Computer Science Study Materials

ACSL Study Materials

Richard Shuai

Computer Science Club *with*
Mr. Schellenberg *as the Lead Teacher*
Walter Murray Collegiate
Saskatoon, Saskatchewan, Canada
2021 - 2022

Collected and Compiled from ACSL Study Materials

Chapter 1: Computer Number Systems

WMC CS Club

Nov 18, 2021

§1 Introduction

All digital computers, from supercomputers to your smartphone, are electronic devices and ultimately can do one thing: detect whether an electrical signal is on or off. That basic information, called a bit (binary digit), has two values: a 1 (or true) when the signal is on, and a 0 (of false) when the signal is off.

Larger values can be stored by a group of bits. For example, there are 4 different values stored by 2 bits (00, 01, 10, and 11), 8 values for 3 bits, and so on. However, large numbers, using 0s and 1s only, are quite unwieldy for humans. For example, a computer would need 19 bits to store the numbers up to 500,000! We use of different number systems to work with large binary strings that represent numbers within computers. In ACSL, we need to know four types of number systems:

- Binary (Base 2, $\overline{abcd}_2$): 0, 1.
- Octal (Base 8, $\overline{abcd}_8$): 0, 1, 2, 3, 4, 5, 6, 7.
- Decimal (Base 10, $\overline{abcd}_{10}$ or $\overline{abcd}$): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Hexadecimal (Base 16, $\overline{abcd}_{16}$): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Consider how we usually represent numbers with decimal number system. If we have a base-10 number $\overline{abcd}$, this number is equal to $10^3a + 10^2b + 10^1c + 10^0d$. This idea works for any bases and any numbers of digits.

§2 Converting with Decimal

The decimal value of a number in an arbitrary base is the sum of each digit multiplied by its place value. Here are some examples of converting from one base into base 10:

Example
Convert 1101_2, 172_8, and $3AD_{16}$ to decimal.

Conversely, the algorithm to convert a number from an arbitrary base requires finding how many times successive powers of the base go into the number, starting with the largest power of the base less than the starting number.

Example
Convert 128, 205 to base 2, 8, and 16, respectively.

§3 Converting between Binary, Octal, and Hex

If you want to convert between number systems other than decimal, you can first convert a number into decimal, then convert that decimal number into the target base. However, there are some shortcuts/ rules. They are fairly straightforward:

1. Converting from octal to binary is simple: replace each octal digit by its corresponding 3 binary bits.
2. Converting from hex to binary is also simple: replace each hex digit by its corresponding 4 binary bits.
3. Converting from binary to either octal or hex is pretty simple as well: group the bits by 3s or 4s (starting at the right), and convert each group.
4. Converting between base 8 and 16 is easy by expressing the number in base 2 (easy to do!) and then converting that number from base 2 (another easy operation)!

Example
Convert 101011_2 to octal and hexadecimal; 54_8 to binary and hexadecimal; $E3_{16}$ to binary and octal.

§4 Practice Problems

1. Convert 156 to binary, octal, and hexadecimal, respectively.
2. Convert $25C_{16}$ to binary, octal, and decimal, respectively.
3. Write 1010111011110101_2 in octal and hexadecimal.
4. Solve for x if $x_{16} = 3676_8$.
5. How many 1s are in the binary representation of 4327_8?
6. Find x if x is a hex that $x = F5AD_{16} - 69EB_{16}$.
7. Evaluate and express the answer in hex: $32_8 + 1011_2 + 352_{10} + AF_{16}$.
8. When convert $AAAAAAAAA_{16}$ to decimal, find the remainder of this number when divided by 5.
9. How many numbers from 100 to 200 in base 10 consist of distinct ascending digits and also have distinct ascending hex digits when converted to base 16?
10. If substring $CADBEF$ is repeated 1000 times in a hex number, find the number of 1s and 0s in its binary representation.

Chapter 2: Pseudocode

WMC CS CLUB

Nov 25, 2021

§1 Introduction to Programs

Frequently, one must use or modify sections of another programmer's code. Since the original author is often unavailable to explain his/her code, and documentation is, unfortunately, not always available or sufficient, it is essential to be able to read and understand an arbitrary program.
The programs are written using a pseudocode that should be readily understandable by all programmers familiar with a high-level programming language, such as Python, Java, or C.

§2 Description of the ACSL Pseudo-code

We will use the following constructs in writing this code for this topic in ACSL:

Operators
! (not) , $\wedge$ or $\uparrow$(exponent), *, / (real division), % (modulus), +, $-$, >, <, >=, <=, !=, ==, && (and), || (or) in that order of precedence.

Functions
`abs(x)` - absolute value, `sqrt(x)` - square root, `int(x)` - greatest integer <= x.

Variables
Start with a letter, only letters and digits.

Sequential statements
```
INPUT variable
variable = expression (assignment)
OUTPUT variabl
```

Decision statements
```
IF boolean expression THEN
Statement(s)
ELSE (optional)
Statement(s)
END IF
```

Indefinite Loop statements
`WHILE Boolean expression Statement(s) END WHILE`

Definite Loop statements

```
FOR variable = start TO end STEP increment
Statement(s)
NEXT
```

Arrays
1 dimensional arrays use a single subscript such as $A(5)$. 2 dimensional arrays use (row, col) order such as $A(2, 3)$. Arrays can start at location 0 for 1 dimensional arrays and location $(0, 0)$ for 2 dimensional arrays. Most ACSL past problems start with either $A(1)$ or $A(1, 1)$. The size of the array will usually be specified in the problem statement.

Strings
Strings can contain 0 or more characters and the indexed position starts with 0 at the first character. An empty string has a length of 0. Errors occur if accessing a character that is in a negative position or equal to the length of the string or larger. The len(A) function finds the length of the string which is the total number of characters. Strings are identified with surrounding double quotes. Use [ ] for identifying the characters in a substring of a given string as follows:
S = "ACSL WDTPD" (S has a length of 10 and D is at location 9)
S[:3] = "ACS" (take the first 3 characters starting on the left)
S[4:] = "DTPD" (take the last 4 characters starting on the right)
S[2:6] = "SL WD" (take the characters starting at location 2 and ending at location 6) S[0] = "A" (position 0 only).
String concatenation is accomplished using the + symbol

§3 Practice Problems

§3.1 Problems

1. After this program is executed, what is the value of B that is printed if the input values are 50 and 10?

```
    input H, R
    B = 0
    if H>48 then
        B = B + (H - 48) * 2 * R
        H = 48
    end if
    if H>40 then
       B = B + (H - 40) * (3/2) * R
       H = 40
    end if
    B = B + H * R
    output B

```

2. After the following program is executed, what is the final value of NUM?

```
A =     BANANAS
NUM = 0: T =
for J = len(A) - 1 to 0 step   1
     T = T + A[j]
next
for J = 0 to len(A) - 1
    if A[J] == T[J] then NUM = NUM + 1
next

```

3. After the following program is executed, what is the final value of C[4]?

```
A(0) = 12: A(1) = 41: A(2) = 52
A(3) = 57: A(4) = 77: A(5) = -100
B(0) = 17: B(1) = 34: B(20 = 81
J = 0: K = 0: N = 0
while A(J) > 0
  while B(K) <= A(J)
    C(N) = B(K)
    N = N + 1
    k = k + 1
  end while
  C(N) = A(J): N = N + 1: J = J + 1
end while
C(N) = B(K)

```

§3.2 Answer Key

1. The final value of B is $2 * 2 * 10 + 8 * 3/2 * 10 + 40 * 10 = 40 + 120 + 400 =$ **560**.
2. The program first stores the reverse of variable A into variable T and then counts the number of letters that are in the same position in both strings. Variable NUM is incremented each time a character at position x of A is the same as the character in position x of string T. There are 5 such positions: 1, 2, 3, 4, and 5.
3. The value of C[4] is **52**.

Chapter 3: Recursive Functions

WMC CS Club

Dec 2, 2021

§1 Definintion of Recursion

A definition that defines an object in terms of itself is said to be recursive. In computer science, recursion refers to a function or subroutine that calls itself, and it is a fundamental paradigm in programming. A recursive program is used for solving problems that can be broken down into sub-problems of the same type, doing so until the problem is easy enough to solve directly.

There are usually two components in a recursive function: the recursion and the base case.

§2 Common Recursive Functions

Fibonacci Numbers
A common recursive function that you've probably encountered is the Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, and so on. That is, you get the next Fibonacci number by adding together the previous two.

Factorials
Consider the factorial function, $n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$, with 0! defined as having a value of 1.

Digit Sum
Try to come up with a recursive function that calculate the digit sum of a number. For example, the digit sum of 2021 is $2 + 0 + 2 + 1 = 5$.

Remark 2.1. Try to write a program using a language you like to solve the above problems.

§3 Recursive Functions in ACSL

Remark 3.1. This ACSL category focuses on mathematical recursive functions rather than programming algorithms. While many mathematical functions can be done iteratively more efficiently than they can be done recursively, many algorithms in computer science must be written recursively.

Example 3.2

Find $f(11)$ if $f(x) = \begin{cases} f(x-3)+1 & \text{if } x > 0 \\ 3x & \text{otherwise} \end{cases}$

Example 3.3

Evaluate $h(13)$ given $h(x) = \begin{cases} h(x-7)+1 & \text{when } x > 5 \\ x & \text{when } 0 \leq x \leq 5 \\ h(x+3) & \text{when } x < 0 \end{cases}$

Example 3.4

Find the value of $f(12,6)$ given $f(x,y) = \begin{cases} f(x-y, y-1)+2 & \text{when } x > y \\ x+y & \text{otherwise} \end{cases}$

§4 Practice Problems

1. Recursive function $L(x)$ gives the x^{th} (1-indexed) term of the sequence 1,3,5,7,9,...
2. Find $g(12)$ given $g(\zeta) = \begin{cases} g(\zeta-2)-3 & \text{if } \zeta \geq 10 \\ g(2\zeta-10)+4 & \text{if } 3 \leq \zeta < 10 \\ \zeta \times \zeta + 5 & \text{if } \zeta < 3 \end{cases}$
3. Find $f(12,7)$ if $f(x,y) = \begin{cases} f(x-1, y+2)+3 & \text{if } x > y \\ 2f(x+1, y-1)-5 & \text{if } x < y \\ x^2+y & \text{if } x = y \end{cases}$
4. Determine $f(15,12)$ given $f(x,y) = \begin{cases} f(y-1, x-2)+4 & \text{if } x > 10 \\ f(x+3, y-3)+2 & \text{if } 5 \leq x \leq 10 \\ 3x-2y & \text{if } x \leq 4 \end{cases}$
5. $f(x)$ takes any positive integers and $f(x) = \begin{cases} 1 & \text{if } x = 1 \\ f(\frac{x}{8}) & \text{if } x \ \% \ 8 = 0 \\ f(x-1)+1 & \text{otherwise} \end{cases}$. Find the maximum value of $f(x)$ for $x \leq 100$.
6. A recursive function $E(x)$ takes a positive integer number, if x is divisible by 11, $E(x) = 0$. Otherwise, $E(x) \neq 0$. Try to come up with at least 2 recursive functions that can do this (Hint: direct recursion and alternating digit sum).
7. $k(x)$ computes the super digit sum of a positive integer. A super digit sum repeatedly calculates the digit sum of the number until it becomes a single digit number. For example, $k(1999) = k(1+9+9+9) = k(28) = k(2+8) = k(10) = k(1+0) = k(1) = 1$.
8. Find a recursive function $f(m,n)$ such that $f(m,n)$ always equals to $\frac{(m+n)!}{m!n!}$ for all positive integers m and n.

Chapter 4: Prefix, Infix, Postfix Notations

WMC CS Club

Dec 16, 2021

§1 The Infix Notation

The way we usually compute math expressions, which is called infix notation, seems easy to us, but it's not the way how computers compute. Instead of infix notation, computers use prefix and postfix notations. Before introducing them, let's review the infix notation.

The expression $5 + \frac{8}{(3-1)}$ clearly has a value of 9. It is written in infix notation as $5 + 8/(3 - 1)$. The value of an infix version is well-defined because there is a well-established order of precedence in mathematics: We first evaluate the parentheses $(3 - 1 = 2)$; then, because division has higher precedence that subtraction, we next do $8/2 = 4$. And finally, $5 + 4 = 9$.

Remark 1.1. The order of precedence is often given the mnemonic of Please excuse my dear Aunt Sue, or **PEMDAS: parentheses, exponentiation, multiplication/division, and addition/subtraction**. Multiplication and division have the same level of precedence; addition and subtraction also have the same level of precedence. Terms with equals precedence are evaluated from left-to-right.

§2 Think Pre/Post-fixedly

What are pre/post-fix notations? Simply put, in prefix notation, each operator is placed before its operands; in postfix notation, each operator is placed after its operand. The prefix and postfix notations of $5 + \frac{8}{3-1}$ are "$+\ 5\ /\ 8\ -\ 3\ 1$" and "$5\ 8\ 3\ 1\ -\ /\ +$", respectively.

§3 Prefix to Infix to Postfix

Here are the algorithms for converting from prefix(postfix) to infix and from infix to prefix(postfix).

From Prefix or Postfix to Infix

One way to convert from prefix (postfix) to infix is to make repeated scans through the expression. Each scan, find an operator with two adjacent operators and replace it with a parenthesized infix expression. This is not the most efficient algorithm, but works well for a human.

From Infix to Prefix or Postfix

1. Fully parenthesize the infix expression. It should now consist solely of "terms": a binary operator sandwiched between two operands.
2. Write down the operands in the same order that they appear in the infix expression.
3. Look at each term in the infix expression in the order that one would evaluate them, i.e., inner-most parenthesis to outer-most and left to right among terms of the same depth.
4. For each term, write down the operand before (after) the operators.

Example 3.1

Convert the prefix expression $\uparrow\ +\ *\ 3\ 4\ /\ 8\ 2 - 7\ 5$ to infix expression and evaluate.

Example 3.2

Convert the postfix expression $3\ 4\ *\ 8\ 2\ /\ +\ 7\ 5\ -\ \uparrow$ to infix expression and evaluate.

Example 3.3

Convert the infix notation expression to prefix and postfix notations: $(X = (((A * B) - (C/D)) \uparrow E))$.

§4 Practice Problems

1. Evaluate prefix expression $*\ -\ 5\ 3\ +\ 4\ 9$.
2. Evaluate postfix expression $7\ 3\ 2\ \uparrow\ 4\ 5\ +\ /\ *$.

3. Convert $6 - \frac{2^3}{5-1}$ to prefix and postfix expression.

4. Convert $A^B - C * D/(E + F * G)$ to prefix and postfix expression.

5. An intuitive assumption might be that the prefix notation is the reversed string of the postfix notation of the same expression. Consider why this assumption might be incorrect.

Chapter 5: Bit-String Flicking

WMC CS Club

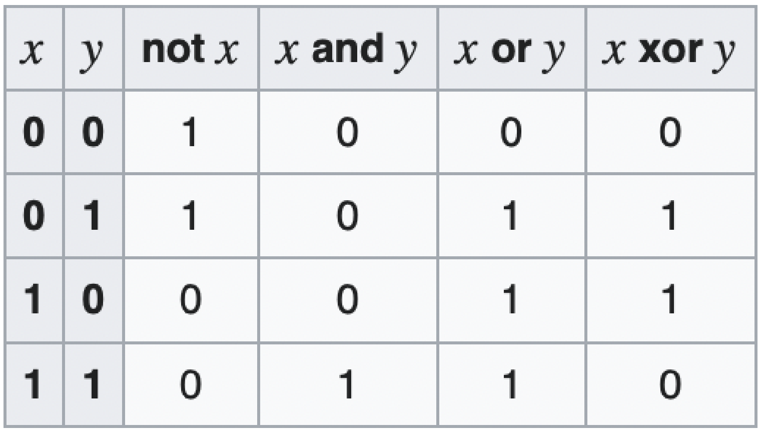
Jan 6, 2022

Bit manipulation is all about binary operations. Recall that there are only two states in the binary world: 0 or 1, on or off, rain or not rain, . . .

§1 Bitwise Operators

The logical operators are NOT ($\sim$ or $\neg$), AND (&), OR ($|$), and XOR ($\oplus$).

- NOT is a unary operator that performs logical negation on each bit. Bits that are 0 become 1, and those that are 1 become 0. For example: **∼101110** has a value of **010001**.
- AND is a binary operator that performs the logical AND of each bit in each of its operands. The AND of two values is 1 only if both values are 1. For example, **1011011 and 011001** has a value of **001001**. The AND function is often used to isolate the value of a bit in a bit-string or to clear the value of a bit in a bit-string.
- OR is a binary operator that performs the logical OR of each bit in each of its operands. The OR of two values is 1 only if one or both values are 1. For example, **1011011 or 0011001** has a value of **1011011**. The OR function is often use to force the value of a bit in a bit-string to be 1, if it isn't already.
- XOR is a binary operator that performs the logical XOR of each bit in each of its operands. The XOR of two values is 1 if the values are different and 0 if they are the same. For example, **1011011 xor 011001 = 110010**. The XOR function is often used to change the value of a particular bit.



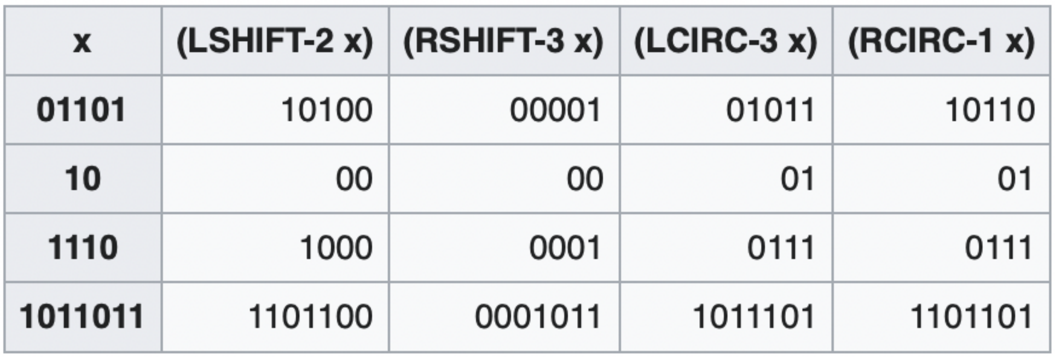
x	y	**not** x	x **and** y	x **or** y	x **xor** y
0	**0**	1	0	0	0
0	**1**	1	0	1	1
1	**0**	0	0	1	1
1	**1**	0	1	1	0

§2 Shift Operators

Logical shifts (LSHIFT-x and RSHIFT-x) "ripple" the bit-string x positions in the indicated direction, either to the left or to the right. Bits shifted out are lost; zeros are shifted in at the other end.

Circulates (RCIRC-x and LCIRC-x) "ripple" the bit string x positions in the indicated direction. As each bit is shifted out one end, it is shifted in at the other end. The effect of this is that the bits remain in the same order on the other side of the string.

The size of a bit-string does not change with shifts, or circulates. If any bit strings are initially of different lengths, all shorter ones are padded with zeros in the left bits until all strings are of the same length.



x	(LSHIFT-2 x)	(RSHIFT-3 x)	(LCIRC-3 x)	(RCIRC-1 x)
01101	10100	00001	01011	10110
10	00	00	01	01
1110	1000	0001	0111	0111
1011011	1101100	0001011	1011101	1101101

Remark 2.1. The shortcuts:
LSHIFT-n: Erase the first n characters, add padding to the back.
RSHIFT-n: Erase the last n characters, add padding to the front.
LCIRC-n: Move n characters from the front to the back.
RCIRC-n: Move n characters from the back to the front.

Remark 2.2. The order of precedence (from highest to lowest) is: NOT; SHIFT and CIRC; AND; XOR; and finally, OR. In other words, all unary operators are performed on a single operator first. Operators with equal precedence are evaluated left to right; all unary operators bind from right to left.

§3 Examples and Practice Problems

Example 3.1
Evaluate: (NOT(000101 AND 101001 OR NOT 000101)).

Example 3.2
Evaluate: (101110 AND NOT 110110 OR (LSHIFT-3 101010)).

Example 3.3
Solve for X (5-bit string): (LCIRC-2 X) OR (RSHIFT-2 01010) = (NOT 00000) AND X.

Practice Problems

1. Evaluate: (010111 XOR NOT 101010).
2. Evaluate: (LCIRC-2 01101) OR (RSHIFT-1 11111)
3. Evaluate the expression: (RSHIFT-1 (LCIRC-4 (RCIRC-2 01101))).
4. Evaluate the following: (RCIRC-2 (LSHIFT-1 (LCIRC-1 (RSHIFT-2 (NOT 10100))))).
5. Evaluate: (R-CIRC-2003((LCIRC-2002(LCIRC-2001(NOT NOT (NOT NOT 10011)))))).
6. List all possible values of x (5 bits long) that solve the equation: (LSHIFT-1 (10110 XOR (RCIRC-3 x) AND 11011)) = 01100.

Answer Key

1. 000010
2. 11111
3. 01010
4. 00010
5. 10011
6. 00000, 00001, 00100 and 00101.

Chapter 6: LISP

WMC CS Club

Jan 13, 2022

LISP (LISt Processing language) is one of the simplest computer languages in terms of syntax and semantics, and also one of the most powerful. LISP presents a very different way to think about programming from the "algorithmic" languages, such as Python, C++, and Java. LISP is a prefix language.

§1 Basic Syntax

As its name implies, the basis of LISP is a list. One constructs a list by enumerating elements inside a pair of parentheses. For example, here is a list with four elements:

```
(10 (wmc cs club) hello 123)
```

The second element is a list within the list. The individual elements are called "atoms." In the list above, `10, `wmc, `cs, `club, `hello, 123` are atoms. Everything in LISP is either an atom or a list (but not both). The only exception is "NIL," which is both an atom and a list. It can also be written as "()" – a pair of parentheses with nothing inside.

All statements in LISP are function calls with the following syntax: `(function arg1 arg2 ... argn)`. To evaluate a LISP statement, each of the arguments (possibly functions themselves) are evaluated, and then the function is invoked with the arguments. For example, `(MULT (ADD 2 3) (ADD 1 4 2))` has a value of 35, since `(ADD 2 3)` has a value of 5, `(ADD 1 4 2)` has a value of 7, and `(MULT 5 7)` has a value of 35. Some functions have an arbitrary number of arguments; others require a fixed number. All statements return a value, which is either an atom or a list.

§2 12 Common LISP Functions

Function	Description	Statement	Result
`SET`	Assign a value to a variable (the first argument must have a quote (since it's a variable), the second argument can be a value (without quote), atom (with quote), or a list (with quote).	`(SET `test 6);` `(SET `test `(a b c))`	6; (a b c)
`SETQ`	Same as `SET`, but it causes LISP to act as if the first argument was quoted.	`(SETQ EX (MULT 2 5))`	10

`CAR`	Returns the **first** item of the list x (and x must be a list or an error will occur).	`` (CAR `(This is a list)) ``	This
`CDR`	Returns the list **without** its first element.	`` (CDR `(This is a list)) ``	(is a list)
`CONS`	The function `CONS` takes two arguments, of which the second must be a list. It returns a list which is composed by placing the first argument as the first element in the second argument's list.	`` (CONS `red `(pink blue)) ``	(red pink blue)
`REVERSE`	The function `REVERSE` returns a list which is its arguments in reverse order.	`` (REVERSE `(a b c)) ``	(c b a)

`ADD`	Sum of all arguments (2 to n arguments.)	`(ADD 1 2 3)`	6
`SUB`	$a - b$ (2 arguments)	`(SUB 9 2)`	7
`MULT`	Product of all arguments (2 to n arguments.)	`(MULT 1 2 4)`	8
`DIV`	$\frac{a}{b}$ (2 arguments)	`(DIV 8 4)`	2
`SQUARE`	a^2 (1 arguments)	`(SQUARE 5)`	25
`EXP`	a^b (2 arguments)	`(EXP 2 3)`	8

The above functions will be the possible ones in ACSL Intermediate Division. Certainly, there are more functions in LISP: `ATOM` (to check if a variable is an atom), `EVAL` (evaluate a variable), `EQ` (to check if a and b are equal), `POS` (to check if a number is positive), `NEG` (to check if a number is negative).

Example 2.1

Evaluate the following expression:

```
(MULT (ADD 6 5 0) (MULT 5 1 2 2) (DIV 9 (SUB 2 5)))
```

Answer: -440

Example 2.2

Consider the following program fragment:

```
(SETQ X `(BC AB SK MB ON))
 (CAR (CDR (REVERSE X)))
```

What is the value of the `CAR` expression?

Answer: MB

§3 User-defined Functions

LISP also allows us to create our own functions using the `DEF` function. For example:

```
(DEF SECOND (args) (CAR (CDR args)))
```

defines a new function called SECOND which operates on a single parameter named `args`. `SECOND` will take the `CDR` of the parameter and then the `CAR` of that result. So, for example: ``(SECOND `(a b c d e))`` would first `CDR` the list to give `(b c d e)`, and then `CAR` that value returning the single character `b`.

§4 Practice Problems

Problems

1. Evaluate: `(ADD (SUB 8 7) 6 5)`.
2. Evaluate: `(ADD (SUB 4 1) (EXP 2 4) (MULT 3 5) (MULT (EXP 3 2) (SUB 2 4)))`.
3. Consider the following program segment:

   ```
   (SETQ score (MULT 5 4 3 2 1))
   ```

   What is the value of the expression `(SUB AMOUNT (DIV 100 5))`?
4. Evaluate the expression ``(CDR `((2 (3)) (4 (5 6) 7)))``.
5. Evaluate: ``(CAR (CDR (REVERSE `(ON (AB QC) (NS (NB MB)) (BC PE SK) NL))))``
6. Write a user-defined function that calculate the length of the hypotenuse c given two legs a and b in a right triangle. Recall that the hypotenuse $c = \sqrt{a^2 + b^2}$. Specifically, function `CALC` has two arguments, `a` and `b`, with `c` returned (assume the arguments passed are legal.)
7. Write a user-defined function `FUNC` with one argument that returns the fourth last element in a list (assume the argument is a list with at least four elements.)

Answer Key

1. 12
2. 16
3. 100
4. ((4 (5 6) 7))
5. (BC PE SK)
6. `(DEF CALC (a,b) (EXP (ADD (SQUARE a) (SQUARE b)) 0.5))`
7. `(DEF FUNC (args) (CAR (CDR (CDR (CDR (REVERSE args))))))`

Chapter 7: Boolean Algebra

WMC CS Club

March 3, 2022

§1 Revisit Boolean Operators

Operators used in Boolean algebra is the same as the ones we mentioned in bits manipulation. These operators are NOT ($\overline{x}$), AND (xy), OR ($x + y$), XOR ($x \oplus y$), XNOR ($x \odot y$).

- NOT is a unary operator that performs logical negation. The NOT of a value is its opposite; that is, the not of a true value is false whereas the not of a false value is true. For example, if $x = 0$, $\overline{x} = 1$.
- AND is a binary operator. The AND of two values is true only whenever both values are true. For example, if $x = 0$, $y = 1$, $xy = 0$.
- OR is a binary operator. The OR of two values is true whenever either or both values are true. For example, if $x = 1$, $y = 1$, $x + y = 1$.
- XOR is a binary operator. The XOR of two values is true whenever the values are different. For example, if $x = 0$, $y = 1$, $x \oplus y = 1$.
- XNOR is a binary operator. The XNOR of two values is true whenever the values are the same. It is the NOT of the XOR function. For example, if $x = 0$, $y = 1$, $x \oplus y = 0$.

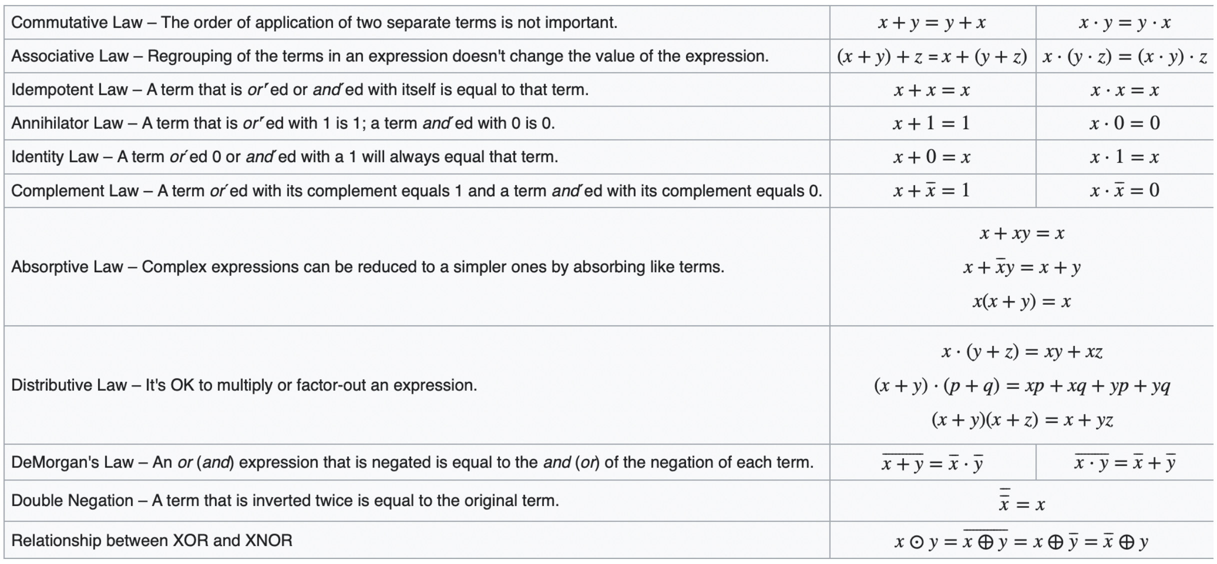
§2 Laws of Boolean Algebra

A law of Boolean algebra is an identity such as $x + (y + z) = (x + y) + z between$ two Boolean terms, where a Boolean term is defined as an expression built up from variables, the constants 0 and 1, and operations and, or, not, xor, and xnor.

Like ordinary algebra, parentheses are used to group terms. When a not is represented with an overhead horizontal line, there is an implicit grouping of the terms under the line. That is, $x \cdot \overline{y + z}$ is evaluated as if it were written $x \cdot \overline{(y + z)}$.

Remark 2.1 (Order of Precedence). The order of operator precedence is not; then and; then xor and xnor; and finally or. Operators with the same level of precedence are evaluated from left-to-right.

Fundamental Identities



Commutative Law – The order of application of two separate terms is not important.	$x + y = y + x$	$x \cdot y = y \cdot x$
Associative Law – Regrouping of the terms in an expression doesn't change the value of the expression.	$(x + y) + z = x + (y + z)$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Idempotent Law – A term that is *or*'ed or *and*'ed with itself is equal to that term.	$x + x = x$	$x \cdot x = x$
Annihilator Law – A term that is *or*'ed with 1 is 1; a term *and*'ed with 0 is 0.	$x + 1 = 1$	$x \cdot 0 = 0$
Identity Law – A term *or*'ed 0 or *and*'ed with a 1 will always equal that term.	$x + 0 = x$	$x \cdot 1 = x$
Complement Law – A term *or*'ed with its complement equals 1 and a term *and*'ed with its complement equals 0.	$x + \overline{x} = 1$	$x \cdot \overline{x} = 0$
Absorptive Law – Complex expressions can be reduced to a simpler ones by absorbing like terms.	$x + xy = x$ $x + \overline{x}y = x + y$ $x(x + y) = x$	
Distributive Law – It's OK to multiply or factor-out an expression.	$x \cdot (y + z) = xy + xz$ $(x + y) \cdot (p + q) = xp + xq + yp + yq$ $(x + y)(x + z) = x + yz$	
DeMorgan's Law – An *or* (*and*) expression that is negated is equal to the *and* (*or*) of the negation of each term.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	$\overline{x \cdot y} = \overline{x} + \overline{y}$
Double Negation – A term that is inverted twice is equal to the original term.	$\overline{\overline{x}} = x$	
Relationship between XOR and XNOR	$x \odot y = \overline{x \oplus y} = x \oplus \overline{y} = \overline{x} \oplus y$	

Example 2.2

Simplify: $\overline{A(A+B)+A}$.

Example 2.3

How many ordered pairs (A, B) satisfy $\overline{\overline{(A+B)} + \overline{A}B} = 1$?

§3 Practice Problems

Problems

1. Simplify $AB + A\overline{B}$
2. Simplify the following Boolean expression: $(A+B)(A+C)$.
3. Simplify $(A+C)(AD + A\overline{D}) + AC + C$.
4. Find all ordered pairs (A, B) that make $\overline{AB}(\overline{A}+B)(\overline{B}+B)$ true.
5. Find all ordered pairs (A, B) that make $\overline{\overline{A(A+B)} + B\overline{A}}$ false.
6. Find the equivalent of $A \oplus B$ using only AND, OR, and NOT. Then, use the fundamental identities to find the equivalent of $A \odot B$.

Answer Key

1. A
2. $A + BC$
3. $A + C$
4. $(0,0), (0,1)$
5. $(0,0), (0,1)$
6. $A \oplus B = A\overline{B} + \overline{A}B$, $A \odot B = \overline{AB} + AB$

Chapter 8: Data Structures

WMC CS Club

March 17, 2022

For the ACSL Intermediate Division, we focus on five data structures: stacks, queues, trees (particularly BST), and priority queues/heaps.

§1 Stacks and Queues

A stack is like placing a plate on a pile of plates, which is a last-in-first-out (LIFO) structure.

A queue is like waiting in a line, which is a first-in-first-out (FIFO) structure.

Both the stacks and the queues support two operations: `POP()` (remove an element; if no element left, returns `NIL`) and `PUSH(param)` (insert an element). Here is an example to show the difference between a stack and a queue.

Example 1.1

```
PUSH("A")
PUSH("M")
PUSH("E")
X = POP()
PUSH("R")
X = POP()
PUSH("I")
X = POP()
X = POP()
X = POP()
X = POP()
PUSH("C")
PUSH("A")
PUSH("N")
```

Solution 1.1 If these operations are applied to a stack, then the values of the pops are: E, R, I, M, A and `NIL`. After all, there are three items still on the stack: the N is at the top, and C is at the bottom. If, instead of using a stack we used a queue, then the values popped would be: A, M, E, R, I and `NIL`. There would be three items still on the queue: N at the top and C on the bottom.

§2 Trees and Binary Search Trees

Trees, in general, use the following terminology: the **root** is the top node in the tree; **children** are the nodes that are immediately below a parent node; **leaves** are the bottom-most nodes on every branch of the tree; and **siblings** are nodes that have the same immediate parent.

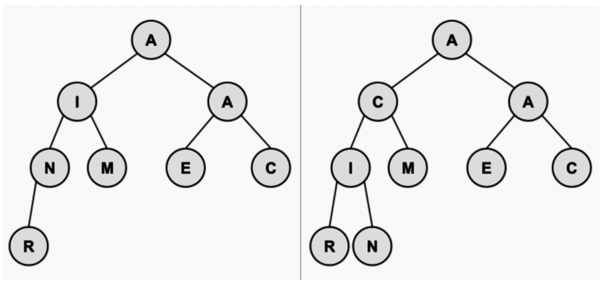
A **binary search tree (BST)** is composed of nodes having three parts: information (or a key), a pointer to a left child, and a pointer to a right child. It has the property that the key at every node is always **greater than or equal** to the key of its left child, and **less than** the key of its right child (that is, in ACSL, when there is a new node with duplicate key, always place it as the left child). The ACSL almost always give the input as a string, so we need to process each character one by one.

Example 2.1

Build a binary search tree for the word `SASKATOON` and determine how many nodes have only one child in the BST.

§3 Priority Queues and Heaps

A **priority queue** (implemented by **min-heap**) is quite similar to a binary search tree, but one can only delete the smallest item and retrieve the smallest item. A min-heap priority queue node has values/keys of both its children **greater than equal** to its own value/key. It is very common used in code to reduce the program run time from $O(n^2)$ to $O(n \log n)$.



The algorithm for insertion is not too difficult: put the new node at the bottom of the tree and then go up the tree, making exchanges with its parent, until the tree is valid. The heap at the left was building from the letters A, M, E, R, I, C, A, N (in that order); the heap at the right is after a C has been added.

§4 Practice Problems

Problems

1. Consider an initially empty stack. After the following operations are performed, what is the value of Z? `PUSH(3); PUSH(6); PUSH(8); Y = POP(); X = POP(); PUSH(X-Y); Z = POP();`
2. Create a min-heap for `PROGRAMMING`. What are the letters in the bottom-most row?
3. Create a binary search tree from the letters in the word `PROGRAM`. What is the internal path length (sum of the depths of all nodes)?
4. When creating a BST for `GREATEXPECTATIONS`, which node(s) have two children?

Answer Key

1. -2
2. `RORN`
3. 12
4. G, A, R, T

Chapter 9: FSAs and Reg Exs

WMC CS CLUB

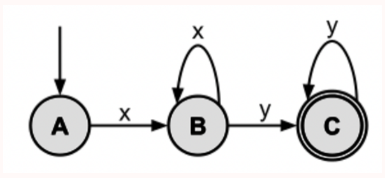
March 24, 2022

§1 Basics of FSA and Regular Expressions

A **Finite State Automaton** (FSA) is a mathematical model of computation comprising all 4 of the following: 1) a finite number of states, of which exactly one is active at any given time; 2) transition rules to change the active state; 3) an initial state; and 4) one or more final states. We can draw an FSA by representing each state as a circle, the final state as a double circle, the start state as the only state with an incoming arrow, and the transition rules as labeled-edges connecting the states.

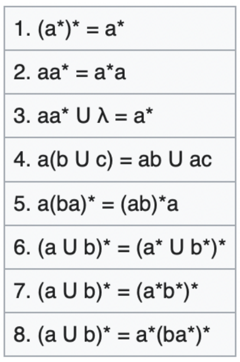
Example 1.1

Parse the following FSA diagram.



Explanation. In the above FSA, there are three states: A, B, and C. The only way to go from state A to B is by seeing the letter x. Once in state B, there are two transition rules: seeing the letter y will cause the FSA to make C the active state, and seeing an x will keep B as the active state. State C is a final state so if the string being parsed is completed and the FSA is in State C, the input string is said to be accepted by the FSA. In State C, seeing any additional letter y will keep the machine in state C. The FSA above will accept strings composed of one or more x's followed by one or more y's (e.g., xy, xxy, xxxyy, xyyy, xxyyyy).

A **Regular Expression** (RE) is an algebraic representation of an FSA. Think of it as an abstraction of FSA, as if numbers are an abstraction of counting. For example, the regular expression corresponding to the FSA given above is xx*yy*. The rules for forming a RE are as shown below. Figure 1 on the right shows some common identities of Regular Expressions.



1. (a*)* = a*
2. aa* = a*a
3. aa* U λ = a*
4. a(b U c) = ab U ac
5. a(ba)* = (ab)*a
6. (a U b)* = (a* U b*)*
7. (a U b)* = (a*b*)*
8. (a U b)* = a*(ba*)*

Figure 1. RE Identities

1. The null string (λ) is a RE.
2. If the string a is in the input alphabet, then it is a RE.
3. if a and b are both REs, then so are the strings built up using the following rules:
   a) **CONCATENATION**. "ab" (a followed by b).
   b) **UNION**. "a ∪ b" or "a | b" (a or b).
   c) **CLOSURE**. "a*" (a repeated zero or more times). This is known as the Kleene Star.

Remark 1.2 (Order of Precedence)**.** The order of precedence for Regular Expression operators is: Kleene Star, concatenation, and then union. For example, "dca*b" generates strings dcb, dcab, dcaab, etc. However, "d(ca)*b" generates strings db, dcab, dcacab, etc.

§2 RegEx in Practice

Programmers use Regular Expressions (usually referred to as regex) extensively for expressing patterns to search for. All modern programming languages have regular expression libraries. Here are the syntax rules that we will use.

Pattern	**Description**
|	A vertical bar separates alternatives. For example, gray | grey can match "gray" or "grey".
*	The asterisk indicates zero or more occurrences of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
?	The question mark indicates zero or one occurrences of the preceding element. For example, colou?r matches both "color" and "colour".
+	The plus sign indicates one or more occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".
.	The wildcard. matches any character. For example, a.b matches any string that contains an "a", then any other character, and then a "b" such as "a7b", "a&b", or "arb", but not "abbb". Therefore, a.*b matches any string that contains an "a" and a "b" with 0 or more characters in between. This includes "ab", "acb", or "a123456789b".
[]	A bracket expression matches a single character that is contained within the brackets. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", or "z".
[∧]	Matches a single character that is not contained within the brackets. For example, [∧abc] matches any character other than "a", "b", or "c". [∧a-z] matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
()	Parentheses define a sub-expression. For example, the pattern `H(ä|ae?)ndel` matches "Handel", "Händel", and "Haendel".

Example 2.1

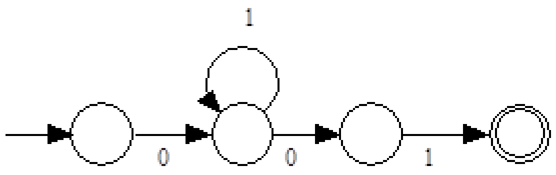
Which of the following strings match the RE pattern "[A-D]*[a-d]*[0-9]"?
1. ABCD8 **2.** abcd5 **3.** ABcd9 **4.** AbCd7 **5.** X **6.** abCD7
7. DCCBBBaaaa5

Explanation. The pattern describes strings the start with zero or more uppercase letters A, B, C, or D (in any order), followed by zero or more lowercase letter a, b, c, or d (in any order), followed by a single digit. The strings that are represented by this pattern are 1, 2, 3, and 7.

§3 Practice Problems

§3.1 Problems

1. Find a simplified Regular Expression for the following FSA:



2. Which of the following strings are accepted by the following Regular Expression "00*1*1 U 11*0*0"?
   A. 000000111111 **B.** 1010101010 **C.** 1111111 **D.** 0110 **E.** 10

3. Which of the following strings match the RE pattern `Hi?g+h+[^a-ceiou]`?
   1. Highb **2.** HiiighS **3.** HigghhhC **4.** Hih **5.** Hghe **6.** Highd
   7. HgggggghX

§3.2 Answer Key

1. The expression 01*01 is read directly from the FSA. It is in its most simplified form.

2. This Regular Expression parses strings described by the union of 00*1*1 and 11*0*0. The RE 00*1*1 matches strings starting with one or more 0s followed by one or more 1s: 01, 001, 0001111, and so on. The RE 11*0*0 matches strings with one or more 1s followed by one or more 0s: 10, 1110, 1111100, and so on. In other words, strings of the form: 0s followed by some 1s; or 1s followed by some 0s. Choice **A** and **E** following this pattern.

3. The ? indicates 0 or 1 "i"s. The + indicates 1 or more "g"s followed by 1 or more "h"s. The ∧ indicates that the last character cannot be lower-case a, b, c, e, i, o, or u. The strings that are represented by this pattern are **3**, **6**, and **7**.

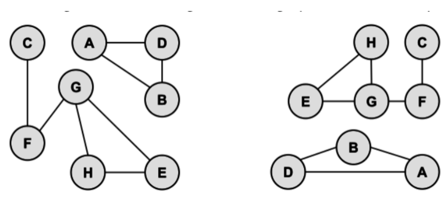
Chapter 10: Graph Theory

WMC CS CLUB

March 31, 2022

§1 Graph 101: Introduction

A **graph** is a collection of vertices and edges. An **edge** is a connection between two **vertices** (sometimes referred to as **nodes**). One can draw a graph by marking points for the vertices and drawing lines connecting them for the edges, but the graph is defined independently of the visual representation.



For the graph on the right, the precise way to represent this graph is to identify its set of vertices $\{A, B, C, D, E, F, G\}$, and its set of edges between these vertices $\{AB, AD, BD, CF, FG, GH, GE, HE\}$.

§2 Graph 102: Terminology

- **Undirected Graph:** The edges of the above graph have no directions meaning that the edge from one vertex A to another vertex B is the same as from vertex B to vertex A. Such a graph is called an undirected graph. Similarly, a graph having a direction associated with each edge is known as a directed graph.
- **Path & Simple Path:** A path from vertex x to y in a graph is a list of vertices, in which successive vertices are connected by edges in the graph. For example, $FGHE$ is path from F to E in the graph above. A simple path is a path with no vertex repeated. For example, $FGHEG$ is not a simple path.
- **Connected Graph & Connected Component:** A graph is connected if there is a path from every vertex to every other vertex in the graph. Intuitively, if the vertices were physical objects and the edges were strings connecting them, a connected graph would stay in one piece if picked up by any vertex. A graph which is not connected is made up of connected components. For example, the graph above has two connected components: $\{A, B, D\}$ and $\{C, E, F, G, H\}$.
- **Cycle:** A cycle is a path, which is simple except that the first and last vertex are the same (a path from a point back to itself). For example, the path $HEGH$ is a cycle in our example. Vertices must be listed in the order that they are traveled to make the path; any of the vertices may be listed first. Thus, $HEGH$ and $EHGE$ are different ways to identify the same cycle. For clarity, we list the start / end vertex twice: once at the start of the cycle and once at the end.

- **Sparse & Dense Graph:** Denote the number of vertices in a given graph by V and the number of edges by E. Note that E can range anywhere from V to V^2 (or $\frac{V(V-1)}{2}$ in an undirected graph). Graphs with all edges present are called complete graphs; graphs with relatively few edges present (say less than $V \log(V)$) are called sparse; graphs with relatively few edges missing are called dense.

- **Weighted Graph:** A weighted graph is a graph that has a weight (cost) associated with each edge.

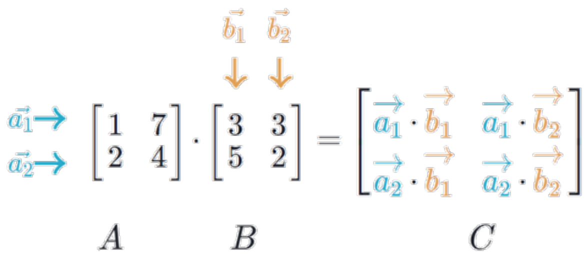
- **Tree and Forest:** A graph without cycles is called a tree. There is only one path between any two nodes in a tree. A tree with N vertices contains exactly $N1$ edges. A group of disconnected trees is called a forest.

- **Spanning Tree:** A spanning tree of a graph is a subgraph that contains all the vertices and forms a tree. A minimal spanning tree can be found for weighted graphs in order to minimize the cost across an entire network.

§3 Graph 103: Adjacency Matrices

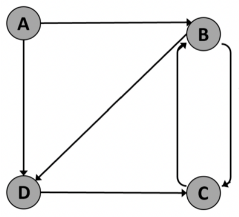
§3.1 Matrices

A matrix is a rectangular arrangement of numbers into rows and columns. Each number in a matrix is referred to as a matrix element or entry. For example, the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ has 2 rows and 3 columns. In order for two matrices A and B to multiply, the number of rows of A must equal to the number of columns of B. Multiplying matrices together is essentially taking the dot product ($\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$, where a_i and b_i are components of vectors $\vec{a}$ and $\vec{b}$) of vectors. The following diagram demonstrates matrix multiplication.


$\vec{b_1}$
$\vec{b_2}$
$\vec{a_1}$
$\vec{a_2}$
$\begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix}$
A
B
C

§3.2 Adjacency Matrices

It is frequently convenient to represent a graph by a matrix known as an adjacency matrix. Consider the following directed graph:



To draw the adjacency matrix, we create an N by N grid and label the rows and columns for each vertex. Then, place a 1 for each edge in the cell whose row and column correspond to the starting and ending vertices of the edge. Finally, place a 0 in all other cells.

$$M = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & & & & \\ B & & & & \\ C & & & & \\ D & & & & \end{array} = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & & 1 & & 1 \\ B & & & 1 & 1 \\ C & & 1 & & \\ D & & & 1 & \end{array} = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 0 & 1 & 1 \\ C & 0 & 1 & 0 & 0 \\ D & 0 & 0 & 1 & 0 \end{array}$$

By construction, cell (i, j) in the matrix with a value of 1 indicates a direct path from vertex i to vertex j. If we square the matrix, the value in cell (i, j) indicates the number of paths of length 2 from vertex i to vertex j.

$$M^2 = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 2 & 1 \\ B & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 1 & 1 \\ D & 0 & 1 & 0 & 0 \end{array}$$

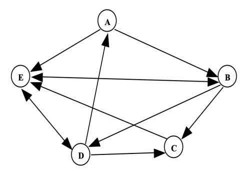
In general, if we raise M to the pth power, the resulting matrix indicates which paths of length p exist in the graph. The value in $M^p(i, j)$ is the number of paths from i to j.

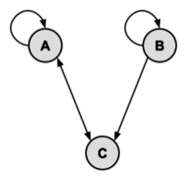
§4 Graph 104: Practice Problems

§4.1 Problems

1. Find the number of different cycles contained in the directed graph with vertices $\{A, B, C, D, E\}$ and edges $\{AB, BA, BC, CD, DC, DB, DE\}$.

2. How many paths of length 2 are in the directed graph shown in the first figure?





3. In the above directed graph (the second figure), find the total number of different paths from vertex A to vertex C of length 2 or 4.

4. Given the adjacency matrix, draw/describe the directed graph: $\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$.

§4.2 Answer Key

1. The cycles are: ABA, $BCDB$, and CDC. Thus, there are 3 cycles in the graph.

2. To find the number of paths of length 2, add the entries in the square of the adjacency matrix. The sum is 24.

3. First construct M, then calculate M^2 and M^4. Observe that there is 1 path of length 2 from A to C (cell $[1, 3]$ in M^2); 3 paths of length 4 (cell $[1, 3]$ in M^4).

4. There must be exactly 4 vertices: $V = \{A, B, C, D\}$. There must be be exactly 7 edges: $E = \{AB, AD, BA, BD, CA, DB, DC\}$.

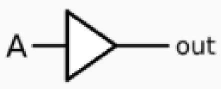
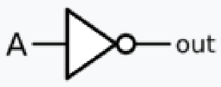
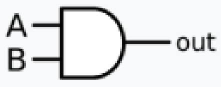
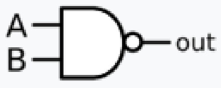
Chapter 11: Digital Electronics

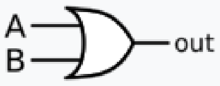
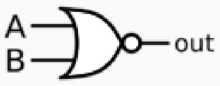
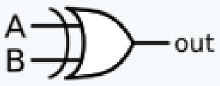
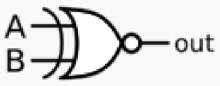
WMC CS CLUB

April 7, 2022

§1 Digital Electronics

A digital circuit is constructed from logic gates. Each logic gate performs a function of boolean logic based on its inputs, such as AND or OR. Each circuit can be represented as a Boolean Algebra expression; this topic is an extension of the topic of Boolean Algebra, which includes a thorough description of truth tables and simplifying expressions.

NAME	GRAPHICAL SYMBOL	ALGEBRAIC EXPRESSION	TRUTH TABLE
BUFFER		X = A	INPUT | OUTPUT; 0 | 0; 1 | 1
NOT		X = $\overline{A}$ or $\neg A$	INPUT | OUTPUT; A | X; 0 | 1; 1 | 0
AND		X = AB or $A \cdot B$	INPUT | OUTPUT; A B | X; 0 0 | 0; 0 1 | 0; 1 0 | 0; 1 1 | 1
NAND		X = $\overline{AB}$ or $\overline{A \cdot B}$	INPUT | OUTPUT; A B | X; 0 0 | 1; 0 1 | 1; 1 0 | 1; 1 1 | 0

			INPUT		OUTPUT
			A	**B**	**X**
OR		X = $A + B$	0	0	0
			0	1	1
			1	0	1
			1	1	1
NOR		X = $\overline{A + B}$	0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR		X = $A \oplus B$	0	0	0
			0	1	1
			1	0	1
			1	1	0
XNOR		X = $\overline{A \oplus B}$ or $A \odot B$	0	0	1
			0	1	0
			1	0	0
			1	1	1

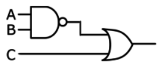
§2 Online Tools

The ACSL proposes the Logisim (http://www.cburch.com/logisim/index.html) application for digital electronics experimentation. Logisim is a wonderful tool for exploring this topic. Logisim is free to download and use; among its many features is support to automatically draw a circuit from a Boolean Algebra expression; to simulate the circuit with arbitrary inputs; and to complete a truth table for the circuit.

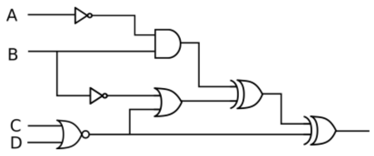
§3 Practice Problems

§3.1 Problems

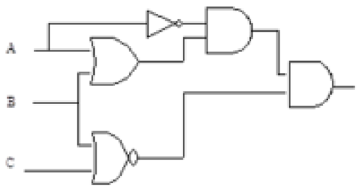
1. Find all ordered triplets (A, B, C) which make the following circuit FALSE:



2. How many ordered 4-tuples (A, B, C, D) make the following circuit TRUE?



3. Simplify the Boolean expression that this circuit represents.



§3.2 Answer Key

1. For the circuit to be FALSE, both inputs to the file OR gate must be false. Thus, input C must be FALSE, and the output of the NAND gate must also be false. The NAND gate is false only when both of its inputs are TRUE; thus, inputs A and B must both be TRUE. The final answer is (TRUE, TRUE, FALSE), or **(1, 1, 0)**.
2. We'll use a truth table to solve this problem. The rows in the truth table will correspond to all possible inputs - 16 in this case, since there are 4 inputs. From the truth table, there are **10** rows where the final output is TRUE.
3. The circuit can be simplified to **0**, or always false.

Chapter 12: Assembly Language

WMC CS CLUB

April 28, 2022 (Year End)

§1 Introduction to Assembly

Programs written in high-level languages are traditionally converted by compilers into assembly language, which is turned into machine language programs – sequences of 1's and 0's – by an assembler. Even today, with very good quality compilers available, there is the need for programmers to understand assembly language. First, it provides programmers with a better understanding of the compiler and its constraints. Second, on occasion, programmers find themselves needing to program directly in assembly language in order to meet constraints in execution speed or space.

ACSL chose to define its own assembly language rather than use a "real" one in order to eliminate the many sticky details associated with real languages. The basic concepts of our ACSL topic description are common to all assembly languages.

§2 Reference Manual

Execution starts at the first line of the program and continues sequentially, except for branch instructions (`BG`, `BE`, `BL`, `BU`), until the end instruction (`END`) is encountered. The result of each operation is stored in a special word of memory, called the "accumulator" (`ACC`). The initial value of the `ACC` is 0. Each line of an assembly language program has the following fields:

```
LABEL OPCODE LOC
```

The LABEL field, if present, is an alphanumeric character string beginning in the first column. A label must begin with an alphabetic character(A through Z, or a through z), and labels are case-sensitive. Valid OPCODEs are listed in the chart below; they are also case-sensitive and uppercase. Opcodes are reserved words of the language and (the uppercase version) many not be used a label. The `LOC` field is either a reference to a label or immediate data. For example, "LOAD A" would put the contents referenced by the label "A" into the `ACC`; "LOAD =123" would store the value 123 in the `ACC`. Only those instructions that do not modify the `LOC` field can use the "immediate data" format. In the following chart, they are indicated by an asterisk in the first column.

OP CODE	**DESCRIPTION**
*LOAD	The contents of LOC are placed in the ACC. LOC is unchanged.
STORE	The contents of the ACC are placed in the LOC. ACC is unchanged.
*ADD	The contents of LOC are added to the contents of the ACC. The sum is stored in the ACC. LOC is unchanged. Addition is modulo 1,000,000.
*SUB	The contents of LOC are subtracted from the contents of the ACC. The difference is stored in the ACC. LOC is unchanged. Subtraction is modulo 1,000,000.
*MULT	The contents of LOC are multiplied by the contents of the ACC. The product is stored in the ACC. LOC is unchanged. Multiplication is modulo 1,000,000.
*DIV	The contents of LOC are divided into the contents of the ACC. The signed integer part of the quotient is stored in the ACC. LOC is unchanged.
BG	The contents of LOC are divided into the contents of the ACC. The signed integer part of the quotient is stored in the ACC. LOC is unchanged.
BE	Branch to the instruction labeled with LOC if ACC = 0.
BL	Branch to the instruction labeled with LOC if ACC < 0.
BU	Branch to the instruction labeled with LOC.
READ	Read a signed integer (modulo 1,000,000) into LOC.
PRINT	Print the contents of LOC.
DC	The value of the memory word defined by the LABEL field is defined to contain the specified constant. The LABEL field is mandatory for this opcode. The ACC is not modified.
END	Program terminates. LOC field is ignored and must be empty.

§3 Practice Problems

§3.1 Problems

1. After the following program is executed, what value is in location TEMP?

TEMP	DC	0
A	DC	8
B	DC	-2
C	DC	3
	LOAD	B
	MULT	C
	ADD	A
	DIV	B
	SUB	A
	STORE	TEMP
	END	

2. If the following program has an input value of N, what is the final value of X which is computed? Express X as an algebraic expression in terms of N.

	READ	X
	LOAD	X
TOP	SUB	=1
	BE	DONE
	STORE	A
	MULT	X
	STORE	X
	LOAD	A
	BU	TOP
DONE	END	

§3.2 Answer Key

1. The ACC takes on values -2, -6, 2, -1, and -9 in that order. The last value, -9, is stored in location TEMP.

2. This program loops between labels TOP and DONE for A times. A has an initial value of X and subsequent terms of N, then values of A-1, A-2, . . . , 1. Each time through the loop, X is multiplied by the the current value of A. Thus, X = A * (A-1) * (A-2) * . . . * 1 or X = A! or A factorial. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120. Since the initial value of A is the number input (i.e. N), the algebraic expression is X = N!.

Thank you for your involvement and support of WMC CS Club this year!
Your hard work and dedication will pay off in the future.
Best wishes to your future endeavours.