TEAM

Language Reference Manual

Wenlu (Lulu) Zheng: lulu.zheng@tufts.edu
Yingjie Ling: yingjie.ling@tufts.edu
Saurav Gyawali: saurav.gyawali@tufts.edu
Naoki Okada: naoki.okada@tufts.edu

Contents

1 Introduction to TEAM

TEAM (Text Extraction And Manipulation) is a domain specific programming language designed for text processing, data extraction, and report generation. With its straightforward syntax and various built-in functions, TEAM offers a clean layer of abstraction for users to perform tasks that are often cumbersome to do in general purpose languages.

TEAM is strongly and statically typed and imperative. Like many other modern languages, TEAM supports four primitive types: `int`, `string`, `bool`, and `char`. Moreover, TEAM has container type `list`. For easy file I/O, we introduce the `file` type to our language. Control flow is carried out by `for` loops, `while` loops, and `if` statements.

The various built-in functions along with the support for regular expressions and file I/O free the users from the ordeal of repetitive tasks such as file parsing and string manipulation. Additionally, TEAM recognizes functions as first-class citizens, which allows users to expand language features for more complicated tasks with minimal effort. TEAM also provides a standard library for basic operations on strings and lists.

2 Language Reference Manual

2.1 LRM Reading Directions

Different font sizes and styles are used throughout this LRM for a clear presentation of its content. In general, prose will be written in plain font like this: "Lorem ipsum dolor sit...". Short snippets and inline codes appear in this font: `int x = 5`. Sometimes, we use italics to denote grammatical placeholders like: *type*, *expression*, and *statement*. Finally, large code blocks are formatted with special syntax highlighting like this:

```
1 string language = "TEAM";
```

Whenever a function is described in detail, the types of the parameters and the return values are explicitly stated in parentheses.

2.2 Lexical Elements

There are six kinds of tokens: identifiers, keywords, constants, strings , operators and other separators. Blank spaces, newlines and comments are generally ignored.

2.2.1 Identifiers

Identifiers must begin with a lowercase or uppercase letter followed by letters, digits, and _.

2.2.2 Keywords

The following are reserved keywords: `if, else, elif, while, for, end, int, bool, char, string, float, list, file, void, and, or, not, in, return, true, false, break, continue`.

2.2.3 Constants

There are two types of constants:

1. Integer constant: a sequence of digits where the leftmost digit is a number between 1 and 9, and every other digit is an integer between 0 to 9.

2. Floating constant: consists of an integer part, a dot denoting a decimal point, and a fraction part, which are all required.

2.2.4 Strings

A string consists of a sequence of characters surrounded by double quotation marks.

2.2.5 Operators

Operators include: `+`,`-`, `*`, `/`, `%`, `^`, `+=`, `-=`, `*=`, `/=`, and `%=`.
More details for each operator are given below in section 2.4.

2.2.6 Separators

Semicolons are used to denote the end of statements.

2.2.7 Comments

Single-line comments are followed by `//`, and block comments are surrounded by `/*` and `*/`, that is, text after `//` in the same line, and any text between `/*` and `*/` are ignored.

2.3 Data Types

2.3.1 Primitive Types

TEAM supports the following six primitive data types:

1. `int`: The 32-bit signed integer.
2. `float`: The 64-bit floating point number.
3. `string`: A sequence of characters.
4. `bool`: The 1-bit boolean.
5. `char`: A character.
6. `file`: A file handle for file reading and writing.

The grammar for primitive types is:

```
  INT    { Int }
| FLOAT  { Float }
| BOOL   { Bool }
| CHAR   { Char }
| STRING { String }
| FILE   { File }
```

All variables should be initialized using this syntax:

type `var = literal;`

2.3.2 Collection Data Types

To provide easy and intuitive access to a collection of data, TEAM offers `list` as a container type. Lists are ordered and store one or more elements of the same type. The size of a list is dynamically determined, but the type of the elements a list stores has to be declared upon initialization. A list is initialized using the syntax: `list <`*type*`> var_name = []`. An element of a list at index i can be easily retrieved with the following syntax:

```
// Note that TEAM uses zero-based indexing
list <int> arr = [0, 1, 2, 3, 4, 5];
int first_element = arr[0]; // => 0
```

Team also supports list slicing with the following syntax:

```
// Note that the right bound is not included
list <int> arr = [0, 1, 2, 3, 4, 5];
list <int> first_three_elements = arr[0:3]; // => [0, 1, 2]
```

The grammar for type `list` is:

```
  | LIST LT typ GT { List $3 }
```

TEAM offers two essential built-in functions for list:

- `length(arr)`: a function that determines the length of a list.
  - Parameter:
    - `arr` (`list <`*type*`>`): a list
  - Returns: length of the list (`int`)
- `append(arr, ele)`: a function that appends a new element to the end of a list. Note that lists in TEAM are immutable, so the original list is unmodified in the end of the function call.
  - Parameter:
    - `arr` (`list <`*type*`>`): a list
    - `ele` (*type*): the element to be appended
  - Returns: a new list (`list <`*type*`>`)

The following code demonstrates the two built-in functions:

```
list <int> arr = [0, 1, 2, 3, 4, 5];
int arr_len = length(arr); // => 6
list <int> arr_new = append(arr, 6);
// arr: [0, 1, 2, 3, 4, 5]
// arr_new: [0, 1, 2, 3, 4, 5, 6]
```

2.4 Expressions and Operators

An expression is made up of one or more operands and zero or more operators. Operands can be literal values, a variable to a value, function calls, or a combination of these. Parentheses are used to group subexpressions as part of a bigger expression. A semicolon terminates an expression. Here are some examples:

```
35;
-42 + 6;
(9 - 2) / square(3);
```

2.4.1 Arithmetic Operators

Usage of arithmetic operators is straightforward. All of the following operators group left-to-right. Here is a list of arithmetic operators:

1. `+` is used for addition. If both operands are `int`, the result is `int`. If both are `float`, the result is `int`. If one is `int` and one is `float`, the former is converted to a `float` and the result is `float`. No other type combinations are allowed.

2. `-` is used for subtraction or negation. When used for subtraction, the same type considerations as for `+` apply. When used for negation, the result is the negative of the expression, and has the same type. The type of the expression must be `int` or `float`. When used for negation, `-` groups right to left.

3. `*` is used for multiplication. It has the same type considerations as `+`.

4. `/` is used for division. It has the same type considerations as `+`.

5. `%` is used for modular division. Both operands must be an `int`, and the result is `int`.

6. `^` is used for exponentiation. Both operands must either be `float` or `int`.

The grammar for the above arithmetic operators is:

```
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr EXP    expr { Binop($1, Exp,   $3)   }
  | expr MOD    expr { Binop($1, Mod,   $3)   }
```

2.4.2 Assignment Operators

Assignment operators store the value of the right operand into the variable specified by the left operand.

The left operand can either be a variable that is already declared or one that is being declared and initialized in this expression. It cannot be a literal value.

The assign operator (=) can be used to store values of list type. All assignments are pass by value and group right-to-left. The type of an assignment expression is that of its left operand.

```
int x;
x = 5;
float y = 5.9;
list <string> stringList = [];
```

Compound assignment operators combine the standard assign operator with another binary operator. The type rules for compound assignment operators follow their corresponding arithmetic operators, which are discussed in the section above. TEAM supports the following compound assignment operators:

1. += adds the value of the right operand to a variable and assigns the result to the left operand.

2. −= subtracts the value of the right operand from the left operand and assigns the result to the left operand.

3. *= multiplies the values of the two operands and assigns the result to the left operand.

4. /= divides the value of the left operand by the right operand and assigns the result to the left operand.

5. %= performs modular division on the two operands and assigns the result to the left operand.

The grammar for compound assignment operators is:

```
  | ID ADDASN expr { AssignOp($1, Add, $3) }
  | ID SUBASN expr { AssignOp($1, Sub, $3) }
  | ID MULASN expr { AssignOp($1, Mult, $3) }
  | ID DIVASN expr { AssignOp($1, Div, $3) }
  | ID MODASN expr { AssignOp($1, Mod, $3) }
```

2.4.3 Relational Operators

The relational operators compares two operands structurally and returns a Boolean value. All relational operators in TEAM group left-to-right. The relational operators supported by TEAM are listed below:

1. == tests the operands for equality.

2. != tests the operands for inequality. The == and != operators are lower in precedence than the other relational operators.

3. < tests if the left operand is less than the right.

4. > tests if the left operand is greater than the right.

5. <= test if the left operand is less than or equal to the right.

6. >= tests if the left operand is greater than or equal to the right.

The grammar for the above relational operators is:

```
  | expr EQ  expr { Binop($1, Equal, $3)   }
  | expr NEQ expr { Binop($1, Neq,   $3)   }
  | expr LT  expr { Binop($1, Less,  $3)   }
  | expr LEQ expr { Binop($1, Leq,   $3)   }
  | expr GT  expr { Binop($1, Greater, $3) }
  | expr GEQ expr { Binop($1, Geq,   $3)   }
```

2.4.4 Logical Operators

Logical operators test the Boolean value of a pair of operands. TEAM supports the ones below:

1. `and` is used to evaluate logical and. Groups left-to-right.

2. `or` is used to evaluate logical or. Groups left-to-right.

3. `not` flips the Boolean value. This operator is applicable only to `bool` typed expressions. Groups right-to-left.

The grammar for the above logical operators is:

```
  | expr AND expr { Binop($1, And,   $3)   }
  | expr OR expr  { Binop($1, Or,    $3)   }
  | NOT expr      { Unop(Not, $2)      }
```

2.4.5 Range Operator

The range operator (`..`) creates a range of successive integers. The left boundary is included while the right is not. The range operator is non-associative.

Here is an example of how the range operator is used:

```
// prints 0, 1, 2, 3, 4
for int i in 0..5:
  println("%d", i);
end
```

The grammar for the range operator is:

```
  | expr RANGE  expr { Binop($1, Range, $3) }
```

2.4.6 Slicing

Slicing is used to extract a substring from a string or a sublist from a list. A string or list is followed by `[`*index1*`:`*index2*`]`, where *index1* and *index2* are integers that denote indices. A slice expression will return a substring/sublist that consists of elements starting at *index1* and ending at *index2* − 1. If *index1* and *index2* are left blank, it is implied that *index1* is 0 and *index1* is the length of string or list. Below are some examples that demonstrate how slicing works:

```
string greeting = "hello";
print("%s", greeting[1:4]); // Prints "ell"

string state = "Massachusetts";
print("%s", state[:4]); // Prints "Mass"

list <string> classes = ["Compilers", "Security", "Algorithms", "
   Entrepreneurship", "Nutrition"];
list <string> csClasses = classes[:3];

for class in csClasses:
    print("%s ", class); // Prints "Compilers Security Algorithms"
```

Here is the grammar for slice expressions:

```
| ID LSQUARE expr RSQUARE { SliceExpr($1, Index($3)) }
| ID LSQUARE expr COLON expr RSQUARE { SliceExpr($1, Slice($3, $5))}
| ID LSQUARE COLON expr RSQUARE { SliceExpr($1, Slice(IntLit 0, $4))}
| ID LSQUARE expr COLON RSQUARE { SliceExpr($1, Slice($3, End)) }
```

2.4.7 Operator Precedence

The rules of precedence determine the order of evaluation of terms in expressions that contain multiple operators. Expressions with multiple of the same operators are evaluated based on the operators' associativity. The following is a set of operator precedence rules, presented in order of highest precedence to lowest precedence. Operators that are shown together on a line have the same precedence.

1. `not`
2. `^`
3. `*, /, %`
4. `+, -`
5. `..`
6. `>, <, >=, <=`
7. `==, !=`
8. `and`
9. `or`
10. `=, +=, -=, *=, /=, %=`
11. `->`

The associativity of the operators are shown below:

```
%left ARROW
%right ASSIGN ADDASN SUBASN MULASN DIVASN MODASN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%nonassoc RANGE
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left EXP
%right NOT
```

2.5 Statements

Statements are executed in sequence from top to bottom. The grammar for statements is:

```
  | vdecl SEMI { $1 }
  | expr SEMI { Expr $1 }
  | RETURN expr_opt SEMI { Return $2 }
  | IF internal_if { $2 }
  | FOR expr IN expr COLON stmt_list END { For($2, $4, Block(List.rev
    $6)) }
  | WHILE expr COLON stmt_list END  { While($2, Block(List.rev $4)) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }
```

2.5.1 expression statement

Most statements have the form:

expression

A semicolon is used to indicate the end of a statement.

2.5.2 `if` statement

The `if` statement is a conditional construct that evaluates statements based on an expression that evaluates to Boolean. `if` statements can have the following three forms:

1. `if` *expression* `:`
   stmt_list
   `end`

   In this case, if *expression* evaluates to `true`, *stmt_list* gets evaluated from top to bottom.

2.

```
if expression:
    stmt_list1
else:
    stmt_list2
end
```

In this case, if *expression* evaluates to `true`, *stmt_list1* will be evaluated; otherwise, *stmt_list2* will be evaluated.

3.

```
if expression:
    stmt_list
elif expression:
    stmt_list
(more elif statements)
else:
    stmt_list
end
```

In this case, the expressions are evaluated sequentially from the top until one is found to be true, at which point the corresponding *stmt_list* is executed. If none of the expressions are evaluated to be `true`, then the *stmt_list* corresponding to `else` is executed. Note that the `elif` and `else` are optional.

2.5.3 `while` statement

The `while` statement is a looping construct that continuously evaluates a statement conditionally based on an expression that evaluates to a Boolean. `while` statement has the following form:

```
while expression:
    stmt_list
end
```

While *expression* evaluates to true, the statements represented by *stmt_list* keep getting evaluated. Once *expression* evaluates to false, the statements stop being evaluated.

2.5.4 `for` statement

A `for` statement is a looping construct that iterates over a list. It has the following form:

`for` *type var_name* `in` *expression:*
    stmt_list
`end`

The *expression*, which has the type `list<`*type*`>`, is evaluated first. *stmt_list* is evaluated once for every item in *expression*. On the first iteration, *var_name* is assigned the first element of the list and *stmt_list* is evaluated. At each subsequent i^{th} iteration, the variable is assigned the i^{th} element of the list and *stmt_list* is evaluated.

2.5.5 `break` statement

The `break` statement is used inside a looping construct to break from the loop regardless of the condition of the loop. It has the following form:

`break;`

When the `break` statement gets evaluated inside a loop, all the statements after it inside the loop are skipped, and the execution moves out of the looping construct.

2.5.6 `continue` statement

The `continue` statement is used inside a looping construct to skip the current iteration of the loop. It has the following form:

`continue;`

After the `continue` statement gets evaluated, all the statements after it inside the loop are skipped, and the execution moves on to the next iteration of the loop.

2.5.7 `return` statement

The `return` statement is used inside a function to exit function. It either returns an expression or nothing. It has the following forms:

1. `return` *expression*`;`

   In this case, *expression* gets evaluated first, the execution exits the current function, and the result is returned.

2. `return;`

   This `return` statement is used in `void` functions, and it simply exits the current function without returning anything.

2.6 Functions

Functions allow you to separate parts of your program into distinct snippets of code that can be called subsequently. Since functions are treated as first class citizens, they can be assigned as values to identifiers, passed as argument to other functions, and returned as values from other functions. A function is globally-scoped and must be defined before it is used.

2.6.1 Function Definitions

A function definition consists of information regarding the function's name, return type, types and names of parameters, and the body of the function. The function body consists of a series of statements and is terminated by the keyword `end`. The types of the parameters and the return type are required at the beginning of the formal function definition. If a function does not return anything, use the keyword `void` in place of the return type.

Here is the general form of a function definition:

type `foo(`*type* `p1,` *type* `p2, ...):`
    statements
`end`

Below defines a simple function that adds two numbers.

```
int add(int x, int y)
    return x + y;
end
```

2.6.2 Calling Functions

A function is called by using its name and supplying any parameter it requires. Here is an example that calls the `add` function defined above:

```
add(1, 2);
```

Calling an function is just another expression, which means that it will return a value and can be used as part of other expressions.

Here is an example where the returned value is assigned to a variable:

```
int x = add(1, 2);
```

2.6.3 Function Parameters

Function parameters can be any expression — a literal value, another function, or a variable. A parameter's value is passed into the body of the function, so it cannot

be changed by changing the value of the local copy. Function parameters can have any data type.

Here is an example to demonstrate that the value of a formal parameter remains unchanged even if we assign a new value inside the function.

```
void swap(int x, int y)
    int temp = x;
    x = y;
    y = temp;
end

x = 2;
y = 3;
swap(x, y);
println("%d", x); // 2
println("%d", y); // 3
```

Once the function definition has been written, the number of arguments stay fixed. Because functions are recognized as first-class citizen, they can be passed as argument to other functions. The code snippet below demonstrates an example of higher order functions in TEAM. When a function serves as a formal parameter, its type is specified by the types of its formal parameters and the return type separated by the arrow keyword (->).

```
string shout(text):
    return upper(text);
end

string whisper(text):
    return lower(text);
end

/*
greet takes a function that takes a string
and returns a string (string -> string)
and returns nothing (void)
*/
void greet((string->string) func):
    string greeting = func("Hello");
    print("%s", greeting);
end

greet(shout); // prints "HELLO"
greet(whisper); // prints "hello"
```

The grammar for a function type is:

```
  | LPAREN typ RPAREN { $2 }
  | typ ARROW typ { Func($1, $3) }
```

2.6.4 Recursive Functions

Functions can be recursive. Here is an example that computes the nth term in the Fibonacci sequence:

```
int fib(n):
    if n <= 1:
       return n;
    else:
       return (fib(n-1) + fib(n-2));
    end
end
```

A recursive function can run infinitely until the program is interrupted or runs out of memory. It is the programmer's job to ensure that a recursively defined function has a well-constructed base case.

2.7 Scope

TEAM is statically scoped. A variable used inside a function that is not local to the function scope resolves to a variable defined outside the function based on the position of the variable in the program.

```
int y = 5; // scope of y is entire program

int global():
    print("%d\n", y); // prints 5
end

int local():
    int y = 10;
    print("%d\n", y); // prints 10
end
```

Variable `y` is referenced inside function `global`, but is not defined inside `global`. Therefore `y` resolves to the first variable named `y` that is defined in the enclosing scope of `global`. Variable `y` is locally defined inside `local`.

```
int y = 5;

int f():
    return y + 1; // y not found in enclosing function
end

void g(f):
    y = 10;
    print("%d", f()); // prints 6
end
```

When `y` is referenced inside of `g`, it resolves to the variable in the closest enclosing space, which is the global variable `y`.

Formal parameters have local scope within a function.

```
void f(y):
    print("%d\n", y + 1); // prints the argument value passed
end
```

An error is thrown when there are duplicate declarations for the same identifier in the current context, that is, there cannot be multiple global variables declared with the same name. The following example demonstrates this error:

```
int age = 10;
print("%d", age);

int age = 5; // ILLEGAL!
```

2.8 Program Structure

A program in TEAM is a list of statements. TEAM does not require a main function. Unless specified otherwise, statements are evaluated from the top of the file.

2.9 Formatted Output

The `print` and `println` functions can be used to print any number of arguments. They accept a string as the first argument, and allow any subsequent arguments to be printed according to the specifications in the template string. Most characters in the string are printed as is, with the exception of two character sequences starting with `%`. Conversion specifications are specified by the `%` character in the template string that results in subsequent arguments to be formatted and written to the standard output.

The use of `%d` specifies that an int argument should be printed in decimal notation. The `%f` conversion allows floating-point numbers to be printed in fixed-point notation. The `%s` conversion specifies printing of a string argument.

Here is an example of how an output can be properly formatted:

```
int i = 5;
float f = 3.14159;
print("The number is %d", i);
println("The first five digits of pi are %f", f);
```

3 Built-in Functions

The following are built-in functions supported by TEAM:

3.1 List

- `len(list)`: a function that takes in a list and returns the number elements in the list
  - Parameter:
    - `list` (`list <`*type*`>`): a list
  - Returns: an integer (`int`) that indicates the number of elements in the list.

3.2 File I/O

It is straightforward to work with files with TEAM. To read from a file, a variable of file type should be defined first; this variable serves as a buffer for the file. Then the content of the file is transferred to the buffer with the built-in `open` function. Quick access to the file content can be achieved with the `readline` function.

- `open(file_name, mode)`: a function that creates a new file or opens an existing file
  - Parameter:
    - `file_name` (`string`): the path of the file to be opened
    - `mode` (`string`): the access mode
  - Returns: a file handle
  - Note: If mode is not specified, the file is opened in read mode by default. Other modes can be explicitly mentioned:
    1. `r`: opens a file for reading only.
    2. `r+`: opens a file for reading and writing. The stream is positioned at the beginning of the file.
    3. `w`: opens a file for writing only and overwrite any existing file with the same name. If the file doesn't exist, a new file is created.
    4. `w+`: opens a file for reading and writing. The stream is positioned at the beginning of the file. The file is created if it does not exist, otherwise it is truncated.
    5. `a`: opens a file for appending new information to it. If the file doesn't exist, a new file is created.
- `readline(file_handle)`: read one complete line from the given file specified by handle `f`
  - Parameter:
    - `file_handle` (`file`): the file handle that refers to the file
  - Returns: the current line of the file with `\n` in the end

- `write(file_handle, content)`: write the given content to the file specified by file_handle
  - Parameter:
    - `file_handle` (`file`): the file handle that refers to the file
    - `content` (`string`): the string to be written
  - Returns: nothing (`void`)
- `close(file_handle)`: closes the opened file specified by file_handle
  - Parameter:
    - `file_handle` (`file`): the file handle that refers to the file
  - Returns: nothing (`void`)

The following code snippet demonstrates the usage of the aforementioned file I/O related functions.

```
file a = open("abc.txt");
file b = open("myText.txt", "w+");
file f;

f = open("example.txt");
string line = "";

while not eof(f):
    string line = readline(f);
    print("%s\n", line);
end

close(f);
```

4 Standard Library Functions

The following standard library functions are all implemented in TEAM. Those functions demonstrate how users can expand the power of TEAM using well-constructed small blocks of built-in functions. Those standard library functions also provide users with an interface to carry out essential tasks with lists and strings.

4.1 List Library

- `contains(arr, ele)`: a function that determines if an element exists in the list.
  - Parameter:
    - `arr` (`list` <*type*>): a list
    - `ele` (*type*): the element to be searched for
  - Returns: a boolean value (`bool`) that indicates the state of existence.
- `remove(arr, ele, all)`: a function that removes an element from the list. The original list is unmodified.
  - Parameter:
    - `arr` (`list` <*type*>): a list
    - `ele` (*type*): the element to be removed
    - `all` (`bool`): when all is `true`, the entire list is traversed, and all duplicates will be removed; otherwise, only the first occurrence is removed. The original list is unmodified.
  - Returns: a new list (`list` <*type*>)
- `reverse(arr)`: a function that reverses a list. The original list is unmodified.
  - Parameter:
    - `arr` (`list` <*type*>): a list
  - Returns: a new list (`list` <*type*>)
- `concat(arr1, arr2)`: a function that concatenate two lists. The original lists are unmodified.
  - Parameter:
    - `arr1` (`list` <*type*>): a list
    - `arr2` (`list` <*type*>): a list
  - Returns: a new list (`list` <*type*>) with two lists concatenated.

The following code snippet demonstrates the usage of the aforementioned list related functions.

```
list <int> courses_taken = [11, 15, 40, 61];
list <int> courses_to_take = [105, 160, 170];

if contains(courses_taken, 160):
    print("This kid knows algorithms\n");
else:
    print("This kid does know algorithms\n");


list <int> courses_required = concat(courses_taken, courses_to_take);
// courses_required is [11, 15, 40, 61, 105, 160, 170]
// courses_taken is still [11, 15, 40, 61]
// courses_to_take is still [105, 160, 170]

list <int> courses_reversed = reverse(courses_required);
// courses_required is still [11, 15, 40, 61, 105, 160, 170]
// courses_reversed is [170, 160, 105, 61, 40, 15, 11]

list <int> cs_mimnor = remove(105,
                              remove(160, courses_required, true),
                              true);
// courses_required is still [11, 15, 40, 61, 105, 160, 170]
// cs_minor is [11, 15, 40, 61, 170]
```

4.2 String Library

The `String` library provides essential functions to perform string manipulation.

- `split(str, delim)`: a function that takes in a string and a delimeter and returns a list of substrings divided by the delimeter.
  - Parameter:
    - `str` (`string`): a string
    - `delim` (`string`): a string
  - Returns: a list (`list <string>`) of strings that are separated by the delimeter

```
list <string> l = split("Hello,World", ",");
// l = ["Hello", "World"]
```

- `join(str_list, delim)`: a function that takes in a list of strings and combines them into one string.
  - Parameter:
    - `str_list` (`list <string>`): a list of strings
    - `delim` (`string`): a string
  - Returns: a new string (`string`)

```
list <string> l = ["Jack", "likes", "fishing"];
string s = join(l, " ");
// s = "Jack likes fishing"
```

- `reverse(str)`: a function that takes in a string and returns the reverse of the string
  - Parameter:
    - `str` (`string`): a string
  - Returns: a new string (`string`)

```
string reversed = reverse("hello world";
// reversed = "dlrow olleh"
```

- `startswith(str, head)`: a function that takes in two strings and returns true if the first string starts with the second string, false otherwise
  - Parameter:
    - `str` (`string`): a string
    - `head` (`string`): a string
  - Returns: a boolean (`bool`)

```
bool b = startswith("hello world", "hello");
// b = true
bool c = startswith("hello world", "hi");
// c = false
```

- `endswith(str, head)`: a function that takes in two strings and returns true if the first string ends with the second string, false otherwise
  - Parameter:
    - `str` (`string`): a string
    - `head` (`string`): a string
  - Returns: a boolean (`bool`)

```
bool b = endswith("chocolate", "late");
// b = true
bool b = endswith("chocolate", "mate");
// b = false
```

- `lower(str)`: a function that takes in a string and returns a string where all characters are lower case
  - Parameter:
    - `str` (`string`): a string
  - Returns: a new string (`string`)

```
string s = lower("HELLO MY FRIENDS");
// s = "hello my friends"
```

- `upper(str)`: a function that takes in a string and returns a string where all characters are upper case

- Parameter:
  - `str` (`string`): a string
- Returns: a new string (`string`)

```
string s = upper("hello my friends");
// s = "HELLO MY FRIENDS"
```

5 Regular Expressions

Regular expressions describe patterns that are used to determine if some other target string has characteristics specified by the pattern. TEAM provides robust built-in features for using regular expressions. Our features offer the ability to match strings to regular expressions and replace matches with another string. A regular expression is passed into a function as a string but is interpreted as a regular expression using the rules described below.

5.1 Meta characters

- `\`: indicates a special sequence
- `^`: matches the beginning of a string
- `$`: matches the end of a string
- `|`: alternation, allows the target to be matched over a number of patterns
- `()`: grouping, allows the target to be matched over a number of patterns specified in the parenthesis
- `.`: any character except newline character
- `*`: zero or more occurrences
- `+`: one or more occurrences
- `?`: zero or one occurrence(s)
- `[]`: a set of characters
- `{}`: denotes number of occurrences expected

5.2 Special sequences

- `\d`: matches if the target string contains a digit
- `\D`: matches if the target string contains a non-digit character
- `\s`: matches if the target string contains a whitespace
- `\S`: matches if the target string contains no whitespace
- `\b`: matches the specified characters to the beginning of a word in the target string
- `\B`: matches the specified characters somewhere in the target string, except at the beginning of a word
- `\t`: matches a tab character in the target string
- `\n`: matches a newline character in the target string
- `\0`: matches a null character in the target string

5.3 Sets

The following rules can be used to define a set (`[]`).

- `[`*num1*`-`*num2*`]`: matches any digit between *num1* and *num2*, inclusive
- `[`*letter1*`-`*letter2*`]`: matches any lower case letter between *letter1* and *letter2*, inclusive
- `[`*digit1*`,` *digit2*`, ...]`: matches any digit included in the sequence
- `[`*letter1*`,` *letter2*`, ...]`: matches any letter included in the sequence

5.4 Regular Expression Examples

The code snippet below demonstrates some of the useful regular expressions that follows the rules described above.

```
// One or more digits between 0 and 9, inclusive
string digits = "[0-9]+";

// Any string that starts with "chachacha"
string echos = "^cha{3}[.]*";

// Any string that ends with "a" or "b"
string ab = "(a|b)$";
```

5.5 Built-in Functions

- `match(regex, target)`: takes a regex and a target string and returns `true` if target matches the regex and `false` if otherwise.
  - Parameter:
    - `regex` (`string`): a string that is interpreted as a regular expression
    - `target` (`string`): a target string that will be matched against the given regular expression
  - Returns: a Boolean value (`bool`) that indicates if the target string matches the regex

```
bool b = match("(g\w+)\W(g\w+)", "guru99 get");
// b = true
```

- `find(regex, target)`: takes a regex and a target string and returns the first substring of the string that matches the regular expression.
  - Parameter:
    - `regex` (`string`): a string that is interpreted as a regular expression
    - `target` (`string`): a target string that will be matched against the given regular expression
  - Returns: If no match is found, return the empty string. Otherwise, return the first match (`string`) found.

```
string s = match("(g\w+)\W(g\w+)", "guru99 get");
// s = "guru99"
```

- `findall(regex, target)`: takes a regular expression and a target string and returns a list of all matches.
  - Parameter:
    - `regex` (`string`): a string that is interpreted as a regular expression
    - `target` (`string`): a target string that will be matched against the given regular expression
  - Returns: If no match is found, return the empty list. Otherwise, return a list (`list<string>`) of all matches found.

```
string a = findall("te", "test");
// a = ["te"]
```

- `replaceAll(regex, target, replc)`: takes a regular expression, a target string, and a replacement string and returns a string that is constructed by replacing all matches in target with `replc`. Note that the original string (`target`) is not modified.
  - Parameter:
    - `regex` (`string`): a string that is interpreted as a regular expression
    - `target` (`string`): a target string that will be matched against the given regular expression
    - `replc` (`string`): a string that will replace the matches found
  - Returns: a string (`string`) constructed from replacing all matches in target with `replc`.

```
string a = replaceAll("u", "a", "jumbo tufts", 2);
// a = "jambo tafts"
```

- `replace(regex, target, replc, count)`: takes a regular expression, a target string, replacement string, and count number and returns a string that is constructed by replacing number of matches in target with `replc` specified by `count`. Note that the original string (`target`) is not modified.
  - Parameter:
    - `regex` (`string`): a string that is interpreted as a regular expression
    - `target` (`string`): a target string that will be matched against the given regular expression
    - `replc` (`string`): a string that will replace the matches found
    - `count` (`int`): an integer that indicates how many matches should be replaced by replc
  - Returns: a string (`string`) that constructed by replacing all matches in target with `replc`.

```
string google = "www.google.com";
string facebook = replace("google", google, "facebook", 1);
// facebook = "www.facebook.com"
```