COMP SCI 4TB3/6TB3 2021/22 Midterm Test 2 [24 points]

Emil Sekerinski, McMaster University, 5 April 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:

```
· × ∑
≤ ≥ ≠ ≡ ≢
¬ ∧ ∨ ∀ ∃
⇐ ⇒ →
∩ ∪ ⊂ ⊆ ∈ ∉ ∅ ∁ 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
```

Part of the grammar of WebAssembly from the course notes:

```
num ::= digit {digit}
int ::= [ '-' ] num
float ::= num '.' [num] [('E' | 'e') ['+' | '-'] num]
name ::= '$' (letter | digit) {letter | digit}
string ::= \" {char} \"
module ::= "(" "module" {import} {global} {func} [table] {elem} [memory] [start] ")"
import ::= "(" "import" string string "(" "func" name func_type ")" ")"
global ::= "(" "global" name "(" "mut" type ")" instr ")"
memory ::= "(" "memory" num ")"
table ::= "(" "table" num "funcref" ")"
elem ::= "(" "elem" instr name ")"
start ::= "(" "start" name ")"
func ::= "(" "func" name func_type { local } { instr } ")"
func_type ::= { "(" "param" name type ")" } { "(" "result" type ")" }
local ::= "(" "local" name type ")"
type  ::= "i32" | "f32"
instr ::= "i32.const" int |
          "i32.add" | "i32.sub" | "i32.mul" | "i32.div_s" | "i32.rem_s" |
          "i32.eqz" | "i32.eq" | "i32.ne" | "i32.lt_s" | "i32.gt_s" | "i32.le_s" | "i32.ge_s" |
          "i32.load" "offset" "=" num |
          "i32.store" "offset" "=" num |
          "global.get" name |
          "global.set" name |
          "local.get" name |
          "local.set" name |
          "block" name { instr } "end" |
          "loop" name { instr } "end" |
          "if" { instr } [ "else" { instr } ] "end" |
          "br" name |
          "br_if" name |
          "return" |
          "call" name |
          "f32.const" float |
          "f32.add" | "f32.sub" | "f32.mul" | "f32.div" |
          "f32.sqrt" |
          "f32.min" | "f32.max" |
          "f32.ceil" | "f32.floor" |
          "f32.abs" | "f32.neg" |
          "f32.eq" | "f32.ne" | "f32.lt" | "f32.le" | "f32.gt" | "f32.ge" |
          "i32.trunc_f32_s" | "f32.convert_i32_s" |
          "f32.load" "offset" "=" num |
          "f32.store" "offset" "=" num
          "call_indirect" func_type
          "br_table" {name} name
```

Question 1 (WebAssembly) [5 points]. The characteristics of WebAssembly are:

- Stack architecture
- Byte code
- Statically typed
- Block-structured
- Non-uniform store

Explain each of these briefly and give for each an instruction or a sequence of instruction to illustrate the point!

YOUR ANSWER HERE

Question 2 (`for` -loops) [4 points]. Consider extending the P0 language with `for` -loops of the form:

```
for v := exp₁ to exp₂ do stat
```

Give one possible translation scheme for this loop and discuss which parts of the P0 compiler (or any other compiler) would need to be modified!

YOUR ANSWER HERE

Question 3 (Parameter Passing). [4 points] P0 passes integer and boolean parameters by value and passes arrays and records by reference (as a note, Java does the same).

- What changes to the translation scheme are needed if one were to pass integer and boolean parameters by reference? What is the disadvantage?
- What changes to the translation scheme are needed if one were to pass arrays and records by value? What is the disadvantage?

Give a brief answer!

YOUR ANSWER HERE

Question 4 (Alignment) [4 points]. Consider following record and array declarations:

```
type R = (b: boolean, i: integer, c: [0 .. 1] → boolean)
type A =  [3 .. 9] → R
```

Assuming that words are 4 bytes, `size(integer) = 4`, and `size(boolean) = 1`, what is `size(A)` with

1. sequential allocation that is not word-aligned,
2. word-aligned sequential allocation (words must be word-aligned, bytes can be arbitrarily aligned),
3. word-aligned allocation with reordering for minimizing the size.

YOUR ANSWER HERE

Question 5 (Extending Regular Expressions with Squaring) [7 points]. Consider following declarations from the course notes and assignments:

In [ ]:
```python
class set(frozenset):
    def __repr__(self):
        return '{' + ', '.join(str(e) for e in self) + '}'

class FiniteStateAutomaton:
    def __init__(self, T, Q, R, q0, F):
        self.T, self.Q, self.R, self.q0, self.F = T, Q, R, q0, F
    def __repr__(self):
        return str(self.q0) + '\n' + ' '.join(str(f) for f in self.F) + '\n' + \
               '\n'.join(str(q) + ' ' + a + ' → ' + str(r) for (q, a, r) in self.R)

def parseFSA(fsa: str) -> FiniteStateAutomaton:
    fsa = [line for line in fsa.split('\n') if line.strip() != '']
    q0 = fsa[0] # first line: initial
    F = set(fsa[1].split()) # second line: final, final, ...
    R = set()
    for line in fsa[2:]: # all subsequent lines: "source symbol → target"
        l, r = line.split('→')
        R |= {(l.split()[0], l.split()[1], r.split()[0])}
    T = {r[1] for r in R}
    Q = {q0} | F | {r[0] for r in R} | {r[2] for r in R}
    return FiniteStateAutomaton(T, Q, R, q0, F)

def minimizeFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    δ = {(q, a): r for (q, a, r) in fsa.R}
    dist = {(q, r) for q in fsa.Q for r in fsa.Q if q != r and (q in fsa.F) != (r in fsa.F)}
    done = False
    while not done:
        done = True #; print(dist)
        for q in fsa.Q:
            for r in fsa.Q:
                if q != r and (q, r) not in dist and any(((q, u) in δ) != ((r, u) in δ) or \
                    ((q, u) in δ) and ((δ[(q, u)], δ[(r, u)]) in dist) for u in fsa.T):
                    dist |= {(q, r)}; done = False #; print('adding', q, r)
    Q′ = {set({q} | {r for r in fsa.Q if (q, r) not in dist}) for q in fsa.Q}
    R′ = {(q′, u, r′) for q′ in Q′ for r′ in Q′ for u in fsa.T if any((q, u, r) in fsa.R for q in q′ for r in r
    q′0 = {q′ for q′ in Q′ if fsa.q0 in q′}.pop()
    F′ = {q′ for q′ in Q′ if (q′ & fsa.F) != set()}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)

def totalFSA(A: FiniteStateAutomaton, t = -1) -> FiniteStateAutomaton:
    T = set('abcdefghijklmnopqrstuvwxyz') # T is vocabulary, t is trap state
    R = A.R | {(q, a, t) for q in A.Q for a in T if all((q, a, r) not in A.R for r in A.Q)}
    if any(r == t for (q, a, r) in R): # transition to t exists
        Q = A.Q | {t}
        R = R | {(t, a, t) for a in T}
    else: Q = A.Q
    return FiniteStateAutomaton(T, Q, R, A.q0, A.F)

def renameFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    m, c = {}, 0
    for q in fsa.Q:
        m[q] = c; c = c + 1
    Q′ = {i for i in range(c)}
    R′ = {(m[q], u, m[r]) for (q, u, r) in fsa.R}
    q′0 = m[fsa.q0]
    F′ = {m[q] for q in fsa.F}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)

def equivalentFSA(a: FiniteStateAutomaton, a′: FiniteStateAutomaton, printMap = False) -> bool:
    a = minimizeFSA(totalFSA(a))
    a′ = minimizeFSA(totalFSA(a′))
    δ = {(q, u): r for (q, u, r) in a.R}
    δ′ = {(q, u): r for (q, u, r) in a′.R}
    m, v = {a.q0: a′.q0}, {a.q0}
    while v != set():
        if printMap: print(m)
        q = v.pop(); q′ = m[q]
        for u in a.T:
            if ((q, u) in δ) != ((q′, u) in δ′): return False
            elif (q, u) in δ: # (q′, u) in δ′
                r, r′ = δ[(q, u)], δ′[(q′, u)]
                if r in m:
                    if m[r] != r′: return False
                elif r′ in m.values(): return False
                else: v.add(r); m[r] = r′
    if printMap: print(m)
    return a′.F == {m[q] for q in a.F}
```

In [ ]:
```
class Choice:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
    def __repr__(self): return '(' + str(self.e1) + '|' + str(self.e2) + ')'

class Conc:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
    def __repr__(self): return '(' + str(self.e1) + str(self.e2) + ')'

class Star:
    def __init__(self, e): self.e = e
    def __repr__(self): return '(' + str(self.e) + ')*'
```

Here is the concrete grammar for regular expressions:

```
expression  →  term { '|' term }
term  →  factor { factor }
factor  →  atom [ '*' | '+' | '?' ]
atom  →  plainchar | escapedchar | '(' expression ')'
plainchar  →  ' ' | '!' | '"' | '#' | '$' | '%' | '&' | '\'' | ',' | '-' | '.' | '/' |
     '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':' | ';' | '<' | '=' | '>' |
     '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
'O' |
     'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '[' | ']' | '^' | '_' |
     '`' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
'o' |
     'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '{' | '}' | '~'
escapedchar  → '\' ( '(' | ')' | '*' | '+' | '?' | '\' | '|' )
```

Here is a parser that constructs the abstract syntax tree of regular expressions. The attribute grammar for this is as follows:

```
expression(e)  →  term(e) { '|' term(f) « e := Choice(e, f) »  }
term(e)  →  factor(e) { factor(f) « e := Conc(e, f) » }
factor(e) → atom(e) [ '*' « e := Star(e) » | '+' « e := Conc(e, Star(e)) » | '?' « e :=
Choice(e, '') » ]
atom(e)  →  plainchar(e) | escapedchar(e) | '(' expression(e) ')'
plainchar(e)  →  ' ' « e := ' ' » | ... | '~' « e := '~' »
escapedchar(e)  → '\\' ( '(' « e := '(' » | ')' | ... | '|' « e := '|' »)
```

The parser in Python is:

In [ ]:
```python
PlainChars = ' !"#$%&\',-./0123456789:;<=>@ABCDEFGHIJKLMNO' + \
             'PQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{}~'
EscapedChars = '()*+?\\|'
FirstFactor = PlainChars + '\\('

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos+1
    else: sym = chr(0) # end of input symbol

def expression():
    e = term()
    while sym == '|': nxt(); e = Choice(e, term())
    return e

def term():
    e = factor()
    while sym in FirstFactor: e = Conc(e, factor())
    return e

def factor():
    e = atom()
    if sym == '*': nxt(); e = Star(e)
    elif sym == '+': nxt(); e = Conc(e, Star(e))
    elif sym == '?': nxt(); e = Choice(e, '')
    return e

def atom():
    if sym in PlainChars: e = sym; nxt()
    elif sym == '\\':
        nxt()
        if sym in EscapedChars: e = sym; nxt()
        else: raise Exception("invalid escaped character at " + str(pos))
    elif sym == '(':
        nxt(); e = expression()
        if sym == ')': nxt()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))
    return e

def parse(s: str):
    global src, pos;
    src, pos = s, 0; nxt(); e = expression()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))
    return e
```

Here is more code from the course notes:

In [ ]:
```python
def REToFSA(re):
    global QC
    if re == '': q = QC; QC += 1; return FiniteStateAutomaton(set(), {q}, set(), q, {q})
    elif type(re) == str:
        q = QC; QC += 1; r = QC; QC += 1
        return FiniteStateAutomaton({re}, {q, r}, {(q, re, r)}, q, {r})
    elif type(re) == Choice:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R2 = {(A1.q0 if q == A2.q0 else q, a, r) for (q, a, r) in A2.R} # A2.q0 renamed to A1.q0 in A2.R
        F2 = {A1.q0 if q == A2.q0 else q for q in A2.F} # A2.q0 renamed to A1.q0 in A2.F
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, A1.R | R2, A1.q0, A1.F | F2)
    elif type(re) == Conc:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R = A1.R | {(f, a, r) for (q, a, r) in A2.R if q == A2.q0 for f in A1.F} | \
            {(q, a, r) for (q, a, r) in A2.R if q != A2.q0}
        F = (A2.F - {A2.q0}) | (A1.F if A2.q0 in A2.F else set())
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, R, A1.q0, F)
    elif type(re) == Star:
        A = REToFSA(re.e)
        R = A.R | {(f, a, r) for (q, a, r) in A.R if q == A.q0 for f in A.F}
        return FiniteStateAutomaton(A.T, A.Q, R, A.q0, {A.q0} | A.F)
    else: raise Exception('not a regular expression')
```

```
def convertRegExToFSA(re):
    global QC; QC = 0
    return REToFSA(re)
```

In [ ]:
```
def deterministicFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    q′0 = set({fsa.q0})
    Q′, R′, visited = {q′0}, set(), set()
    # print(Q′, R′, visited)
    while visited != Q′:
        q′ = (Q′ - visited).pop(); visited |= {q′}
        for t in fsa.T:
            r′ = {r for (q, u, r) in fsa.R if u == t and q in q′}
            if r′ != set(): Q′ |= {set(r′)}; R′ |= {(q′, t, set(r′))}
        # print(Q′, R′, visited)
    F′ = {q′ for q′ in Q′ for f in fsa.F if f in q′}
    return FiniteStateAutomaton(fsa.T, Q′, R′, q′0, F′)

def accepts(fsa: FiniteStateAutomaton, τ: str) -> bool:
    δ = {(q, a): r for (q, a, r) in fsa.R}
    q = fsa.q0
    for t in τ:
        if (q, t) in δ: q = δ[q, t]
        else: return False
    return q in fsa.F
```

In [ ]:
```
def equalRegEx(E1, E2):
    a1 = deterministicFSA(convertRegExToFSA(parse(E1)))
    a2 = deterministicFSA(convertRegExToFSA(parse(E2)))
    return equivalentFSA(a1, a2)
```

Let us extend regular expressions with squaring, for example:

- `a²` is `a` repeated twice, i.e. `aa`;
- `(ab)²` is `ab` repeated twice, i.e. `abab`
- `(a²)²` is `a²` repeated twice, i.e. `aaaa`.

We let `²` bind the same way as `*`, `+`, and `?`, for example:

```
ab² = a(b²)
```

Extend above grammar for regular expressions accordingly! Use the cell below. [1 point]

YOUR ANSWER HERE

Extend above attribute grammar above to construct an abstract syntax tree of regular expressions with squaring! The type of the nodes should not be extended. Use the cell below. [1 point]

YOUR ANSWER HERE

Extend above Python parser accordingly! Use the cell below. [2 points]

In [ ]:
```
# YOUR CODE HERE
raise NotImplementedError()
```

Here are some test cases: [3 points]

In [ ]:
```
assert str(parse('a²')) == '(aa)'
assert str(parse('(ab)²')) == '((ab)(ab))'
assert str(parse('(a²)²')) == '((aa)(aa))'
assert str(parse('ab²')) == '(a(bb))'

assert equalRegEx('a²', 'aa')
assert equalRegEx('(ab)²', 'abab')
assert equalRegEx('(a²)²', 'aaaa')
assert equalRegEx('ab²', 'a(b²)')
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js