Towel Reference Manual

Zihang Chen (zc2324) Baochan Zheng (bz2269) Guanlin Chen (gc2666)

October 26, 2015

DON'T
PANIC

Contents

Chapter 1

Lexical Elements

1.1 Keywords

Keywords in the Towel programming language is as follows:

```
if>=0 if>0 if<=0 if<0 if~0 if=0 ift iff ife ifne
match fun bind type also then
```

The corresponding tokens are:

```
IFGEZ IFGZ IFLEZ IFLZ IFNEZ IFEZ IFT IFF IFE IFNE
MATCH FUNCTION BIND TYPE ALSO THEN
```

1.2 Punctuations

Punctuations used in the Towel programming language are as follows:

- Whitespace characters are simply ignored.
- These characters have special meanings in the Towel programming language: `‘ ’ ‘‘ , ; . ( ) [ ] \ @ eof`. This means that you cannot use these characters in names and atoms. [1]
- Any unprintable character is reserved and won't be used.

Specially,

```
TERMINATOR ::= ['.' '\n' '\r' '\r\n' eof]
```

1.3 Names

Names are used for naming (or to be more precise, referencing to) values. Valid names should not start with reserved punctuations, lowercased letters, and numbers.

More formally,

```
unprintable ::= [all the unprintable ASCII characters]
whitespace ::= ['\n' '\t' ' ' '\r']
reserved_punct ::= ['‘' ''' '"' ',' ';' '.' '\' '@'
                    '(' ')' '[' ']' '{' '}' whitespaces unprintables]
valid_punct ::= ['!' '~' '#' '$' '%' '^' '&' '*' '-' '_' '+' '=' '|'
                 ':' '<' '>' '?' '/']
```

[1] In other words, you can use any other punctuation characters in names and atoms.

```
BQUOTE ::= '`'
SQUOTE ::= '''
DQUOTE ::= '"'
COMMA ::= ','
SEMICOLON ::= ';'
PERIOD ::= '.'
SLASH ::= '\'
AT ::= '@'
LPAREN ::= '('
RPAREN ::= ')'
LBRACKET ::= '['
RBRACKET ::= ']'
LBRACE ::= '{'
RBRACE ::= '}'

digit ::= ['0'-'9']
hexdigit ::= ['0'-'9' 'a'-'f' 'A'-'F']
bindigit ::= ['0'-'1']
lc_chars ::= ['a'-'z']
NAME ::= [^ '-' reserved_punct digit lc_chars] [^ reserved_punct]*
```

1.4 Literals

Most easy-to-use languages support a wide variety of literals (Python is a good example and Java is not). The Towel programming language supports literals for atoms, integers (fixed)[2], floats, strings, and lists. They are defined as follows (rule for list literals will be revealed later):

```
ATOM ::= lc_chars [^ reserved_punct]*

signed ::= ['+' '-']
INT ::= signed? (("0d"? digit+) | ("0x" hexdigit+) | ("0b" bindigit+))

dot ::= '.'
int ::= digit+
frac ::= digit+
exp = 'e' signed? int
dot_float = ((dot frac) | (int dot frac)) exp?
exp_float = int (dot frac)? exp
FLOAT ::= signed? (dot_float | exp_float)

string_char ::= [^ '\' ''']
string_esc_seq ::= '\' string_char
string_item ::= string_char | string_esc_seq
STRING ::= ''' string_item* '''
(Rules for strings is from the lexical parsing section of the Python
 language reference manual.)
```

Note that because terminator is also a dot, a float number literal cannot be written as `int dot`, otherwise any integer before a period terminator will be scanned as a float. Although it's feasible to handle this case, but that involves some more parser rules for integer literals alone.

[2]Fixed integers are interpreted as signed integers. There is no literal provided for unsigned fixed integers for now.

1.5 Comments

Comments are defined as follows:

```
__COMMENTS ::= '"' [^ '"']* '"'
```

1.6 Lexical Error

When the scanner encounters any other character not mentioned above, it will raise a `LexicalError` exception.

Chapter 2

Data Types

Types are important and inferred in Towel. If the type check fails, the compiler will refuse to compile the program. However, when compilation finishes, all the type information is thrown away.

This chapter covers the basics on types in Towel.

2.1 Primitive Types

Towel provides to the user the following primitive built-in types:

- Atom
- FixedInt
- Unsigned FixedInt
- Int (won't be implemented until later versions)
- String
- Float
- List
- Tuple

Internally, these types are phony type values bound to names above. For example, `Atom` binds to the value `type-value:atom` so that you can reference to types in Towel.

2.1.1 Atoms

Atoms are special names uniquely bound to integer constants. But they are not comparable to integers.

2.1.2 `FixedInts`, `Ints`, `Floats`

Fixed integers and floats are simply OCaml integers and floats. `Int`s are signed integers of arbitrary precision (like those `int`s in Python). These types are subclass of the class `Number`. A lot of arithmetic functions take `Number`s as their arguments. However, bitwise functions will take `FixedInts` as arguments.

There is also an unsigned version of `FixedInt`. It will be internally interpreted as a `Stdint.Uint32` of library `ocaml-stdint`.

2.1.3 Strings

Strings are implemented as OCaml strings. String items are surrended by single quote, rather than double quote.[1]

2.1.4 Lists and Tuples

Lists and tuples are implemented as OCaml lists. Types of tuples, for instance, `PT_Tuple1` for 1-element tuples, are decided at compile time, and cannot be changed later, whereas the lengths of lists are not fixed.

2.2 `PT_Any`

In current version, without generic typing and algebraic types implemented, `PT_Any` is the superclass of the type of every value in Towel.

2.3 Algebraic Data Types

Users can use the `type` special form to define custom algebraic data types, see also section 3.9 and chapter 4. However, type hierarchy between algebraic data types hasn't been designed yet.

2.4 Functions

The type of functions is a list of types, which consists of the types of input arguments along with the return type. When applying arguments to functions, if the arguments are not enough (i.e. the caller's stack exhausts before all the arguments are satisfied), a partial applied function is returned.

2.5 Type Checking

Type checking is an operation done between two types, and is not commutative. When we say type A is checked against type B, type checking succeeds under the following circumstances:

- The two type being exactly the same,
- Or, type A is a subclass of type B,
- Or, check the type of each item of type A and type B, if any of the type checkings succeeds, the whole type checking between type A and type B succeeds.

For a program, each name or value is type checked against the respective argument type of a function that is applied with the names or values. For example, in the following program:

```
bind B 'string'
also A fun,
  (2.0 B +)
then A.
```

`2.0` and `B` are applied to `+`. Because `+` is of `Number -> Number -> Number`, so `2.0` is type checked against the first argument `Number` type and succeeds, `B`, a string, is type checked against the second argument `Number`, which fails. So the type check of the program fails. Compiler complains about this and exits.

[1]Because you don't have to hit the *shift* key when inputing single quotes. Same goes for brackets.

Chapter 3

Elements of Programs

3.1 Basic Concepts

A Towel program consists of one or multiple so-called **words** and one terminator. A terminator can be a period (dot) or the end-of-file character.

When encountered multiple words, they are always evaluated one by one in the order they appear. Although in most times, Towel remains in a postfix fashion, but for the sake of convenience, some parts of the grammar is of prefix or infix style (e.g. the bind form and namespaced name invocation).

A word, can be one of the following:

- literal
- name
- sequence
- backquote
- `match` and `if` forms
- function form
- bind form
- type declaring form

3.2 Rules for Evaluation

When evaluating a Towel program, each word is evaluated in the original order:

- For integer, float, string, and atom literals, return references to them directly;
- For each list literal, return a reference to the list after you have evaluated each item of the list;
- For a backqoute, return whatever is quoted (without evaluating);
- For a sequence, create a new function out of the sequence and evaluate that function (i.e. evaluate the sequence on a new stack) and return the TOS of the new stack;
- For a function, apply[1] required arguments to it (if not, a backquoted partial applied function will be returned), then evaluate the return value of the function with the caller's stack and return the evaluated value;

[1]A stack is always created along with a function invocation, this is to avoid the potential corruption of shared stacks.

- For a name, look up the value it references to, evaluated that value and return the evaluated value.
- For `match` and `if` forms, match or test against the TOS and evaluate the word in the matched branch;
- For bind-then form, evaluate the value that gets bound and bind the value evaluated to the name, then evaluate the then section with current scope.

After evaluating the words, you push the result onto the stack.

3.3 Literal

A literal is a literal value whose type is of the data types we have talked about in chapter 2.

3.3.1 Atoms

You can create atoms by writing any lowercased letter followed by arbitrary length of characters that are not reserved punctuations and keywords.

Atoms are unique across the entire program. Because atoms are assigned with a unique integer, so you can have no more than 2^{31} of them in your program.

Atoms are good for pattern matching.[2] Note that bool types are implemented as atoms (`true\Bool` and `false\Bool`).

3.3.2 Numbers and Strings

You can create number and string literals by writing like this:

```
1 -1 2 -3 5 -8 13 -21
1.1e1 -0.1 'don\'t panic'
```

3.3.3 Lists

When creating a list literal, you must write a list of words separated with spaces in a pair of brackets, like the following code:

```
[arthur-dent ford-prefect betelgeuse]
[Spam Spam Spam]
[Spam ifne (More Spam)‘, (Less Spam)‘]
```

Note that in the last list literal, there exists an `ifne` form. This is totally valid, because `if` forms are also words, so as long as the **type** of the value the `ifne` form evaluates to match, you are good. And you should be aware that `if` forms always test against the top element of current data stack, so the name `Spam` before `ifne` has nothing to do with the value the `ifne` form evaluates to.

3.3.4 Tuples

Tuples are fixed length lists, whereas you can append new items to lists to create new lists. Create tuples like this:

```
[\ arthur-dent ford-prefect betelgeuse] "a 3-tuple of type PT_Tuple-3"
[\] "a 0-tuple of type PT_Tuple-0"
```

You can use tuple in match forms like this:

```
match [\ a b A], A
```

This match form matches a 3-tuple and if the first and second element are atom `a` and `b` respectively, it binds the third element to the name `A`. See also subsection 3.6.2 for more details on `match` form.

[2]This idea is from Erlang.

3.3.5 Literal of Algebraic Type

You can create literals of algebraic type by listing the required parameters in a pair of backet with an at symbol followed by the left bracket. And creating constructors by concatenating the constructor atom and the type name with a slash. For instance[3]:

```
[@ chapman
  [@ cleese
    [@ gilliam
      [@ idle
        [@ jones
          [@ palin nil\List]
           cons\List]
         cons\List]
       cons\List]
     cons\List]
   cons\List]
 cons\List
```

3.4 Sequence

Sequences are short-hand forms for creating anonymous functions with no arguments. You can create a sequence by writing the function body between a pair of parenthesis.

Towel also provides another kind of sequences, the shared sequences. These kind of sequences share the same context (such as stack and scope) with the caller. When creating such sequences, you add an at symbol right after the left parenthesis. For example,

```
((A B - if>0 1, 0) (A B + if<0 2, 3) Bitand) "non-shared regular sequences"
(A B - (@ if>0, 1, 0) Println) "prints 1 or 0"
```

You may want to create a shared sequence when defining tail-recursive functions, otherwise, you lose the advantage of not stacking up contexts.

3.5 Backquote

Towel evaluates and pushes everything it encounters, you can use backquotes the values to prevent Towel from evaluating them so that Towel pushes them directly onto the data stack. Backquotes are created by appending a backquote to the values you want to backquote.

You can backquote only limit types of words:

- literals
- names
- sequences
- backquotes

You can create backquoted shared sequence by replacing the parentheses with braces and dropping the at symbol. See also subsection 4.2.1 and subsection 4.2.2.

3.6 `match` and `if` Forms

In order to be a Turing-complete language, one must provide condition branching mechanisms. They are called `if` and match forms here in Towel.

[3] I know this seems ridiculous. But they are (and was) great comedians.

3.6.1 `if` Forms

Towel has 10 kinds of `if` forms for the sake of readability and convenience. They are of the same form, while differing in the predicate they use.

An `if` form contains two words separated by a comma. When evaluating an `if` form, Towel tests the TOS and see if it satisfies the condition. If the condition is satisfied, the first word (called true branch) is evaluated and the second word is ignored[4], and vice versa. By default, `if` forms does not consume TOS, see section 3.10 for more detail.

The predicates used by `if` forms are as follows:

- `if>0`, if TOS is a number and greater than 0
- `if>=0`, if TOS is a number and greater than or equal to 0
- `if<0`, if TOS is a number and less than 0
- `if<=0`, if TOS is a number and less than or equal to 0
- `if=0`, if TOS is a number and equal to 0
- `if~0`, if TOS is a number and not equal to 0
- `ife`, if the stack is empty
- `ifne`, if the stack is not empty
- `ift`, if TOS is an atom and equal to `true`
- `iff`, if TOS is an atom and equal to `false`

See chapter 4 for examples on `if` forms.

3.6.2 `match` Form

A `match` form is a type of advanced condition branching, which uses instead of simple predicates like greater than or less than but more complicated ones, for example, whether a value can be deconstructed as a single element and the rest of the list (like pattern matching `x::xs` in Haskell). As what `if` forms do, `match` forms also match against TOS.

To use `match` form, you write the `match` keyword followed by the pattern-action pairs separated by semicolons, while patterns and actions are interleaved by commas. To be more intuitive:

```
match pattern-1, action-1;
      pattern-2, action-2;
      ...
      pattern-n, action-n
```

Each pattern can be a literal, or a list of words. The rules of `match` form are:

- If a pattern is a simple literal, it matches if TOS is strictly equal to this literal.
- If a pattern is a list or tuple literal, it matches if TOS is a list or tuple respectively, and if corresponding items are strictly equal; if so, items that are names in pattern will be bound to respective value in TOS.
- In case of a list of words, the last word of the word list is called a pattern engine.
- If the pattern engine is a name, the name is invoked with the argument of L (note that this does not pop L off the caller's stack), then the function bound to the name will return a tuple of values, and for the rest of words:

[4] This is basically why you want a designated condition form

  - if the word is a value, the word matches if the value is strictly equal to the corresponding value in the returned tuple
  - if the word is a name, the name is bound to the corresponding value in the returned tuple

  If all words match, this pattern matches.

- If the pattern engine is an algebraic data type constructor, the values in the parameter are tested to see if they are equal (pattern matches if this is true), while the names are bound to the respective values. This is called **deconstruction**.

If a pattern matches, the respective action is evaluated. See chapter 4 for a concrete example on pattern matching in Towel.

Note that actions can only contain certain types of words for the sake of unambiguous grammar:

- name
- backquote
- literal
- sequence

3.7 Function Form

You use function forms to define anonymous functions, you may want to use bind forms jointly to create recursive functions. To define a function, first type the keyword `fun`, and a list of argument declarations and finally a word for the body of the function. You can optionally tag a type to the argument by enclosing the type in a pair of parentheses after the name of the argument. If not, types will be inferred. If not enough information was provided and the type cannot be inferred, an error will occur.

Because function form creates and evaluates function in place, the following code is valid[5]:

```
fun A B,
  (A B fun X, (3 X +))
```

See chapter 4 for concrete examples.

3.7.1 Tail Recursive Function Calls

Any practical functional programming language provides tail recursion optimization. So does Towel. However, Towel is unable to identify[6] whether a function call is tail recursive, so users are responsible for tagging tail recursive calls with an at symbol at the end of the name of the function, like this:

```
bind Loop fun F It End,
  (@ - if=0 (It F),
           (@ It F F It 1 + End Loop@))
then (("looping" Println) 1 10 Loop).
```

3.8 Bind Form

Use bind forms to add new name bindings in current scope. Names can be bound to any kind of values such as functions, atoms, and all kinds of literals. Simply type `bind` followed by the name and the value. Bind forms have a compulsory `then` clause, which is followed by a word. You can do your computation under the name scope after this name binding in the `then` clause. Top-level name bindings, i.e. names bound by the outmost bind form, are visible across modules, you may want to take advantage of this behavior.

If you encountered a situation where names bindings must be done in a cross referencing manner, use `also` clause. For example:

[5]But not semantically correct.

[6]I admit that it's not very difficult to implement this, but we are undermanned.

```
bind A 5
then bind A (B 4 -)
     also B (A 5 +)
     then (A B -)
```

3.8.1 Namespace

Namespace[7], or module, in Towel, is a set of names. Mechanisms like this prevent names of various files from colliding into each other.

To open and close a namespace (in other words, import and unimport a module), use `Import` and `Unimport` function. To invoke a name in a certain namespace, interleave the names with a backslash. You can invoke a namespace from another namespace, resulting in a chained list of names.

For instance:

```
bind Library ("library" Import)
then bind Towel Towel\The-guide\Library
  "bind the name Towel from namespace The-guide from namespace Library"
then (Towel Println
      Library Unimport)

... some more code ...
```

3.9 Algebraic Data Type Declaration

Algebraic data types will be basically the same with that of OCaml except the syntax is different. The syntax here in Towel is designed in a postfix fashion. First, use `type NAME` to bind the type you are declaring to the name `NAME`. Then declare constructors using atoms. Before the constructor, put the types of parameters in a list with an at symbol in the beginning. If the types of parameters are parameterized, put the parameters in a pair of braces before the type, like this:

```
type Option [@ a] some, none
type List [@ a {a}List] cons, nil
type BinTree [@ {a}BinTree {a}BinTree] tree, [@ a] leaf
```

You can only put atoms, parameterized names (potential namespaced) into the parameter list. See also subsection 4.2.3.

3.10 Switches

Towel provides some switches to change the default behavior of the compiler or the virtual machine:

- `hungry`
- `share-stack`
- `optimize-seq`, on by default

If you want to turn on/off these switches, enter the switch names on first line followed by the word `on` and a period, leave an empty line next to it, then go on with your code, like this.

```
hungry share-stack on.

bind Something-new (1 2 -)
then Something-new.
```

[7]Namespace won't be implement until later versions.

If `hungry` is turned on, `if` and `match` forms will consume the TOS they test against when the test finishes, i.e. immediately after falling in particular branches or after failing to fall in any of the branches. This is useful when you want to be thrifty about stack spaces.

When `share-stack` is on, every function (including sequence) uses the same stack, you get more classic stack-based language programming experience[8] out of this switch, but you may want to make sure functions don't leave extra elements on the stack, so you'd better turn on `hungry` switch along with this.

If you turn on `optimize-seq` switch, when you create a sequence as the body of a function, `if` or `match` form, this sequence is optimized to disappear, leaving the body of it as the body of the form, because the point of putting a sequence here as the form body is purely syntactic: a list of words must be quoted by a pair of parentheses so that Towel knows where the function body ends.[9] However, by doing so, you may risk polluting the name scopes and this may not be what you want.

[8] And probably faster execution, because the context switching is done a lot faster without data stacks pushing and popping.
[9] Not knowing where to end is called ambiguity in compiler jargon.

Chapter 4

Examples

The `traverse` tool produces correct syntax tree with these examples. And be aware that these examples are not guaranteed to run properly with switch `hungry` and `shared-stack` turned on.

4.1 Concrete Examples

4.1.1 Quicksort

```
bind Quicksort fun _,
  match
    [], [];
    Head Tail ::,
      (Tail (Head <) Filter Quicksort
       [Head]
       Tail (Head >=) Filter Quicksort
       ++3)
then ([5 4 3 2 1] Quicksort).
```

Note that `++3` is a trinary version of function `++` (list concatenation), you can always replace `++3` with two `++`s. Also the second pattern action is written as a sequence, which creates an anonymous function whose body is the forms in the sequence, then the anonymous function is evaluted.

4.1.2 Greatest Common Divisor

```
bind Greatest-common-divisor fun X(Int) Y,
  (- if=0 X,
     if>0 (X Y - Y Greatest-common-divisor@),
          (X Y X - Greatest-common-divisor@))
then (42 24 Greatest-common-divisor).
```

Note that `X` and `Y` are already in the stack by default (because Towel pushes arguments onto stack), so we can immediately evaluate function `-`.

Pay attention to the tail recursive call here. And because `optimize-seq` is on by default, we can use regular sequence here as the body of the function form and `if>0` form.

4.1.3 Fibonacci Numbers

```
bind Fib fun A B N,
  if=0 A, (A B + A 1 N - Fib@)
then (1 1 10 Fib).
```

Note how tail recursive calls are done by adding an at symbol at the end of `Fib`.

4.2 Advanced Examples

4.2.1 Backquotes

```
bind SomeFun fun A,
  if~0 +‘, -‘
then bind AnotherFun fun A B,
  (A B A SomeFun)
then (1 5 AnotherFun).
```

A quick explanation: when `SomeFun` is called with `A`, it returns either evaluated backquoted name `+` or `-`, in other words, it returns either name `+` or `-`.

What the **returning** actually does is that it cleans up the current function, and pushes whatever is on top of the stack (a name `+` or `-` in this case) of the current function (in this case, it is the stack of `SomeFun`) onto the caller's stack (in this case, it is the stack of `AnotherFun`). And finally jump to the instruction next to the last one. So there is a name `+` or `-` on top of the function `AnotherFun`.

And because we are evaluating return values of functions, `+` and `-` are evaluated (derefenced to) some function values, for example `fv1:0x0001` and `fv2:0x0002`. And again because we are evaluating values that the names are pointing to, one of `fv1:0x0001` and `fv2:0x0002` is called[1] with `A` and `B`, thus resulting in either adding or substracting `A` and `B`.

4.2.2 Macro

```
bind A-Macro fun,
  (@ if~0 +, -)‘
also B-Macro fun,
  {if~0 +, -}
then bind AnotherFun fun A B,
  (A B A-Macro)
then (1 2 AnotherFun).
```

A quick explanation: `(@ if~0 +, -)` is returned as we want, so a sequence (which is essentially an anonymous function that test whether the value on top of the stack is zero), and by adding a `@` we define the sequence to be a shared sequence which shares the same stack with the caller, so the overall effect is like we have done a code replacement.

`B-Macro` is a short-hand version of `A-Macro`.

You can define `++3` we talked about before as follows:

```
bind ++3 fun, {++ ++}
then ([1 2 3] [2 3 4] [3 4 5] ++3).
```

4.2.3 Algebraic Data Type

```
type BinTree [@ {a}BinTree {a}BinTree] tree,
             [@ a] leaf
also MyList [@ a {a}MyList] cons,
            nil
then bind A [@ [@ [@ 42 nil\MyList] cons\MyList] leaf\BinTree
               [@ [@ 42 nil\MyList] cons\MyList] leaf\BinTree]
            tree\BinTree
then (A export).
```

This is equivalent to the following OCaml code:

[1] It is worth mentioning that because we have exited `SomeFun`, we are now evaluating the function value with the stack of `AnotherFun`

```
type 'a bin_tree = Tree of a bin_tree * a bin_tree
                 | Leaf of a;;
type 'a my_list = Cons of a * a my_list
                | Nil;;
let a = Tree(Leaf(Cons(42, Nil)), Leaf(Cons(42, Nil)))
```