FUNctions, Loops, and Expressions

Lecture 04 – Fall 2018

Announcements

- UTA Tech Talk - "Things I Wish I Knew When I Took COMP110"
  - **Tomorrow - Wednesday 9/5 - 5pm in Fred Brooks 009**
  - Panel of UTAs talking about early experiences in Computer Science
  - Pointers on things to be doing and thinking about early in CS career
  - Open Q&A to ask students in the major the questions on your mind

- Worksheet 0 - Posted and Due Saturday 9/8 at 11:59pm
  - You will scan on your phone or library scanner and hand-in PDF via Gradescope
  - You must handwrite your submissions

- Problem Set 1 - Dwight's Beet Farm
  - Goes Out Tonight
  - Due Monday 9/10 at 11:59pm

Videos for Thursday's Warm-up Questions

- V08 while Loop Control Statement (8m)
- V09 Functions: Overview (10m)
- V10 Functions: Parameters and Arguments (3m)
- V11 Functions: Return Statement (5m)

- We will introduce while Loops and Functions in today's lecture
  - The videos will dive into the details of each

Functions are *inspired by* their mathematical relatives...

$$f(x) = (x \times 3) + 1$$

- What is $f(3)$? What is $f(f(1))$? Answer on PollEverywhere.

- We know that to compute $f(5)$ we
  1. Assign 5 to x such that the expression becomes $(5 \times 3) + 1$
  2. Using arithmetic, simplify to $15 + 1$
  3. Using arithmetic, simplify to 16. Final answer.

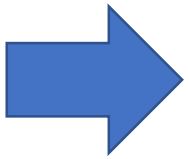
- Let's express the same function in code.

Follow along: Our First Function

- Open 04 / 00-function-demo-app.ts
- Let's define the function ***f*** together!
  - Notice: It is defined outside of the *main* function!
  - We will break down the syntax next.
- Then let's call function ***f*** from within main.

```
export let main = async () => {
  let input: number = await promptNumber("?");
  let answer: number = f(input);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

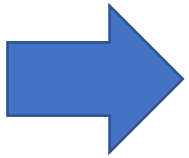
Tracing a function call (1/10)



```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(input);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

The user is prompted for a number. The number entered is assigned to the **input** variable. Let's imagine **5** was entered.

Tracing a function call (2/10)



```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(input);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

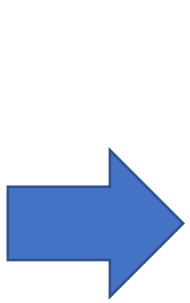
5

input

The variable answer is declared and initialized to be **f(input)**

But what is f(input)? The computer must compute it!

Tracing a function call (3/10)



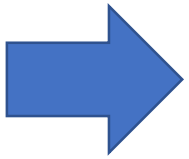
```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(5);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

input

First, it is going to substitute the **input** variable reference with its value **5**.

Note: You will never *see* this happen. The computer is doing this as it runs your program.

Tracing a function call (4/10)



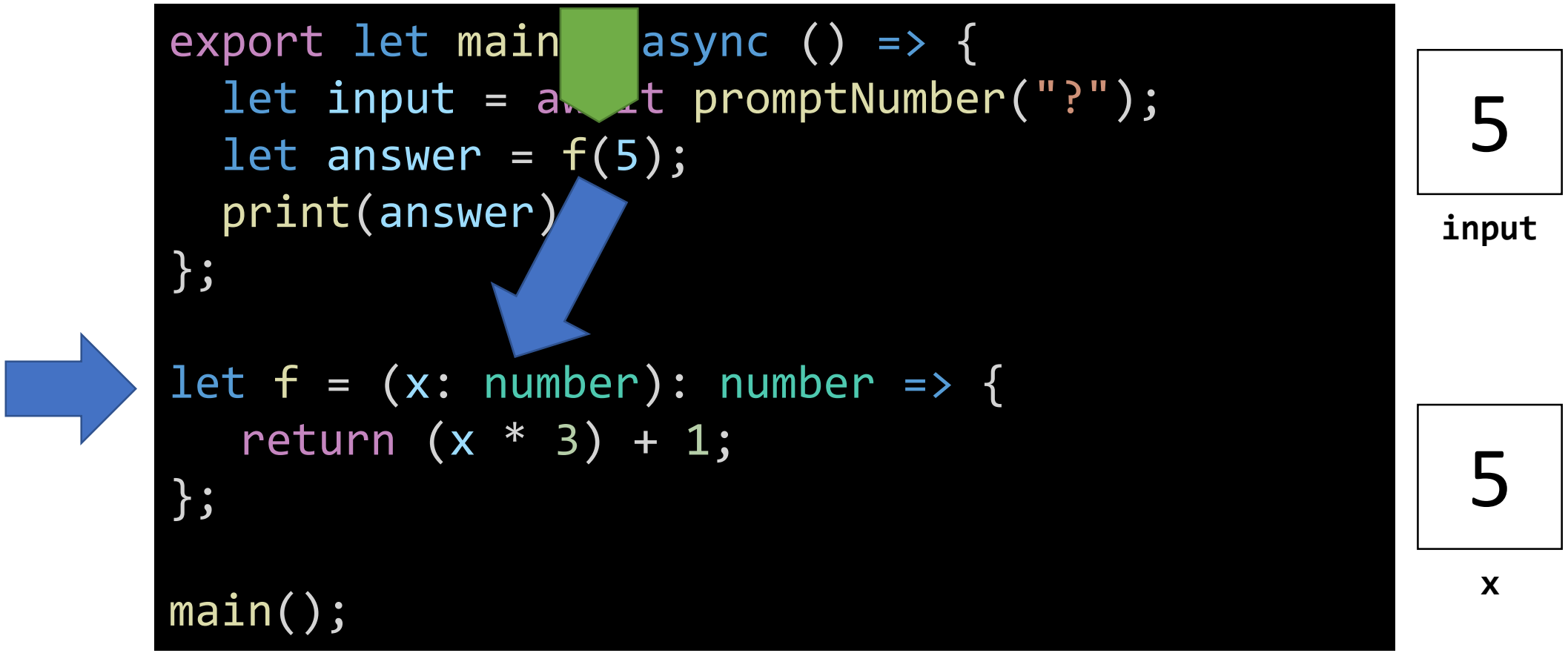
```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(5);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

5

input

Now the computer is ready to call function ***f.*** It drops a bookmark.

Tracing a function call (5/10)

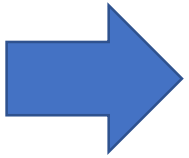


Then, we assign the value in parenthesis **(5)** to ***f***'s variable ***x***.

We will cover this process in depth next.

Tracing a function call (6/10)

```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(5);
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```



5

input

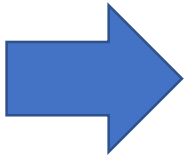
5

x

The function is entered and return statement is reached.
The computer needs to calculate the result of this expression.

Tracing a function call (7/10)

```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(5);
  print(answer);
};

let f = (x: number): number => {
   return (5 * 3) + 1;
};

main();
```



5
input

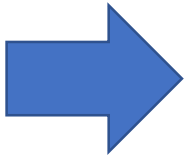
5
x

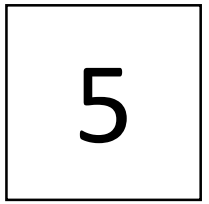
First, it will substitute the ***x*** variable with its current value.

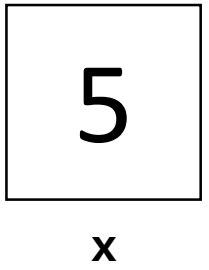
Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (8/10)

```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = f(5);
  print(answer);
};

let f = (x: number): number => {
   return 16;
};

main();
```





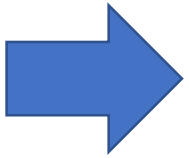


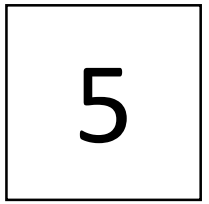
Then, it will follow PEMDAS.

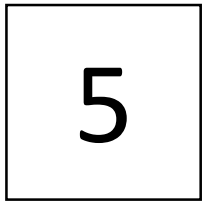
Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (9/10)

```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = 16;
  print(answer);
};

let f = (x: number): number => {
   return 16;
};

main();
```







Once the return statement is computed down to a single value, it is ***returned*** *to the function call's bookmark and replaces it.*

Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Tracing a function call (10/10)

```
export let main = async () => {
  let input = await promptNumber("?");
  let answer = 16;
  print(answer);
};

let f = (x: number): number => {
   return (x * 3) + 1;
};

main();
```

5	16
input	answer

Once the return statement is computed down to a single value, this is *returned* to the *function call's bookmark and replaces it.*

The computer resumes processing the program at this line and initializes answer to 16.

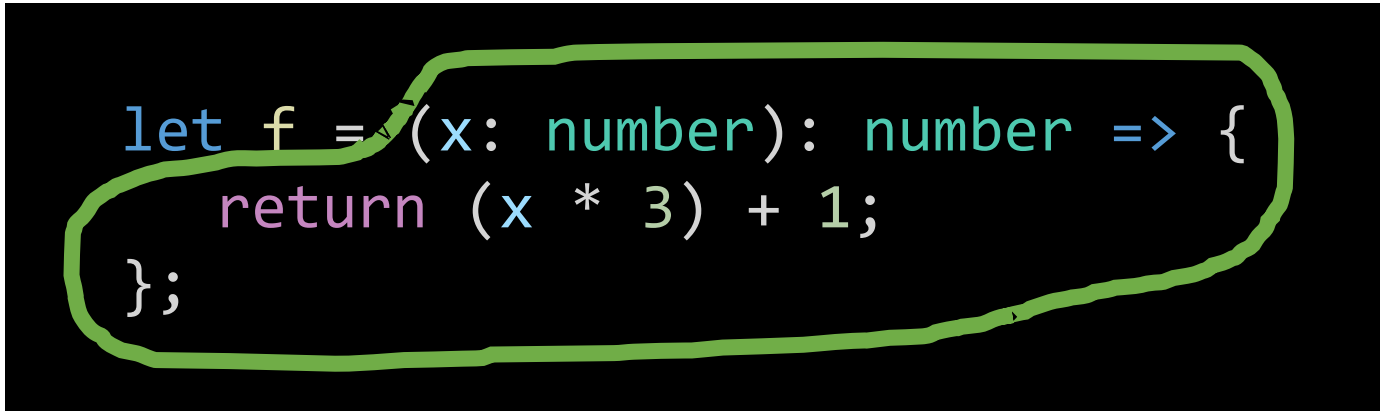
Note: You will never *see* this happen in your code. The computer is doing this as it runs your program.

Function Definition Walk through (1/5)

```
let f = (x: number): number => {
    return (x * 3) + 1;
};
```

"Let ***f*** be..."

Function Definition Walk through (2/5)



"Let ***f*** be... **a function**..."

Function Definition Walk through (3/5)

```
let f = (x: number): number => {
    return (x * 3) + 1;
};
```

"Let ***f*** be... a function...

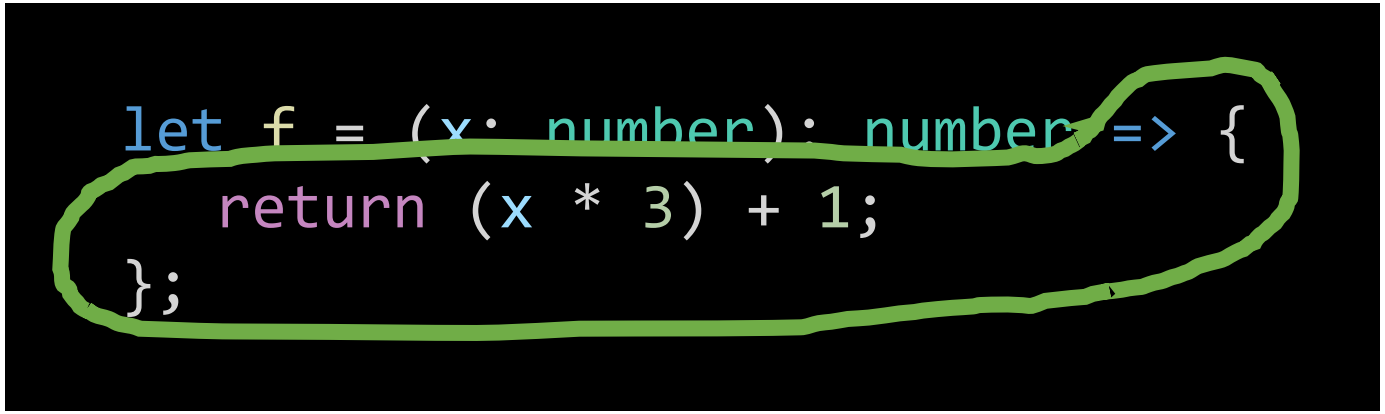
that needs a number value named *x*..."

Function Definition Walk through (4/5)

```
let f = (x: number): number => {
   return (x * 3) + 1;
};
```

"Let ***f*** be... a function... that needs a number value named *x*...
and will return a number value when called."

Function Definition Walk through (5/5)



"Let ***f*** be a function that needs a number value named *x* and will return a number value when called."

"When ***f(x)*** is called, the result of computing **(x * 3) + 1** will be returned to the caller."

Typing Speed Contest

- Open up 00-contest.txt and await further instruction…
- Rule #1 – You can't use copy/paste!
- Rule #2 – Wait until I say go!

Introducing: **`while`** Loops

- General form of a **`while`** loop statement:

```
while (<boolean expression "test">) {
    <repeat block – statements in braces run when test is true>
}
```

- ***Like*** an **`if-then`** statement:
  - the test you place in the parenthesis must be a `boolean` expression
  - if the test evaluates to **`true`**, the computer will move to the first line of code in the repeat block
  - If the test evaluates to **`false`**, the computer will *jump* over the repeat block

- ***Important! Unlike*** an if-then, **after the last statement in the repeat block** completes, the computer will next ***jump backwards*** **up to the test** and start afresh.

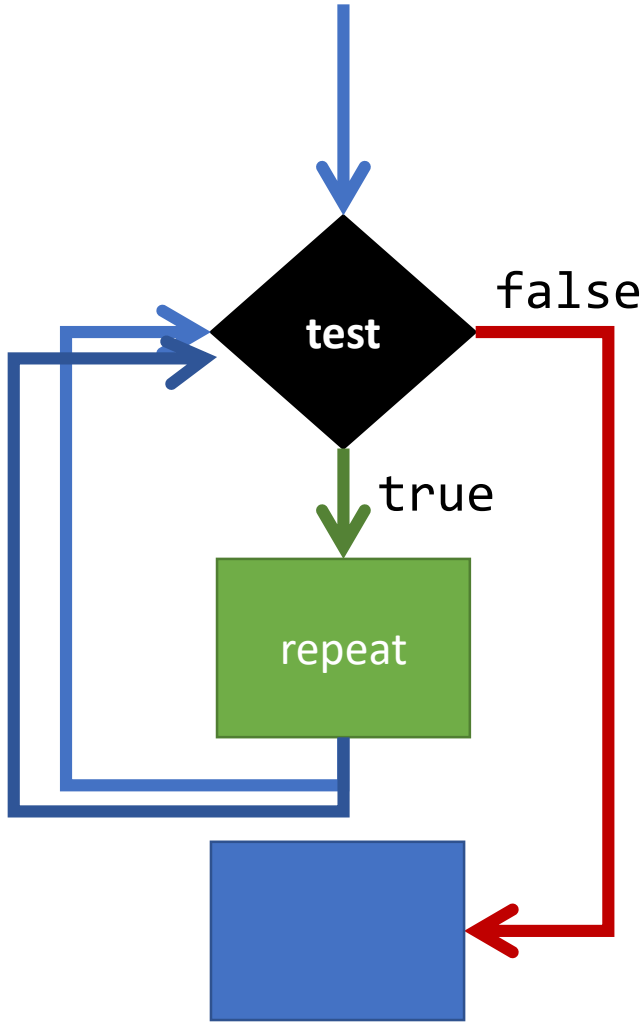
- A **`while`** loop statement can be used *anywhere* you can write a statement.

while loop Flow of Control

1. When a **`while`** statement is encountered, its **`boolean`** **test** expression is evaluated

2. If the **test** is **`true`**,
   a) then the processor will **proceed into the repeat block**.
   b) At the end of the repeat block, the processor jumps back to **step 1**.

3. If the **test** is **`false`**, the processor will jump over the repeat block and continue on.



Follow-along: Computers Love Looping

- Open 01-while-love-app.ts

- Let's try making it print "I love you" a lot…

```
import { print, promptNumber } from "introcs";

export let main = async () => {

    print("Hello. It's me again... your computer...");
    let n = await promptNumber("How much do you love me?");

    let i = 0;
    while (i < n) {
        print("I love you, too ;) " + i);
        i = i + 1;
    }

};

main();
```

Follow Along: Improving the 8-Ball

- Open 02-better-8-ball-app.ts and let's move the generation of a random response string to its own function.

```
export let main = async () => {
    let question = await promptString("Ask a Yes/No Question");
    print(randomResponse());
};
```

```
let randomResponse = (): string => {
    let responseCode = random(0, 2);
    if (responseCode === 0) {
        return "Most definitely not.";
    } else {
        if (responseCode === 1) {
            return "Ask again later.";
        } else {
            return "It is certain.";
        }
    }
};
```

Follow Along:

How can we make it so that we can continue asking the 8-ball questions without refreshing?

Repeating a Game

```
export let main = async () => {
   while (true) {
      let question = await promptString("Ask a Yes/No Question");
      print(randomResponse());
   }
};
```

Hands-on: Stopping the Loop

1. Open 04-stopping-8-ball-app.ts

2. Notice the `while` loop's condition is the current value of **`isPlaying`**

3. Underneath the TODO, implement the following logic:

4. When `shouldContinue` is equal to "yes", `isPlaying` should be assigned `true.` Otherwise, **`isPlaying`** should be assigned **`false`**.

5. Save and test. You should be able to respond "no" and the game stops.

6. Check-in on PollEv.com/compunc and try to talk through *why* the loop stops with a neighbor.

Repeating a Game

```
export let main = async () => {
    let isPlaying = true;
    while (isPlaying) {
        let question = await promptString("Ask a yes / no question...");
        print(randomResponse());

        let shouldContinue = await promptString("Ask another? yes / no");
        if (shouldContinue === "yes") {
            isPlaying = true;
        } else {
            isPlaying = false;
        }
    }

    print("Have a great day.");
};
```

Expressions

- **Expressions** are a fundamental building block in programs

- Expressions are analogous to the idea of clauses in English
  - Single clause sentence:
    "I am a student."
  - Multiple clause sentence:
    "I am a student and I am currently sitting in COMP110."
  - In English, *Sentences* are *more expressive* through the creative use of *clauses*

- In code, ***statements*** are *more expressive* through creative uses of ***expressions!***

How can we compute the volume of a cube using different expressions?

```
let answer: number;
answer = 3 * 3 * 3;
```

1. We can "hard-code" the expression with exact numbers.

How can we compute the volume of a cube using different expressions?

```
let answer: number;
let length = 3;
answer = length * length * length;
```

2. We can use a variable to hold the length of a side of the cube.

Notice, in doing so, our *expression* has more meaning:
`length * length * length` is more expressive than `3 * 3 * 3`

How can we compute the volume of a cube using different expressions?

```
let answer: number;
let length = await promptNumber("Length:");
answer = length * length * length;
```

3. We can use the **`promptNumber`** function to allow *any number*!

Our program is more generally useful.

How can we compute the volume of a cube using different expressions?

```
let cubeVolume = (side: number): number => {
   return side * side * side;
};
```

```
let answer: number;
let length = await promptNumber("Length:");
answer = cubeVolume(length);
```

4. We can write a *function* to compute the volume and *call the function*.

This has two benefits:

1. It reads more naturally: "answer is assigned the result of calculating `cubeVolume` using the given `length`"
2. We can *reuse* the `cubeVolume` function without rewriting the equation!

Expressions

There are two **big ideas** behind expressions:

1. *Every* expression *simplifies to a single value at runtime*
   - Thus, every expression has a *single type*.
   - This occurs *only* when the program runs (runtime) and when the processor reaches the expression in the program.

2. Anywhere you can write an expression you can substitute any other expression *of the same type*

Where have we *used* expressions?

- Assignment operator:

```
let <name>: <type> = <expression of same type>;
```

- We are able to assign *any* of the expressions below because each results in a single *number* value:

```
let x: number = 1;
let y: number = x + 1;
let cubeY: number = y * y * y;
```

- Notice that we are combining *multiple* expressions in the same line.
- After each line completes, the declared variable has a *single* value.

Where else have we *used* expressions?

- if-then statement

```
if (<boolean expression>) {
   // ... elided ...
}
```

- ***Any boolean expression*** can be used as the test expression in an if-then statement
  - `if (age >= 21) { // ...`
  - `let is21 = (await promptNumber("Age")) >= 21;`

    `if (is21) { // ...`

- When the computer reaches the boolean expression of an if-then statement, it evaluates the expression down to the single value of either **`true`** or **`false`**.

Repeating a Game – Alternative Implementation

```
export let main = async () => {

    let isPlaying = true;
    while (isPlaying) {
        let question = await promptString("Ask a Yes/No Question");
        print(randomResponse());
        isPlaying = (await promptString("Continue? yes / no")) === "yes";
    }

    print("Have a great day.");

};
```

Expressions of Various Kinds

- Literal Values
  - `3.14`
  - `true`
  - `"hi"`
- Variable Access
  - `x`
  - `compCourseNumber`
- "Unary" operators
  - `-x` (number *negation)*
  - `!is21` (boolean negation)
- Function Calls
  - `cubeVolume(x)`
- "Binary" Operators
  - Arithmetic
    - `1 + 2`
  - Concatenation
    - `"Hello " + name`
  - Equality
    - `x === 1`
    - `x !== 1`
  - Relational
    - `age >= 21`
    - `age < 13`