Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 7

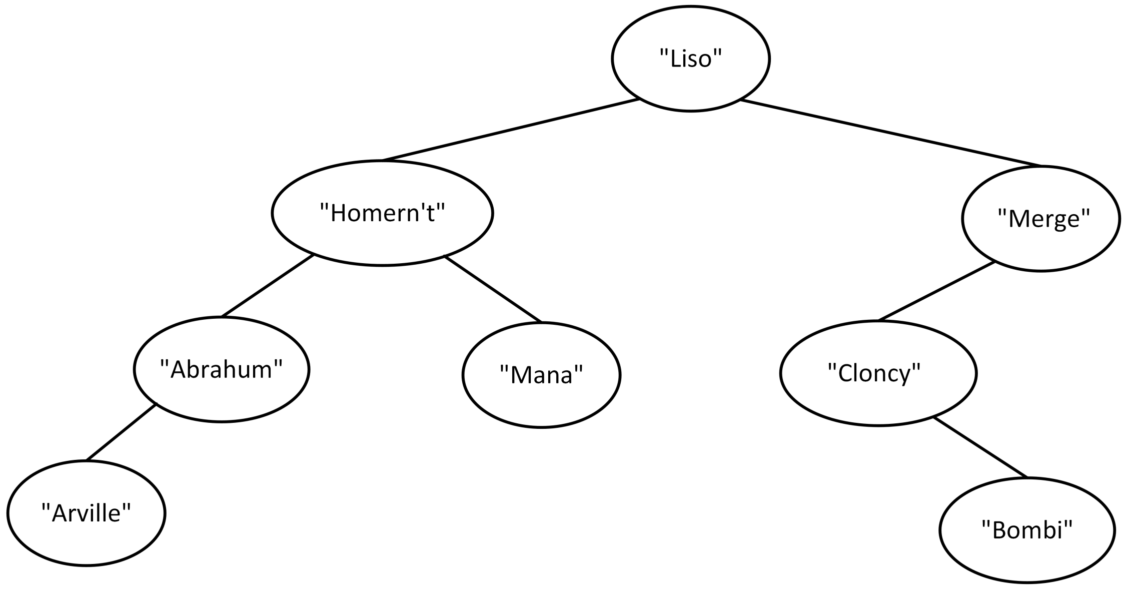
Due:	Tuesday, November 15, 2022 9:00 pm
Coverage:	End of Module 14
Language level:	Intermediate Student
Allowed recursion:	Simple, Accumulative and Mutual Recursion; Generative as guided
Files to submit:	`ancestor-trees.rkt`, `bstd.rkt`, `behaviour-trees.rkt`, `locals-only.rkt`

- Make sure you read the A07 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Your solutions for assignments will be graded on both correctness and on readability. This assignment will be auto-graded and hand-marked.
- Policies from Assignment A06 carry forward.
- Submit examples for all questions by **Friday, November 11 at 8:00AM** (before the due date of assignment A07). See the Example Submissions and Basic Test Results post on Piazza for examples of correct and incorrect example submissions.
- For this assignment, you **cannot** use the built-in Racket function `reverse` or `append` unless otherwise instructed.

Here are the assignment questions that you need to submit.

1. In this question, we will be working with ancestor trees. Each node in an ancestor tree represents a person in a family. The children of a node are the parents of the person stored at that node.

   Consider the following example tree, `copyright-free-ancestors`:



   Since the terms "child", "parent", "grandparent", etc. are overloaded[1] in this problem, we will need to decide how to be more clear about what these terms mean.

   We will continue to use the same terms as in lectures to discuss relationships between tree nodes. To qualify a <relationship> within a family, we will refer to it as a "familial <relationship>". For example, in the tree `copyright-free-ancestors`, "Merge" is the familial child of "Cloncy" and "Mana" is the familial grandparent of "Liso".

   We provide the following data definition for an ancestor tree (`AT`):

```
(define-struct anode (name father mother))
;; An ANode is a (make-anode Str AT AT)

;; An ancestor tree (AT) is one of:
;; * empty
;; * ANode
;;
;; Requires: each name (Str) is unique
```

[1] **overloaded:** when a term is defined multiple times with different meanings

In the problems below, we will refer to the example tree just provided as `copyright-free-ancestors`.

(a) Write the templates `anode-template` and `at-template` for the above data definitions.

(b) To start, write a function `find-subtree` that consumes an `AT` and a `name`, and produces the subtree of `AT` that is rooted at the node labelled `name`. If no node in the `AT` has the provided `name`, you should produce `empty`.

You should make use of a helper function or **`local`** to ensure that your solution is efficient.

(c) Write the function `get-f-generation` that consumes an `AT` and a natural number, and produces the list of family members that have depth equal to the provided number, from left to right. The depth of a node is determined by how many edges need to be followed from the root to get to that node – the depth of the root is 0. If there are no nodes at the provided depth, the function should produce `empty`.

For example, `(get-f-generation copyright-free-ancestors 2)` would produce `(list "Abrahum" "Mana" "Cloncy")`.

You may use `append` for this part.

(d) Write the function `get-f-descendants-path` that consumes an `AT` and a `name`, and produces a list of the familial descendants of `name`, starting with `name` and ending at the root node. If the provided `name` does not appear in the `AT`, produce `empty`.

For example, `(get-f-descendants-path copyright-free-ancestors "Arville")` would produce `(list "Arville" "Abrahum" "Homern't" "Liso")`.

You should make use of a helper function or **`local`** to ensure that your solution is efficient. Remember that you may **not** use `reverse`.

As a reminder, you may not use `reverse` in this assignment.

Submit your solutions in a file named `ancestor-trees.rkt`.

2. In this question we will use an augmented version of a binary search tree that stores a dictionary. Our dictionaries will store natural number keys and string values.

Below, we provide the data definition for a binary search tree dictionary (`BSTD`):

```
(define-struct node (key val left right))
;; A Node is a (make-node Nat String BSTD BSTD)
;; requires: key > every key in left BSTD
;;           key < every key in right BSTD

;; A binary search tree dictionary (BSTD) is one of:
;; * empty
;; * Node
```

We will also be using the following data definition for a sorted association list (SAL):

```
;; A sorted association list (SAL) is one of:
;; * empty
;; * (cons (list Nat Str) SAL)
;;
;; Requires: each key (Nat) is unique
;;
;;           list elements in SAL are sorted by key (Nat)
;;           in increasing order
```

(a) Given a SAL, we can use a simple algorithm to construct a BSTD:

i. Find the median of the keys. If the SAL has an even number of elements, take the smaller of the two middle values. Note that following this rule guarantees that the median will always be a single key value from the SAL.

ii. Use the list of elements before the median value to construct the left subtree.

iii. Use the list of elements after the median value to construct the right subtree.

Write the function build-bstd that consumes an SAL and produces a BSTD, using the algorithm described above. Implementing this algorithm requires generative recursion. Generative recursion is **only** allowed for this problem.

You **may** use reverse in this part, but it is not required. **Hint:** You may find a helper that produces a fixed-length list helpful. An alternate solution may use list-ref.

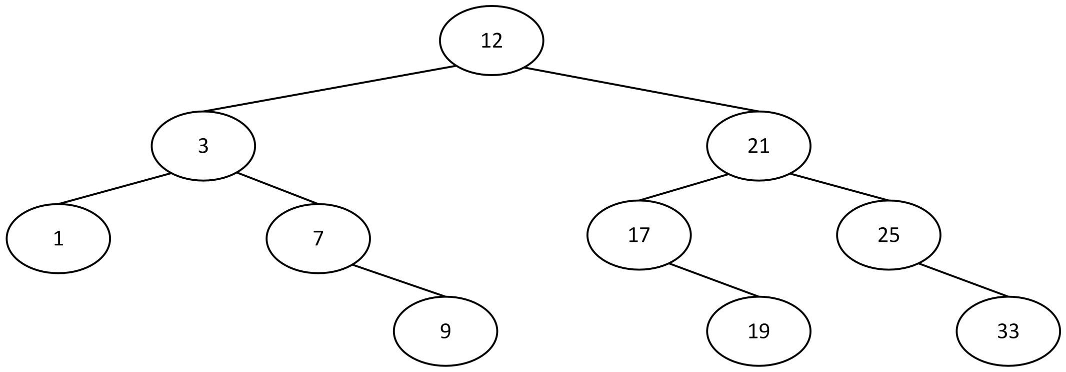
For example, here we have a SAL with some of the top songs of all time, according to Rolling Stone. The key value pairs are the ranking and name of a song, respectively.

```
(define sal-top-songs
    (list (list 1 "Respect")
          (list 3 "A Change is Gonna Come")
          (list 7 "Strawberry Fields Forever")
          (list 9 "Dreams")
          (list 12 "Superstition")
          (list 17 "Bohemian Rhapsody")
          (list 19 "Imagine")
          (list 21 "Strange Fruit")
          (list 25 "Runaway")
          (list 33 "Johnny B. Goode")))
```

Given this sorted association list, `(build-bstd sal-top-songs)` would produce the value of the constant `bstd-top-songs` presented below:

```
(define bstd-top-songs
    (make-node 12 "Superstition"
        (make-node 3 "A Change is Gonna Come"
            (make-node 1 "Respect" empty empty)
            (make-node 7 "Strawberry Fields Forever"
                empty
                (make-node 9 "Dreams" empty empty)))
        (make-node 21 "Strange Fruit"
            (make-node 17 "Bohemian Rhapsody"
                empty
                (make-node 19 "Imagine" empty empty))
            (make-node 25 "Runaway"
                empty
                (make-node 33 "Johnny B. Goode" empty empty)))))
```

We can also draw this tree. We present a diagram depiction of `bstd-top-songs` below, with only the keys displayed inside of the nodes to conserve space.



If it is unclear how the presented `SAL` and `BSTD` are related, you should use the provided algorithm to work through the example with a pencil and paper.

(b) Now create a function `range-query` that consumes a `BSTD` and two natural numbers, the first smaller than the second, and produces a list of all the corresponding values (`Str`) in that range, inclusively, ordered from the value with the smallest key to the largest key.

For example, if we wanted to query the values with keys that fall between 3 and 18 in `bstd-top-songs` – presented in part (a) – `range-query` would produce:

```
(check-expect (range-query bstd-top-songs 3 18)
    '("A Change is Gonna Come" "Strawberry Fields Forever" "Dreams"
       "Superstition" "Bohemian Rhapsody"))
```

You **may** use append in your solution for this part. Try to use the BST ordering property so that you do not traverse the entire tree.

Submit your solutions in a file named `bstd.rkt`.

3. Behavior trees (BT) offer a powerful method for modeling robots and NPCs (non player/-playable characters) in robotics and video games, respectively. In this question, we will simplify behavior trees and consider them to be directed trees with some limited features. Our behavior trees will consist of composite and leaf nodes. Composite nodes can have one or more child nodes that are either other composite nodes or leaf nodes. The leaf nodes cannot have child nodes and in our behaviour trees only depict an action.

   Common composite nodes are sequence and selector nodes. A sequence node delineates a set of actions that must be completed in its entirety and are represented by the actions of the leaf nodes. Consider, for example the simple behaviour tree as illustrated below:

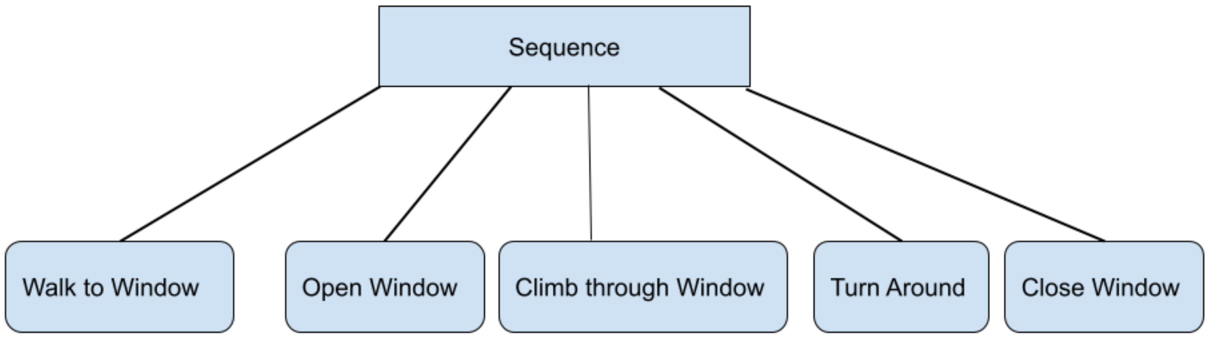


Figure 1: A Simple BT.

   Here, the sequence node is being used to model the following series of actions for an NPC:

   (a) Walk to Window
   (b) Open Window
   (c) Climb through Window
   (d) Turn Around
   (e) Close Window

   The actions must be performed in order from left to right. Each action must be completed before the next action can be initiated. In the simple behaviour tree shown above, the expected actions can be summarised in the string `"(Walk to Window and Open Window and Climb through Window and Turn Around and Close Window)"`. Here the series of actions of the sequence node are simply the action of its leaf nodes, connected with `" and "` (a single space, followed by "and", followed by another single space) and encapsulated within a pair of parentheses.

In contrast to the sequence nodes, selector nodes allow behavior trees to model a selection operation from amongst a series of actions of their child nodes. The selection operation continues to initiate actions from left to right in its child nodes, until the first action that completes successfully. Consider for example, a more elaborate example of a behaviour tree:

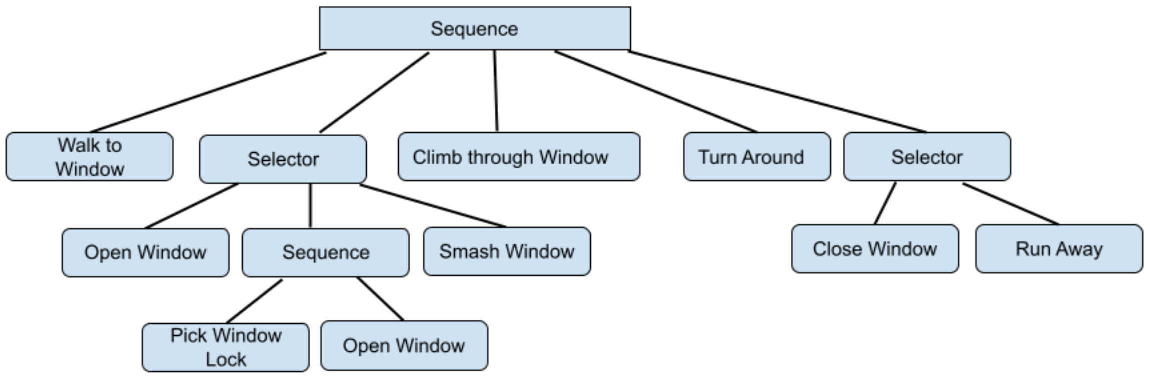


Figure 2: A more elaborate BT

We have provided the BT in Figure 2 in the support file. We have defined it as `npc2-aggressive-window-entry`.

Here, the expected actions can be summarised in the string,

```
"(Walk to Window and (Open Window or (Pick Window Lock and Open Window)
   or Smash Window) and Climb through Window and Turn Around and (Close
   Window or Run Away))"
```

Note, that the series of actions of every composite node is encapsulated within a pair of parentheses, and that the series of actions of a sequence node are connected with an `" and "`, whereas the series of actions of a selector node are connected with an `" or "` (single space followed by "or", followed by another single space).

It should be noted that behavior trees are more powerful than the description provided, however, for the context of this question we will only consider these simplified modelling behaviours.

There can be multiple sequence and selector nodes, so we will identify them with an `id`.

A behaviour tree and its composite and leaf nodes are defined as follows:

```
(define-struct cnode (type id children))
;; a CNode (composite node) is a (make-cnode Sym Nat (ne-listof BT))
;; Requires:
```

```
;;   type is (anyof 'Sequence 'Selector)
;;   id is unique in BT

;; A BT (Behaviour Tree) is one of:
;;   * Str ;; a leaf node (an action)
;;   * CNode
```

The BT of Figure 1 can be represented as:

```
(define npc1-through-window
        (make-cnode 'Sequence 1 (list "Walk to Window"
                                      "Open Window"
                                      "Climb through Window"
                                      "Turn Around"
                                      "Close Window")))
```

(a) Write a function definition action-exists? that consumes a BT and a Str for an action, in that order, and produces a true if the action exists in the BT and false otherwise.

```
(check-expect (action-exists? npc1-through-window "Open Door") false)
(check-expect (action-exists? npc1-through-window "Open Window") true)
```

(b) Write a function definition for summarize-bt that consumes a BT and produces a summarized string of the series of actions in the BT. For example:

```
(check-expect (summarize-bt npc1-through-window)
    "(Walk to Window and Open Window and Climb through Window and \
Turn Around and Close Window)")
```

On the other hand the expected output for the BT in Figure 2 is:

```
(check-expect (summarize-bt npc2-aggressive-window-entry)
    "(Walk to Window and (Open Window or (Pick Window Lock and Open \
Window) or Smash Window) and Climb through Window and Turn \
Around and (Close Window or Run Away))")
```

Note, that you can split a string in Racket using the backward-slash. Recall, that the series of actions of every composite node is encapsulated within a pair of parentheses, and that the series of actions of a sequence node are connected with `" and "` whereas the series of actions of a selector node are connected with `" or "`.

(c) Write the function definition `(add-action a-cnode id action n)` that consumes the following:

- a `CNode` a-cnode,
- a `Nat` id,
- a `BT` action, and
- a `Nat` n

and produces a `CNode` that has the `action` inserted as a leaf node in the n^{th} position in the immediate children of `a-cnode`, with `cnode-id = id`. [Added November 9, 9am]: Note that if there is no node in `a-cnode` with `cnode-id = id`, produce `a-cnode` unchanged.

For example [Note: second example updated November 10, 9am]:

```
(check-expect
     (add-action npc1-through-window 1 "Look through Window" 2)
     (make-cnode 'Sequence 1 (list "Walk to Window"
                                   "Look through Window"
                                   "Open Window"
                                   "Climb through Window"
                                   "Turn Around"
                                   "Close Window")))

 (check-expect
     (add-action
         (make-cnode 'Selector 10 (list "Walk Left" "Walk Right"))
         10
         npc1-through-window
         1)
     (make-cnode 'Selector 10
                 (list (make-cnode 'Sequence 1
                                   (list "Walk to Window"
                                         "Open Window"
                                         "Climb through Window"
                                         "Turn Around"
                                         "Close Window"))
                       "Walk Left"
                       "Walk Right")))
```

As another example, using the BT of Figure 2:

```
(check-expect
    (add-action npc2-aggressive-window-entry 4 "Stop to Catch Breath"
       3)
    (make-cnode 'Sequence 1
          (list "Walk to Window"
                (make-cnode 'Selector 2
                      (list "Open Window"
                            (make-cnode 'Sequence 3
                                   (list "Pick Window Lock"
                                         "Open Window"))
                            "Smash Window"))
                "Climb through Window"
                "Turn Around"
                (make-cnode 'Selector 4
                        (list "Close Window"
                              "Run Away"
                              "Stop to Catch Breath")))))
```

You can hold the following requirements if:

- $1 \le n \le$ `((length (cnode-children bt)) + 1)`.
- If the action is a `CNode` then all the ids in the `action CNode` are distinct and unique from all the ids in `bt`.

(d) Write the function definition `rewind` that lists the actions of all the leaf nodes in a BT in reverse order. For example, for the BT in Figure 1, the expected output will be:

```
(list "Close Window" "Turn Around" "Climb through Window" "Open
   Window" "Walk to Window")
```

For the BT of Figure 2, the expected output will be:

```
(list "Run Away" "Close Window" "Turn Around" "Climb through Window"
      "Smash Window" "Open Window" "Pick Window Lock" "Open Window"
      "Walk to Window")
```

You can use `append`, however, as a reminder, you cannot use `reverse` for this question.

Place all your functions in the file `behaviour-trees.rkt`.

4. In this question, you will practise using **`local`** helper functions.

   (a) In this question you will perform step-by-step evaluations of Racket programs, as you did in assignment one. Please review the instructions on stepping in A01 and complete the five required questions under the "Module 12: Locals" category on the CS135 Stepping Practice website.

(b) Write the function `normalize` that consumes a non-empty list of numbers, `lon`, and produces a normalized list of those numbers. Let the list of numbers be $x_0, x_1, ..., x_n$. Then the normalized list is $y_0, y_1, ..., y_n$, where $y_i = \frac{(x_i - min)}{max - min}$ where *min* and *max* are the minimum and maximum numbers in `lon`, respectively.
Consider the following example:

```
(check-expect (normalize '(2 4 6))
                         '(0 0.5 1))
```

Normalization is an important technique used to scale data in many applications, such as machine learning and statistics. If there is only one element or if the `min` and `max` of the list are the same then that implies all elements in the list are the same. In such a case, the list containing the original elements would be the normalized list.

This function must be completely self-contained using **`local`**. In other words, all constants and helpers must be **`local`**.

Place all your functions for this question in the file `locals-only.rkt`.

This concludes the list of questions for which you need to submit solutions.
Remember to always check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

The material below first explores the implications of the fact that Racket programs can be viewed as Racket data, before reaching back seventy years to work which is at the root of both the Racket language and of computer science itself.

The text introduces structures as a gentle way to talk about aggregated data, but anything that can be done with structures can also be done with lists. Section 14.4 of HtDP introduces a representation of Racket expressions using structures, so that the expression `(+ (* 3 3) (* 4 4))` is represented as

```
(make-add
  (make-mul 3 3)
  (make-mul 4 4))
```

But, as discussed in lecture, we can just represent it as the hierarchical list `'(+ (* 3 3) (* 4 4))`. Racket even provides a built-in function `eval` which will interpret such a list as a Racket expression and evaluate it. Thus a Racket program can construct another Racket program on the fly, and run it. This is a very powerful (and consequently somewhat dangerous) technique.

Sections 14.4 and 17.7 of HtDP give a bit of a hint as to how `eval` might work, but the development is more awkward because nested structures are not as flexible as hierarchical lists. Here we will use the list representation of Racket expressions instead. In lecture, we saw how to implement `eval` for expression trees, which only contain operators such as `+,-,*,/`, and do not use constants.

Continuing along this line of development, we consider the process of substituting a value for a constant in an expression. For instance, we might substitute the value `3` for `x` in the expression `(+ (* x x) (* y y))` and get the expression `(+ (* 3 3) (* y y))`. Write the function `subst` which consumes a symbol (representing a constant), a number (representing its value), and the list representation of a Racket expression. It should produce the resulting expression.

Our next step is to handle function definitions. A function definition can also be represented as a hierarchical list, since it is just a Racket expression. Write the function `interpret-with-one-def` which consumes the list representation of an argument (a Racket expression) and the list representation of a function definition. It evaluates the argument, substitutes the value for the function parameter in the function's body, and then evaluates the resulting expression using recursion. This last step is necessary because the function being interpreted may itself be recursive.

The next step would be to extend what you have done to the case of multiple function definitions and functions with multiple parameters. You can take this as far as you want; if you follow this path beyond what we've suggested, you'll end up writing a complete interpreter for Racket (what you've learned of it so far, that is) in Racket. This is treated at length in Section 4 of the classic textbook "Structure and Interpretation of Computer Programs", which you can read on the Web in its entirety at `http://mitpress.mit.edu/sicp/` . So we'll stop making suggestions in this direction and turn to something completely different, namely one of the greatest ideas of computer science.

Consider the following function definition, which doesn't correspond to any of our design recipes, but is nonetheless syntactically valid:

```
(define (eternity x)
  (eternity x))
```

Think about what happens when we try to evaluate `(eternity 1)` according to the semantics we learned for Racket. The evaluation never terminates. If an evaluation does eventually stop (as is the case for every other evaluation you will see in this course), we say that it *halts*.

The non-halting evaluation above can easily be detected, as there is no base case in the body of the function `eternity`. Sometimes non-halting evaluations are more subtle. We'd like to be able to write a function `halting?`, which consumes the list representation of the definition of a function with one parameter, and something meant to be an argument for that function. It produces `true` if and only if the evaluation of that function with that argument halts. Of course, we want an application of `halting?` itself to always halt, for any arguments it is provided.

This doesn't look easy, but in fact it is provably impossible. Suppose someone provided us with code for halting?. Consider the following function of one argument:

```
(define (diagonal x)
  (cond
    [(halting? x x) (eternity 1)]
    [else               true]))
```

What happens when we evaluate an application of diagonal to a list representation of its own definition? Show that if this evaluation halts, then we can show that halting? does not work correctly for all arguments. Show that if this evaluation does not halt, we can draw the same conclusion. As a result, there is no way to write correct code for halting?.

This is the celebrated *halting problem*, which is often cited as the first function proved (by Alan Turing in 1936) to be mathematically definable but uncomputable. However, while this is the simplest and most influential proof of this type, and a major result in computer science, Turing learned after discovering it that a few months earlier someone else had shown another function to be uncomputable. That someone was Alonzo Church, about whom we'll hear more shortly.