**CSCI 465**

**Exam I**

**Due October 25, 2020**

**Take Home**

**(Total Points: 120)**

# First Name: Sam

Last Name: Dressler

1. (10 points) Regular expressions are closed under many operations--- that is, if we apply these operations to an RE or a collection of REs, the result is an RE.

1. What are these operations?

The primary operations that we use to build more complex regular expressions are union, concatenation, Kleene closure, and positive closure. Additionally, regular expressions are closed under difference and intersection.

What are the main incentives to use these operations?

Regular expressions allow us to process patterns and the operations listed above allow us to build regular expressions that can accept more complex patterns. They also give us methods to prove that languages using these operations are regular.

1. Can you provide one example for each operation?

for the following examples let L­1 and L2 denote finite languages where L1 = {0,1, 2, …, 9} and L2 = {a, b, …, z, A, B, …Z}

Concatenation: L1L2 is the set of strings generated by the languages where a number is followed by a letter.

Union: L1 U L2 is the set of digits and letters.

Kleene Closure: L1\* is the set of all strings with zero or more numbers, including the empty set.

Positive Closure: L2+ is the set of all strings of letters excluding the empty set, i.e. must have at least one letter.

Combined example: L2(L1 U L2) \* is the set of all identifiers that my lexical analyzer program accepted. This regular expression classifies a token as an identifier if it contains a letter followed by zero or more letters or numbers.

Intersection: L1 ⋂ L2: the set of all strings containing letters and numbers.

Difference L1 – L2: The set of all strings containing just numbers.

1. (20 points) Transform the following (recursive) grammar into Iteration (i.e., {} or \*)

(a) E→ T | E addop T

addop→+ | -

E -> T (addop T)\*

addop -> + | -

(b) RepeatStmt → **Repeat** Stmt Rest **Until** expr

Rest → ; Stmt Rest | ∈

expr→boolean

RepeatStmt -> **Repeat** Stmt Rest **Until** expr

Rest -> stmt B | ∈

B -> Rest

Expr -> boolean

1. (30 points) Consider the following grammar

S→A d

A→c |a A B

B→ b B |∈

1. Find FIRST(B) and Follow(B)

**First (B) = {b, ∈}**

Follow(S) = {d, $}

Follow(A) = {First(B)- ∈} U first{d} = {b, d, $}

**Follow(B) = Follow(A) = {b, d, $}**

1. Why is it harder to compute FOLLOW sets?

Computing the FOLLOW set calls for the handling of additional cases and it also calls for the computing of the FIRST sets to get the answer for some of the cases.

1. Why do we need to Compute FIRST and FOLLOW sets?

We need both FIRST and FOLLOW sets to be able to construct a predictive parsing table for a grammar. Through using both functions we can avoid the complication that occurs when we encounter an ∈.

These two sets also allow us to select the appropriate production rules for recursive parsing.

1. (10 points) Remove any issue from the following grammar so it can work with LL(1) parser.

expr → expr addop term| term

term→ ID | ID ‘(‘ expr ‘)’

addop→ + |-

A grammar is said to be LL(1) if it can be generated by a LL(1) grammar that is non ambiguous and not left-recursive. We can get this language by removing the left recursion in expr and by eliminating the left factoring in term(ID).

expr → expr’ | term

expr’ → expr addop term

term → ID term’

term’ → ‘(‘expr’)’ | ԑ

addop→ + |-

1. (20 points) Consider the following grammar

B→0B1

B→∈

(a). Transform the above grammar to PDA

* + P = (∑, Q, ▲, H, h0, q0, F)
    - ∑: Alphabet
    - Q: states
    - **▲: transition functions**
    - H: finite stack alphabet
    - h0: initial symbol in H (stack)
    - q0: Initial state
    - F: finite set of final states

▲= {

1. T (q, ∈, B) = (q, 0B1)

2. T (q, ∈, B) = (q, ∈)

3. T (q, 0, 0) = (q, ∈)

4. T (q, 1, 1) = (q, ∈)

}

(b). Parse the input **00110** with the initial configuration **(q, 00110, B)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Input** | **Stack** | **Transition** |
| **1** | .00110 | B | -- |
| **2** | .00110 | 0B1 | 1 |
| **3** | 0.0110 | B1 | 3 |
| **4** | 0.0110 | 0B11 | 1 |
| **5** | 00.110 | B11 | 3 |
| **6** | 00.110 | 11 | 2 |
| **7** | 001.10 | 1 | 4 |
| **8** | 0011.0 | -- | 4 |
| **9** | 0011.0 | B(no match) | No match |

There is no transition that would be able to accept the final 0 so the language would not be accepted. We would need to be able to push another production variable onto the stack but there is no available transition.

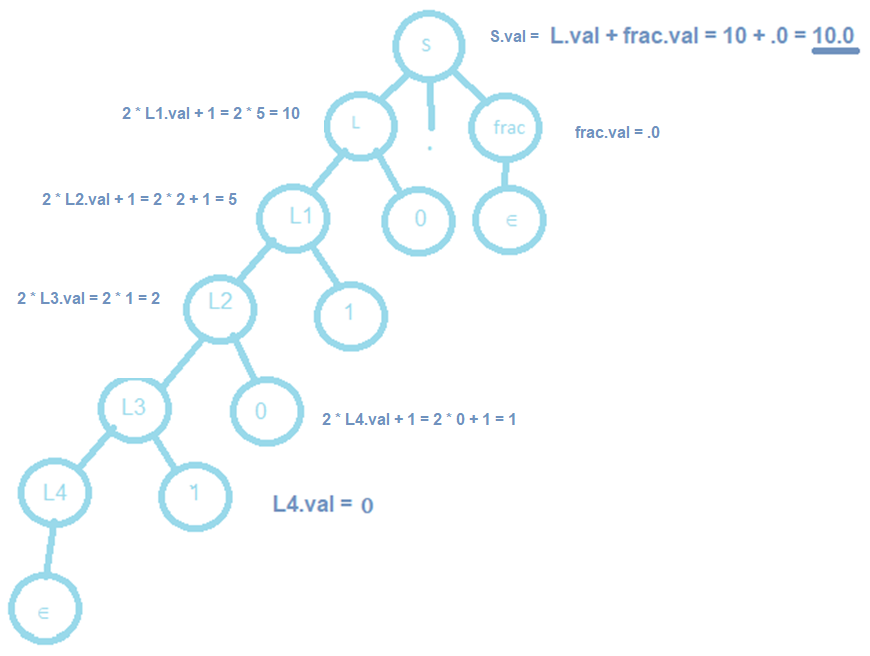
1. (30 points) consider the following **syntax-directed definition** which can be useful in converting from binary to decimal

(**Note**: L and frac represent strings of binary digits):

|  |  |
| --- | --- |
| Production | Semantic Rules |
| S → L. frac | S.val = L.val + frac.val |
| L → L1 **0** | L.val = 2\*L1.val |
| L → L1 **1** | L.val = 2\*L1.val + 1 |
| L →∈ | L.val = 0 |
| frac→ ∈ | frac.val = 0 |
| frac→ **0** frac1 | frac.val = frac1.val /2 |
| frac→ **1** frac1 | frac.val = 0.5 + (frac1.val/2) |

DRAW annotated parse tree based on the above syntax directed definition table. Using annotated tree with synthesized value **val**, show how to convert each of the following binary numbers (i.e., input) to the corresponding decimal number (i.e., output).

1. 1010.



1. 1.011

