Final Assignment (option): ADS for Control Structures

Design an abstract data structure for program control structures.

Caution: This exercise is too difficult to assign a legitimate due date or time allocation, because of its exploratory nature. No one has been able to do the task of control structure abstraction well. Thus, this exercise requires a research mentality. Think about how to do it, what the task requires, the tools you have available, and where the hard and easy parts are. Sketch a partial solution and do the parts that you believe are possible. Almost all time should be spent puzzling about what the exercise means. Do not expect to complete any subparts of this assignment.

Definition: Program control structures are those components of a program which determine the sequence and style of program execution. The *semantics*, or meaning, of a program is defined by its behavior, which is guided by control structures. The major program-level control structures are:

- logic (Boolean primitives such as AND, OR, NOT, EQUALS, IF-THEN-ELSE, CASE)
- assignment (assigning names to return values)
- loops (structures which repeat instructions a specific number of times) including iteration, recursion, and {DO, FOR, WHILE, UNTIL}
- sequencing (calls that are executed sequentially as they are written in the program)
- function invocation (calls to specific functions)
- mapping (applying a function to a collection of items)
- catch and throw (jumps from one part of the program to another, usually in exceptional circumstances)

Level of effort: First try to design and implement an ADS which generalizes one of the above control structures. The above structures are elementary, similar to the elementary data structures of bits, arrays and pointers. The idea is to build higher-level control structures from these primitives. The following discussion may help you to choose a difficulty level.

Logic is expressed by propositional calculus or by Boolean algebra. An ADS for logic would identify possible Boolean structures (the propositions in propositional logic can be functions that return a specific value). Most languages extend the semantics of logic so that any return is viewed as True. But how would you specify a logic structure abstractly?

Assignment looks simple, but has been one of the most difficult concepts to understand. Naming is inherently tricky, and naming to a memory location is trickier still. Scoping rules make the duration of assignment tricky also. Several languages do not use assignment, it is not an essential concept. Good programming isolates all assignments. ASSIGNMENT is still very much in use, but it is beginning to look like a bad idea. It is being replaced by LET and by parameter passing in FUNCTION INVOCATION.

Loops are a dominant tool for repetitive actions. They come in many varieties. It is easy translate between FOR, DO, WHILE and UNTIL. Converting between iteration and recursion is significantly harder, but not tricky. Iteration repeats over a data structure (the loop index); recursion repeats over a function invocation. Newer repetitive techniques use an *iterator* data/control structure which generates items as needed. *Streams* and *mappings* can also drive a repetitive call without introducing looping.

Sequencing is fairly easy if it is not mixed with other control structures. Assignment and goto make sequencing particularly difficult.

Function invocation is captured by the rules of lambda calculus, and is not difficult. Since lambda calculus provides two evaluation regimes, there are significant design decisions about what to evaluate when. This has driven the debate between *eager and lazy evaluation*.

Mapping is easy and is widely under utilized in languages. It is very similar to implicit looping.

Catch and throw are dynamic exits into non-local environments. They are principled and easy to model since they essentially throw away intermediate results. But since they cross algorithmic boundaries, they have very limited use (ie for exception handling and occasionally when a loop must be interrupted). GOTO (jumps out of a program without encapsulation) is a control structure which is antiquated and no longer in use.

Algorithms: When the simple control structures above are combined to make a compound program, that program is an algorithm. Your ADS should be able to handle algorithms. The essential utility of the control structure ADS is to convert algorithms which do the same task into different programming approaches or metaphors. This is called a *meta-protocol*. We all know that we can choose between different ways of implementing a specification, iteration vs recursion for example. Your ADS should be able to isolate the choice of a particular control structure from the meaning of the specification.

Examine several SORT algorithms. How do they differ in control structure? How can the concept of sorting be separated from the implementation of sorting? How can the concept of sorting be separated for the various sorting algorithms?

Examine several tree traversal algorithms. How do they differ? How can they be abstracted into one algorithm which steers the search based on a parameter?

What other compound control structures do you commonly use? How would you design a program which could switch between members of a family of algorithms (e.g. the SORT family) depending on the incoming data structure, expectations about the average data structure being processed, and the available computing resources?

Discussion: This assignment is intended to let you think about a hard problem, to use the ADS template on difficult structures in unique ways. Be sure to follow the ADS guidelines. Be sure that you understand the limitations of your language of choice; often language design will preclude your access to manipulating control structures (this is because it is easier to implement languages when control is fixed by a design choice). For example, C++ uses assignment, sequence (or *compound*), logic, and for-loops as elementary control structures.

Stick to a subset of your programming language which avoids the difficult constructs. (Then learn to program like this all the time, never using the problematic ideas of assignment, goto, and possibly loops and variables.) Assume a **pure language** which does not include very tricky concepts like memory allocation and deallocation, destructive operations on memory, dynamic scoping, global variables, and non-local jumps.