2/22

TPL: Table Programming Language

LANGUAGE REFERENCE MANUAL

HAMZA JAZMATI

1. Introduction

Table Programming Language, or TPL, is based on the C programming language, with the main difference that TPL supports an extra data type called *Table.* On the contrary, TPL does not include all the features supported by C programming language, only a limited subset described in the rest of this document. This document will serve as a manual for TPL.

2. Lexical Conventions

Just like C, which TPL is based on, there are six types of tokens:

- Identifiers
- Keywords
- Constants
- Strings
- Expression operators
- Other separators.

White space including tabs, newlines, blanks, and comments only purpose is to separate tokens. Otherwise, they are ignored by the compiler.

2.1 Comments:

Two types of comments will be supported:

- The characters /∗ introduce a comment, which terminates with the characters ∗/ This type is borrowed from C.
- The characters // introduce a comment, which terminates at the end of the line. This is borrowed from C++.

2.2 Identifiers:

Identifiers can be described as sequence of characters that start with a letter (lower or upper case) and the rest of it can be a combination of letters and numbers. Identifiers are case sensitive.

2.3 Keywords:

The following list includes all the keywords in TPL. These keywords cannot be used as identifiers.

list
bool
else
float
int
if
string
table

while

2.3 Constants:

These constants include:

2.3.1 Integer Constants:

An integer constant is a sequence of digits. An integer is always taken to be decimal.

2.3.2 Floating Constants:

A floating constant consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. Only the fraction part can be missing.

2.3.3 Strings:

A string is a sequence of characters surrounded by double quotes ‘‘"’’. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end.

3. Conversions:

The only supported conversion in TPL is between integers and floats. Any *int* can be converted to a *float* without a significate loss. *float* to *int* conversion is supported but can produce undesirable results if the trunked *flaot* value is not in the range of integers.

4. Expressions:

4.1 Primary Expressions:

4.1.1 Identifier:

An identifier is a primary expression, provided it has been suitably declared. Its type is specified by its declaration.

4.1.2 constant:

A decimal that can be represented with *int*, or a floating point constant represented as *float. String* is also another type of constant.

4.1.3 (expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

4.2 Unary operators:

Expressions with unary operators group right-to-left.

4.2.1 – Expression:

The result is the negative of the expression, and has the same type. The type of the expression must be *int* or *float.*

4.2.2 ! expression:

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is 1. The type of the result is bool. This operator is applicable only to bool.

4.3 Multiplicative Operators:

The multiplicative operators *, /, and % group left-to-right.

4.3.1 expression * expression:

The binary * operator indicates multiplication. If both operands are *int*, the result is *int*; if one is *int* and one is *float*, the former is converted to *float*, and the result is *float*; if both are *float*, the result is *float*. No other combinations are allowed.

4.3.2 expression / expression:

The binary / operator indicates division. The same type considerations as for multiplication apply.

4.3.3 expression % expression:

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be *int*, and the result is *int*. In the current implementation, the remainder has the same sign as the dividend.

4.4 Additives Operators:

The additive operators + and − group left-to-right.

4.4.1 expression + expression:

The result is the sum of the expressions. If both operands are *int*, the result is *int*. If both are *float*, the result is *float*. If one is *int* and one is *float*, the former is converted to *float* and the result is *float*. No other type combinations are allowed.

4.4.2 expression - expression:

The result is the difference of the operands. If both operands are *int*, or *float* the same type considerations as for + apply.

4.5 Rational Operators:

The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to.

4.5.1 expression < expression

4.5.2 expression <= expression

4.5.3 expression > expression

4.5.4 expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

4.6 Equality Operators:

4.6.1 expression == expression

4.6.2 expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

4.7 expression || expression:

4.8 expression && expression:

4.9 Assignment Operators:

lvalue = expression.

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, or float.

5. Statements:

5.1 Expression Statement:

Most statements are expression statements, which have the form

expression ;

5.2 Compound Statement:

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement: { statement-list }

5.3 Conditional Statement:

The two forms of the conditional statement are

if (expression) statement

if (expression) statement else statement

In both cases the expression is evaluated and if it is 1, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered elseless if.

5.4 While Statement:

The while statement has the form

while (expression) statement The substatement is executed repeatedly so long as the value of the expression remains 1. The test takes place before each execution of the statement.

6. Lists and Tables:

The two main differences between C/C++ and TPL are the ways lists and tables are handled.

6.1 Lists:

lists are a collection of objects that do not have a fixed length. In TPL, a list can have only one type. Lists can be of any of the types: *int*, *float*, or *string*.

6.1.1 Declaring Lists:

Declaring a list can be as follows:
list string header = {“Hamza” , “Jazmati” , “Edward”, “Snowden”}
list int primes = {2,3,5,7,11}

6.1.2 Getters/Setters:

To obtain a value from the list, we use the following syntax:
primes[3]
This will get us the fourth element of the list, which is 7.
primes[4] = 13
This will set the value of the fourth element of the list to 13.

6.2.3 Appending:

To append a new value to the end of the list, we do as follows:
primes.append(17)
This statement should return the index of the newly added item, in this case, 5.

6.2 Tables:

Tables are two dimensional lists where the header is a string list and each column is a list of a specific type.

6.2.1 Declaring a Table:

table mytable = {“First_Name”, “Last_Name”, “Grade”} {string, string,int}
Column names cannot be the used more than once in the same table.

6.2.2 Getters/Setters:

To get the value of a specific cell in the table, we do as follows:
mytable[“First_Name”][0]
To get a specific column, we can use:
mytable.”First_Name”
To set a value of a specific cell in the table, we do as follows:
mytable[“First_Name”][0] = “John”

6.2.3 Appending Rows:

To append a row to the table, we use the following syntax:
mytable.appendrow (“Hamza”, “Jazmati”, 100)

6.2.3 Generate a column:

To generate a new column from other columns in the table, we use the following syntax:
mytable.newcolumn (“Grade_Out_Of_Ten” , “Grade”/10)

6.2.4 Sorting:

To sort a column according to a column, we use the following syntax:
mytable.sort(“Grade”, asc)

mytable.sort(“Grade”, desc)
asc indicates that the sorting is ascending. dsec indicates descending.

6.2.5 Getting Row Count:

To get row count of the table, mytable.rowcount.