The Selection Control Structure

There are a few fundamentally different ways control may flow through a program. We call them control structures. In all of the work you've done thus far, every statement you wrote simply executed in order. That pattern of control is the first of the fundamental control structures and we call it the sequence control structure. Sometimes we don't want that kind of behavior, but instead only want to select to execute particular code based on meeting certain conditions. This gives rise to the second fundamental control structure: selection. Here, we'll examine how selection works and look at a few different implementations of the selection control structure.

I. Relational Operations and Boolean Expressions

The first thing we'll need are the relational operators:

\[
\geq \quad > \quad = \quad < \quad \leq \quad \neq
\]

They simply allow us to compare two operands of the same type. The result is necessarily true or false in all cases; either the comparison is a correct one or it is not.

We use the relational operators to build Boolean expressions or logical statements, i.e. sentences that evaluate to true or false. To build a simple Boolean expression, all we need to do is put something (technically called an operand) on both sides of a relational operator. So, what can operands be? We have choices:

- constants, like 7 or 2 or even named constants like PI
- variables, like a or height, where when the Boolean expression is evaluated, the value of a variable is substituted
- arithmetic expressions, usually built up from arithmetic operators and variables and constants, like a + 7, where the arithmetic expression is evaluated and the result is substituted in evaluating the Boolean expression

Whatever the operands are, it is vital that both operands evaluate to the same type. Usually, that means both must be numeric, but it's also possible to compare characters to other characters, for example.

Examples: Let's write down a few examples of simple Boolean expressions:

These operators are treated strictly as binary operators; that is, they must have two operands. Furthermore, our definition of a simple Boolean expression allows exactly one relational operator and exactly two operands. So, while in the pure math view, we can say things like \(0 < x < 5\), we disallow that here, as many programming languages disallow such expressions.

---

1 I'm intentionally capitalizing "Boolean" and you should too. It comes from George Boole's name, and he was one of the foundational contributors to logic.
II. Three Fundamental Patterns for Making Decisions

Great, so we have Boolean expressions, but how do we use them? The basic way is to use them in if statements. Here's a basic template for a Boolean expression expr:

```plaintext
if expr
    what to do when expr is true
```

Perhaps we also want to do something particular when an expression is false. In such cases, we can add an else clause:

```plaintext
if expr
    what to do when expr is true
else
    what to do when expr is false
```

But, that's not all! There's a third fundamental pattern. Say we do a first test, via expr_1, and have something in mind for when it passes. But if it fails, we instead check a second test, via expr_2. If that passes, we know what to do, but if it fails, we check something else (say, via expr_3). This process could go on for as many tests as we like, but we may have something in mind for when all the tests fail. Here's a basic template for this kind of conditional thinking:

```plaintext
if expr_1
    what to do when expr_1 is true
else if expr_2
    what to do when expr_2 is true
else if expr_3
    what to do when expr_3 is true
    ...
else if expr_n
    what to do when expr_n is true
else
    what to do when all of expr_1, expr_2, ..., expr_n are false
```

Note that in the last pattern, there may be cases where we don't have anything in mind to do in the final else clause. In that case it's simply omitted. (When it's there, think of it as the "when all else fails" case.)

Also note (extremely important), that the "what to do" clauses are always indented.

III. Basic Examples

Example: Let's say we have a variable temp defined to be today's temperature in degrees Fahrenheit. Suppose a user has been prompted to enter a value for temp. Write pseudocode to display a message saying that it's freezing today or it is not.

Problem: Given this piecewise function

```plaintext
f(x) = \begin{cases} 
  x - 5 & \text{when } x < 10 \\
  0 & \text{when } x < 100 \\
  3x^2 & \text{when } x \geq 100 
\end{cases}
```

along an initialized variable x for the x in the function and a variable y for the f(x) as defined, write equivalent pseudocode.
Example: Say we have a variable `examScore` defined to be an exam grade between 0 and 100 and initialized and a variable `letterGrade` defined to be the appropriate letter grade for `examScore`. Initialize `letterGrade`.

IV. Logical Operators and Compound Statements

We called the Boolean expressions we wrote above *simple* Boolean expressions (or, officially, atomic statements). They were the shortest kind we could write: operand, relational operator, operand. What if we want to test something more complex? To do so, we need to use the **logical operators** -- and, or, and not. We can use these operators to join other Boolean expressions to make more complicated Boolean expressions. Let's say `p` and `q` are some Boolean expressions. Then we can write things like…

- `p` and `q`
- `p` or `q`
- `not p`

Examples: Let's look at several examples, where I'll use relatively descriptive variable names that hopefully make sense to you:

Expressing that something is moving only horizontally, not vertically

Expressing that something has no velocity in the horizontal direction or no velocity in the vertical direction

Expressing the opposite of the last condition

Expressing that something does not have a zero horizontal velocity and does not have a zero vertical velocity

Expressing that we have a flush in poker.

Expressing that we have a straight in poker.
Note that some of the examples had the same meaning, no matter what values the variables had. Such statements are said to be logically equivalent. There are several laws for determining logical equivalence and transforming statements (and they come up in a discrete math course), but one group in particular that's worth mentioning and fits here is De Morgan's Laws. For statements $p$ and $q$, they say that "not \((p \text{ and } q)\)" is equivalent to "not $p$ or not $q$" and that "not \((p \text{ or } q)\)" is equivalent to "not $p$ and not $q$.

V. Investigating the Logical Operators: Truth Tables

Tools called truth tables formally define the meanings of logical operators. They spell out all of the possible cases of all of the atomic statements that make up compound expressions. Let's fill in the basic truth tables:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \text{ and } q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \text{ or } q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>not $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

We could build up truth tables with multiple columns to investigate compound logical statements. Here's an example table, whose 4th and 7th columns essentially prove one of De Morgan's Laws:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \text{ and } q$</th>
<th>not ((p \text{ and } q))</th>
<th>not $p$</th>
<th>not $q$</th>
<th>not $p \text{ or } not q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Why should you care? Well, aside from the definitions, if you think you have a logical statement that's equivalent to some other statement, you can check that equivalence by making sure all rows match for columns with supposedly equivalent statements.

VI. Practice with Compound Statements

**Problem:** Express $0 < x < 5$.

**Problem:** Draw a number line and illustrate the opposite of the above condition. Express it in two different ways.
Problem: Say that one way you can pass some competency is to have an exam average (examAvg) of 90 or above. Alternatively, you pass with an exam average of 75 or above, as long as you also have an attendance score (attScore) of 100. Express this condition.

VII. Syntax Interlude

Now that we've looked at the fundamental computation concept of selection, we must pause to consider some matters of programming language syntax.

Some of the standard relational operators are a bit tough to type, so let's first note the syntax for each of the relational operators:

<table>
<thead>
<tr>
<th>≥</th>
<th>&gt;</th>
<th>=</th>
<th>&lt;</th>
<th>≤</th>
<th>≠</th>
</tr>
</thead>
</table>

Here are some notes of caution:

- When relational operators consist of two characters, the order matters!
- Equality vs. assignment is a tricky matter, for beginning programmers and experienced programmers alike. In many languages, = means assignment and == means equality. A common pitfall is to write something like a = 5 where you want a Boolean expression, when in fact you want a == 5. (Is the former a Boolean expression?) Worse yet, some compilers will accept the former, won't give you a warning or error, generally return true, and change the value of a in the process. Be VERY careful!

Next up, we have the logical operators:

<table>
<thead>
<tr>
<th>and</th>
<th>or</th>
<th>not</th>
</tr>
</thead>
</table>

In programming language syntax, you can use parentheses just as in theory to establish precedence. Always be explicit about this. (Some compilers/languages have precedence rules for the logical operators, but we haven't talked about them, and I'd rather you get in the habit of thinking problems through explicitly than memorizing some precedence rules for now that you'll forget and have to look up in a year anyway.)

Finally, translating the basic patterns of if and else clauses and statements to code is relatively natural. There are a few caveats, though, dealing with the placement of the Boolean expressions and the fact that what we do conditionally can turn out to be multiple lines of code. Let's look at an example to illustrate everything:
VII. A Selection Structure for Finite Cases

Problem: Consider the following mapping:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>'S'</td>
<td>1</td>
</tr>
<tr>
<td>'C'</td>
<td>2</td>
</tr>
<tr>
<td>'H'</td>
<td>3</td>
</tr>
<tr>
<td>'D'</td>
<td>4</td>
</tr>
</tbody>
</table>

Write a multiple selection block to set the variable output, assuming input is known.

So, while that worked, it had a little bit of redundancy. Since the input variable takes on only finite values, every condition would have the same kind of test, i.e. input = (something). Most programming languages provide a simpler vehicle for branching on an input variable with a finite number of cases. We'll call this a case structure or a switch structure. First, let's note some caveats:

- Since the cases of the input variable are finite, the input must be of _________________ or _________________ type
- The goal is to map the input to an output. _________________ may execute or apply.
- As with if/else structures, it's nice to have a "when all else fails" case. This is called _________________.

Here's a general form of the structure:

Example: Let's write a case structure that solves the problem above.
Problem: Recall the mapping from the last page:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>'S'</td>
<td>1</td>
</tr>
<tr>
<td>'C'</td>
<td>2</td>
</tr>
<tr>
<td>'H'</td>
<td>3</td>
</tr>
<tr>
<td>'D'</td>
<td>4</td>
</tr>
</tbody>
</table>

Use a case structure to implement the inverse of this mapping. Additionally, map anything that's not in the domain of the inverse to "?"

When your input variable is a character,
- Don't forget to put it in quotes to distinguish it from an identifier.
- It's often nice to allow a user to choose both uppercase and lowercase characters. Thus, a common practice is to convert all inputs to either uppercase or lowercase. This is done via some fun with ASCII, as per a homework problem.

One More Example: Another example will be on the course web site.

X. Choosing Selection Structures and Good Practice

Let's say we have a problem where we know multiple selection is the solution. There are two different ways to solve this problem. How should we decide which to use?

When we're using if blocks, the choice between having else if statements vs. only if statements is a matter of logic. In fact, we could program having only the if statement and no elses whatsoever. What challenge would that add?

When making the decision about using else statements, think about whether things you want to test are related to one another or not, or, bringing in the language of probability, if you have events that are dependent or independent. In dependent cases, taking advantage of else makes for simpler code.

The amount of indentation within each branch should be consistent and should be consistent with the amount of indentation in your main program, i.e. both could be 3 spaces.
Use whitespace in your code to illustrate whether if statements are related. In other words, if you have several branches that are all related (such as multiple selection structure), don't leave any white space between them. But if you happen to have two independent things being tested that just happen to occur sequentially in a program, do leave white space between them as an extra indicator to readers that the two branches have nothing to do with each other. Here's a quick example:

Finally, it is good practice (and a requirement in this course) to comment each branch of every selection structure you write to the right with a brief comment interpreting the branch's purpose in context. Don't repeat the logical statement or translate it to words, but rather interpret and give meaning. For example:

- You might have a branch guarded by temp < 32, and it would be appropriate to document this as freezing case to give meaning and imply that all the computations you'll do in that branch fit the freezing case. (It would be silly to comment this when temp is less than 32, because that says exactly what the code does and doesn't provide meaning.)
- You might have case of 1 for a structure implementing an menu option, and it would be appropriate to comment this as whatever the menu option is, e.g. get new temperature. Similarly, you might comment a case 'Q' as quit. This documentation provides good organization without forcing a reader (or maintainer) to hunt down your menu to find just which case he/she wants to read or debug.

XI. Nesting Selection Structures

Note that the body of any of the branches of any selection structure can be any valid statement or statements, for example...

- An assignment statement
- Several assignment statements
- An input statement
- An output statement

We could also put another selection structure, or even several of them, inside a branch.

Example: Say temp is the outside temperature in Fahrenheit and load is a measure of how much weight in lbs. is currently on some structure. The structure is always safe when temperatures are above freezing, but runs a risk of breaking when temperatures are freezing and the load exceeds 1000 lbs. Use nested selection structures to set a string status to "safe" or "unsafe," appropriately.
XII. Testing Programs with Selection

When we had programs that used only the sequence control structure, testing was less complex. It was wise for us to try out a few different inputs for our variables, but we could be reasonably confident with only a few tests. When selection comes into play, we need to be more careful in our testing. There are two fundamental strategies:

• **Branch Coverage**
  • What: Test at least one value that gets into every branch of the logic
  • Why: Because we can't be certain every branch works unless we test every branch

• **Boundary Testing**
  • When: When we have data that branch based on ranges
  • What: Test values that land at the cutoff points between ranges
  • Why: It's a common error to mix up operators like ">" vs. "\geq". Boundary testing exposes such errors so that we may correct them.

Example: Consider the letter grades example at the top of p. 3 and determine a smallest comprehensive test suite.

Test values for branch coverage:

Test values for boundary testing:

Problem: Consider the problem at the top of p. 6. Determine a smallest comprehensive test suite.

---

XIII. Boolean Variables

There's another fundamental data type beyond the ones we've studied: a **Boolean variable** or **Boolean flag**. The premise is simple: A Boolean variable can hold a value of **true** or **false** and nothing else. A Boolean variable alone can be a Boolean expression. (After all, it evaluates to **true** or **false**.) Why should you care? Well…

• You can set some flag to **true** but switch it to **false** upon detection of a problem. Then you can use that flag to guard off some code later.
• You can store the result of some Boolean expression in a Boolean variable and remember it for later.
• You can use Boolean flags to "remember" whether inputs to a program were valid and thus computation is safe.

We'll look at a case study of the last application in a future class.

For now, let's conclude with a very simple example.

**Example:** Declare a Boolean variable called **isWeekend** and initialize it appropriately, given that you have an initialized character variable **day**, which contains a one-letter abbreviation of the day of the week using PSU's usual abbreviations.
XIV. A Final Word: Why Indentation Matters

Indentation is a wonderful thing that helps us produce code we can understand. Most languages allow us to indent, but not all do. While it wouldn't be considered a programming language, most spreadsheet software allows the use of if statements in formulas, but it's generally not possible to indent there. For example, a common syntax used in spreadsheets is

```plaintext
if(condition, value when condition is true, value when condition is false)
```

One way to achieve multiple selection, then, would be to nest several if statements. Here's an example of code that might compute a letter grade from a numeric grade stored in A1:

```plaintext
```

While tedious, this is easy enough to code, but imagine trying to read it, debug it, or fix a problem with parentheses. (Fortunately, there are better ways to write this in a spreadsheet, and, fortunately, general-purpose programming languages allow us to indent to make even code like the above readable.)