CSC 208 – Programming in C and Python.
Undergraduate in-course project(s) – 17 April 2021

Remit:
We indicate below the remit of the various categories of projects we may use though, for pedagogic convenience, we freely make adjustments. Specifically, the choice of programming language (Python or C) need not be adhered to, nor the design decomposition approaches. In the 2020/2021 academic year, we do NOT consider Python programming for group projects. Instead, simple but relatively elaborate examples are presented in class.

Programming projects:
Convert an algorithm into executable code, typically formulating problems as abstract data types. For each project, the essentials of the main algorithm, and maybe data structures, are given. For problem analysis, deep knowledge of the application domain is not expected, though an understanding beyond CSC 207 might be needed. For this academic year, programming projects are implemented in C, and so better mastery of the C language is an objective.

Program development projects.
Here projects emphasise: analysing and formulating simple specifications; design and conceptual creativity of algorithms and architectures (data structures); and the usability of the finished product. Ironically, mastery of program development techniques is incidental. Capacity to explore/research an issue that may eventually be used (or not) is also expected. Implementation is in Python: one may possibly reimplement in Python algorithms from a programming project or from other programming languages and/or use foreign function interfaces to invoke program components implemented under programming projects. Also, introduction to object-oriented programming concepts, better mastery of Python, and use of scripting and non-imperative programming paradigms are possible objectives.

Programming and program development projects:
Each project consists of two parts: a programming part and a program development part. In the programming part, students produce working code from, demonstrate an ability to develop and implement algorithms as working code, and show minimal understanding of the problem domain, possibly from independent research. Focus is on demonstrating programming skills, and using appropriate tools.

Notation (for algorithms copied from Cormen et al):
Given an object (or variable) x, x.y indicates that y is an attribute (or field of) x. in x.y.z, z is an attribute of y, which is an attribute of x. (If x is a pointer, then x.y indicates that x points to y.)

Use the notation y(x) to stand for (or to extract/get) the value of attribute y from object x. Thus, if y is an attribute that holds the value 42, then y(x) [can be treated as a function that] returns 42. More generally, y(x) denotes a function y over x, that returns a possibly computed attribute of x.

Notation in Cormen et al.	Notation from class
Tab indentation shows nesting. Thus, e.g.: **while** *x* != *y* *x = x.left* **return** *x*	while x != y do x = x.left end return x;

CSC 208 In-course Project:
A Simple Projectional Tree Editor

William S. Shu
Department of Computer Science
University of Buea, P.O. Box 63 Buea, CAMEROON.

17 April 2020

Introduction

In conventional text-based editing, text is held in a buffer where it is edited and the changed text stored in a file when editing is done. The text is presented as line-based and does not have any additional structure over it. In *structured editing* (or *syntax-directed editing* of programs), the text edited is structured—it has a structure over it, such as by the syntax of the programming language that is used. For a programming language (PL), one needs to identify its abstract syntax tree (AST) in order to verify or validate its structure. (In this document, we use the term abstract structure, just in case the entity being edited is not a language, such as a book. Technically though, we still have/require a [formal] language to describe the book, but that is another story!) As such, changes to the text have to be repeatedly parsed, as changed, to ensure the edits are consistent with the syntax or grammar of the language. Such editing is parser-based.

Projectional editors generalise structured or syntax-directed editors and today the terms are used as synonyms. Here, the user applies edit operations (changes) directly to the AST of a program, rather than on their concrete syntax tree. Since they do not require a parser for the edits, assorted notations (tables, diagrams, mathematical formulae, etc.) can be mixed and edited, and various languages can be variously [mathematically] composed for assorted uses. In particular, languages could be aligned to specific domains (giving rise to domain specific languages, DSLs), which is essential to model-driven software development and are usually exploited via language workbenches.

Language workbenches provide high-level mechanisms to implement DSLs. They are "tools that support the efficient definition, reuse and composition of languages and their IDEs" and make it easier to develop new languages with comparatively little effort (Sebastian Erdweg et al, "The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge", 2011). These, in turn, enhance intentional software development where software content is expressed as relevant domains involved and separated out from its implementation, so that domain experts and programmers can collaborate and provide solutions.

In this project, we seek to develop a simple projectional editor based on the tree structure that typically characterises entities edited. We use slightly more general terminology (such as abstract structure for abstract syntax tree) to deliberately remain general. We also involve programming language features and mathematical concepts that can better help the student develop constructs and organise information needed. However, such features and concepts are typically not directly referenced. For instance, aspects of the zipper data structure usual to data structure edits in functional programming is exploited in navigating trees but never explicitly referenced.

Some notions may not have been studied to any significant working depth (e.g., trees and parsing) though most were touched upon in CSC 207. In any case, the lecturer gives additional information as necessary, or explicitly requests the students to find out more. However, such exercises are not intended to teach the students the concepts, but to help them develop skills to acquire concepts, etc.

as needed. The students miss the point if they insist on conventional learning, rather than refining his/her mastery in the mechanics and application of software development techniques and processes.

Conventional Text Editors

Conventional text editors can be abstractly viewed as having a window over a file which reveals the text to be edited. A file is a sequence of bytes (characters) folded (or split) into lines that are visible in the window. The window can be seen as having an *edit buffer* where some text is loaded from a file, manipulated and optionally saved back to file.

<Teacher helps students develop diagrammatic representation>

Figure 1. Abstract view of Text Editor and its Navigation and Edit Operations © wss, 2021.

One navigates through the text by moving the *current cursor position* (called *current point* or just *point*) up, down, left, right to a new position within the visible text area (where the window is). Point is immediately to the left of the current character the cursor is on and corresponds to character/text positions within the file. If point moves beyond the visible text area, then the window over the file is adjusted so that the point (and surrounding text) is visible within it. Commands which move point but do not change the content or presentation of the text or file, such as the up and right commands above, are called *navigation commands*. Other navigation commands move point (and possibly the window) to the top, bottom or some relative position in the file.

Basic Edit Operations

One can edit the text found at the current cursor position. The basic edit operations are *insert*, *delete* and *replace* operations used to insert, delete and replace the character found at point. (Other edit operations can be built over them.) Space is created in the buffer at point in order to insert a character, and adjacent characters are shifted to close the gap created by a deleted character. (How this is actually implemented depends on the data structure used to implement the buffer. In an array implementation, array entries to the right of point are shuffled right to create space or shuffled left to close gaps.) The *undo edit* operation undoes the last *n* edit operation(s) that were applied, and the *redo operation* reapplies the last *n* operations that were undone, for some maximum natural number *n*. The edit operations and changes they made are stored in a list (a stack actually) and their inverse operations applied in undo operations. (This is more efficient than storing each copy of the changed document.) So, for example, when we add text to a file, we record at the start of the list: the change operation, the location where it was applied, and the text actually added.

Edit operations are also applied to non-character objects such as words, lines, paragraphs and other sequences (or collections) of text fragments. Such objects are generically called *blocks*, and are typically identified via markers. *Markers* are identified positions in a text that are possibly named and used to identify the range of characters or text found in a block. So, a *selected text* is the text identified from the marker where the chosen text starts to the [current] point where the text ends. For a rectangular selection of text, these positions are typically the top-left and bottom-right positions (or vice-versa) enclosing the text found in a rectangular box.

File Operations

A file [in the underlying operating system] is *opened* by associating it with the editor program and loading its content (or part thereof) into an edit buffer. The file is *closed* when it is no longer needed by disassociating it from the file in the underlying file system. (c.f., File processing in CSC 207 and/ or file systems as discussed under CSC 205.) Text can be loaded from a file into a buffer, and the possibly modified content saved to the file (or to another file) via appropriate load and save operations. Also, a file content (or another buffer) can be loaded into a given buffer at point.

Presentation Views
Editing is carried out on a file by loading it into the edit buffer for edits, and suitably presented for manipulation by the user through a window(s). Typically, the text is stored in a file structure, loaded into a buffer that has a suitable data structure to manipulate it, and presented (displayed) in a window with an organisation suitable for viewing. These different presentations (views) of the file have to be coordinated to remain consistent as the user edits. In effect, certain types of transformations and operations have to be carried out as the user loads files into buffers, saves buffers to files, presents changes in buffers to windows, and apply edit operations [from windows] to buffers.

Projectional Editors
Inspired by the way conventional text editors work, projectional editors could be described as editing a document, such as a [programming language] program, which has an associated structure that constrains how navigation and edit operations take place. Furthermore, the document may be presented (viewed) in any of a number of possible ways, depending on what the user wants or expects. As such, we need an *abstract structure* (description) that captures the document structure we would navigate—that is, move point, etc., as discussed under conventional text editors—and hence locate the contexts in which to apply edit operations. Given the abstract structure, the document can be projected (viewed) in alternative ways (presentations), as the user may desire.

So, for example, if the document structure is a book, whenever we delete a book section, we must locate the book section within a book chapter, and also delete all the paragraphs within it. If we insert a book chapter, we must locate it within the book and then give it its chapter number (and perhaps chapter heading) as well. In short, implementation of each edit operation must reckon with the nature and context of the component edited. To view the document, we may represent it (and its structure) as a hierarchical tree of book components or a linear sequence of book components.

More abstractly, to develop a projectional editor for an entity such as a program text or book, we have to allow for the following as loosely captured in the diagram below. We identify the concrete structure of the entity—what most people generally perceive as the structure of the entity—and exploit it to manipulate the entity (c.f., concrete syntax in PLs). Typically, this is the form in which we often want to store the entity in our system.

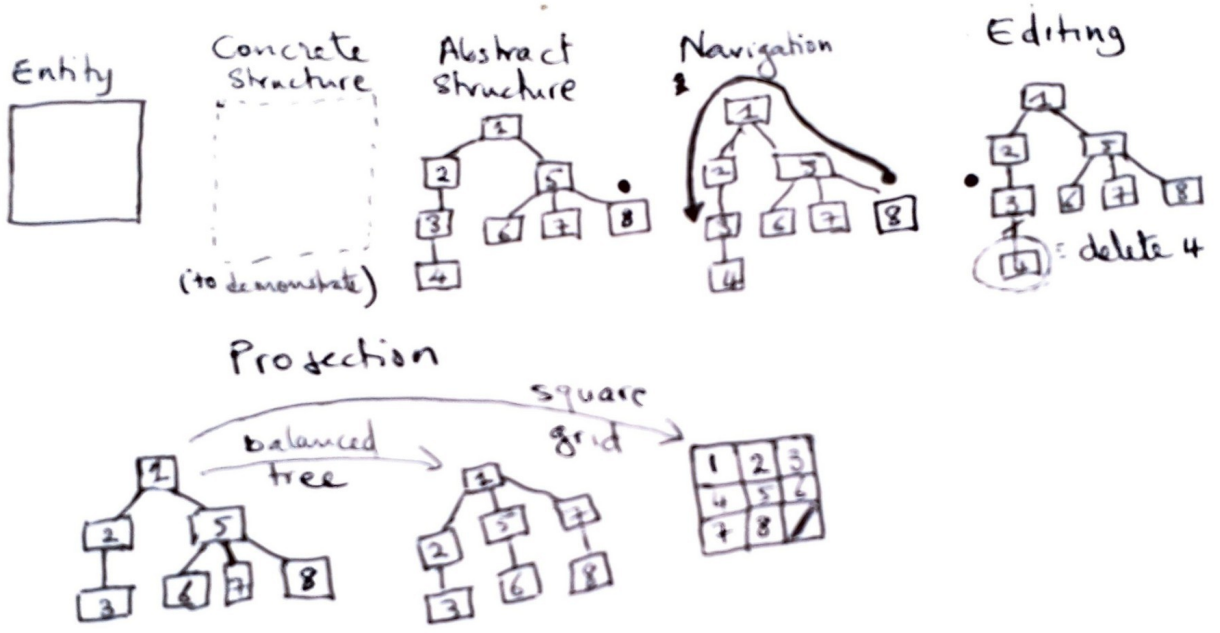


Figure 2. Steps in Projectional Editing © wss, 2021.

From the concrete structure, we identify the entity's *abstract structure* (c.f., abstract syntax in PLs), which mostly captures the logical relationship between components of the structure, and hence key factors to reckon with when editing. From this structure, one defines how best to navigate the entity for edit purposes; that is, to move from one part of the structure to the next. One figures out how edit operations could be applied once one navigates to an *edit focus* (the current position or point where editing takes place), and what information, constraints and contexts to reckon with. (Loosely, *context* here is the surrounding components in which the editing takes place, such as the chapter in which a book section was deleted.) Editing takes place in the abstract structure, but its presentation is seen as a projection into some "space" where the abstract structure is reexpressed. As shown in the diagram, alternative projection views could be balanced trees or square grids. However, more elaborate transformations could also be applied to the entity's components and structure. For example, graphical icons or animated pictures could be used, or commands given to carry out other operations.

Tree Projectional Editor Approach
For our purposes, we can visualise the projectional tree editor as editing a program or writing/editing a book.

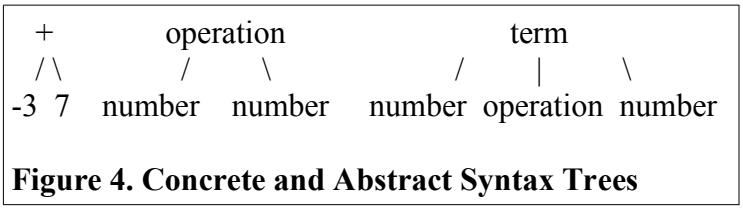


Figure 4. Concrete and Abstract Syntax Trees

Concrete vs Abstract Structure
Our approach is to apply the projectional editor unto a tree structure which represents the entity edited. The abstract structure is essentially the type of components in the entity, and their relative positions to each other in the entities structure. The concrete structure has instances of the data held in components and their relationships.

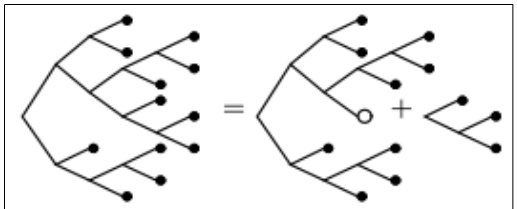


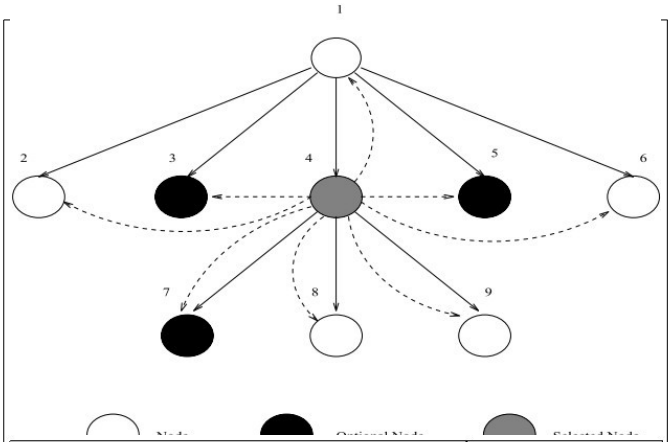
Figure 5. A tree is a context with subtree at an edit focus © McBride.

Edit Contexts and Operations
Editing is done on a node of a tree and potentially affects all the nodes under it. Any such location defines an edit context which partitions the tree into a wider tree context and a subtree rooted at the edit focus. This can be held in a data structure which we can call a *location structure* that has: (a) the subtree whose root is the edit focus, and down which further edits could be done; and (b) a path from the edit focus to the root of the tree. This path can be obtained by keeping a reverse pointer to the root of the tree.

Note: *The location structure is [deliberately] ambiguously defined, to be refined, if need be by the programmer*. It could be seen as a path which defines the successive contexts of the various edit foci considered so far, and hence a path over the latest edit history. The foci could be also seen as defining paths to the root node of the edit tree (possibly via inverse tree pointers). Conceptually however, we can see the location structure as a list of indices (say) which correspond to nodes in path(s) and so permit us to separate (or focus on) navigation requirements from edit ones as need be.

Navigation Commands
Given the tree structure, we need to position and navigate to parents and siblings in a predetermined order as explained earlier. However, some node positions could be optionally filled because the tree component is optional (as in the omitted else-part in if statement). Referring to tree components as in genealogical trees (parent, siblings and children), the edit focus in Figure 6 is node 4 (grey). Node 1 is its parent; nodes 2,3, 5 and 6 are its siblings; and nodes 7,8 and 9 are its children. Filled circles are optional nodes. So in navigating nodes, we can skip positions for optional nodes if they are not there; it is an error for compulsory nodes to be absent, unless the edit process defers updates/checks.

Figure 7 indicates which node each navigation command transitions from node 4 to. For structure editing, tree traversal is mainly by forward preorder and backward preorder strategies. These strategies better define what it means to move forward and backward; and also to move left and right when one is already at the left-most and right-most nodes respectively. The forward preorder traversal strategy is left-to-right and depth-first. That is, the default forward movement (to siblings) is left-to-right. Multiple forward movements to children are depth-first: a child is [somehow] selected, then its child, then the child's child, and so on. Correspondingly, the backward preorder traversal is a right-to-left, depth-first strategy. (Study Figure 7 carefully to make sure you understand how commands affect traversals, with and without optional siblings.)



Command	Leads to node
ascend-to-parent	1
backward-preorder	9
backward-sibling	2
backward-sibling-with-optionals	3
backward-with-optionals	1
forward-preorder	8
forward-sibling	6
forward-sibling-with-optionals	5
forward-with-optionals	7

Figure 7. Navigation commands to neighbouring tree nodes © Suodenjoki, 1994.

Projecting the Views

In projecting views, we essentially want our abstract structure to be expressed and presented in a certain way (formally as a map), for some well-defined perspective (scenario or context). While the options are myriad, we keep things simple by considering 3 main types (classes) of transformations that can be applied to component values. The first is that the component to be presented could be *filtered* in that the range and form of accepted component could be restricted, without changing the meaning of the components so restricted. For instance, one could restrict numbers from the range [1, 100] to the range [5, 10].

Second, components could also be *formatted* (c.f., the printf format statements in C), for a more suitable presentation, possibly using additional information, structures and constraints. Typically, the meaning of the component formatted is not changed and the original component could be recovered from its new format. For example, the integer 1000 could be formatted as 1,000 or 1×10^3. Finally, a component could be *presented* (or *projected*) by mapping it, possibly mathematically, unto some other structure or "space" in which it is described, interpreted, or expressed via alternative structures and components. For example, the text of a program could be re-expressed as an animation on a 2-D screen. Presentations could imply filter and formatting, and can be seen as the most general form of view projections. Therein, one exploits the syntax and semantics of the space(s) projected into.

Consistency among Views

Consistency among views requires that at least the syntactic expression of a component is consistent with the view, and that what the component means (or how it can be interpreted) derives from its abstract structure and the edit operations applied to it. In mathematical terms, a homomorphism(s) preserves intended meanings and possibly the applied edit operations. Such consistency requirements help define invariants that help develop algorithms. *Coherent conditions* may also be expressed; they ensure that the outcome of operations applied do not depend on their application order (e.g., their derivation steps to solve a problem) if such an application order does not matter.

Syntactic constraints, for example, could require that each component type has a unique representation in a projection or is classified uniquely. Semantic constraints may require that

function application be *referentially transparent*. That is, applying the same function multiple times on the same argument value gives the same result every time. Of course, what constraints are applicable depends on what is being edited/represented. The developer looks out for them.

Project Directives
We need to meet project goals. However, in order to stay within the competencies expected of CSC 208 students, and better focus on the pedagogic objectives, we give some project directives under the sections "Simplifying Assumptions", "Essence of Projectional Editors" and "Software Development Hints". Additional ones could be introduced, as need be, in the course of project development.

Simplifying Assumptions
A number of assumptions could be made, at least initially, without significantly compromising pedagogic objectives. Some concepts, such as pointers and binding, probably need to be recalled from CSC 207 (Introduction to algorithms) and additional information on tree data structures (normally taught in CSC 301, Data structures and algorithms) may need to be introduced. However, they are not triggers for students to go learn about such topics, unless they failed to master the basics required of them from CSC 207 and CSC 208.

- If pointer management is problematic, tree nodes could be obtained from a statically allocated array of tree nodes (which could serve as a "node pool"; see below), and the array indices used as if they were node pointers.
- Normally, one obtains an AST from the concrete syntax tree (CST). Unfortunately, obtaining a working version could be rather tedious. Instead one could assume the CST and its AST are the same (equivalent, at least initially) and be manually matched later. Alternatively, one assumes a very simple language and develops conversion procedures between its ASTs and CSTs. Note: *We assume here that we are dealing with PL's, but the principles apply to other abstract-concrete structure pairs*.
- One could assume navigation follows the tree structure, meaning that navigation from node A to node B moves up from node A to a common root node (or the root node of the whole tree), and then down another path to node B.
- If implemented, a "goto operation" to jump to a new, non-neighbouring node—as when one clicks on the node in a tree's graphical representation—requires that one somehow identifies the node and go to it. This could be implemented via one of a number of possibly inefficient techniques. The crudest and most inefficient is to search the whole tree for the node. Another technique is to store in a "node pool" a pointer (or index) to each node used in the edit tree. From the node pool, any node to goto can be accessed. Though not expected in this project, the student should be aware that more elaborate techniques involve the use of *finger trees* (which assume the new node to go to is nearby) and *least common ancestor* (*lca*) *queries* (which find the nearest node that is on the paths to the tree nodes considered).

Essence of Projectional Editors
Some of the essential aspects of a projectional editor include the following:
- Have an ordering for components which is followed for navigation. This way, one knows precisely how to get to a component, and what neighbouring components to adjust when a change alters an entity's structure. In more technical language, we need a partial ordering over [access to] the tree components. We should be able to navigate to any position (edit component) in our entity; abstractly, a traversal operation lets us do that, possibly interleaved with edit operation.
- Have an [*edit*] *context* in which edits take place. The context permits us to effect changes and also capture their possible [side] effects on the rest of the structure. A context must therefore let us: (a) identify the value/object to change; (b) locate the structure in which this

value/object is changed; and (c) and be able to navigate to (from) future (past) edit contexts. In this sense, navigation takes you from one edit context to another along a "history path" that records the edit contexts. Note: *In the discussion so far, edit focus has served the purpose of edit contexts*.

- Have a family of projection maps to/from other representations of the entity (content) to provide projection views. Conceptually, this is a family of operations that could be added or removed, depending on what view/presentation we want of the edit structure. In this sense, the map between the concrete and abstract structures (actually, a map and its inverse) is one such map which is always required.

Given this abstract view of projectional editing, we can develop the editor as operations at this abstract level, generalising to fill in details as needed. In this wise, navigation is just a composition of paths over the pre-determined ordering over tree nodes. Edit contexts are defined by enclosing structures (tree node links) and the edit contexts so far. We can develop projectional editing in terms of just the basic edit operations (change, delete, etc.) and later add more specific details for individual node requirements. Projection maps are individual operations whose requirements and consistency conditions are specified by the target presentations envisaged.

Software Development Hints

Software development should start small, with simplifying assumptions, with a view to generalising later to the original problem. Of course, different emphases will emphasise different aspects of the software development process, and could give different but equivalent results. Possible simplifying assumptions include the following:

- Consider Navigation, Edit and Maps as more or less independent modules (tasks) over the tree structure, so students (or different groups of students) can focus on one of them at a time. The lecturer will therefore have to make sure that main decisions in one module that could impede the functioning of other modules (or the whole system) are explained and avoided.
- Initially work with a homogenous tree of nodes which hold single atomic value. Therefore, a subproblem is to navigate to nodes and then edit the nodes (as linked to the tree) and/or its value.
- Abstractly see (reexpress) a node data type as a tree structure whose root is the outermost data type, with pointers to its nested data types. The data structures related to these data types are then expressed as this tree structure plus the data [instances] they contain. This way, editing the tree structure/node links is handled in the same way as editing node structures and values.
- Proper interfacing between modules calls for communication among teams of developers, if project components are developed by distinct teams.

The End