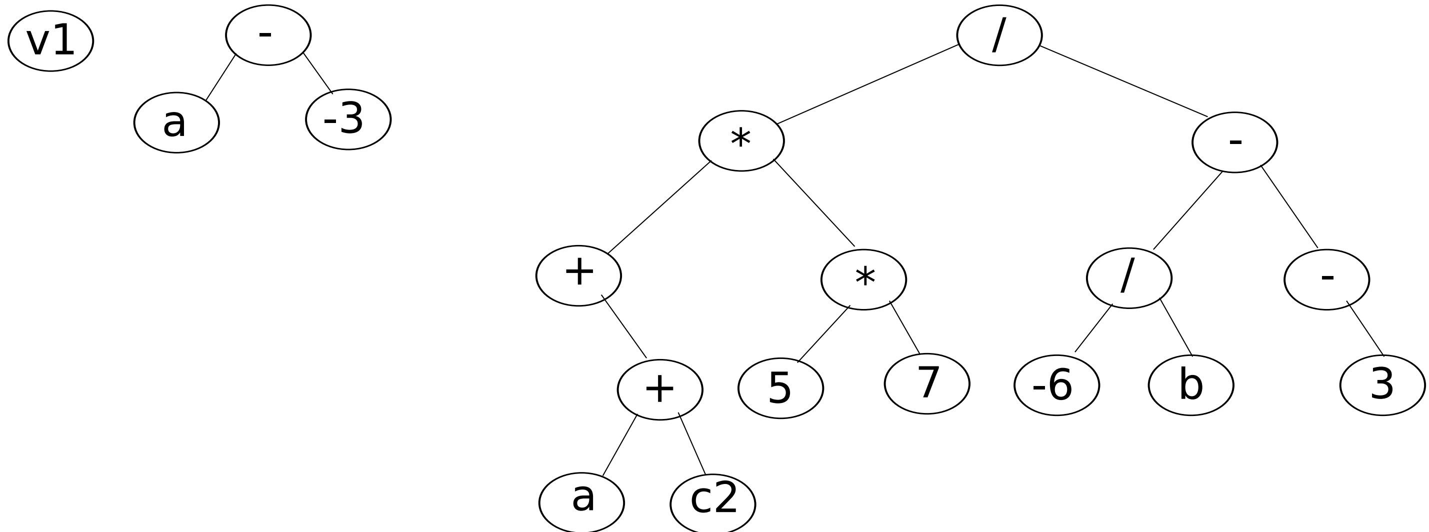
CSC150 2009F
ASSIGNMENT 1

Expression Trees

A tree is a natural way to represent arithmetic expressions. This assignment considers the expressions formed from numbers, variable names, binary (two argument) operations $+$, $\times$, $-$ and $\div$, as well as unary (one argument) versions of $+$ and $-$.

Below are three example expression trees. As is common in programming languages, `*` and `/` denote multiplication and division.



These trees represent the expressions below, shown in *fully-parenthesized* form for ease of parsing and display.

- `v1`
- `(a - -3)`
- `(((+ (a + c2)) * (5 * 7)) / ((-6 / b) - (- 3)))`

These can also be considered as Scheme values built from numbers, symbols, and lists of 2 or 3 elements. They can be entered literally by prefixing with a single quote:

- `'v1`
- `'(a - -3)`
- `'(((+ (a + c2)) * (5 * 7)) / ((-6 / b) - (- 3)))`

Try each of these in the DrScheme REPL to see.

Code to Write

For each datatype or function you are asked to create on the next page, make sure that:

- The name is *exactly* as specified.
- Structure fields are named *exactly* as specified, and occur in *exactly* the same order as (first) mentioned.
- Functions arguments are in *exactly* the order as (first) mentioned.
- Functions work for *at least* the type of arguments specified.
  You may make them work in *more* cases for your own convenience.
  Clearly indicate in a comment before each function the precondition(s) for which it works.
  You may create and name other "helper" functions.
- Functions have side-effects only where specified.

(1) Define a structure datatype `operation`, with three fields `lhs`, `operator` and `rhs`.
An *expr-tree* below is now one of:
- `#f`, meaning "empty" expr-tree
- a number
- a variable name, as a *string*
- an `operation`
  - the `lhs` and `rhs` are expr-trees
  - the `rhs` must *not* be the empty expr-tree
  - the `operator` is one of the *symbols* `+`, `*`, `-`, or `/`.

(2) Create a function and name it `expression->expr-tree`, that takes an expression built from numbers, symbols, and lists of 2 or 3 elements as shown in the examples eariler. It returns the expr-tree that its argument represents. You will find most or all of the following PLT Scheme functions useful: `number?` `symbol?` `symbol->string` `list?` `length` `first` `second` `third`.

(3) Create a function and name it `expr-tree->string`, that takes a non-empty expr-tree and returns a fully-parenthesized *string* representation of the expression. Include exactly one space on each side of each binary operator, one space after each unary operator, and a pair of parentheses around each sub-expression built from a binary or unary operator. You will find most or all of the following PLT Scheme functions useful: `string-append` `symbol->string` `number->string`.

(4) Define a structure datatype `binding` with fields `name` and `value`.
A *substitution* is now a `bst` (as defined in lecture) containing `binding`s, whose `name`s are unique strings (representing variables) and the `value`s are numbers. The `bst` is ordered according to the `name`s, using `string<?`.

(5) Create a function and name it `substitution-bind`, that takes a substitution, a string name and a numeric value.
- If the substitution contains a `binding` with the given name, the `binding`'s value is updated with the given value.
- Otherwise, a new binding is inserted with the given name and value.

The function returns the (altered) substitution.

(6) Create a function and name it `expr-tree-value`, that takes an expr-tree and a substitution containing bindings for (at least) the variables in the expr-tree. It returns the value of the expression, according to the values of the variables in the substitution.

(7) Create a function and name it `expr-tree-simplified`, that takes an expr-tree and returns the result of performing the following simplifications, as *an entirely new expr-tree leaving the original one unchanged.*
- non-empty sub-expressions containing no variables are replaced with their values
- the forms `(x + 0)`, `(0 + x)`, `(x - 0)`, `(x * 1)`, `(1 * x)`, `(x / 1)` are replaced by `x`
- the forms `(x * 0)`, `(0 * x)`, `(0 / x)` are replaced by 0
- the form `(0 - x)` is replaced by `(- x)`
- the forms `(- (- x))`, `(+ x))` are replaced by `x`

Do *only* these simplifications. But be sure that none of the simplifiable forms are left in the result.