Turtle Tango (TT)

Language Reference Manual

Jeff Bender | UNI: jrb2211 | COMS W4115
6/29/2012

Contents

1. Introduction

Tango (TT) is a line-drawing language that uses audio input to modulate graphical output. Turtle, the relative cursor, follows programming instructions and the modulations of a program's associated Tune to generate a PostScript representation of the specified dance in his world, the Disco.

A TT program consists of variable and routine declarations composed in a text file with a .tt extention. Control initiates in the `main~ filePath` routine, which accepts as an argument the `FilePath` to .wav audio input. The output of compilation is a PostScript file with extension .ps depicting Turtle's dance in graphical form when viewed in a PostScript viewer.

2. Lexical Conventions

2.1. Tokens

The five classes of tokens are identifiers, keywords, constants, operators, and separators. Whitespace is ignored; TT is free-format.

2.2. Comments

The sequence `'/*'` begins a comment, which ends with the sequence `'*/'`.

2.3. Identifiers

An identifier is a sequence of letters `'a'-'z'` and `'A'-'Z'` and digits `'0'-'9'`. Identifiers are case-sensitive, e.g. `'a'` is not the same as `'A'`.

2.4. Keywords

The following keywords are reserved:

```
int, routine, return, if, else, for, while, break, continue
setDiscoSize, setDiscoColor
fd, bk, lt, rt, setPosition, setOrientation, setRate
pu, pd, setColor, setLineStyle, setLineWidth
cue, pause, play, stop, setVolume
```

2.5. Constants

Constants are integers consisting of a sequence of digits `'0'-'9'`.

3. Scope

Variable and routines are statically scoped.

Variable declarations are allowed only within routines; all variables are considered local. Variables can only be accessed within the routine in which they were declared, and cannot be accessed before declared. Variable names must be unique within any single scope.

Routines must have globally unique names and cannot be declared within other routines. A routine can be called before it is declared.

4. Statements

Statements are executed in sequence unless otherwise noted.

4.1. Expression

Expressions are described in detail in section 5. An expression statement consists of an expression followed by `';'`.

4.2. Compound

A compound statement allows several statements to be used in places in which one statement is expected, but more are needed. It consists of a `'{'` followed by zero or more statements and a closing `'}'`.

4.3. Conditional

Conditional expressions take one of two forms:

```
if (expression) statement
if (expression) statement else statement
```

The expression is first evaluated. Any resulting value other than zero is considered true, and the following statement is executed. If the expression evaluates to zero, in the first conditional form, the statement is skipped; in the second conditional form, the statement following the `else` is executed. `else` binds to the most recent `if` not associated with another `else`.

4.4. Loop

4.4.1. For

The `for` statement has one form:

```
for (expression1; expression2; expression3) statement
```

`expression1` specifies loop initialization. `expression2` specifies a test, executed before each iteration, that evaluates to 0 to signal that the loop should exit. `expression3` usually mutates a variable declared in `expression1`, so that `expression2` eventually evaluates to 0. However, none of the three expressions is required. If `expression2` is empty, the loop

iterates until the `break` statement is executed.

4.4.2. While

The while statement has one form:

```
while (expression) statement
```

The expression is evaluated once before each of zero or more iterations. If the expression evaluates to zero the loop terminates; otherwise, the statement is executed, followed again by the expression.

4.5. Break

The statement `break;` terminates the nearest enclosing `for` or `while` statement and passes control past the scope of that terminated statement.

4.6. Continue

The statement `continue;` passes control to the end of the nearest enclosing `for` or `while` statement, skipping any in-between statements. The loop statement then continues normal iteration behavior.

4.7. Return

A routine can terminate execution and return to its caller with the `return` statement, which has two forms:

```
return;
return expression;
```

When the expression is omitted, zero is returned.

5. Expressions

Expressions are left-associative unless otherwise specified.

5.1. Constants

Integer constants are valid expressions.

5.2. Variables

A variable is accessed by specifying its identifier, which is a valid expression.

5.3. Unary Operators

The unary operator `'-'` negates its right-associated expression.

5.4. Binary Operators

The following binary operators are listed from highest to lowest precedence.

5.4.1. Multiplicative

`'*'` indicates: multiplication.

`'/'` indicates: division.

5.4.2. Additive

`'+'` indicates: addition.

`'-'` indicates: subtraction.

5.4.3. Relational and Equality

The relational and equality operators return 1 when true and 0 when false.

`'<'` indicates: less than.

`'>'` indicates: greater than.

`'<='` indicates: less than or equal.

`'>='` indicates: greater than or equal.

`'=='` indicates: equal.

`'!='` indicates: not equal.

5.5. Assignment

`'='` indicates: assignment. It is used to assign an integer variable or constant to a variable, and is right-associative.

5.6. Routine Calls

A routine is called by specifying its identifier, followed by a `'~'` and a comma-separated list of arguments. The number of the arguments in the comma-separated list must match the routine declaration.

6. Variable Declarations

A variable declaration has one form:

```
int identifer;
```

The keyword `int` prefixes a variable declaration, and a semi-colon follows the identifier. Variables contain the value zero when first declared.

7. Routine Declarations

A routine declaration has the form:

```
routine identifer ~ arg1, arg2, ..., argN
{ variable declarations and statements }
```

The keyword `routine` is followed by an identifier, a `'~'`, and an optional comma-separated list of arguments. A mandatory opening brace then begins a series of zero or more variable declarations and statements, followed by a closing brace.

All variables specified in the argument list are accessible as local variables; these variable names remain unique within the scope of the routine. The main routine requires one argument, `filePath`, which specifies the `FilePath` used to initiate this program's Tune.

```
routine main ~ filePath { }
```

8. Built-in Routines

8.1. Disco

`setDiscoSize~ x,y`

Adjusts the rectangular size of Turtle's dance floor to width `x` and height `y`.

`setDiscoColor~ c`

Sets the background color to `c`.

8.2. Turtle

`fd~ p, bk~ p`

Turtle moves forward or back positive `p` pixels while drawing if Pen is down.

`setPosition~ x,y`

Turtle jumps to the position `x,y` without drawing.

`getOrientation~`

Retrieves Turtle's orientation in degrees.

`setOrientation~ d`

Turtle turns `d` degrees relative to value returned by `getOrientation~`.

`setRate~ r`

Sets Turtle's rate of movement to `r`.

8.3. Pen

`pu~, pd~`

Sets the Pen Up, Down.

`setColor~ c`

Sets the Pen's Color to `c`.

`setLineStyle~ s`

Sets the Pen's LineStyle to `s`.

`setLineWidth~ w`

Sets the Pen's Width to `w`.

8.4. Tune

cue~ filePath

Prepares the .wav file at `filePath` for playback.

pause~

Pauses the Tune in the Disco.

play~

Resumes play of the Tune in the Disco.

stop~

Closes and releases the Tune's .wav file.

setVolume~

Sets the volume of the Tune to `v`.

9. Sample Program

Example TT Program:

```
/* SquareDance.tt */

routine square ~ x,y
{
  int entryOrientation = getOrientation~;
  setOrientation~ 0;
  setPosition~ x,y;
  for(int i = 0; i < 4; i = i + 1;)
  {
    fd~ 50;
    rt~ 90;
  }
  setOrientation~ entryOrientation;
}

routine main ~ filePath
{
  setDiscoSize~ 400,400
  setRate~ 10;
  cue~ filePath;
  play~;
  pd~;
```

```
  int i = 1;
  while (i <= 5)
  {
    square~ i*80,i*80;
    i = i + 1;
  }

  stop~;
}
```

This example draws five squares positioned along the diagonal from top left to bottom right. Each square is modulated by the amplitude of the .wav file input specified by `filePath`. The image below represents a loose depiction of a possible output, driven by a specific audio input: