COMP SCI 4TB3/6TB3 2021/22 Midterm Test 1 [24 points]

Emil Sekerinski, McMaster University, 1 March 2022, 10:30 - 12:20

- Complete the test on jhub4tb3, do not download the test. Don't forget to submit. You can submit multiple times, the last version will be taken.
- If you have questions for clarification, you can ask those by raising your hand or by asking Yaminah, qureshiy@mcmaster.ca, on Teams. Teams can be used only for that purpose.
- You may add cells as needed. If your kernel hangs, restart the kernel by `Kernel → Restart`, don't just close the window and open a new window.
- Please have your student card ready. You may use paper and pencil, but not a calculator (it would not be of use).
- Before you start, on jhub4tb3 go to the "Running" tab and shut down all terminals and notebooks that you have used previously but not exited. Your midterm test may not be able to run properly if you are using too many resources.
- You may also use python.org, although you will be pressed for time and it may not be of much use.
- This is a "closed book" test: your browser must have only one tab with Jupyter on jhub4tb3 and one tab with python.org. Accessing any other material on jhub4tb3, on your computer, through the web, or through any other means is not allowed. This includes the course notes and the assignments of this course. Any attempt to access other material will be treated as Academic Dishonesty.

Math symbols to copy and paste:

```
· × ∑
≤ ≥ ≠ ≡ ≢
¬ ∧ ∨ ∀ ∃
⇐ ⇒ →
∩ ∪ ⊂ ⊆ ∈ ∉ ∅ ℂ ε
0 1 2 3 4 5 6 7 8 9 i j + m ⁰ ¹ ² ³ ⁴ ⁵ ⁶ ⁷ ⁸ ⁹ ⁱ ⁿ
```

Question 1 (Derivation) [4 points]. Let `G = (T, N, P, S)`, where `T = {a, b}`, `N = {A, B, S, X, $}`, and productions `P` are:

```
S → X$
    X → ε
    X → aXA
    X → bXB
    Aa → aA
    Ab → bA
    Ba → aB
    Bb → bB
    A$ → a$
    B$ → b$
    $ → ε
```

- Give the derivation of `abab`!
- Describe the language generated by `G`!
- Can you construct a recursive-descent parser for directly for `G` or by rewriting `G` into an equivalent grammar? Explain briefly; you do not need to write a parser!

Derivation of abab:

S → X$

→ aXA$

→ aXa$

→ abXBa$

→ abXaB$

→ abab$

→ abab

The language generated by G is any a's and b's that are even in a string.

You can constructs a recursive descent parser directly for G, because this is an LL(1) grammar, it is a context free grammar and has no left recursion. It is already in a form that can be directly parsed, however we can edit this grammar if we so wish if it makes the algorithm more efficient:

S -> X$

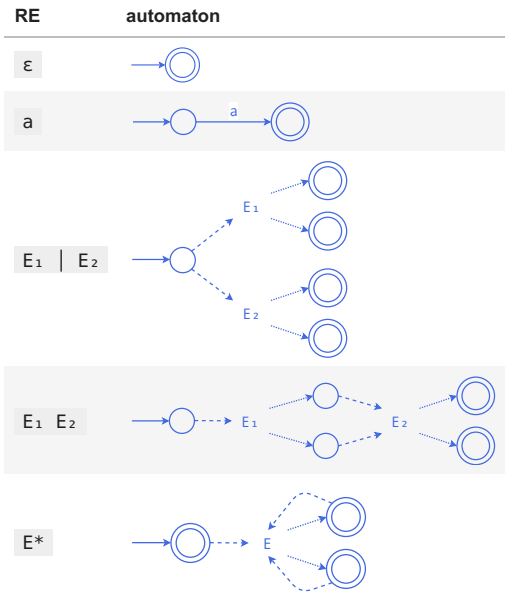
X -> ε | aXA | bXB

A -> aA | bA

B -> aB | bB

Question 2 (Equivalence of Regular Expressions) [10 points]. Consider regular expressions `E1 = ab(ab)*` and `E2 = a(ba)*b`. Show that they are equivalent by following steps:

- Construct finite state automata `A1` and `A2` equivalent to `E1` and `E2`, respectively, following the procedure from the course notes in terms of `(T, Q, R, q₀, F)`. Write the transitions in the form `0 a → 1`. Give only the resulting automata. A cheat sheet is to the right. [3 points]



A1 = (T, Q, R, q_0, F) with T = {a, b}, Q = {0, 1, 2, 3}, F = {2} and the set R of transitions as follows:

0 a -> 1

1 b -> 2

2 a -> 3

3 b -> 2

A2 (T, Q, R, q_0, F) with T = {a, b}, Q = {0, 1, 2, 3}, F = {3}and the set R of transitions as follows:

0 a -> 1

1 b -> 2

1 b -> 3

2 b -> 1

- Construct deterministic finite state automata `D1` and `D2` from `A1` and `A2` following the procedure from the course notes. Give only the resulting automata. Label the new states such that their correspondence to the old states is visible. [3 points]

`D1 = (T, Q, R, {0}, F)` with `T = {a,b}`, `Q = {{0},{1},{2},{3}}`, `F = {{2}}` and the set `R` of transitions as follows:

{0} a -> {1}

{1} b -> {2}

{2} a -> {3}

{3} b -> {2}

D1 is already deterministic, nothing changes.

`D2 = (T, Q, R, {0}, F)` with `T = {a,b}`, `Q = {{0},{1},{2},{3}}`, `F = {{2, 3}}` and the set `R` of transitions as follows:

{0} a -> {1}

{1} b -> {2,3}

{2, 3} -> {1}

- Construct now `M1` and `M2` by minimizing `D1` and `D2`, respectively, following the procedure from the course notes. Give only the

resulting automata. Label new states such that their correspondence to the old states is visible. [3 points]

`M1 = (T, Q, R, {0}, F)` with `T = {a,b}`, `Q = {{0},{1,3},{2}}`, `F = {{2}}` and the set `R` of transitions as follows:

{0} a -> {1, 3}

{1, 3} b -> {2}

{2} a -> {1, 3}

`M2 = (T, Q, R, {0}, F)` with `T = {a,b}`, `Q = {{0},{1},{2,3}}`, `F = {{2,3}}` and the set `R` of transitions as follows:

{0} a -> {1}

{1} b -> {2,3}

{2, 3} -> {1}

M2 is already minimized so it stays the same.

- Show that `M1` and `M2` are equivalent by giving a bijection between their states. [1 point]

The bijection of states between `M1` and `M2` is as follows:

```
{
    {0}: {0},
    {1,3}: {1},
    {2}: {2,3}
}
```

Question 3 (Recursive Descent Parsing) [10 points]. Give the rules that an EBNF grammar has to satisfy for recursive descent parsing. Use following template. [2 points]

`A`	`condition(A)`
`[E]`	
`{E}`	
`E₁ E₂ …`	
`E₁ | E₂ | …`	

`A`	`condition(A)`
`[E]`	`first(E) ∩ follows(A) = {}`
`{E}`	`first(E) ∩ follows(A) = {}`
`E₁ E₂ …`	`first(E₁) ∩ first(E_{i+1}) ∩ first(E_{i+1}) .... = {}` for any `E₁` that is nullable, provided `A` is not nullable.
`E₁ | E₂ | …`	`first(E₁) ∩ first(Eⱼ) = {}` for all $i \neq j$

Consider following EBNF grammar for balanced parenthesized expressions, with '`(`', '`)`', '`[`', '`]`', as parenthesis, with '`e`' as expressions, and with expressions separated by '`,`':

```
p ::= a {',' a}
a ::= 'e' | '[' p ']' | '(' p ')'
```

As expressions, only the character `'e'` is allowed. Argue that this grammar satisfies the conditions for recursive descent parsing. State explicitly what the needed `first` and `follow` sets are and be rigorous in writing down all the conditions. [2 points]

The grammar satisfies the following conditions for recursive descent parsing:

For p ::= a {',' a} we have the conditions

```
1. first(a) ∩ first({',' a}) = {} if a is nullable, which it is not. So the condition holds
vacuously.
2. first(',' a) ∩ follows({',' a}) = {} since {',' a} is nullable. first(',' a) = {','} and
follows({',' a }) = {']',')'}. These sets are disjoint and thus the condition holds.
3. first(',') ∩ first(a) = {} if ',' is nullable, which it is not. So the condition holds
vacuously.
```

a ::= 'e' | '[' p ']' | '(' p ')'

```
1. first('e') ∩ first('[' p ']') ∩ first('(' p ')') = {}. first('e') = {'e'}, first('[' p ']')
```

= {'['} and first('(' p ')') = {'('}. The 3 sets are disjoint thus the condition holds.
2. first('[') ∩ first(p) ∩ first(']') = {} if '[' is nullable, which it is not so the condition holds vacuously. Similar condition for '(' p ')'.

The *depth* of an expression is the maximal level of nested parenthesis or brackets, see examples below. Extend the grammar with attribute rules that compute the depth. Delineated the attribute rules by « and ». [2 points]

```
p(d) ::= a(x) « d := x » {',' a(y) « d := max(d,y) »}
a(d) ::= 'e' « d := 0 » | '[' p(d) « d := d+1 » ']' | '(' p(d) « d := d+1 » ')'
```

Write a recursive descent parser in Python for that grammar. The parser uses characters as input symbols and has to call `getCh()` to read the next character and has to inspect `ch` as the next input symbol: [4 points]

In [25]:
```
def getCh():
    global ch, source
    if len(source) > 0: ch, source = source[0], source[1:]
    else: ch = None

def depth(s) -> int:
    global source
    source = s; getCh()
    return p()

def a() -> int:
    if ch == 'e':
        getCh(); return 0
    elif ch == '[':
        getCh()
        d = p()
        if ch != ']':
            raise Exception()
        getCh()
    elif ch == '(':
        getCh()
        d = p()
        if ch != ')':
            raise Exception()
        getCh()
    else:
        raise Exception()
    return d+1

def p() -> int:
    d = a()
    while ch == ',':
        getCh()
        y = a()
        d = max(d, y)
    return d
```

In case there is an error in the input, your parser can stop right away (by `raise` in Python), you do not have to include any error treatment. Here are some test cases:

In [26]:
```
assert depth('e') == 0
```

In [27]:
```
assert depth('e,e') == 0
```

In [28]:
```
assert depth('(e)') == 1
```

In [29]:
```
assert depth('[e]') == 1
```

In [30]:
```
assert depth('(e,e,e)') == 1
```

In [31]:
```
assert depth('[e,e]') == 1
```

In [32]:
```
assert depth('([e],(e),e,e)') == 2
```

In [33]:
```
assert depth('[e,e,[e,[e,e]]]') == 3
```

In [ ]:
```

```