Tranquility Programmer's Manual

Introduction

The language Tranquility has been created as a language to teach the basic concepts of programming languages and programming technique. It has relatively few features to keep the learning curve short, and its features are focused on reinforcing the core ideas of how programming languages work and how computers run the programs expressed in them. It is important to understand that Tranquility has not been created to make it easy to create production applications. It has been created to help you develop a deeper understanding of how the hardware, the programming language, and the code you write work together. The basic usage model of the language is an unusual combination of the classic edit-compile-run development with code execution in the web browser. Although created for the CS164 course at Drexel University, Tranquility is a good language for anyone wanting to learn how the hardware and software fit together.

The name Tranquility has been chosen for several reasons:

1. Tranquility is a pun on the name of the BLISS programming language. BLISS is a systems programming language that was used heavily at Digital Equipment Corporation (DEC). This connection is highlighted in the name because Tranquility uses the same model of variables as does BLISS.
2. The summer during which Tranquility was developed (2019) was the same summer that saw the 50th anniversary of the Apollo 11 flight, and the landing site for that mission was the Sea of Tranquility.
3. Although the word 'tranquility' does not appear in the classic song *Bridge over Troubled Water,* it does seem an appropriate description of the sense of bridging across troubles. It would be natural to ask what any of this has to do with a programming language. The answer is that much of the motivation for creating this language was in bridging a large conceptual chasm between the world of binary numbers, adders, and instructions on the one hand and the world of popular high-level languages on the other. It is hoped that students using Tranquility will be able to approach other languages with less of a sense of mystery and more of an understanding of what's really going on.

Hello World

As has become traditional in the presentation of programming languages, we begin with a very simple program to simply print a string to the output. In the course of examining and running this example, we will get a feel for several aspects of the language and how to use it. We begin by creating a file containing the source code for the program:

```
fun init() {
    sprint("Hello World\n")
}
```

For the sake of discussion, let's say we wrote this source code into a file called `hello.t`. There are two things about this that are worth highlighting. First, the `.t` extension is not required by the language; it's just a convention that Tranquility source file names end that way. Second, remember the file is really just a sequence of binary numbers with one 8-bit number (a byte) for each character in the file. The correspondence of what numbers are used for what characters is defined by the ASCII character set.

Compiling Hello World

Before getting into the details of how the program is written, we'll take a look at how we would run this program. As with many languages, before we can run the program, we must compile the program. A compiler translates a program from the representation that you see in your editor into another representation that is more easily understood by the computer. In many cases, this target representation is the set of machine language instructions directly understood by the CPU, much like the set of ten instructions available on the CARDIAC. In the case of Tranquility, the compiler translates the source code into the machine language of a different fictitious machine called the Tranquility Virtual Machine, or tvm. The command to compile this Tranquility program is

```
tranqc hello.t
```

After you run that program, you will find that a new file named `hello.json` has been created in the current directory. This file contains a representation of the tvm machine code for this program in a form suitable for the program that simulates the tvm.

Running Hello World

Up to this point, using Tranquility is much like using many other languages. It departs from more conventional languages in that the program that simulates the tvm runs in your web browser. As you might guess, this requires that you have an HTML file that loads your program to be run. A very basic HTML file that can be used for this purpose looks like:

```
<html>
<head>
<script src="hello.json"></script>
<script src="https://www.cs.drexel.edu/~bls96/tvm.js"></script>
</head>
<body>
</body>
</html>
```

This HTML file loads two other files, one being the file created by the Tranquility compiler and the other being a JavaScript program that simulates the tvm. The file `hello.json` is expected to be in the same directory as the HTML file and the file `tvm.js` file is in the professor's web pages. To load it in the browser, you need to have these files in your `public_html` directory. If the HTML file is named `hello.html`, then getting them there can be accomplished with the command

```
cp hello.json hello.html ~/public_html/
```

If your user name is `abc123` then you can load this page with the URL

```
http://cs.drexel.edu/~abc123/hello.html
```

When the page loads, this is what it looks like:

Pressing the button labeled Start begins running the code and results in a screen showing the message in the standard output window.

Hello World Code

Returning to the source code for our program, the file begins with the keyword `fun`. In Tranquility, `fun` introduces the declaration of a function. Functions are named sections of code that may take inputs and may produce outputs. The next thing after the keyword `fun` is the name of the function being declared. In this case, the function is called `init`. This is a special function name in Tranquility. The first function that is run (called) in Tranquility is always the function `init`. After the name of a function in a declaration, there must be a list of arguments in parentheses. If there are no arguments to the function, then the parentheses will be empty. The remainder of a function declaration is the function body enclosed in braces.

Notice the placement of the braces. Some languages have strict rules for formatting, and some languages allow very flexible formatting. Regardless of which approach a language takes, it is important for the programmer to develop disciplined style in formatting. To reinforce the importance of such discipline, Tranquility requires a fairly strict formatting. In Tranquility, the opening brace must always go on the line that begins whatever is enclosed in the braces. Conversely, the closing brace must always go on a line by itself. Although not required by the language, statements enclosed in braces should be indented.

In general, every function declaration will take the form:

`fun` *name*`(`*args*`) {`

where *name* is the name of the function being declared and *args* is a possibly empty list of parameter names.

In this function, there is only a single statement in the function body. That statement is

```
sprint("Hello World\n")
```

This statement is a function call. In other words, it causes the code for another function to be run. In this case, the function is `sprint`, which is a built-in function in Tranquility. It prints a string of characters to the standard output window of the browser. In this case, the string being printed is a literal string given in double quotation marks. The string is an input to the function and is enclosed in parentheses following the name of the function.

Another Example

To illustrate more features of Tranquility, let us consider another example.

```
fun sq(n) {
    return .n * .n
}

fun init() {
    var i

    sprint("Table of squares:\n")
    i : 1
    loop {
        until .i > 10
        iprint(.i)
        sprint(" squared equals ")
        iprint(sq(.i))
        sprint("\n")
        i : .i + 1
```

```
    }
}
```

This program has two functions, one called `sq` and the other called `init`. The simpler of the two is `sq` so we'll look at that one first. From the declaration of the function, we can see that `sq` takes a single argument as input. Inside of `sq`, we call that argument `n`. For those coming with experience in other languages, it may seem strange that there is no type information given in the argument list. More will be said on this later, but in Tranquility, all variables and parameters are of a single type, the word. This characteristic of the language was chosen to reinforce the understanding that internally, everything is a binary number. We will see later, how that can be used for characters and strings of characters.

Function Returns

The body of `sq` contains only a single line that reads:

```
return .n * .n
```

This statement illustrates several features of the language. First, the keyword `return` indicates that we are instructing the system to go back to the point from which this function was called. In Tranquility, a function may return a value back to the caller. If a function is not returning a value, the `return` statement can simply be that with nothing else after it. Here, the function does return a value and the `return` keyword is followed by an expression. At the time the function returns, the expression is evaluated and the result of the evaluation is the value that the caller gets from the function it called.

Variable Addresses and Values

In `sq` the expression is a multiplication as indicated by the `*` character. So the return value of this function is a product of two values. That much is easy, almost obvious. How the two values to be multiplied are specified requires a little more explanation. Remember that in programming, it's better to think of a variable as a name for a memory location, rather than a name for a value. In Tranquility, that's exactly how variables work. An expression that consists of just a variable name evaluates to be the address of the location named by the variable. To look inside that memory location and get the value stored there, we have an operator that's denoted by the dot (`.`) character. So in Tranquility, the expression `n` evaluates to be the memory address associated with the name `n`, and the expression `.n` evaluates to be the data stored in the memory location named `n`. This is the characteristic of the language BLISS that is copied in Tranquility. To evaluate the expression `.n * .n`, we look in the memory location identified by the name `n` and get the value out, then we do that again, and finally we multiply the two values together. So the overall operation of the function `sq` is to compute and return the square of the number that was passed into it.

The `init` function in the program is more involved than the other two functions that we've examined, so we'll take it one line at a time. The first line reads:

```
var i
```

This line creates a variable and gives it the name `i`. Because it appears inside a function, it is local to that function. In other words, it cannot be seen outside the function, it comes into being when the function is called, and it ceases to exist when the function returns. Of course, the memory location that the variable name identifies actually exists throughout the program, but association with the variable name has a limited life. In Tranquility, all variable declarations must appear before any executable code in a function.

The first statement of executable code in `init` is a call to the function `sprint` much as we saw in the previous example. We don't need to say any more about it or the other call to `sprint` later in the function. It is followed by a very simple appearing statement:

```
i : 1
```

The experienced programmer will tend to read this as "set `i` to 1," but it is worthwhile to look at it a little more carefully. An assignment statement uses the : (`:`) to indicate an operation like the `STO` instruction on the CARDIAC. This is very much like the use of the equals sign in other languages. It works by using two numbers, one on the left and one on the right. In CARDIAC terms, the number on the right needs to be loaded into the accumulator and then it's stored into the memory location given by the number on the left.

So in this statement we take the value of 1 on the right and store it into the memory location identified by the variable `i`.

Variables in Tranquility can also be declared globally. Any variables declared in the file before the first function is declared will be visible to all code in the program. However, if a function parameter or local variable has the same name as a global variable, then it will "hide" the global variable in that function. References to that name inside the function will be resolved to the parameter or local variable, rather than the global variable.

Repetition

The next statement, however, needs some discussion. It begins with the keyword `loop` and it contains other statements within it. In Tranquility, any time statements appear inside other statements, they must be enclosed in braces. Unlike some other languages, you may not omit the braces even when there is a single statement in that grouping. This characteristic of Tranquility is much like that found in the language Go. The meaning of the `loop` statement is that it identifies a sequence of statements that will be repeated. This is essentially the same sort of thing we saw in CARDIAC code where we had a sequence of instructions and there was a `JMP` instruction at the bottom of it to go back to the top. The statements in the body of the loop statement are simply listed on consecutive lines. Where some languages use a semicolon to terminate a statement, Tranquility uses the newline to terminate it. This means that all statements in Tranquility must be on a single line. If you find that you have a line that is getting unreasonably long, then you need to break it down into multiple statements.

In the body of the loop, the first statement is

```
until .i > 10
```

This is another unusual aspect of Tranquility when compared to most other languages. This statement determines the condition on which we break out of the loop and go down to the code that follows the loop. It can appear anywhere inside the loop. Many languages have looping structures that put a condition at the beginning of the loop or at the end of the loop. When the condition is at the beginning, we call it a pre-test loop and the loop is performed zero or more times. When the condition is at the bottom, we call it a post-test loop and the loop is performed one or more times. In Tranquility, the same effect is achieved by placing the `until` statement as either the first statement or the last statement in the loop body. However, there are times when it's most natural to have a part of the loop that's done n times and another part of the loop that's done $n-1$ times. In these cases, being able to put the `until` statement in the middle of the loop is more flexible. It also more closely mirrors what you saw when working with the machine language instructions in the CARDIAC. This particular use of `until` means that at the point where we get to this statement and the memory location `i` has a value greater than 10, the loop stops and the program continues with whatever follows the loop.

The next line of the loop body is a call to `iprint` which is like `sprint` except that it prints out a representation of a number as a base-10 numeral. Make sure you understand here why this line is written as `iprint(.i)` rather than `iprint(i)`. The second call to `iprint` in this function:

```
iprint(sq(.i))
```

is more interesting. The value to be printed is given by the expression `sq(.i)` that's in the argument list for `iprint`. Such an expression looks the same as the calls to `sprint` and `iprint` that we've seen, and it really is. In all of these cases, we first evaluate the arguments, then pass the argument values to the functions jumping to their code, and then come back to where we left off when they finish. That's the process of calling a function. The difference in this case is that we expect the function we're calling to return a value to us as a result, and that value becomes the value of the expression which, in turn, gets passed to `iprint` for printing. The `sprint(\n)` that follows prints a newline to finish the line that's being printed.

At the end of the loop, we have the statement

```
i : .i + 1
```

Based on what has already been discussed, this statement fetches the value stored in the memory location identified by the variable `i`, adds one to it, and stores the result into the memory location identified as `i`.

The effect is to increment `i`. Putting it all together, the program takes each integer value starting at 1 and going up to and including 10, computes the square of that number and prints it out. Running the program produces the output:

```
Table of squares:
1 squared equals 1
2 squared equals 4
3 squared equals 9
4 squared equals 16
5 squared equals 25
6 squared equals 36
7 squared equals 49
8 squared equals 64
9 squared equals 81
10 squared equals 100
```

More on Using `tranqc`

Of course, things don't always go as smoothly as we've portrayed here. Programming is an error-prone activity. However, some of the simplest types of errors are ones that the compiler can catch for us and help us resolve. These are ones where the text we've given the compiler is not a valid program in that language. This includes things like syntax errors and mistyped variable names. For example, suppose we meant to declare a variable called `b`, but instead typed `v` (the next key over) in the declaration. Then in every place we use `b`, the compiler won't know what we're talking about and we might get some error messages like this:

```
prog.t:6: unresolved symbol b
prog.t:7: unresolved symbol b
prog.t:13: unresolved symbol b
```

Each of these lines indicates an error that the compiler found. The first thing on the line is the name of the file in which the error occurred. After that is the line number where the error was detected, and the remainder of the line attempts to describe the error to help you correct it. Be careful about the line numbers, however. In some cases, the error is actually detected when the compiler gets to a later line in the program (usually the next line). So if the error message doesn't seem to make sense on the line reported, scan upward and see if there's an earlier line for which the message makes sense.

The more difficult errors to deal with are the logical errors. These are ones where we've not designed the logic of the program correctly, and the compiler can't find them for us. There are many ways in which such errors can arise, but most of the time they can be traced to either carelessness or a misunderstanding on the part of the programmer. For cases where there's a misunderstanding about the language and how programs in the language are run, the Tranquility compiler can print out its internal representation of the program and let you see how the compiler sees it. To do so, run the compiler with the `-v` option:

```
tranqc -v sq.t
```

Assuming that the example in the previous section is in the file `sq.t`, then the compiler will print out the following:

```
Globals:


Functions:
0 sq na:1 nl:0: args:  n local:
    0x1(n:L) @ 0x1(n:L) @ * RETURN
1 init na:0 nl:1: args:  local: i
    "Table of squares:
"(S) -102(sprint) CALL POP
    0x1(i:L) 1(I) !
```

```
    [
        0x1(i:L) @ 10(I) > BREAK
        0x1(i:L) @ -101(iprint) CALL POP
        " squared equals "(S) -102(sprint) CALL POP
        0x1(i:L) @ 0(sq) CALL -101(iprint) CALL POP
        "
"(S) -102(sprint) CALL POP
        0x1(i:L) 0x1(i:L) @ 1(I) + !
    ]GO
```

The notation here is a bit cryptic. It's based on some ideas from the programming language Forth and is connected with the details of the tvm discussed later. However, it provides a thorough and exact expression of how the compiler has understood your program. Spending a little time understanding how your original program got translated into this form will help you determine how well you've managed to express what you intended to in the language.

Another option that is available to you is the `-a`. Compiling with the `-a` option directly creates the HTML file in your `public_html` directory and embeds the JSON tvm machine code into the HTML file. This streamlines the development process at the expense of seeing how all the pieces fit together.

One other thing that is often helpful is to look at what the program is doing as it's running. The starting point in learning how to do that is to add `iprint` and `sprint` statements into the program to see if you're reaching the places in the code that you expect and if the variables are getting set to the values you expect. In later courses and throughout your career, you will be exposed to languages and environments that will have debuggers that you can use like a microscope to look at what's going on. However, for now, it's good to learn how to work without them to give you a better sense of what they can and cannot do for you.

Yet Another Example

We will take a look at one more example in the tutorial portion of this manual. It illustrates several more features and techniques that you will find useful.

```
    fun fact(n) {
        if .n == 0 {
            return 1
        }
        else {
            return .n * fact(.n - 1)
        }
    }

    fun genfacts(n) {
        var i, ftab

        ftab : alloc(.n)
        i : 0
        loop {
            until .i >= .n
            (.ftab+.i) : fact(.i)
            i : .i + 1
        }
        return .ftab
    }

    fun filltable() {
        var i, ftab, istr
```

```
    html("<center>\n")
    html("<table border=1><tr><th>n</th><th>n!</th></tr>\n")
    ftab : genfacts(13)
    istr : alloc(12)
    i : 0
    loop {
        until .i > 12
        html("<tr><td>")
        i2s(.istr, .i)
        html(.istr)
        html("</td><td>")
        i2s(.istr, .(.ftab+.i))
        html(.istr)
        html("</td></tr>\n")
        i : .i + 1
    }
    html("</table></center>\n")
}

fun init() {
    html("<center>")
    button("Make Table", filltable)
    html("<p>Factorials<p>\n</center>\n")
}
```

Conditionals

We'll start with the `fact` function at the top of the program. The main new feature here is the `if` statement. It allows us to selectively perform some operations depending on some conditions. In this case, the statement looks like:

```
    if .n == 0 {
        return 1
    }
    else {
        return .n * fact(.n - 1)
    }
```

Here, the statement means that if the number given to us as an argument is zero, then we return the value one. Otherwise, we calculate a value that is the product of the number we got and the factorial of the next smaller number. That product is what we return.

In `if` statements, the `else` part is optional. Furthermore, if you want to have sequence of tests and perform only one of the actions, then you can write it like this:

```
if cond1 {
    action1
}
else if cond2 {
    action2
}
else if cond3 {
    ...
else {
    unmatched action
}
```

Arrays

The `genfacts` function illustrates how arrays are done in Tranquility and how arrays work behind the scenes in all languages. In reality, the language has no arrays, per se; the programmer constructs arrays from other language features. An array is like a mini memory space that has a name. Each location in the array is identified by a number, like the address in memory. In the case of an array, we call that number the index. Internally, the array is a subest of memory starting at some base address. So to get to a particular element of the array, we just have to add the base address to the index, giving us the address of the element.

Here the array is created by a call to the built-in function `alloc` which takes the number of memory locations we want to allocate as an argument.

```
ftab : alloc(.n)
```

At this point, the memory location `ftab` contains the address of the first memory location we allocated. So the expression `ftab` evaluates to the address of the memory location labeled `ftab`, and the expression `.ftab` evaluates to the address of the first element in the array. We store the values of factorials into the array with the statement:

```
(.ftab+.i) : fact(.i)
```

Remember that in an assignment the thing on the left is the address where we're storing a value. So here, we just add the base address we got back from `alloc` and the index to get that address. The parentheses around the expression on the left aren't strictly necessary. Later in the `filltable` function, we fetch the value at a position in the array with the expression `.(.ftab+.i)`, and here the parentheses are necessary, because the dot operator has a higher precedence than the addition.

Connecting to HTML

Several of the statements in both `filltable` and `init` are calls to built-in functions that populate elements on a web page. The first time one of these functions is called, a new tab/window is opened in the browser for rendering the new HTML elements. This means that calls to `iprint` and `sprint` still go to the standard output in the original window. The relevant statements in `init` in this example are

```
html("<center>")
button("Make Table", filltable)
html("<p>Factorials<p>\n</center>\n")
```

which uses two of the built-in functions, `html` and `button`. The `html` function just appends the argument string to the newly created tab/window. In principle, you could create a button on the page using the `html` function, but we need a way to specify what the button does. The approach we've chosen for Tranquility is to have a separate button creating function where the first argument is the label on the button, and the second argument is the Tranquility function to call when the button is pressed. Notice here, that we have the name of the function without the parentheses after it. That's because we're not calling the function at the time we're creating the button. We're just telling it what function to call when the time comes. After pressing the button labeled "Make Table," this is what appears in the HTML window:

Element Identifiers

For buttons, images, labels, tables, and timers, the functions that create them all return identifiers that can be used in later calls to other built-in functions that deal with them. Normally, these return values are saved in variables that are then later passed to the functions that modify the respective element. As an example of how all of these are used, consider an image we might place on an HTML page. We might create the image with a call such as:

```
img1 :  makeimg()
```

The URL for the image that is displayed there is established by a call to `setimg` such as:

```
setimg(.img1, "cat.png")
```

Timers

Timers are single events, rather than continuous regular events. If you want to make a timer fire continuously on a regular interval, then you want to restart the timer inside the timer handling function. To illustrate this and other identifier use, here is a small program that implements a simple stopwatch.

```
var tim, lab, labstr, secs

fun init() {
    labstr :  alloc(10)
    secs :  0
    html("<center>\n")
    lab :  makelabel("0")
    html("<br>\n")
    button("Start", swstart)
    button("Stop", swstop)
    button("Reset", swreset)
}

fun swstart() {
    tim :  timer(1000, timestep)
}

fun swstop() {
    stoptimer(.tim)
}

fun swreset() {
    secs :  0
    setlabel(.lab, "0")
}
```

```
fun timestep() {
     tim :  timer(1000, timestep)
     secs :  .secs + 1
     i2s(.labstr, .secs)
     setlabel(.lab, .labstr)
}
```

Language Reference

This section is a terse description of the details of Tranquility.

Lexical Elements

Tranquility has the following seven reserved keywords: `else`, `fun`, `if`, `loop`, `return`, `until`, and `var`. These reserved words cannot be used as identifiers for variable or function names. The remaining lexical elements of the language include:

1. *Character constants* are enclosed in single quotes (`'`) and consist of a single character, or a special character denoted by a backslash followed by a regular character, (e.g. `'\n'`). The supported special characters include `\b`, `\n`, `\r`, `\t`, and `\\`.
2. *Integer constants* are composed of one or more digits, optionally preceded by `0x`. Constants that begin with `0x` are hexadecimal constants, and the letters `a-f` are allowed as digits. Constants beginning with `0` (but not `0x`) are taken to be octal. All other constants are interpreted as decimal.
3. *Literal strings* are enclosed in double quotes (`"`) and contain zero or more characters including the escaped special characters discussed in the character constants.
4. *Identifiers* consist of unquoted alphanumeric characters beginning with an alphabetic character. Both upper and lower case alphabetic characters are allowed, and the underscore character (_) is considered an alphabetic character.
5. *Comments* are denoted by a pound sign (`#`). The comment character through the end of the line are ignored.

Syntax

The following is the syntax for Tranquility given in BNF notation:

```
<program> ::= <var-list> <fun-list>

<var-list> ::= ϵ | "var" <id-list> "\n" <var-list>

<id-list> ::= ID | ID "," <id-list>

<id-list-e> ::= ϵ | <id-list>

<fun-list> ::= <fun-decl> | <fun-decl> <fun-list>

<fun-decl> ::= "fun" ID "(" <id-list-e> ")" "{" "\n" <var-list> <stmt-list> "}" "\n"

<stmt-list> ::= ϵ | "\n" <stmt-list> | <stmt> <stmt-list>

<stmt> ::= <expr> ":" <expr> "\n" | <expr> "\n" | <if-stmt>
     | "until" <expr> "\n" | "loop" "{" "\n" <stmt-list> "}" "\n"
     | "return" "\n" | "return" <expr> "\n"

<if-stmt> ::= "if" <expr> "{" "\n" <stmt-list> "}" "\n"
     | "if" <expr> "{" "\n" <stmt-list> "}" "\n" "else" "{" "\n" <stmt-list> "}" "\n"
     | "if" <expr> "{" "\n" <stmt-list> "}" "\n" "else" <if-stmt>

<expr-list> ::= <expr> | <expr> "," <expr-list>

<expr-list-e> ::= ϵ | <expr-list>

<expr> ::= INTEGER | CHARACTER | STRING | ID | ID "(" <expr-list-e> ")"
     | "(" <expr> ")" | "." <expr> | "-" <expr> | "~" <expr>
```

```
| <expr> "+" <expr> | <expr> "-" <expr> | <expr> "*" <expr>
| <expr> "/" <expr> | <expr> "%" <expr> | <expr> "&" <expr>
| <expr> "|" <expr> | <expr> "^" <expr> | <expr> "==" <expr>
| <expr> "!=" <expr> | <expr> "<" <expr> | <expr> "<=" <expr>
| <expr> ">" <expr> | <expr> ">=" <expr> | <expr> "<<" <expr>
| <expr> ">>" <expr>
```

Statements

There are relatively few statements in Tranquility. Here, we look at what each of them does.

- **Assignment Statement:** When executing the assignment statement, the expressions on both the left and right side are evaluated. The value on right side of the assignment is stored into memory at the address given by the value of the left-hand expression.
- **Expression Statement:** The expression statement only exists to allow for function calls where the return value is ignored. Other expressions can appear as statements, but no other expressions will have any effect.
- **If Statement:** The expression after the `if` keyword is evaluated first. If the result of the expression is non-zero, then the statements in the true branch of the if statement are executed. Otherwise, (the expression evaluates to zero), the statements in the false branch are executed.
- **Loop Statement:** The statements in the body of the loop statement are executed repeatedly.
- **Until Statement:** The expression in the until statement is evaluated. If the result is non-zero, then control breaks out of the enclosing loop statement and continues with the statements following that loop body.
- **Return Statement:** This statement immediately returns control away from the currently executing function back to the calling function. If an expression follows the `return` keyword, then it is evaluated and the result becomes the function's return value. Otherwise, the return value is zero.

Expressions

As with most languages, expressions in Tranquility are sequences of symbols that represent the computation of values. The syntax given earlier shows that expressions are recursively built from atomic values and from combining expressions with operators. The different types of expressions are summarized as follows:

- Any instance of an integer constant, character constant, or literal string is an expression. The value of an integer constant expression is the numeric value represented by the sequence of digits in the relevant base. The value of a character constant is the number defined in the ASCII character set for that character. The value of a literal string is the memory address of the first character in the string. Strings are terminated by a zero value.
- An identifier is an expression whose value is the memory address that identifier labels. The memory spaces for functions and variables are separate, so programmers should not attempt to do arithmetic on the addresses of functions.
- An identifier followed by a possibly empty list of expressions enclosed in parentheses is a function call. Each of the argument expressions is evaluated and passed to the function named by the identifier. The value of a call expression is the return value from the function called.
- Any expression may be enclosed in parentheses to override the natural precedences of the operators.
- The dot (`.`) operator performs a fetch. It, along with other unary operators, has the highest precedence among operators. The value of a fetch expression is the contents of the memory location whose address is given by the value of the expression following the operator.
- Tranquility supports the usual arithmetic operators with their conventional precedences. Addition is denoted by `+`, subtraction by `-`, multiplication by `*`, division by `/` and remainder by `%`. A unary `-` operator performs negation as usual.
- The usual comparison operators are also present in the language with `==` for equality, `!=` for non-equality, `<` for less-than, `<=` for less-than or equal to, `>` for greater-than, and `>=` for greater-than or equal to. The value of a conditional expression is one if the condition tests true and zero if false. Comparison operators have lower precedence than arithmetic operators, but higher precedence than bitwise operators.

- The final class of expressions in Tranquility is the bitwise expression. Boolean AND is denoted by `&`, OR by `|`, and XOR by `^`. The unary operator for bitwise complement is `~`. Bitwise Boolean operators have the lowest precedence. There are also two bitwise shift operators: left shift (`<<`) and right shift (`>>`). Their precedence is the highest among binary operators.

Built-In Functions

These are the built-in functions in Tranquility. They are listed in alphabetical order for easy reference and include descriptions of each function and its argument use.

- `alloc(n)`: This function allocates a block of memory with `n` locations and returns the address of the first location.
- `button(label, fun)`: Create a button on the HTML page with the label given by the first argument. When the button is pushed, the Tranquility function identified by the second argument is called. This function returns an identifier for the button that can be used in later to `buttonlabel`.
- `buttonlabel(b, label)`: Set the label on the button identified by the first argument to the string passed as the second argument. The first argument should be a previously saved return value from a call to `button`.
- `free(p)`: Returns the previously allocated memory block to the free list. The argument is the memory address returned by the earlier call to `alloc`. (Currently unimplemented)
- `html(s)`: Send the HTML code in the argument string to the HTML window for Tranquility.
- `i2s(str, n)`: Produce a string that contains the decimal (base-10) representation of the integer value passed as the second argument. The first argument should be the address of a block of memory large enough to contain the string.
- `iprint(n)`: Print the integer in decimal in the standard output window.
- `iread(s)`: Bring up a pop-up dialog asking the user to enter an integer, using the argument string as the message in the dialog. The numeric value of the entered integer is returned from the function.
- `makeimg()`: This function creates an image without source specified. Its return value can be used to later specify the source of the image through calls to `setimage`.
- `makelabel(s)`: Create a label setting its contents to the string passed as an argument. It returns an integer label identifier that can be passed to `setlabel` to change the text in the label.
- `maketable(r,c,f)`: Create a table with `r` rows and `c` columns, returning an integer table identifier. The function `f` is called each time the user clicks on a cell of the table. It is passed the row and column as arugments. The return value can be used in later calls to `setcell` and `setcellcolor` to identify which table is being affected.
- `random(n)`: Returns a random number in the range $[0, \mathtt{n})$.
- `setcell(t,r,c,s)`: Set the contents of the cell at row `r` and column `c` of table `t` to the string `s`. The first argument should be a value returned from `maketable`.
- `setcellcolor(t,r,c,color)`: Set the background color of the cell at row `r` and column `c` of table `t` to the color named by the string `color`. The first argument should be a value returned from `maketable`.
- `setimg(n, src)`: Set the source property of an image. The return value from the earlier `makeimg` call should be provided as the first argument. The second argument should be the string specifying the image source.
- `setlabel(n, s)`: Changes the text in the label identified by the first argument (a value returned from an earlier call to `makelabel`) to the string specified by the second argument.
- `sprint(s)`: Print the string whose address is passed as the argument on the standard output.
- `sread(s, p)`: Pop up a dialog prompting the user for a string as input, using the second argument string as the message in the dialog. The string is copied into the memory starting at the address passed as the first argument.
- `stoptimer(n)`: Cancels a previously created timer that has not yet fired. The argument passed should be the value returned from an earlier call to `timer`.
- `timer(ms, fun)`: This function creates a timer that will fire `ms` milliseconds after it is created. When it fires, it will call the function identified in the second argument. The function returns an integer identifier that can be used to cancel the timer with a call to `stoptimer`.

Under the Covers

The compiler for Tranquility is a fairly basic compiler using `lex` to generate the lexical analyzer and `yacc` to generate the parser. The grammar used by `yacc` is the same as the syntax given in BNF earlier. Output from the parser consists of a list of syntax trees, one for each function in the source program. After parsing, the syntax trees are traversed twice. The first traversal resolves all identifier references. Global identifiers are resolved in terms of their absolute memory addresses. Local variables and parameters are resolved as offsets onto the stack. The second traversal over the syntax trees generates tvm code. The tvm is implemented in JavaScript and is contained in the file `tvm.js`.

The Tranquility Virtual Machine

In the tvm, instructions and data live is separate memory spaces (much like what is often called a Harvard architecture). A function "address" is actually an index into a list of functions. Data is stored in a 64Kword array. Global data is allocated starting at address 0 and the stack grows down from the top of memory.

There are only two registers in the tvm, a stack pointer (`sp`) and a frame pointer (`fp`). All expression evaluation takes place on the stack, obviating the need for general purpose registers. The frame pointer is used as the base address for offset references to local variables and function parameters. No program counter is present because instruction execution is handled through recursive evaluation of instruction lists, rather than sequential instruction flow with jumps.

The tvm instruction set is as follows:

Opcode	Instruction	Description
1	PUSH	Push the following value onto the stack
2	FETCH	`x = pop(); push(mem[x])`
3	STORE	`v = pop(); a = pop(); mem[a] = v`
4	IF	`x = pop();` If `x` $\neq 0$ evaluate the next list and skip the second; otherwise skip the first list and evaluate the second
5	LOOP	Repeatedly evaluate the following list of instructions
6	BREAK	`x = pop();` If `x` $\neq 0$ exit the enclosing loop list
7	RETURN	Return from a called function
8	CALL	Call the function identified in the following word
9	FPPLUS	Add the `fp` to the value on the top of the stack
10	ADD	`y = pop(); x = pop(); push(x+y)`
11	SUB	`y = pop(); x = pop(); push(x-y)`
12	MUL	`y = pop(); x = pop(); push(x*y)`
13	DIV	`y = pop(); x = pop(); push(x/y)`
14	MOD	`y = pop(); x = pop(); push(x%y)`
15	NOT	`x = pop(); push(~x)`
16	AND	`y = pop(); x = pop(); push(x&y)`
17	OR	`y = pop(); x = pop(); push(x|y)`
18	XOR	`y = pop(); x = pop(); push(x^y)`
19	EQ	`y = pop(); x = pop();` If `x` $=$ `y` `push(1)`, otherwise `push(0)`
20	NEQ	`y = pop(); x = pop();` If `x` $\neq$ `y` `push(1)`, otherwise `push(0)`
21	LT	`y = pop(); x = pop();` If `x` $<$ `y` `push(1)`, otherwise `push(0)`
22	LEQ	`y = pop(); x = pop();` If `x` $\leq$ `y` `push(1)`, otherwise `push(0)`
23	GT	`y = pop(); x = pop();` If `x` $>$ `y` `push(1)`, otherwise `push(0)`
24	GEQ	`y = pop(); x = pop();` If `x` $\geq$ `y` `push(1)`, otherwise `push(0)`
25	POP	`pop()`
26	LSHIFT	`s = pop(); x = pop(); push(x << s);`
27	RSHIFT	`s = pop(); x = pop(); push(x >> s);`

Note that the IF instruction must be followed by two lists of instructions, and the LOOP instruction must be followed by one list of instructions.

Calling Sequence

The following steps are performed when calling a function:

1. Each argument expression is evaluated and pushed onto the stack from right to left. This leaves the first argument on the top of the stack.
2. Zero is pushed onto the stack n times, where n is the number of words of local storage necessary in the function.
3. The frame pointer is pushed onto the stack.
4. The stack pointer value (prior to pushing `fp`) is copied to the frame pointer.
5. The instruction list for the function is evaluated.

Upon completing evaluation of the function list or execution of a RETURN instruction, the return value is expected to be on top of the stack. The following sequence accomplishes the return:

1. The top of the stack is popped into r.
2. The frame pointer is copied into the stack pointer.
3. The top of the stack is copied into the frame pointer.
4. The stack pointer is incremented by the number of local variables and parameters for the function.
5. r is pushed onto the stack.