

Unit D3: Functions, 11/25/03

Exercise 1: Propositional Logic

As discussed in class, recursive function definition is one of the most powerful tools in Computer Science. Anything computable can be defined recursively. Although we spent more time discussing recursive functions on numbers and some other simple objects, they can also be used for more complex objects. In this exercise, we define a few operations for analyzing formulas in propositional logic.

First, consider a function *isTrue* that would return the truth value (either **T** or **F**) of the given wff. For example, *isTrue* would evaluate various wff's as follows (when the propositional variables **p** and **q** are assigned **T** and **F**, respectively):

- $isTrue(\mathbf{p}) = \mathbf{T}$
- $isTrue(\mathbf{q}) = \mathbf{F}$
- $isTrue(\mathbf{p} \vee \mathbf{q}) = \mathbf{T}$
- $isTrue((\neg \mathbf{q} \vee \neg \mathbf{p}) \leftrightarrow (\mathbf{q} \rightarrow \mathbf{p})) = \mathbf{F}$

This function *isTrue* can be defined recursively as follows (with a few blank lines):

$isTrue(f) =$

- **T** if *f* is a propositional variable and its truth assignment is **T**
- **T** if *f* has the form $\neg g$ and $isTrue(g)$ is **F**
- **T** if *f* has the form $g \wedge h$, and if both $isTrue(g)$ and $isTrue(h)$ are **T**
- **T** if *f* has the form $g \vee h$, and if _____(i)
- **T** if *f* has the form $g \rightarrow h$, and if _____(ii)
- **T** if *f* has the form $g \leftrightarrow h$, and if _____(iii)
- **F** otherwise

Note1: The above formatting is slightly different from the ones used on lecture slides. However, you must be able to understand how to read it.

Note2: To clarify the evaluation process, some wff's involve parenthesis, (and). The above recursive function definition omits reference to parentheses for simplicity.

- Fill in the three blank lines labeled (i) through (iii) so that the recursive function definition works as we expect.
- Explain how to use *isTrue* above in order to find out the truth value of $\mathbf{p} \wedge \mathbf{q}$ (assuming the same truth value assignments, i.e., **p** is assigned **T**). Do this *step by step*.
- Explain how to use *isTrue* above in order to find out the truth value of $\neg \mathbf{p}$ (assuming the same truth value assignments, i.e., **p** is assigned **T**). Do this *step by step*.

In the above, we assumed that the input for *isTrue* is well-formed. However, imagine you write a program to implement the above recursive function. The user may enter non-wff's. In order to use only forms that are acceptable for *isTrue*, you will need to check whether the input is a wff. To do this, we can also define another recursive function *isWff* that would return a truth value (either **T** or **F**) based on its well-formedness, given any formula (wff's and non-wff's). For example, *isWff* would evaluate various formulas as follows:

- $isWff(\mathbf{q}) = \mathbf{T}$
- $isWff(\mathbf{p} \wedge \mathbf{q}) = \mathbf{T}$
- $isWff((\neg \mathbf{q} \vee \neg \mathbf{p}) \leftrightarrow (\mathbf{q} \rightarrow \mathbf{p})) = \mathbf{T}$
- $isWff((\neg \mathbf{q} \vee \neg \mathbf{p}) \leftrightarrow (\mathbf{q} \neg \rightarrow \mathbf{p})) = \mathbf{F}$

Note: The return values reflect the *well-formedness* of the formulas, **not** their truth values. That is, a wff may be **T** or **F**. On the other hand, it is impossible to evaluate non-wff's with respect to truth values.

D. Recursively define *isWff*.

Hint: The definition is expected to be simpler than that for *isTrue*.

Exercise 2: String Operations

[optional, but highly recommended]

There are many areas where recursive functions are very useful. Here is one such example. If you attempt and understand this exercise reasonably well (maybe after the instructor's feedback), it is likely that you can do well in many intermediate/advanced Computer Science courses, because the exercise touches upon several important points in the field.

In this exercise, you will recursively define a string operation *replace* (with the type $\Sigma^* \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$) that would replace all the occurrences of the specified substring with another substring. For example, *replace* would perform as follows:

- $replace(\text{inputString}, \text{in}, \text{out}) = \text{outputString}$ (multiple occurrences)
- $replace(\text{inputString}, \text{i}, \text{ee}) = \text{eeinputStreeng}$ (multiple occurrences)
- $replace(\text{inputString}, \text{in}, \text{in}) = \text{inputString}$ (no change, i.e., should not loop)
- $replace(\text{inputString}, \text{out}, \text{in}) = \text{inputString}$ (no change, i.e., not found)

Hint: First, identify the base case (e.g., no replacement possible). For the induction step, replace the first occurrence, and then recursively call the function for the remaining part of the input string. Compare the recursive approach with the iterative approach. The latter typically involves more variables to keep track of the status (e.g., positions), which can complicate coding.

You can assume the following functions as well as arithmetic operations (e.g., addition) discussed in class, and use them in your definition:

attachChar: $\Sigma^* \times \Sigma \rightarrow \Sigma^*$

- *attachChar*(ϵ , **a**) = **a**
- *attachChar*(**abc**, **d**) = **abcd**

'+' (string concatenation): $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

- $\epsilon + \mathbf{a} = \mathbf{a}$
- **abc** + **d** = **abcd**

length: $\Sigma^* \rightarrow \mathbf{N}$

- *length*(ϵ) = 0
- *length*(**a**) = 1
- *length*(**abcd**) = 4

indexOf: $\Sigma^* \times \Sigma^* \rightarrow (\mathbf{N} \cup \{-1\})$

- *indexOf*(**inputString**, **in**) = 0
- *indexOf*(**inputString**, **String**) = 5
- *indexOf*(**inputString**, **none**) = -1

substring: $\Sigma^* \times \mathbf{N} \times \mathbf{N} \rightarrow \Sigma^*$

- *substring*(**inputString**, 0, 5) = **input**
- *substring*(**inputString**, 5, 1) = **s**
- *substring*(**inputString**, 100, 3) = ϵ

Note: Most of the above functions are defined in the Java **String** class as methods (naturally, you can also define these recursively). So, you could implement and test your definition in Java fairly easily. Your code may consist of only several lines.

<End>