Midterm Exam 1

601.428/628 Compilers and Interpreters

October 4, 2021

Complete all questions.

Time: 75 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: _____

Print name: reference solution

Date: _____

Question 1. [20 points] A language for doing computations on binary (base 2) values has the following kinds of tokens:

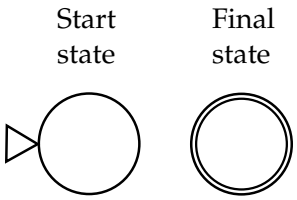
- An identifier starts with one of the letters a, or b, which may be following by 0 or more letters (a or b) or binary digits (0 or 1)
- A literal binary integer is a sequence of 1 or more occurrences of the binary digits 0 and 1
- Operators are +, -, *, and /

(a) Show regular expressions for generating identifier and literal binary integer tokens as described above. Note that you may use character classes such as [a-b] and [0-1] in your regular expressions.

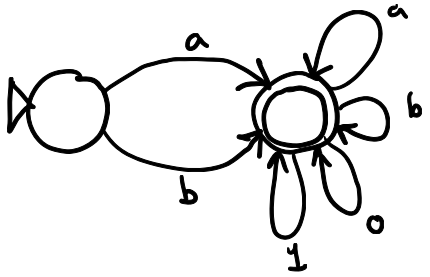
identifier: [ab] ([ab] | [01])*

literal binary integer: [01] [01]*

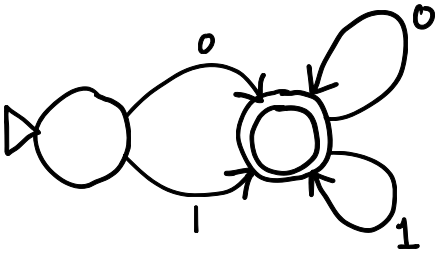
(b) Show two finite automata which recognize, respectively, identifier and literal binary integer tokens. Make sure each transition indicates a direction and is clearly labeled with an input symbol, that the start state and final (accepting) state(s) are clearly indicated using the notation shown on the right.



identifier:



literal binary integer:



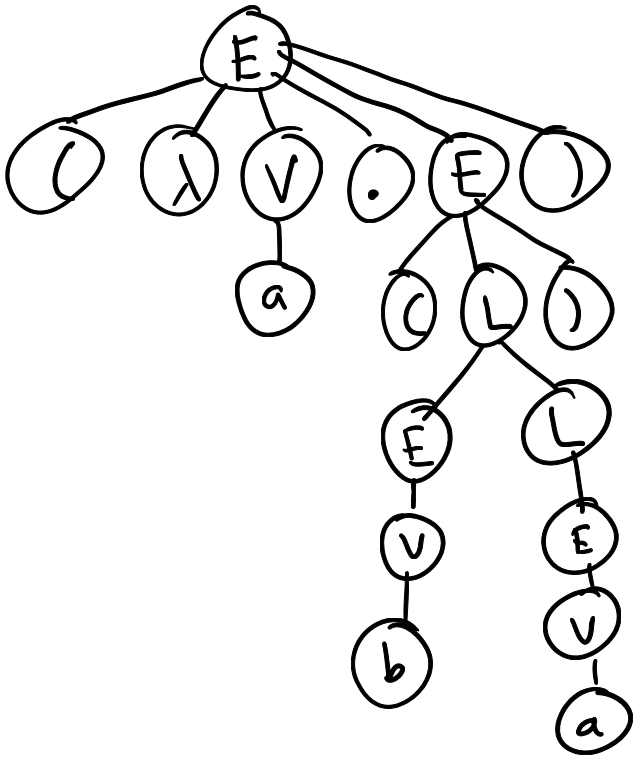
Question 2. [20 points] Consider the following context-free grammar with nonterminal symbols E L V (E is the start symbol) and terminal symbols ( ) λ . a b :

E → (λ V . E)
E → (L)
E → V
L → E
L → E L
V → a
V → b

(a) Show a derivation for the input string (λ a . (b a))

Working string	Production
E	E → (λ V . E)
(λ V . E)	V → a
(λ a . E)	E → (L)
(λ a . (L))	L → E L
(λ a . (E L))	E → V
(λ a . (V L))	V → b
(λ a . (b L))	L → E
(λ a . (b E))	E → V
(λ a . (b V))	V → a
(λ a . (b a))	

(b) Show the parse tree for the derivation you found in part (a). Make sure that each symbol is represented by exactly one node, and that connections from parent to child nodes are indicated clearly.



Question 3. [10 points] Show the FIRST and FOLLOW sets for the E nonterminal symbol in the context-free grammar in Question 2. Recall that

- the FIRST set of a nonterminal symbol is the set of terminal symbols that could begin an expansion of the symbol, including ϵ if the symbol could expand to an empty string, and
- the FOLLOW set is the set of terminal symbols that could follow an expansion of the symbol, including the special *eof* symbol if the expansion could occur at the end of the derived string of terminal symbols

FIRST(E) = { (, a , b }

FOLLOW(E) = { eof, a, b, (}

Question 4. [20 points] Show a pseudo-code implementation of a recursive descent parse function for the E nonterminal in the context-free language from Question 2. Assume that the lexical analyzer has

- a **next** operation to consume the next token (raising an exception if the end of input has been reached)
- a **peek** operation which returns the next token without consuming it (returning a null value if the ned of input has been reached)
- an **expect** operation which is passed a kind of token, calls **next**, and raises an exception if the returned token is not the expected kind of token

Your parse function doesn't need to build a tree. It just needs to choose and apply a production. For reference, the grammar productions are:

$E \rightarrow (\lambda V . E)$	$L \rightarrow E$	$V \rightarrow a$
$E \rightarrow (L)$	$L \rightarrow E L$	$V \rightarrow b$
$E \rightarrow V$		

```
parseE() {
    t = peek()
    if (t is null) { error("unexpected eof") }
    if (t is '(') {
        next()
        t = peek()
        if (t is null) { error("unexpected eof") }
        if (t is 'λ') {   // E → ( λ V . E )
            expect('λ')
            parseV()
            expect('.')
            parseE()
            expect(')')
        } else {          // E → ( L )
            parseL()
            expect(')')
        }
    } else if (t is 'a' or t is 'b') {   // E → V
        parseV()
    } else {
        error("unexpected token" + t)
    }
}
```

Question 5. [15 points] Consider the following grammar for an infix expression language where primitive operands are **a**, **b**, **1**, **2**, and **3**, and the infix operators are **+**, **-**, *, and **/**:

E → E **+** T	F → **a**
E → E **-** T	F → **b**
E → T	F → **1**
T → T * F	F → **2**
T → T **/** F	F → **3**
T → F	

(a) Show how to add parenthesized expressions to this language. For example, input strings such as **a * (3 - b)** should be allowed. Explain what productions would need to be added or modified to the original grammar.

add one production:

F → (E)

(b) Show how to add a unary minus (**-**) operator to the language. For example, input strings such as **3 * - a** should be allowed. Explain what productions would need to be added or modified to the original grammar.

in orig grammar, replace all references to 'F' in the right hand side of any production with 'U'

add productions:

U → - U

U → F

Question 6. [15 points] Consider the following program, which is written in the programming language you are implementing in Assignment 2:

```
var a, b;

function f(a) {
  a + b;
}

b = 2;
a = f(3);
println(a);
```

(a) Show the contents of the static environments for the global scope and the scope of the `f` function. (You may assume that `println` is the only intrinsic function defined in the global environment.) Make sure that within each environment, each symbol has a unique integer offset.

Global environment:

Name	Offset
println	0
a	1
b	2
f	3

Function `f` environment:

Name	Offset
a	0

(b) Annotate each reference to a name or function in the program above with its lexical address. Recall that a lexical address is a pair (*depth*, *offset*).

In orange above