction I: `11001001 | 10011011 = 11011011`
   exclusive disjunction ^: `11001001 ^ 10011011  = 01010010`

   Unary operators:
   Negation ~: `~11001001 = 00110110`

For these operators, you should only use with char, int types, and for maximum portability, use unsigned int types.

g. Type Casts
You can use the type cast in the same way of C, note that, there will be precision lost if you down casts.
```
int i = (int) 3.5;
```

Same with C, type casting only works for scalar types, such as int, float or reference type. The following type casting will fail:
```
list[int] arr = create(int, 8);
(double[]) arr; // FAIL
```

h. Array initialization and subscripts
When creating arr ???

i. Function calls as Expressions
Functions which return values can be expressions.
```
int function(int);
a = 9 + function(9);
```

j. Comma Operator
You use the comma operator , to separate two expressions. The first expression must take effect before being used in the second expression.
```
int x = 1, y = 2;
x += 1, y += x;
```
The return value of comma expression is the value of second expression. In the above example, the return value should be 4.

If you want to use comma expression in function, you should use it with parentheses because in function call, comma has a different meaning, separating arguments.

k. Member Access Expressions
You can use access operator dot . to access the members of a structure variable.
```
struct foo {
      int x, y;
};
struct foo bar;
bar.x = 0;
```

l. Conditional expressions

You use the conditional operator to cause the entire conditional expression to evaluate on either second operand or the third operand. If a is `true`, then the expression will evaluate `b`, otherwise `c`.
`a ? b : c`
The return type of `b` and `c` should be compatible, meaning the same type in our language.

m. Operator Precedence

1. **Function calls or grouping**, array subscripting, and membership access operator expressions.
2. Unary operators, including logical negation, bitwise complement, unary positive, unary negative, indirection operator, type casting. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x). **(right to left)**
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Bitwise shifting expressions.
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
7. Equal-to and not-equal-to expressions.
8. **Bitwise** AND expressions.
9. Bitwise exclusive OR expressions.
10. Bitwise inclusive OR expressions.
11. **Logical** AND expressions.
12. Logical OR expressions.
13. Conditional expressions (using ?:). When used as subexpressions, these are evaluated right to left.
14. All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left. **(right to left)**
15. Comma operator expressions.

We took the experience here:
http://www.cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html

5. **Statements**
   a. Expression Statements
   Similar to C, any expression with a semicolon appended is considered as a statement.

b. `if-else` statements
   `if-else` statements are in the following forms.
   `if (statement) { statement }`
   or
   `if (statement) { statement } else {statement}`
   or
   `if (statement) {statement} [elseif (statement){ statement }]+ [else {statement}]`
c. `while` statements
   Format: `while (statement) { statement }`
d. `for` statements
   `for` statements can take the form of
   `for (statement; statement; statement) {statement}`
   We also support C++11's for-each statement style, namely,
   `for (element : list-like) {statement}`
   to iterate over any list-like data structure.
e. Blocks
   Similar to C, Harmonica uses braces to group zero or more statements. Blocks can be nested. Variables declared inside a block are local to that block.
f. `break, continue`
   `break` terminates a while or for control structure;
   `continue` terminates an iteration of the for or while loop and begins the next iteration.
g. `return`
   Used to end the execution of a function. It should be followed by a return value matching the declared return type (no return value is needed for a function returning `void`).
h. `typedef`
   `typedef` is used to create new names (aliases) for data types. The syntax is:
   `typedef oldType newType`
i. `import`
   Includes source code from another module (modules are defined by a single source file, see Program Structure section).

6. **Functions**
   You can write functions to separate parts of your program into distinct sub-procedures. Every program requires at least one function named `main`, which is where the program begins execution.
   a. Definition
      Functions are defined with the following syntax:
```
returnType functionName (parameterList) {
  functionBody;
}
```

A parameter list is a comma-separated list of parameters, where each parameter consists of a parameter type followed by a parameter name.

b. Calling a function
Functions are called by their name and appropriate parameters. For example:
```
foo (1, "bar");
```

c. Lambda functions
Functions can also be defined with the `lambda` keyword. However, lambda functions are restricted to a single line and meant to be used for quick, one-liner functions just like what python does. For example:
```
list[int] onePlus = map(lambda a -> (a+1), [1,2,3]);
```

d. First-class functions
Functions are first-class members in Harmonica, which means that they can be assigned to variables and passed as parameters just like any other variable. Type declaration for functions compose of a sequence of types representing the types of function parameters followed by a single return type. However, we maintain a bottom line to prevent any abuse of higher than 2^31-1 orders of functions.

7. **Concurrency Support**
   a. `parallel`
   The `parallel` keyword spawns multiple child threads that execute the same function. It takes 2 required parameters: a function, and an iterable collection of elements. The function would be called on each of the elements in the collection in a separate thread. An optional 3rd argument can be specified to control the number of threads spawned from `parallel`.
   b. `channel`
   `channels` are pipes that make it easy for different threads to pass data around. You can declare a channel of capacity 5 with `channel[T] c = chan(T, 5);`. Channels support two basic operations: push and pop. Push inserts an element into the channel while pop retrieves an element from the channel. There is no guarantee about the order of elements inserted/retrieved. Push will block if the channel is full, and pop will block if the channel is empty.

8. **Program Structure**
A Harmonica source file consists of a list of statements and function definitions. Each source file defines a module that can be imported in other files. The compiler compiles several source files, with a single definition of the main function, into an executable program.

9. **Sample Program**

```
int foo(int int void f, bool b) {
      print(“Hello, foo.”);
}
int int tuple[int int] f0 = lambda a b -> (a, b);

bool bar(list[int] arr) {
      if (List.contains(arr, 42)) {
            print(“Hoo, bar.”);
      }
}

typedef list[int] boo;
boo aoo, foo;
aoo = [1,2];
foo = [41,42];
list[boo] boos = [aoo, foo];
parallel(bar, boos); #should print “Hoo, bar.” only once



int sum(list[int] lst, int i, int j, channel[int] c) {
      int sum = 0;
      for (int k = i; k < j; k++) {
            sum += lst[k];
      }
      c.push(sum);
}

int main() {
      channel c = chan(int);
      List[int] lst = [1,2,3,4];
      # two threads to sum on different parts of lst
      parallel(sum, lst, 0, 2, c);
      parallel(sum, lst, 0, 2, c);
      # return the sum of entire list
      return c.pop() + c.pop();
}
```