Thinking about Conditional Thinking

Stephen Hughes
AppsLab, John Pappajohn
Entrepreneurial Center
University of Northern Iowa
Cedar Falls, IA 50614-0130
stephen.hughes@uni.edu

J. Philip East
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50614-0507
east@cs.uni.edu

Abstract

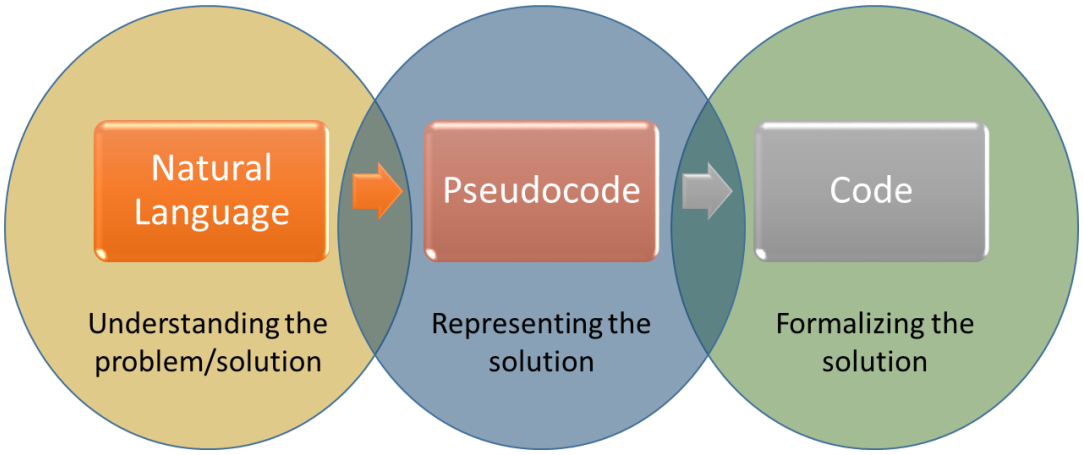
In revising our course on introductory programming we have been updating our instructional approach to consider more deeply student thinking and difficulties. We have found that students generally understand program control structures—conditional execution, repetition, and modularization—at the conceptual level. When asked to use these constructs to solve their own problems, their difficulty often lies with the representation of the problem and the formulation of the conditional expressions necessary to make the decisions. This paper will describe the problem-centric approach that led up to this point and the rationale for our focus on *asking questions of the data*. We will discuss assignments and class activity related to paying explicit attention to conditional expressions.

1. Background

In the fall of 2012, the authors were assigned to teach introductory programming as a service course to the broader university. In planning for this course, we sought to better relate this course to the motivations and goals of its audience. From previous experiences, we understand that students enrolled in this course are less excitable by syntax tricks, subtle optimizations or other nuances of programming languages. Serious students from this population are instead driven by learning to how use programming as a problem solving tool within their native discipline.

We also have observed that many introductory programming courses are taught from the perspective of language capabilities and features. As instructors, we tend to organize the course by working through a progression of data types, I/O, arithmetic operators, if statements, for loops and so on. While this structure is not inherently bad or inappropriate, it tends to emphasize syntax and mechanics – perhaps at the expense of engaging students in higher order problem solving strategies. As reported last year (East & Hughes, 2013), we set out to design our course to emphasize problem-solving over language specifics.

One of the guiding principles that we adopted was to provide an ongoing collection of problems and ask the students to articulate their solutions in English. Once a solution had been described, the students could shift their attention to translating it to the programming language. This sounds like pseudocoding, which is a traditional technique for focusing on algorithm development and is linked to critical thinking skills (Tasneem, 2012) . In many ways it is similar, however, in our initial discussions, we did not prescribe any formal structures or vocabulary to be used in expressing a solution. Instead, we were looking for a natural-language solution; we typically asked, "How would you do this in English?" For example, when dealing with the problem of computing your weekly gross pay, a reasonably acceptable answer may be, "Multiply the hours by the pay rate, unless there is overtime; you get time-an-a-half for overtime hours". This approach ensures that the students at least understand the problem and a general solution before they invest much time in formalizing their solution. This progression, mapped onto student thinking tasks is illustrated in the following figure.



We are not suggesting that this is a new model for computer science instruction. Instead we are interested designing a course that deliberately shifts the focus of student activities. A course that adopts a language-centric approach places more emphasis on the boundary between pseudocode and code. We sought to conduct a class that placed more emphasis on the boundary between natural language and pseudocode.

Through this experience, we have found that students generally understand, recognize and use problem-solving techniques such as sequential processing, conditional execution, repetition and modularization in their natural-language descriptions. However, while these descriptions are (often) good enough for human communication, they often are vague and require additional inferences which cannot be processed in a standard programming language. While translating these statements into the programming language is challenging, a more fundamental problem is identifying the representation that accurately and precisely represents the problem. Missing from the example above is the explicit recognition that overtime pay is due when someone works more than 40 hours. Moreover, to properly model this problem requires a comparison to determine if the hours exceeds that threshold. Having knowledge and command of syntax rules will not be helpful if the student is unable to bridge this gap.

Representation of the problem is an underlying challenge in many stages of introductory programming. I/O-based problems that require a simple arithmetic manipulation mirror many of the basic word problems that students may already be familiar with. A reasonable background in Algebra may mute some students' difficulties in deriving an appropriate representation. In our experience, students are less familiar with representing conditional expressions. While it is part of their natural-language problem-solving repertoire, the more than likely have little experience formalizing these relationships. This paper outlines some of the effort that we have employed in the recent introductory programming courses to bolster students' ability to represent conditional relationships.

2. Conditional Thinking: (Conditional Expressions != "if statements")

Traditional instruction often introduces conditional expressions in conjunction with "if" statements. In fact, an informal survey our bookshelves revealed that an overwhelming majority of textbooks introduce the if-then programming construct even *before* any discussion, let alone a comprehensive discussion, of relational operators or logical operators. We propose that this is a detrimental practice for several reasons.

- It is easy to think of the language feature as the "new" concept that must be taught and focus our emphasis on the mechanics of the control structure. However, students seem to be able to quickly understand the if-then construct; they lack experience with developing sound conditional expressions. Without the ability to formulate meaningful conditional expressions, mastery of the if-then construct is of little value.

- When learning about conditional statements in the context of an "if" statement, students are forced to simultaneously confront the formulation of a conditional statement as well as managing the resulting behaviors. Efforts to formulate the resultant clauses of the if-then construct can be a distraction from the primary task of conceptually identifying the conditional expression.

- Conditional expressions are used in multiple language constructs. By conflating conditional expressions with the branching control structure, students may fail to appreciate the role that conditional expressions fill in other control structures, e.g. loops. They are also quite useful when used to populate Boolean variables.

Based on these observations, we have made a deliberate decision to decouple our presentation of conditional expressions from the if-then construct. Moreover, we spend a considerable amount of instruction focused on formulating conditional expressions from natural-language problem statements.

3. Representing conditional thinking.

We begin our exploration of conditional thinking by considering some the role of questions in problem solving. We recognize that to solve some problems, it is necessary to ask a simple yes/no question, i.e., "Is this person eligible for a scholarship?" In other cases, we need to ask questions and then use the answer to change the outcome of the program, i.e., "Orders $50 or more get free shipping; what is the total bill?" At the heart of solving these kinds of problems is the ability to *ask questions of our data*. Our course design includes several steps to cultivate this skill in our students.

3.1. Identifying variables and writing conditional statements

In an initial activity students are asked to provide English descriptions for a set of questions such as:

- Is this person a senior citizen?
- Did we make a profit?

The phrasing used for these questions is critical. In this early stage, we ensure that the questions all have yes/no answers and do not imply further processing; we can deal with the consequences of the answers at a later time. In a discussion of these questions, students should see that their answers all involve comparisons, which motivates a discussion of relational operators. Additional questions can be added to the discussion to introduce a need for logical operators, e.g.:

- Is this person a teenager?
- Is my grade in the B-range?

Once we have a discussion of natural-language solutions and have established the concept of relational and logical operators, we are ready to consider representation of these in a more structured format. To develop students' ability to represent conditional

expressions we ask them to work through a series of questions using the following framework:

1. Identify the relevant data by name (i.e., what are the variables needed). Are these values provided by user directly or are they derived/calculated?
2. Are there any assumptions made when the expression is developed? Identify any literal values that are relevant to the question.
3. Represent the question using relational and logical operators to combine your variables and literal values.

For example given the question "Is overtime pay called for?" one appropriate response would be:

1. hours: how many hours were worked in the past week – provided by user
2. 40: The number of hours in a standard work week.
3. Hours > 40

This framework allows us to have a rich discussion with the students about the origin of the data that is needed to represent the problem. Under many circumstances, it is possible to have the relevant data directly input by the user. However, in many cases we may wish to ask questions about composite data. A good example of this is the question "Am I Obese?" There are multiple ways that the programmer could choose to model this question; they could directly enter the BMI score or they could ask the user to supply weight and height values and compute the BMI.

Using the framework, students that directly enter the BMI would answer:

1. BMI – provided
2. 30: The cutoff for obesity level I
3. BMI >= 30

Students who derive the BMI from other data might reply:

1. Height (in m) – Provided; Weight (in kg) – Provided; BMI - derived
2. 30: The cutoff for obesity level I
3. BMI >= 30

Still others might wonder if it is possible to combine the operations into a single expression:

1. Height (in m) – Provided; Weight (in kg) – Provided;
2. 30: The cutoff for obesity level I
3. Weight / (Height*Height) >= 30

It is valuable for students to experience multiple representations a problem. This strategy requires students to explicitly consider the representation in terms of both data and relationships among the data. Again, it is worth noting that for this exercise we are strictly practicing the formulation of the question; not the implications of the answers. In the examples above, we did not actually compute overtime pay or recommend diet plan.

The emphasis of this lesson is to formulate conditional expressions, so it is important to stop with the calculation of a Boolean value.

3.2. Developing test cases and common pitfalls

Developing effective test cases is a valuable skill that needs to be cultivated in younger programmers. Likewise, any seasoned programmer knows that there are several well-worn pitfalls that we fall into from time to time. Novice programmers are more prone to these, and less likely to identify them when they occur. They would benefit from direct exposure to these when they are first learning to construct conditional expressions. We address these issues in the second phase of our exploration of conditional thinking. This includes exposing students to an initial battery of examples for them to evaluate, which include the initial natural-language phrase, the conditional statement and a set of conditions to evaluate the statement. These examples span a range of complexities, including statements such as:

Did I earn a passing score?

```
(score >= 60)
```

score	56.4	T/F
score	73.0	T/F
score	59.9	T/F
score	60.2	T/F

Is this student a sophomore?

```
(credits >= 30 AND credits < 60)
```

credits	56.4	T/F
credits	73.0	T/F
credits	59.9	T/F
credits	60.2	T/F

Is it a leap year?

```
(year Mod 4 = 0 And year Mod 100 <> 0 Or year Mod 400 = 0)
```

year	1999	T/F
year	2012	T/F
year	2000	T/F
year	1900	T/F

This activity not only reinforces the connection between natural-language questions and their code/pseudocode representation, it also give students practice with executing the statements with given values to ensure that they get the expected result.

As a follow on to this assignment students are given a set of natural-language statements paired with conditional expressions that do not match; they do not accurately represent the question. Students are asked to provide test values that show the flaw in the representation of the conditional relationship and then create an expression that is appropriate. An example problem could be:

Is the password at least 8 characters?

```
(passwd.length > 8)
```

By carefully designing the examples on this worksheet, students can simultaneously refine their ability to develop meaningful test cases while being exposed to common pitfalls. It is easy to produce erroneous conditional statements by randomly inverting relational operators, however, this kind of mistake is probably infrequent, even among novice programmers. In our experience, it is somewhat common for students to confuse "and" and "or" operators. Likewise, inappropriately substituting < for <= is also a predictable mistake as we transition from natural language to a more structured representation. We observe that this is often the foundation for the well-known "off-by-one" error; perhaps if we train students to recognize the error in the conditional statement, it will be easier to find in the context of the looping construct. Beyond these simple structural mistakes, it is also advantageous to the students to be exposed to more complex logical errors such as a misapplication of DeMorgan's Law or misinterpreting inclusiveness/exclusiveness in a boundary check.

3.3. (Conditional Expression != Your Program's Question)

We understand every program to be born of an informational need or question. As we observed earlier in the paper, the central question of a program may or may not be the same as the one answered by a conditional expression. If you want to want to know how a student scored on a test, you may need to ask if they provided the correct answer to question #3. Likewise, if you want to know how much you owe for your taxes, you may need to ask what your filing status is first. Up until this point, we have worked to teach students how to answer relatively simple yes/no questions, without considering the implications. Introducing this next level of abstraction is a critical leap in student thinking about conditionals. When approaching a broad problem, we have found it useful to ask the students to consider what additional questions they need to ask before they can write a program.

What is my gross pay for this week?

Did you work overtime?

Thing-a-ma-bobs cost $0.45 each; Orders of 50 or more, get a 10% discount. How much is my total bill?

Did you order more than 50?

Which students in the class earned an A on the last exam? Did student X earn an A? Have I looked at all the students in the class? (Is student X the last student on the roster?)

This process is intended to get students to think deliberately about framing their questions; what is this question "really" asking? Introducing this shift motivates the need for if-then statements (and possibly loops). As we observed earlier, we believe that students have a reasonably good intuition about conditional processing and branching, but the additional focus on formulating conditional expressions will ease the transition of this concept from natural language to code.

Additionally, looking at the gap between the "program's question" and the conditional expression exposes students to the concept that answering a question sometimes requires them to gather additional information, or ask "sub" questions. When teaching programming, we often want students to consider problem decomposition – the process of breaking a problem into smaller, more manageable components. However, as professionals, this process has become so intuitive that it is often difficult to communicate this intuition to our students. Conditional execution can offer a great opportunity to engage students in a discussion about how computer programmers really think. Now that students have some background in writing conditional expressions, they are ready to consider how to use these expressions to add generality to their programs.

4. Conclusions

This paper has not suggested any radically new approach to teaching. Many readers may consider that they are already doing much of what we propose; it is a question of degree. Recognizing that many students – particularly in a non-majors setting – struggle with effectively translating conditional thinking into code, we have attempted to reorganize and prioritize our instructional approach.

A program can be made more robust by handling additional cases or exceptions; we can add generality by defining alternate behaviors under different circumstances. This generalizability is one of the truly awe-inspiring aspects of computing. However, we believe that many students will fail to appreciate the impact of this discussion, when they are still wrestling with the formulation of a conditional expression, i.e. how to ask questions of the data or the state of the program. Therefore, we have advocated for a more measured approach to learning how to represent conditional expressions.

Without a formal study on the impact of this approach, we cannot report on its effectiveness beyond saying that it feels like we are doing a better job of "designing" our instruction. Through this process, and the discussions shared by the authors, our course design has benefited from a thoughtful examination of the student perspective and conscious attempts to relate that perspective to programming concepts.

5. References

East, J. P., & Hughes, S. B. (2013). An Experience Report of Our Teaching Visual BASIC using a Problem-Oriented Approach. *Proceedings of the 46th Annual Midwest Instruction and Computing Symposium.* LaCrosse, WI.

Tasneem, S. (2012). Critical Thinking in an Introductory Programming Course. *Journal of Computing Sciences in Colleges*, 81-83.