https://www.google.com/search?q=manatee&biw=1242&bih=522&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjn987L1vTRAhVCySYKHUHiAcMQ_AUIBigB#imgrc=lZU3VsrZH0F2WM:

ManiT

Manager: Akiva Dollin - acd2164
Language Guru: Irwin Li - izl2000
System Architect: Seungmin Lee - sl3254
Tester: Dong Hyeon (Paul) Seo - ds3457

Language Reference Manual

Table of Contents:

Introduction -- *3*

Types -- *3*

Lexical Conventions -- *3*

Comments ---*3*

Identifiers -- *4*

Keywords -- *4*

Literals --- *4*

Statements -- *5*

Declarations -- *5*

Expressions --- *6*

Operators -- *7*

Standard Library Functions --- *9*

Introduction:

ManiT is a C-like language which enables easy manipulation of large integers and linear systems and compiles into LLVM. This language would be extremely useful specifically for quick manipulation of 2D and 3D graphs as well as general purpose number manipulation. ManiT will include a simple garbage collection as well as type inference.

Types:

Type	**Description**
Integer	Numbers without a decimal point (also known as Integers)
Float	Number that contains a decimal point.
String	A sequence of characters
Array	A container to hold multiple data of the same datatype
Character	A character variable

Lexical Conventions:

When processing a program written in the ManiT language, the program is reduced to a sequence of tokens. There are five classes of tokens: identifiers, keywords, literals, operators, and other separators. In our language, spaces, tabs, newlines, single and multi-line comments are considered to be white-space. In general, white-space is ignored by program. Some white-space, however, is required to separate otherwise adjacent identifiers, keywords and constants.

Comments:

The ManiT language supports both single-line and multi-line or block commenting. Comments do not nest, and cannot exist within a string or character literals.

`// ...`	Single-line comment
`/* ... */`	Block comment.

Identifiers:

An identifier is a sequence of letters or digits that identifies some data that the programmer will interact with. Identifiers can be any length and use any combination of letters and numbers, but must start with a letter.

Keywords:

All types such as `int` and `float` are keywords. In addition, statement keywords such as `if`, `while`, and `for` are also keywords that are reserved.

Literals:

Any sequence of one or more digits in decimal is treated as an integer literal or constant. A negative sign is used to specify a negative integer (`-1`). The integer literal corresponds to the `int` type. A character literal consists of a single ASCII character surrounded by single-quotes. Special characters such as the newline character can be specified using the backslash (`\`) character. The character literal has the `char` type. The following escape sequences may be used in character and string literals:

Newline	`NL`	`\n`
Horizontal tab	`HT`	`\t`
Backslash	`\`	`\\`
Double quote	`"`	`\"`
Single quote	`'`	`\'`

Floating literals consist of a integer part, decimal part, and a fraction part. The integer and fraction portions consists of a sequence of one or more digits. The decimal portion delimits the integer and fraction portions and is specified using the period character (`.`). A floating literal may be written as `1.5`.

A string literal is written as a sequence of zero or more ASCII characters surrounded by double quotes. Special characters such as the newline character may be defined using the same escape sequences used for character literals. The string literal is of the `string` type.

Boolean literals are explicitly the identifiers `true` and `false`. The former represents the logical true and the latter represents the logical false. These identifiers are reserved.

Declarations:

To declare functions and variables, use the appropriate syntax.

Variable Declarations:

Variables are declared with the following structure:

```
var  <variable name> = expression;
var example = “hello”;
var anotherExample = 2;     //2
```

Once a variable is declared, it can be assigned and updated:

```
example = “different string”;
anotherExample = anotherExample + 10;       //12
```

NOTES: updating a variable must remaining type consistent:

```
anotherExample = “this is wrong”   //ERROR
```

Function Declarations:

Functions are declared in the following manner:

```
functionName ( input1, input2, … , inputN) {
        //do stuff here;
}
```

Function Calls:

```
result = functionName(input1, input 2, input 3, ..);
```

Statements:

ManiT allows for many different types of statement, all that all similar to C.

End of Statement:

All statements must be closed by semicolons:

‘;’

Expression Statement:

Any assignment or function calls are expression statements. The syntax for expression statements are:

expression;

Conditional Statement:

*If*s and *Elses* are chained to create conditional statements. Expressions are given for each case as well as statements to be executed. If the expression evaluates to true, the statements provided are executed. If the expression evaluates to false, then the else's provided statement is executed is there is an else:

if (expression) { statement; }
else { statement;}

For Statement:

The *for* structure is similar to C and Java. Three expressions must be provided. An initializer, a condition, and an increment. The *for* loop executes until the condition evaluates to false which is evaluated at the beginning of each loop. At the end of each execution, the increment is evaluated. During the loop, the provided statement is executed:

for(expressionInit; expressionCondition; expressionIncre){ statement; }

While Statement:

While expressions are evaluated after each execution of the provided statement. The statement is executed for as long as the expression evaluates to true:

while (expression) { statement; }

Return Statement:

return statements exits functions and returns to the function call. If *return* is given without an expression, the function returns to the call. If an expression is given, it is evaluated and then returned.

return;
return (expression);

Expressions:

Primary Expressions:

identifier
The type of the identifier is specified by the type identifier.
constant, null
A constant refers to data type literal. The keyword null represents a non-existing reference.
(expression)
The pair of opening and closing parenthesis can be used to group expressions
expression[expression]
Element index access. Indexing is only supported for String literals, integer literals, and float literals. The index must be an integer. If the expression is a negative integer, the indexing starts from the right. *(right → left)*

identifier[expression]

Element index access. The expression must be any integer. If the expression is a negative integer, the indexing starts from the right. *(right → left)*

identifier[start:end:increment]

Returns the elements in a list from the start position element to the end position element with increments upon each element of size increment.

identifier.feature

A feature can be replaced by one of:

.left

The .left feature is used for both Integer and Float. The .left denotes the value to the left of the decimal point.

.right

The .right feature is used for both Integer and Float. The .right denotes the value to the right of the decimal point.

.pow

The .pow feature is used for both Integer and Float. The .pow denotes the exponent of the value itself. E.g.: 3.pow(2) is 3^2. By calling pow separately, it can also return the exponent power of the value itself.

.coeff

The .coeff feature is used for both Integer and Float. The .coeff denotes the coefficient of the value itself. E.g.: a = 3; a.coeff() will return 3.

Unary operators:

Expressions with unary operators group from right-to-left.

expr++, expr--

A unary expression followed by a ++ or a -- is a unary expression. The operand is incremented or decremented by 1.

!expr

The operand of the ! operator must be a boolean type. The result is `false` if the value of the operand compares to `true` and the result is `true` if the value of the operand compares to `false`.

Multiplicative operators:

expr * expr

The * operator denotes either multiplication or multiple concatenation. If the left-hand-side expression is string or char type and the right-hand-side expression is integer type, the entire expression evaluates to a string type that contains concatenation of the left-hand-side expression right-hand-side times. Otherwise, * denotes multiplication. The expression on either side must be either integer or floating point expressions.

expr / expr

The / operator denotes division. Integer division truncates fractional portion of the result.

expr % expr

The % operator denotes the integer modulo operator. Expressions on both sides must be integers.

Additive Operators:

The operands are either integers or floating point numbers. Integers are promoted to floating point type if one is type integer and other is floating point type.

expr + expr

The + operator denotes integer and floating point addition.

expr - expr

The - operator denotes integer and floating point addition.

Equality Operators:

expr == expr, expr != expr

The == and != operators denote equal to and not equal to comparison. Both operands must be either both integers or floating point types.

Relational Operators:

expr < expr, expr <= expr, expr > expr, expr >= expr

The < and > operators denote less than and greater than comparisons. The <= and >= operators denote greater than or equal to and less than or equal to comparisons.

Boolean Operators:

expr && expr, expr || expr

The && operator denotes the AND operation and || denotes the OR operation. Both operands need to be boolean type.

Assignment Operator:

identifier = expr

The = operator denotes assignment of the expressions to a variable. The variable must be declared. NOTE: Overloading a variable with a different type expression throws an ERROR.

Ex:

```
var temp = 3;
temp = “hello”; //error
temp = 4; //4
```

Standard Library Functions

Some Built-in Functions:

map

map (Function, List 1, List 2, List 3, ..);
map (Function, Array 1, Array 2, Array 3, ..);
map (Function, List 1, Array 1, ...);

The map function takes a function and applies the function to every element in the List or Array. If there are more than 1 List or Array, the map function applies to the rest of the List or Array in the inputs.

print

print(expression);

The print function prints the expression to standardoutput.

sqrt

sqrt(n); //int
sqrt(n); //float

The sqrt function computes the square root of the integer or float n.

length

len (input);

The length function takes an input and returns the length of the input. For instance, for Strings, it returns the length of the String. For Integers, it would return the length of the (String) of Integer.

sort

sort (any list);
sort (any array);

The sort function takes either a List or an Array and sorts the List and Array in ascending order. For Integer and Float, the sort returns a List or Array in ascending elements. For Strings, the sort returns a List or Array in the ordering a standard dictionary would.

compare

compare ((Integer) element 1, (Integer) element 2);
compare ((Float) element 1, (Float) element 2);

The compare function compares two input elements. It will return an integer in which if the first element is greater than the second element, the return value is positive. If the first element is equal to the second element, the return value is zero. If the first element is less than the second element, the return value is negative.