U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Compilers (L.EIC026)

Spring 2023

Faculdade de Engenharia da Universidade do Porto (FEUP)
Departamento de Engenharia Informática (DEI)

First Test Solution

March 30, 2022 from 17:30 to 19:30

Please label all pages you turn in with your name and student number.

Name: _____ Student ID: _____

Problem 1: _____
Problem 2: _____
Problem 3: _____
Problem 4: _____

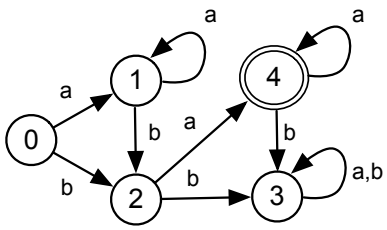
Total Grade: _____

INSTRUCTIONS:

1. **This is an open-notes exam with a single A4 (typed or hand-written) sheet**.
2. The test consists of 4 questions for a total of 100 points.
3. Please identify all the pages where you have answers that you wish to be graded and make sure to label each of the additional pages with the problem you are answering.
4. Use black or blue ink. No pencil answers are allowed.
5. Staple or bind additional answer sheets together with this document to avoid being misplaced or worse, lost. Make sure this cover page is stapled at the front.
6. Please avoid laconic answers so that we can understand you understood the concepts being asked.

Problem 1: Finite Automata and Regular Expressions [20 points]

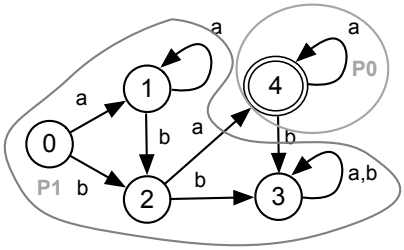
Given the Finite Automaton below with starting state 0 and alphabet {a,b} answer the following questions:

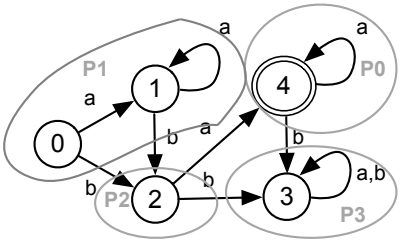


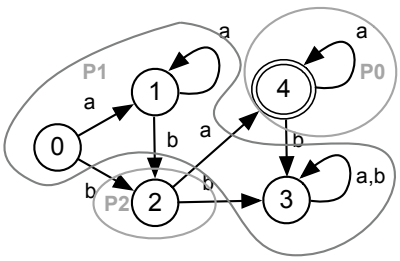
(a) [05 points] Is this a NFA or a DFA? Explain why or why not.
(b) [10 points] If needed convert it to a DFA and minimize it.
(c) [05 points] What is the Regular Expression matched by this NFA? You are advised **not to** use the automatic Kleene construction or try to look at the input NFA but rather the correctly minimized DFA.

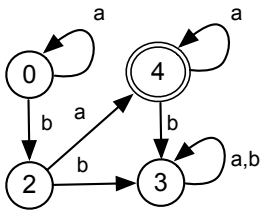
Solution:

(a) For every state, the FA has univocally defined transitions for every input character, so this is a fully specified DFA.
(b) The refinement sequence below shows the minimization of the original DFA with a final DFA with 4 states.
(c) A sequence with exactly one b proceeded by zero or more a characters and ending in at least one a character, which can be captured as the regular expression: a*.b.a+









Problem 2: Top-Down Predictive Parsing [30 points]

Given the following CFG grammar G = ({S ,L}, S, {a, (,), ","), P) with the set of productions P:

S → (L) | a
L → L, S | S

Answer the following questions:

a) [05 points] Is this grammar suitable to be parsed using the recursive descendent parsing method? Justify and modify the grammar if needed.
b) [10 points] Compute the FIRST and FOLLOW set of non-terminal symbols of the grammar resulting from your answer in a)
c) [10 points] Construct the corresponding parsing table using the predictive parsing LL method.
d) [05 points] Is the original grammar G suitable to be parsed using a shift-reduce bottom-up parser as the ones we discussed in class? Why or why not?

Solution:

a) No because it is left-recursive. You can expand L using a production with L as the left-most symbol without consuming any of the input terminal symbols. To eliminate this left recursion we add another non-terminal symbol, L' and productions as follows:

S → (L) | a
L → S L'
L' → , S L' | ε

b) FIRST(S) = { (, a }, FIRST(L) = { (, a } and FIRST(L') = { , , ε}
FOLLOW(L) = {) }, FOLLOW(S) = { , ,) , $ }, FOLLOW(L') = {) }

c) The LL table, without any conflicts, is as shown below.

	(	)	a	,	$
S	S→(L)		S → a		
L	L→S L'		L→ S L'		
L'		L'→ε		L'→ , S L'	

d) We would not have to modify the original grammar, as shift-reduce parsers can handle grammars which are left recursive (unlike predictive parsers).

Problem 3. Attributive Grammar and Syntax-Directed Translation [30 points]

You are asked to develop a CFG G that can parse a list of comma-separated positive integer values. Then, based on this CFG you need to develop an attributive grammar that adds in this sequence, only the values that are (strictly) greater than their two immediate neighbors. The value of the addition, should be assigned to the entire sequence through an attribute of the grammar's start symbol. As an example, in the sequence "34, 51, 22, 1, 4, 2, 7" only the values 51 and 4 are larger than their immediate neighbors. As such, the value 51 + 4 = 55 should be assigned to the sequence.

It is suggested that you structure your answer by carrying out the following:

a) [10 points] develop a simple CFG that can parse the list of comma-separated integers, assuming you have a terminal symbol, say value, whose value is captured in its integer "val" attribute. The scanner should be responsible for successfully scanning this value from the input and converting it to an integer value through a library function such as "atoi" in the standard C library.
b) [15 points] Develop the attributes and the semantic rules associated with each grammar production indicating if the attributes and either synthesized or inherited. Only primitive data typed attributes, such as integer or Boolean are allowed. No attributes such as lists, sets or other are allowed.
c) [05 points] Depict the parse tree corresponding to the input "3, 2, 10, 9, 9" indicating the value of each attributed you have defined in b) and discuss which parsing approach would be most suitable to evaluate your attribute grammar in a single pass.

Solution:

a) A possible solution is to define a simple CFG grammar G with a single non-terminal symbol S, which is also the start symbol S and a terminal symbol num and productions shown below.

$S \rightarrow S$ ',' num
$S \rightarrow$ num

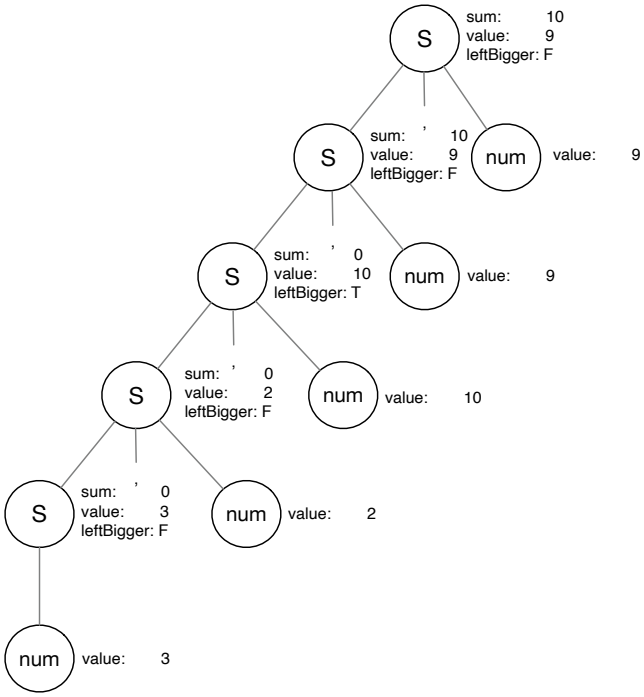
b) For this non-terminal S we define three synthesized attributes, namely, two integer attributes named value and sum, and a boolean attribute named leftBigger. The sum attribute tracks the summation of the values that have the desired property (both neighbors are smaller), the value attribute captures the value of the number currently being considered, and the leftBigger boolean captures the fact that the current value is bigger than the value on the left of the current value. The terminal symbol num has as single integer attribute value the integer number the corresponding token represents.

$S_0 \rightarrow S_1$ ',' num

```
{   if (S1.leftBigger == true){
        if (num.value < S1.value){
           S0.leftBigger = false;
           S0.sum = S1.sum + num.value;
        } else { // num.value > S1.value
           S0.leftBigger = true;
           S0.sum = S1.sum;
        }
    } else { // leftbigger = false
        if (num.value > S1.value){
            S0.leftBigger = true;
        } else {
            S0.leftBigger = false;
        }
        S0.sum = S1.sum;
    }
    S0.value = num.value;
}
```

$S \rightarrow$ num { S.value = num.value; S.leftBigger = false; S.sum = 0; }

c) The parse tree corresponding to the input "3, 2, 10, 9, 9" is depicted below and as all the attributes used in this attribute grammar are synthesize, the evaluation could be done in a single pass using a post-order DFS traversal, or even during bottom-up parsing as the production are "reduced" during parsing.



Problem 4. Code Generation [20 points]

Consider the following grammar G for expressions and lists of statements (StatList) using assignment statements (Assign) and basic expressions (Expr) using the productions presented below. Here const stands for an integer constant and id stands for a scalar variable. The input program is assumed to be syntactically correct.

(0) StatList → Stat ; StatList
(1) StatList → Stat
(2) Stat → Assign
(3) Expr → id
(4) Expr → id + Expr
(5) Expr → const
(6) Assign → id = Expr
(7) Assign → id += Expr

(a) [25 points] Using a three-address instruction target machine write a syntax-directed definition to generate code for StatList, Assign and Expr. Describe briefly the attribute of your translation and auxiliary functions used. Your solution should use the minimal amount possible of temporary registers.
(b) [05 points] Regarding arithmetic expressions, is this grammar left or right associative? Justify your answer.

Solution:

(a) A way to minimize the number of temporaries used is to use the constants and scalar variables as much as possible in the code generation rather than used a very simple approach where you would save in every instance of the Expr symbol the results in a newly allocated temporary variable. Instead we emit the code of constants and identifiers in strings that we use to carry upwards in the parse tree. There are therefore three synthesized attributes we used, the place attribute that will be non-nil only for expression that are bound to temporary variables and two other attributes, the kind of expression (temp, var, const) and a value for identifiers (the index into a symbol table) and integer constants. We make the further improvement that in a binary operator involving constant the parser immediately applies constant-folding and computes the value of the sub-expression. We make use of the auxiliary function newTemp() and emit() with obvious meanings.

```
Expr → id  { Expr.place = nil; Expr.kind = var; Expr.value = id.value }

Expr → const  { Expr.place = nil; Expr.kind = const; Expr.val = const.value; }

Expr0 → id + Expr1          {
  if(Expr1.kind == const) {
     Expr0.kind = const; Expr0.place = nil; Expr0.value = id.value + Expr1.value; }
  if(Expr1.kind == var) { // repeat for (var,const) and (var,var)
     t = newTemp();
     Expr0.kind = temp; Expr0.place = t; emit('t = id.value + IdString(Expr1.value));
   }
 }

Assign → id = Expr {
   if(Expr.kind == const) emit('IdString(id.value) = Expr.value');
   if(Expr.kind == var) emit('IdString(id.value) = IdString(Expr.value)'); // could check if diff
   if(Expr.kind == temp) {
     emit('IdString(id.value) = Expr.place');
     releaseTemp(Expr.place);
  }
}

 Assign → id += Expr1{
   if(Expr1.kind == var) /* could check if diff */
      emit('IdString(is.value) = IdString(id.value) + IdString(Expr1.value)');
   if(Expr1.kind == const)
      emit('IdString(is.value) = IdString(id.value) + Expr1.value');
   if(Expr1.kind == temp){
      emit('IdString(is.value) = IdString(id.value) + Expr1.place);
      releaseTemp(Expr1.place);
   }
 }
```

There are no attribute or semantic actions for the Stat and StatList symbol as these merely organize the statements and assignments in the program.

(b) This is a right-associative grammar as the rule: Expr0 → id + Expr1 implies that all the expressions are aggregated at the right-hand-side. The corresponding parse tree is thus right-skewed.