Quetzal Language Specification

A Tangible Programming Language for Education

—DRAFT—

Michael S. Horn

Department of Computer Science

Tufts University

`michael.horn@tufts.edu`

June 15, 2004

Contents

1 Introduction **3**

1.1 A Language of Physical Objects 4

1.1.1 Form 4

1.1.2 Color 4

1.1.3 Material 5

2 Data Types & Variables **6**

2.1 Number 7

2.2 Boolean 7

2.3 Text 8

2.4 Other Types 8

3 Operators & Expressions **9**

3.1 Numeric Operators 10

3.1.1 Additive Operators 10

3.1.2 Multiplicative Operators 11

3.1.3 Optional Operators 11

3.2 Comparison Operators 11

3.3 Logical Operators 12

3.4 Expression Trees 13

4 Statements & Flow-of-Control **16**

4.1 Connectors & Sockets 17

4.2 Begin & End Statements 17

4.3 Assignment Statements 18
4.4 Base-ten Statement 18
4.5 If-Statement 19
4.6 Merge Statement 19
4.7 Else-If Constructions 20
4.8 Loops 21
4.8.1 While Loops 21
4.8.2 Repeat-Until Loops 21
4.8.3 For Loops 22

5 Glossary **23**

Chapter 1

Introduction

Quetzal is a general-purpose tangible programming language designed for children and first-time programmers to be used in collaborative educational environments. The Quetzal language specification outlines the elements of a physical language designed to be expanded and customized to suit any number of application domains. As such, only the most fundamental data types, operations, and functionality are described here. Quetzal is a small language in the sense that it supports only a limited number of constructs and abstractions, and it is best for describing smaller algorithms. While this may constrain the language's utility, we believe that it enhances its instructional value in that students and instructors should strive for a deeper understanding of fundamental concepts of logic, flow-of-control, and algorithm.

Unlike many other tangible programming languages[1], Quetzal is a symbolic language. This means is that the Quetzal programming elements are not themselves computers, and Quetzal programs have no ability to execute the algorithms that they describe. Some other pieces of technology, namely the Quetzal compiler (described in chapter **??**), must be used to convert the symbolic representation of an algorithm into actual machine code which runs on some host computer. While in some ways a limitation, the symbolic nature of Quetzal gives us a certain degree of flexibility and expressive power that we might not have had otherwise. From a practical standpoint, it allows us to manufacture fast and inexpensive prototypes and even full-fledged implementations. Strategies for manufacturing working Quetzal Implementations are discussed in chapter **??**.

[1]e.g. McNerney, 2000; Resnick et al; 1998, Wyeth and Purchase; 2002

1.1 A Language of Physical Objects

Quetzal is a language completely expressed in the forms and arrangements of physical objects. Each distinct physical object is called a **programming element**. While Quetzal does make use of a limited number of written words and symbols, these are not fundamental to the expression of the language. The syntax of the Quetzal language is almost entirely expressed in terms of physical shapes and the way that they connect to each other. The elements of the language, variables, expressions, operators and statements, are three-dimensional, physical artifacts that connect to form programs.

1.1.1 Form

Unlike text-based languages which consist of a finite grammer and lexicon—bounded, in most cases, by the symbols produced by an ASCII keyboard—implementations of Quetzal could concievably be crafted with endless varieties of materials, colors, and dimensions. In Quetzal, the choice of color, size, and material used for the various programming elements is inconsequential, just as the choice of font used to type a text-based language is unimportant. A Quetzal implementation designed for young children might consist of large, brightly colored programming elements, while an implementation for slightly older students might use smaller objects with more nuanced colors. However, there are certain features of programming elements that should be consistent from one Quetzal implementation to another. For example, a number in Quetzal is represented by a cylinder. The top and bottom faces of a number should be flat and circular. The radius and height can vary from one Quetzal implementation to another, but it is essential that the token fit snugly into the round holes of a numeric operator.

1.1.2 Color

Quetzal places no restrictions on the use of color for various programming elements except to say that, for the sake of students with vision imparements or color blindness, color should never be the sole means for distinguishing one programming element from another—for example, the “red” variable versus the “green” variable. Designers, however, are encouraged to use color to *reinforce* the differences between data types, constants and variables, statements and operators, etc.

1.1.3 Material

Care should be taken in the selection of materials used for the various Quetzal programming elements. Most importantly, materials should be durable and non-toxic. In addition, if used carefully and consistently, materials, can subtly reinforce symantic relationships between programming elements. Some of the many properties of materials that can be exploited to convey meaning include texture, translucency, heat conductivity, rigidity, heft, and the sharpness of edges.

Chapter 2

Data Types & Variables

Variables and constants in Quetzal are represented by physical **tokens** with specific shapes that indicate their data type. For our purposes, a token is defined as a simple physical object which is used to represents some piece or collection of data. For constants and variables in particular, the tokens represent a single piece of information of a specific type (a number for example). Quetzal is a **strongly-typed** language in that every variable, constant, and operator has a data type that is unambiguously expressed through its physical form both at compile time and during program construction. This differs from strongly-typed text-based languages such as Java and C++ in which a variable must be declared to be of a certain type, but the type in not implicit in the variable's symbol itself. For example, numbers (both variable and constants) take the form of a circular disks. A programmer will always know when she is working with a number because of its round shape.

Variables in Quetzal may be used without first being declared. Accordingly, all variables start out with a consistent default value that is dependent on the data type. It is often the case that many tokens might be necessary to represent the same variable in the same program. For example, in the algebraic equation $y = 3x^2 + 5x - 1$, the variable x is written twice. To express the same equation in Quetzal, a programmer must use two tokens representing the variable x. As a result, language designers should manufacture multiple tokens to represent each variable name.

All Quetzal implementations must support two primitive data types: **number** and **boolean**. Language designers may wish to include other data types (such as **text**, for example) as needed by the application domain.

Figure 2.1: Variables and constants of type number are represented by cylindrical tokens with the constant value or variable name clearly marked on the top face. Although tokens representing variables and constants must be identical in size and shape, a language designer may choose to use material or color to distinguish one from another. This figure shows the constant 4, the variable x, and the constant π.

2.1 Number

All variables and constants of type **number** are characterized by physical tokens which take the form of solid disks with flat top and bottom surfaces, and the tokens may be manufactured in any size, material, or color as long as the the top and bottom surfaces are circular. A label identifying the variable name or constant value should be clearly marked on the top face. Quetzal provides ten constants for the digits 0–9 as well as optional constants such as 10, 100, 1000, $\pm\infty$, and π. For arbitrary numeric constants, students should use the base-ten statement (section 4.4) to assign desired numeric values to variables. There is only one data type for both integer and decimal (floating point) number.

Quetzal also provides programmers with several numeric variables, each clearly labeled with a letter or other appropriate symbol. Common algebraic symbols such as x, y, z, i, j, k, and n might be reasonable choices for variable names, however, designers should take into account the age and mathematical experience of the intended programmers as well as the application domain when making this decision. Tokens for variables must have the exact same shape and size as tokens for constants, although designers may wish to make use of color, texture, or material to distinguish the two. The default value for a numeric variable (before it has been set for the first time) is zero.

2.2 Boolean

All variables and constants of type **boolean** are characterized by physical tokens which take the form of rectangular prisms with equilateral diamonds for the top and bottom surface. Labels identifying the variable names and constant values should be clearly marked on the top surface of each token. There are two constant values for the boolean type: TRUE and FALSE. Quetzal implementation should also provide a small number of boolean variables marked with letters or other appropriate symbols. As with numbers, the shape and size of the tokens for boolean variables should be identical to the tokens for boolean

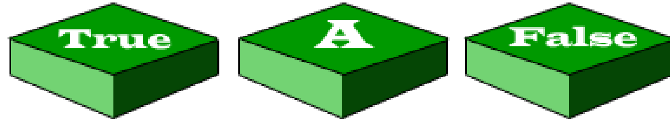


Figure 2.2: Variables and constants of type boolean are represented by rectangular prisms with equilateral diamonds for the top and bottom surface. The constant value of variable name should be clearly marked on the top face. Although tokens representing variables and constants must be identical in size and shape, a language designer may choose to use material or color to distinguish one from another. This figure shows the constant TRUE, the variable A, and the constant FALSE.

constants, although, color, texture, and material may be used to distinguish the two. The default value for a boolean variable (before it has been set for the first time) is FALSE.

2.3 Text

This version of the Quetzal specification does not include a **text** data type. However, the pentagonal prism shape (shown below) is reserved for text tokens in future versions of the language.

2.4 Other Types

As of this version of Quetzal, there are no data types specified other than boolean and number. However, for most application domains, these two types will be insufficient for even moderately complicated programs. Language designers may therefore need to create their own data types and associated operators. In general, the tokens for new data types should conform to the following guidelines.

- The shape used for tokens of a new data type should be distinct from those of existing data types.
- The tokens for variables should be the same size and shape as the tokens for constants.
- The top and bottom surfaces of tokens should be flat.
- A constant value or variable name should be clearly marked on the top surface.
- It is better if tokens have rotational symmetry around at least two axes. This makes it easier for programmers to build composite expressions (see section 3.4 on expression trees).

Chapter 3

Operators & Expressions

The previous chapter discussed variables and constants, Quetzal programming elements which represent single pieces of information with specific data types. In algebra, these numbers and variables can be combined with **operators** to form **expressions** such as $2 \times 4 + x$. The symbols $2, 4$, and x are the variables and constants, while $\times$ and $+$ are the operators. In Quetzal, operators are programming elements which perform mathematical and logical operations such as multiplication and division. Like variables and constants, operators also represent single pieces of information with specific data types. However, the information that an operator represents is usually based on the values of its operands. For example, the division operator accepts two numeric operands, the dividend and the divisor, and yields a single numeric value, the quotient. In general, operators can accept any number of operands, even zero, but they must always yield a single result of a specific data type. This result is called the **return value**. This definition of operator differs slightly from other common languages such as Java where it is possible to have an expression with no return value (the `void` type).

An operator is represented by a rectangular tile or block with a flat top and bottom surface and an identifying symbol clearly marked on the top surface. Operators also contain holes or slots in their top face into which operands can be inserted—one slot corresponds to each operand. Thus, a unary operator will have one slot and a binary operator will have two slots. Finally, an operator has a protrusion from its bottom surface which represents its return value. Both operands and return values must have specific Quetzal data types. Therefore, the shapes of the slots and the protrusion should match the data type for the corresponding operand or return value. Figure 3.1 shows a multiplication operator. The two circular slots in the top face indicate that the multiplication operator accepts two numeric operands. Likewise, the

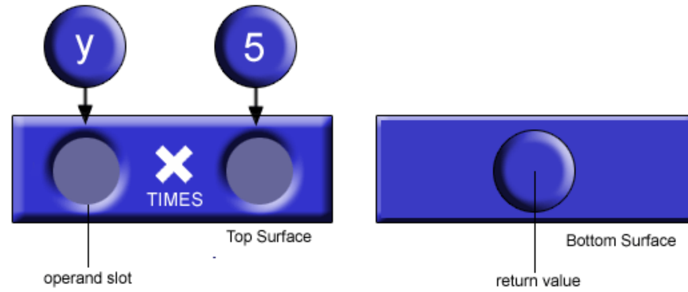


Figure 3.1: This figure shows how a programmer might represent the algebraic expression $(y \times 5)$. Both the top and bottom surface of the multiplication operator are shown. The operator accepts two numeric operands, represented by the circular slots on the top surface, and yields a single numeric return value (the product), represented by the circular protrusion on the bottom surface.

protrusion on the bottom face indicates that the operator yields a numeric return value.

3.1 Numeric Operators

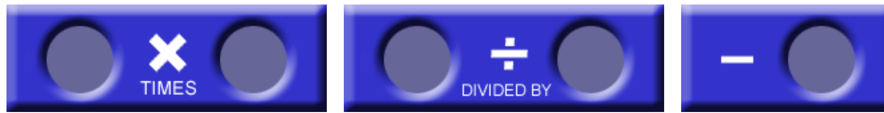
The basic set of **numeric operators** is listed below. All numeric operators accept numeric operands and yield a single numeric return value. Language designers may wish to include any number of additional numeric operators (some of which are listed below) depending on the application domain and the age and mathematical experience of the intended students. For example, trigonometric functions might be useful for programs that control robots with rotational motion.

3.1.1 Additive Operators

The **addition** $(+)$ operator computes the sum of two numbers and the **subtraction** $(-)$ operators computes the difference of two numbers. Note that the left-to-right order of the operands is important for many operations such as subtraction. In other words, $(3 - 2) = 1$ is quite different from $(2 - 3) = -1$. In such cases, language designers should take care to clearly label which operand is which. For example, in the diagram below, the word "MINUS" is written along the bottom edge of the subtraction operator to indicate an *up* orientation. Designers may also choose to use a **postfix** or **prefix** notation instead of an **infix** notation (used in this document) to allieviate some of this confusion.

3.1.2 Multiplicative Operators

The **multiplication** ($\times$) operator computes the product of two numbers while the **division** ($\div$) operator computes the quotient of two numbers. The **negation** ($-$) operator is a unary operation which multiplies its operand by -1.



*Note: at present, there is no specification for integer division or the remainder operator (*MOD*). These should be incorporated into a future version of the Quetzal specification.*

3.1.3 Optional Operators

Language designers may also wish to provide some or all of the following operators:

- The **square root** ($\sqrt{x}$) operator computes the square root of its operand.
- The **exponent** (x $\uparrow$ y) operator computes the value of the first operand raised to the power of the second operand.
- The **absolute value** operator ($|x|$) returns the absolute value of its operand.
- The **round** operator returns the closest integer value to its operand.
- The **floor** ($\lfloor x \rfloor$) operator (also called **round down**) returns the highest integer value less than or equal to its operand.
- The **ceiling** ($\lceil x \rceil$) operator (also called **round up**) returns the lowest integer value greater than or equal to its operand.
- The **random** operator accepts no operands and produces a psudo-random number between 0 and 1, inclusive. The random number generator is automatically seeded at the beginning of program execution with the current system time.

3.2 Comparison Operators

Comparison operators always accept two operands which represent values to be compared. Instead of returning a number, comparison operators return a boolean value (TRUE or FALSE) indicating the

Figure 3.2: A selection of numeric operators that language designers may choose to implement depending on the application domain and the age and mathematical background of the intended students.

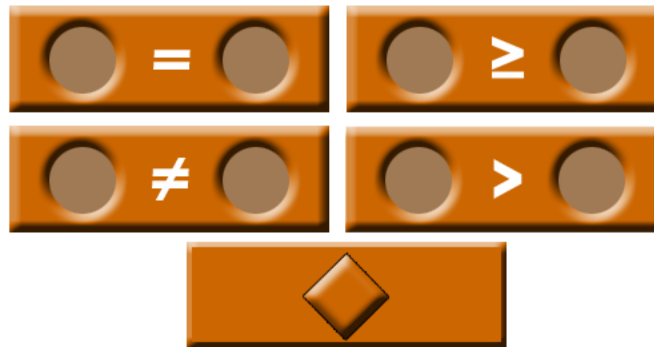


Figure 3.3: This figure shows the four comparison operators: equals ($=$), not equals ($\neq$), greater-than ($>$), and greater-than-or-equal-to ($\geq$). The diamond-shaped protrusion on the bottom surface of comparison operators (shown below the other operators) indicates a boolean return value (TRUE or FALSE).

outcome of the comparison. For example, the **greater-than** ($>$) operator returns TRUE if and only if the first operand is numerically greater than the second operand. Otherwise it returns FALSE. Likewise, the **greater-than-or-equal-to** ($\geq$) operator returns TRUE if and only if the first operand is numerically greater than or equal to the second. There are no less-than or less-than-or-equal-to operators in Quetzal because these operations can be represented by simply turning the greater-than and greater-than-or-equal-to operators upside-down. The **equals** ($=$) operator returns TRUE if and only if the first operand is exactly equal to the second. The **not equals** operator returns TRUE if and only if the first operand is not equal to the second.

3.3 Logical Operators

Logical operators accept one or two boolean operands and always return a boolean value. The set of logical operators supported by Quetzal includes the following:

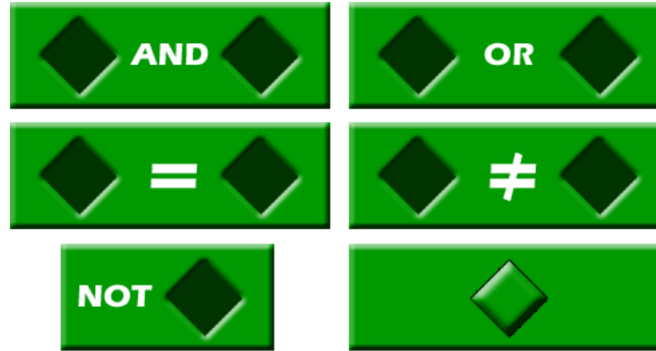


Figure 3.4: This figure shows the basic logic operators supported by Quetzal: AND, OR, NOT, equals (=), and not equals (≠). The diamond-shaped protrusion on the bottom surface of logic operators (shown lower-right) indicates a boolean return value.

- The **AND** operator returns TRUE if and only if both of its operands are TRUE.
- The **OR** operator returns TRUE if and only if at least one of its operands is TRUE.
- The **NOT** operator returns the opposite value of its operand.
- The **equals** (=) operator returns TRUE if either both operands are TRUE or if both operands are FALSE. This is equivalent to the expression $(a \wedge b) \vee (\overline{a} \wedge \overline{b})$
- Finally, the **not equals** (≠) operator returns TRUE if and only if exactly one of its operands is TRUE and one is FALSE. This is the same as the **exclusive or** (XOR) operation.

3.4 Expression Trees

Variables and constants are not the only programming elements that can be used as operands. The return value from other operators can be used as well, as long as the data types correspond. For example, suppose a programmer wanted to represent the algebraic expression $(2 \times x)/3$. The programmer would first select a multiplication operator and insert the constant 2 and the variable x. She would then select a division operator and use the multiplication operator as the dividend and the constant 3 as the divisor (shown in figure 3.5).

This is a simple example of an **expression tree**, a stack of operators, variables, and constants that represents a composite numeric or logical expression. In a similar way, composite logic expressions can be formed using boolean operations. Figure 3.6 shows a logic expression formed with both boolean and

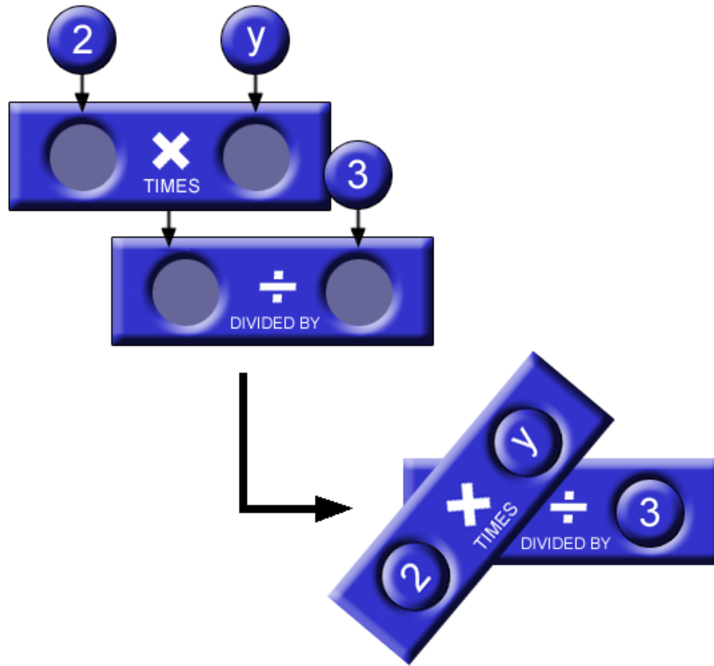


Figure 3.5: This figure shows how a programmer might construct the composite expression $(2 \times y)/3$. The resulting expression is shown at bottom right.

numeric variables and constants. Unlike text-based languages which must rely on order of operation rules or nested parentheses to correctly evaluate expressions, the physical structure of Quetzal expression trees unambiguously convey the order in which operations should be processed. Expression trees are evaluated from the top down at runtime. Operands are evaluated from left to right and all operands are fully evaluated, even for conditional logic operations[1]. Constants and variables will be evaluated immediately, while other operations (sub-expressions) must be evaluated recursively. Figure 3.7 shows an expression which uses the Pythogorean Theorem to calculate the length of the hypotenuse of a right triangle. The Quetzal expression tree is shown on top, and the equivalent expressions in Java and Lisp are shown below.

[1] This is not the case with many text-based languages such as Java and C++ in which conditional logic operations may be *short-circuited* if the first operand evaluates to TRUE (for conditional OR) or FALSE (for conditional AND).

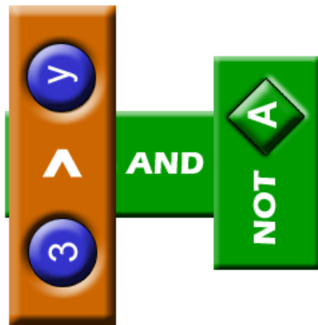
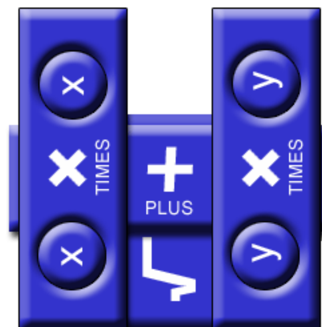


Figure 3.6: This figure shows an expression tree for the conditional logic expression $(3 > y) \wedge \overline{A}$. Tokens representing both numeric and boolean variables and constants are used.



In Java: `Math.sqrt(x * x + y * y);`

In Lisp: `(sqrt (+ (* x x) (* y y)))`

Figure 3.7: This figure shows an expression which uses the Pythogorean Theorem to calculate the length of the hypotenuse of a right triangle. The Quetzal expression tree is show with the equivalent expressions in Java and Lisp below.

Chapter 4

Statements & Flow-of-Control

Up to this point, we haven't dealt with how Quetzal programs are constructed. How do we use Quetzal to actually do something? We'll begin to answer this question by describing Quetzal **statements** and the way in which program execution flows from one statement to another. Quetzal programs are formed from connected chains of statement, physical programming element which performs some task or flow-of-control operation. A statement may be any shape or size provided that it has a flat top and bottom surface with the the statement's name or identifying symbol clearly marked on the top surface. Like operators, statements may accept any number of parameters. Each parameter should be represented by a slot on the top surface which corresponds to the size and shape of the parameter's data type.

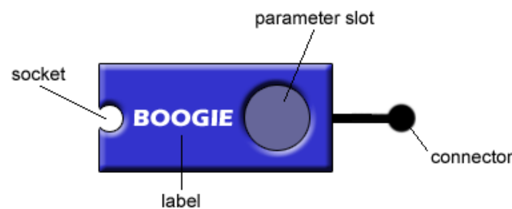


Figure 4.1: This figure shows a simple Quetzal statement with one flow-of-control socket (on the left side) and one connector (on the right). This statementa accepts a single numeric operator and performs the *boogie* operation. The physical form of sockets and connectors may differ in appearance from these diagrams.

4.1 Connectors & Sockets

Quetzal statements are designed to be physically connected to form a program's flow-of-control. Statements are executed sequentially starting with a single **begin** statement and ending with an **end** statement. In fact, the most simple Quetzal program consists of a begin statment connected directly to an end statement (as shown in figure 4.2). Most statements have a single outward flow-of-control **connector** which extends from the right side of the physical element. Connectors can *plugged into* flow-of-control **sockets**, which are positioned on the left side of most statements. A program's flow-of-control always moves *out* from a connector in one statement and *into* the socket of the next statement.

The physical form of the flow connectors and sockets should be determined by the language designer and may be influenced by the technology used to implement the compiler. Connectors may be rigid or flexible, but they should always form a secure connection with a socket. At the same time, they should be easy to plug and unplug. Household electric sockets and phone cables jacks are two good examples of socket and connector designs. The diagrams in this document will employ visual conventions to indicate sockets and connectors, but the real life physical components may have a different shape and appearance. Figure 4.1 shows a generic Quetzal statement with a socket (on the left side) and a connector (on the right).

With the exception of begin and end, all statements must have at least one socket and one connector—otherwise, there would be no way to connect them to a program. However, some statements such as the if-statement and the merge statement (sections 4.5 and 4.6) may have more than one socket or connector.

4.2 Begin & End Statements

The **begin** statement is a special Quetzal statement with a flow-of-control connector, but no socket. It takes the form of an oval-shaped block or tile. A program's execution always starts with a single begin statement. Programs constructed with more than one begin statement will result in a syntax error.

Likewise, **end** statements have a single flow socket, but no connector. End statements are represented by elongated octagonal blocks or tiles. Programs may include multiple end statements, and if a program's flow-of-control ever reaches one of the end statements, execution will halt immediately.

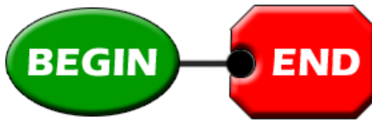


Figure 4.2: The simplest possible Quetzal program consists of a begin statement connected directly to an end statement. The begin statement has a flow connector but no socket, while an end statement has a socket but no connector.

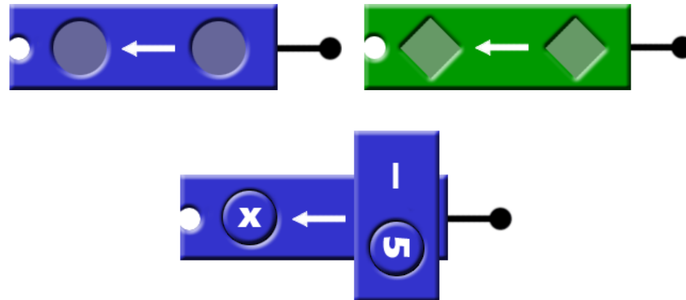


Figure 4.3: This figure shows a numeric assignment statement and a boolean assignment statement. The bottom construction shows how a programmer might execute the statement $x \leftarrow -5$.

4.3 Assignment Statements

The **assignment** statement sets the value of a variable (the left parameter) equal to the value of the right parameter. The assignment statement is represented by a rectangular block or tile with two operand slots. The $\leftarrow$ symbol points from the right parameter to the left parameter. Assignment statements have implied syntax that is not directly expressed through a physical constraint. Namely, while the right parameter may be any Quetzal expression, variable, or constant, the left-hand parameter must be a variable. Quetzal implementations should include an assignment statement for every data type.

4.4 Base-ten Statement

Similar to the numeric assignment statement, the **base-ten** statement allows programmers to set the value of numeric a variable to an arbitrary constant value using a base-ten number representation. The base-ten statement is unique in that any parameter slot that is not filled is assigned an implicit value of zero. In the figure below, the variable x would be set to the number $6,001.3$ Inserting a variable into any slot to the right of the $\leftarrow$ symbol will produce a syntax error. Likewise, inserting a non-variable into the left-most

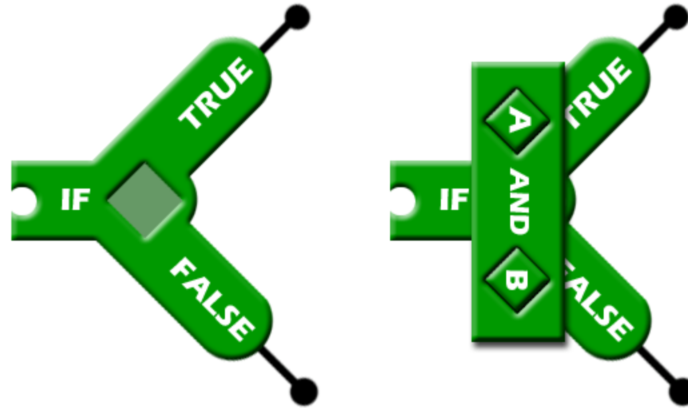
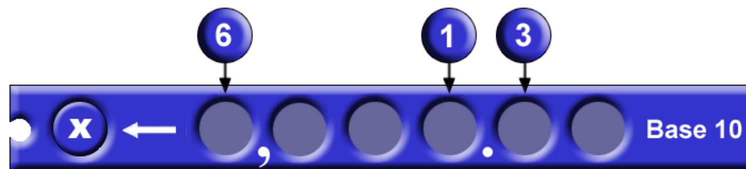


Figure 4.4: An if-statement can be used to branch a program's flow of control down one of two paths based on the value of a boolean predicate. For example, the if-statement on the right will follow the TRUE branch if the variables A and B are TRUE. Otherwise, it will follow the FALSE branch.

slot will produce a syntax error.



4.5 If-Statement

An **if-statement** splits a program's flow-of-control down one of two branches based on the value of a boolean expression, variable, or constant. If-statements are represented with boomerang-shaped blocks or tiles. If-statements have a single flow socket and two flow connectors labeled TRUE and FALSE. If the boolean operand evaluates to TRUE, the program's flow-of-control will continue through the TRUE connector, otherwise it will continue through the FALSE connector. In Quetzal, there is no need for an `else` statement; flow-of-control simply continues down the TRUE path or the FALSE path.

4.6 Merge Statement

A **merge statement** combines two branches of a program into a single branch. This is equivalent to the `end if` statement of many text-based languages. The merge statement is represented by a triangular block or tile. It has a single flow connector and two flow sockets. Merge statements can be used to support a

Figure 4.5: The merge statement combines two branches of code split by an if-statement.

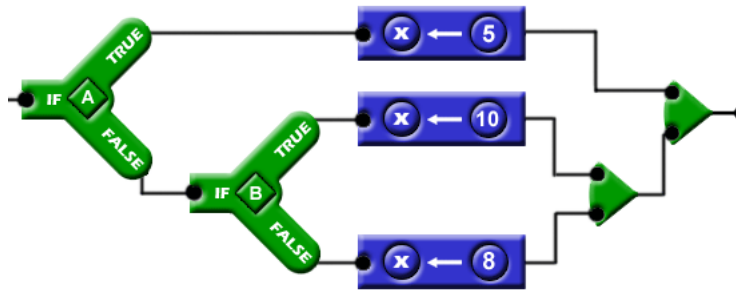


Figure 4.6: An example if, else-if, else, construction using Quetzal if-statements and merge statements.

variety of programming constructions including chained if-statements and loops (see sections 4.7 and 4.8).

4.7 Else-If Constructions

Most programming languages provide `else` and `else if` syntax that allows for multiple branch statements to be easily *chained* together. In Quetzal, equivalent stuctures can be created by connecting multiple if-statements and merge statements. Figure 4.6 shows how a programmer might translate this Java code fragment into Quetzal:

```
if (A) {
    x = 5;
} else if (B) {
    x = 10;
} else {
    x = 8;
}
```

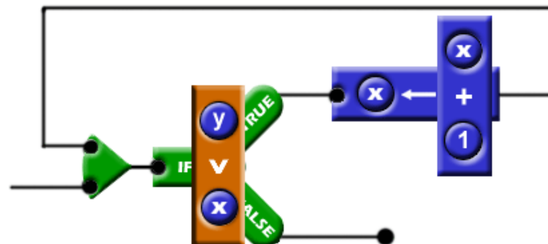
4.8 Loops

Quetzal has no special statements or syntax to support loops, unlike most text-based languages and many visual languages. Instead, loops are formed with if-statements and merge statements and form actual, physical loops in the connected statements of a program's flow-of-control. This section will give examples of three common loop constructions found in text-based languages: while loops, repeat-until loops, and for loops.

4.8.1 While Loops

A **while loop** repeats some sequence of statements (the loop body) while a boolean predicate holds true. The boolean predicate is tested before each iteration of the loop. The diagram below shows how a programmer might construct the following while loop writen in Java.

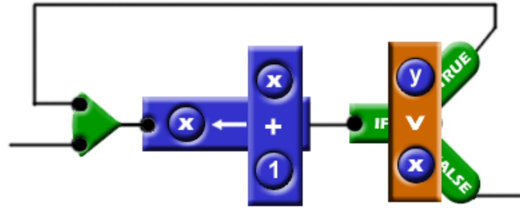
```
while (x < y) {
    x++;
}
```



4.8.2 Repeat-Until Loops

A **repeat-until loop** (or do-while loop) is similar to a while loop except that the boolean predicate is tested immediately *after* each iteration. The loop repeats as long as the boolean expression evaluates to TRUE. The diagram below shows how a programmer might construct the following do-while loop writen in Java.

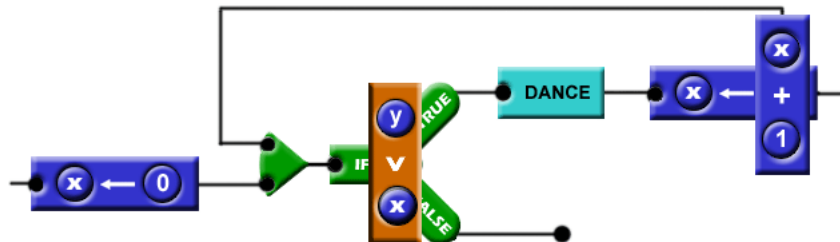
```
do {
    x++;
} while (x < y);
```



4.8.3 For Loops

A **for-loop** in a language like C++ or Java is a multi-part statement that allows programmers to create a variety of loop structures and conditions. In what is perhaps the most common use of the for-loop, some variable is initialized and then incremented every time the loop iterates. For example, the following Java code fragment initialized the variable $x = 0$, tests that $x < y$ before each iteration, and increments x after every iteration of the loop body. The equivalent construction is shown in Quetzal in the diagram below.

```
for (int x = 0; x < y; x++) {
    dance();
}
```



Chapter 5

Glossary

programming element (page 4)

token (page 6)

number data type (page 6)

boolean data type (page 6)

text type (page 6)

text data type (page 8)

expression (page 10)

operator (page 10)

operand (page 10)

return value (page 10)

numeric operator (page 11)

postfix notation (page 11)

prefix notation (page 11)

infix notation (page 11)

comparison operator (page 12)

logical operator (page 13)

exclusive or (page 14)

expression tree (page 14)

statement (page 17)

flow connector (page 17)

flow socket (page 17)

Begin statement (page 18)

End statement (page 18)

if-statement (page 20)

Merge statement (page 20)