Show that there are no formulas of length 2, 3, or 6, but that every other length is possible.

Let $p$, $q$, and $r$ be atomic propositions. Then $p$ is a formula of length 1, $(\neg p)$ is a formula of length 4, and $(p \land q)$ is a formula of length 5. We first prove by induction that some formula of length $n$ exists for all $n > 6$.

**Basis:** $(\neg(\neg p))$ is a formula of length 7, $(\neg(p \land q))$ is a formula of length 8 and $(p \land (q \land r))$ is a formula of length 9.

**Induction Hypothesis:** For all $6 < k \leq n$, there exists a formula of length $k$.

**Inductive Step:** Let $\varphi$ be a formula of length $n - 2$. Such a formula exists by the induction hypothesis. Then $(\neg \varphi)$ has length $n + 1$.

We next prove that formulas of length 2, 3 or 6 can not exist. All formulas have one of the following forms: $p$, $(\neg \varphi)$, $(\varphi_1 \circ \varphi_2)$ where $p$ is a proposition, $\varphi$, $\varphi_1$, $\varphi_2$ are formulas and $\circ$ is one of the binary connectives. A formula that has the first form has length exactly 1 and a formula that has the second or third form has length at least 4. Thus no formula of length 2 or length 3 exists. Lets assume $\theta$ is a formula of length 6. Then $\theta$ is a proposition and has either the second or the third form above. Assume $\theta = (\neg \varphi)$. Then $\varphi$ is a formula with length 3 which is not possible. Next assume $\theta = (\varphi_1 \circ \varphi_2)$. Then $\varphi_1$ and $\varphi_2$ have combined length 3. Thus one of them is a proposition and the other has length 2, which is not possible. Therefore $\theta$ cannot be a formula. and no formulas of length 6 exist.

We have shown that every length except 2, 3 or 6 is possible.

Show that between every left parentheses and right parentheses in a formula $\varphi$ there is at least one occurrence of a propositional connective.

We prove this by structural induction.
Basis: If \( \varphi \) is a proposition then it has no parentheses and the statement is trivially true.

**Inductive Step:** We have the following two cases.

**Case 1:** \( \varphi = (1 \neg \psi)_2 \). By induction, the proposition is true for all pairs of parentheses that lie within the formula \( \psi \). Between \((1\) and every right parentheses in \( \psi \), and likewise between \((1\) and \)2\), we have the primary connective \( \neg \). The only remaining case is when the left parentheses is in \( \psi \) and the right parentheses is \)2\). Then \( \psi \) cannot be an atomic proposition and so \( \psi \) must end with a closing right parentheses. By IH, there is a connective between any left parentheses in \( \psi \) and the closing right parenthesis of \( \psi \). This connective then also lies between any left parentheses in \( \psi \) and \)2\).

**Case 2:** \( \varphi = (1 \psi \circ \theta)_2 \). By induction, the proposition is true for all pairs of parentheses where both members of the pair lie within \( \psi \) or both lie within \( \theta \). For either \((1\) or a left parentheses in \( \psi \), and correspondingly \( )_2\) or a right parentheses in \( \theta \), the primary connective of \( \varphi \) fulfills the statement. Between \((1\) and a right parentheses in \( \psi \) (whereby \( \psi \) is not an atomic proposition, and hence starts with an opening parentheses), there is a primary connective since, by IH, there is one between every right parentheses in \( \psi \) and the opening left parentheses of \( \psi \). By the same argument, the statement also holds for every left parentheses in \( \psi \) and \)2\).

By structural induction, the proposition is true for every formula.

---

3. **Let** \( \varphi \) **be a formula, let** \( p \) **be the number of parentheses (left and right) in** \( \varphi \) **and let** \( l \) **be the length (number of symbols) of** \( \varphi \). **What are the minimum and maximum values of** \( \frac{p}{l} \). **The minimum value of** \( \frac{p}{l} \) **is 0, which is obtained when** \( \varphi \) **is an atomic proposition. In this case,** \( p = 0 \) **and** \( l = 1 \), **so** \( \frac{p}{l} = 0/1 = 0 \).

We show that \( \frac{p}{l} \) has no maximum, but its value does have a least upper bound of \( 2/3 \). We prove this in two parts: first we show, using induction, that for every formula \( \varphi \), \( \frac{p_\varphi}{l_\varphi} < \frac{2}{3} \), and then we show that we can create formulas that have a \( p/l \) value arbitrarily close to \( 2/3 \).

**Claim:** Every formula \( \varphi \) has \( \frac{p}{l} < \frac{2}{3} \).

**Proof by structural induction:**

**Base Case:** \( \varphi \) is an atomic proposition. Then \( \frac{p}{l} = 0 < \frac{2}{3} \).

**Inductive step:** Consider the two possible forms of \( \varphi \).

- **\( \varphi = (\neg \theta) \):** Then, by IH, we have \( \frac{p_\varphi}{l_\varphi} < \frac{2}{3} \). So,

\[
\frac{p_\varphi}{l_\varphi} = \frac{p_\theta + 2}{l_\theta + 3} < \frac{\frac{2}{3} \cdot l_\theta + 2}{l_\theta + 3} = \frac{2}{3}
\]
• \( \varphi = (\theta \circ \psi) \): Then, by IH, we have \( \frac{p_\theta}{l_\theta} < \frac{2}{3} \) and \( \frac{p_\psi}{l_\psi} < \frac{2}{3} \). So,

\[
\frac{p_\varphi}{l_\varphi} = \frac{p_\theta + p_\psi + 2}{l_\theta + l_\psi + 3} < \frac{2}{3} \cdot \frac{2}{3} + \frac{2}{3} \cdot \frac{2}{3} + 2 \cdot \frac{2}{3} = \frac{2}{3}
\]

Therefore, for every formula \( \varphi \), \( 0 \leq \frac{p}{l} < \frac{2}{3} \).

Now let \( q \) be an atomic proposition. We define \( \varphi_0 = q \), and \( \varphi_n+1 = (\neg \varphi_n) \), for all \( n \in \mathbb{N} \). Let \( p_n \) be the number of parentheses in \( \varphi_n \) and let \( l_n \) be the length of \( \varphi_n \). Then the sequence \( \frac{p_n}{l_n} = \frac{2}{n+1} \) converges to \( \frac{2}{3} \) as \( n \to \infty \).

Thus we can create formulas that have \( p/l \) ratios arbitrarily close to the upper bound of \( \frac{2}{3} \).

4 Let \( \varphi \) be a formula, let \( c \) be the number of occurrences of binary connectives (\( \land, \lor, \rightarrow, \text{and} \leftrightarrow \)) in \( \varphi \), and let \( s \) be the number of occurrences of atomic propositions in \( \varphi \). Show that \( s = c + 1 \).

Let \( \varphi \) be a formula. Then \( c(\varphi) \) denotes the number of occurrences of binary connectives in \( \varphi \) and \( s(\varphi) \) is the number of occurrences of atomic propositions in \( \varphi \). We show by structural induction that \( s(\varphi) = c(\varphi) + 1 \).

**Basis:** \( \varphi = p \) where \( p \) is a proposition. Then \( s(\varphi) = 1 \) and \( c(\varphi) = 0 \). So \( s(\varphi) = c(\varphi) + 1 \).

**Inductive Step:** We have two cases.

• \( \varphi = (\neg \varphi') \). Then \( c(\varphi) = c(\varphi') \), \( s(\varphi) = s(\varphi') \) and, by the inductive hypothesis, \( s(\varphi') = c(\varphi') + 1 \). Therefore \( s(\varphi) = c(\varphi) + 1 \).

• \( \varphi = (\varphi_1 \circ \varphi_2) \). Then \( c(\varphi) = c(\varphi_1) + c(\varphi_2) + 1, \ s(\varphi) = s(\varphi_1) + s(\varphi_2) \) and, by the induction hypothesis, \( s(\varphi_1) + s(\varphi_2) = c(\varphi_1) + c(\varphi_2) + 2 \). Therefore \( s(\varphi) = c(\varphi) + 1 \).

5 Give examples of formulas of \( \alpha \) and \( \beta \) and expression \( \gamma \) and \( \delta \) such that \( (\alpha \land \beta) = (\gamma \land \delta) \) but \( \alpha \neq \gamma \).

Let \( p, q, r \) be propositions. Let \( \alpha = (p \land (q \land r)) \) and \( \beta = ((p \land q) \land r) \). Then the expressions

\[
\gamma = (p \land (q \land r)) \land ((p \land q) \land r)
\]

satisfy the condition \( (\alpha \land \beta) = (\gamma \land \delta) \).

In general given any \( \alpha \) and \( \beta \), the number of different pairs of \( \gamma \) and \( \delta \) that satisfy the given condition is equal to the number of occurrences of \( \land \) in the expression \( \alpha \land \beta \).
Formulate and prove unique readability theorem for formulas in prefix notation.

We first prove a version of the prefix lemma for the prefix notation.

**Prefix Lemma:** No strict prefix of a formula in prefix notation is a formula itself.

While proving the original prefix lemma in class, we used the invariant that for a strict prefix of a formula, the number of left parentheses is strictly greater than the number of right parentheses. We need a similar syntactical invariant here. Since the prefix notation does have any parentheses, we look for a relation between propositions and connectives instead. The statement of Problem 4 gives us a hint. We first extend the result of Problem 4 to formulas in prefix notation.

That is we want to show that $s(\varphi) = c(\varphi) + 1$ where $\varphi$ is a formula in prefix notation, $s(\varphi)$ denotes the number of atomic propositions in $\varphi$ and $c(\varphi)$ denotes the number of binary connectives in $\varphi$. We give a brief proof by induction that closely parallels the proof in Solution 4 above: If $\varphi = p$ then $s(\varphi) = 1 = 0 + 1 = c(\varphi) + 1$. If $\varphi = \neg \varphi'$ then $s(\varphi) = s(\varphi') = c(\varphi') + 1 = c(\varphi) + 1$. If $\varphi = \bigcirc \vartheta \psi$ then $s(\varphi) = s(\theta) + s(\psi) = (c(\theta) + 1) + (c(\psi) + 1) = (c(\theta) + c(\psi) + 1) + 1 = c(\varphi) + 1$.

Now we can use this relation between $c(\varphi)$ and $s(\varphi)$ as the basis of our invariant. We claim that if $\alpha$ is a strict prefix of $\varphi$, then $s(\alpha) \leq c(\alpha)$. We prove this claim by induction. Note that the claim is always true for the empty prefix, so in the inductive step below, we only consider the non-empty cases.

**Basis:** If $\varphi = p$, then $\epsilon$ is the only strict prefix and the claim is trivially true.

**Inductive Step:** There are two possible cases.

1. $\varphi = \neg \varphi'$. Let $\neg \alpha$ be any non-empty strict prefix of $\varphi$. Then $\alpha$ is a (possibly empty) strict prefix of $\varphi'$ and so $s(\alpha) \leq c(\alpha)$ by the inductive hypothesis. Since $s(\neg \alpha) = s(\alpha)$ and $c(\neg \alpha) = c(\alpha)$, therefore the claim is true for $\neg \alpha$.

2. $\varphi = \bigcirc \vartheta \psi$. Let $\bigcirc \alpha$ be any non-empty strict prefix of $\varphi$. Then $\alpha$ is a (possibly empty) strict prefix of $\theta \psi$ and $s(\bigcirc \alpha) = s(\alpha)$ and $c(\bigcirc \alpha) = c(\alpha) + 1$. So we need to prove that $s(\alpha) \leq c(\alpha) + 1$. There are three sub-cases:

   • $\alpha$ is a strict prefix of $\theta$. Then by induction hypothesis, $s(\alpha) \leq c(\alpha)$. So the claim is true for this case.

   • $\alpha = \theta$. Then $s(\alpha) = c(\alpha) + 1$ and the claim holds.

   • $\alpha = \theta \beta$ where $\beta$ is a strict prefix of $\psi$. Then by the induction hypothesis, $s(\beta) \leq c(\beta)$ and because $\theta$ is a formula, $s(\theta) = c(\theta) + 1$. Now $s(\alpha) = s(\theta) + s(\beta) \leq c(\theta) + 1 + c(\beta) = c(\alpha) + 1$. Thus the claim holds for this case as well.

The prefix lemma follows directly from the above two results, since for a strict prefix $\beta$ of $\varphi$ to be a formula, it would have to satisfy both the conditions, $s(\beta) = c(\beta) + 1$ and $s(\beta) \leq c(\beta)$, which is impossible.
We now state and prove a unique readability theorem for prefix notation.

**Theorem:** Unique readability for formulas in prefix notation.

**Statement:** Every composite formula in prefix notation has a unique primary connective and immediate subformulas.

**Proof:** By definition, the first symbol in any composite formula is the primary connective. The first symbol is unique, so the primary connective is unique. Let \( \varphi \) be a composite formula. Then we have the following two cases.

**Case 1:** \( \varphi = \neg \varphi_1 \). Then \( \varphi \) cannot also be of the form \( \circ \theta \psi \) because these expressions have different first symbols. If \( \varphi = \neg \varphi_2 \) then from syntactical equality, we have \( \varphi_1 = \varphi_2 \). Thus unique readability for this case is quite trivial.

**Case 2:** \( \varphi = \circ \theta \psi \). Then \( \varphi \) cannot also be of the form \( \neg \varphi' \) because these expressions differ in their first symbols. Let \( \varphi = \circ \theta' \psi' \). Then \( \circ = \circ' \) and \( \theta \psi = \theta' \psi' \). If \( \theta \neq \theta' \) then either \( \theta' \) is a strict prefix of \( \theta \) or \( \theta \) is a strict prefix of \( \theta' \), both of which violate the prefix lemma. Thus \( \theta = \theta' \) and this in turn implies that \( \psi = \psi' \). Thus the immediate subformulas are unique.

This proves our version of the unique readability theorem for formulas in prefix notation.

Let \( \alpha \) and \( \beta \) be formulas and \( \beta \) be a subexpression of \( \alpha \). We need to show that \( \beta \) is a subformula of \( \alpha \). From here on, subexpression means proper subexpression. The proof will use structural induction on \( \alpha \). We denote the number of left (resp. right) parentheses in an expression \( \eta \) by \( l(\eta) \) (resp. \( r(\eta) \)). We also use the following results proved in class to simplify the proof.

- If \( \varphi \) is a formula, then \( l(\varphi) = r(\varphi) \).
- If \( \beta \) is a non-empty strict prefix of formula \( \varphi \) then it cannot itself be a formula and \( l(\beta) > r(\beta) \).

Now if \( \alpha = \beta \delta \) then \( \beta \) is a strict prefix of a formula and cannot itself be a formula. Also if \( \alpha = \delta \beta \), then \( \delta \) is a strict prefix of a formula and \( l(\delta) > r(\delta) \). Since \( l(\alpha) = r(\alpha) \), therefore \( l(\beta) < r(\beta) \) and again \( \beta \) cannot be a formula. Thus the only possible case is \( \alpha = \gamma \beta \delta \) where \( \gamma \) and \( \delta \) are both non-empty. The rest of the proof proceeds by induction. In what follows, \( \gamma \) and \( \delta \) are always non-empty.
Basis: $\alpha = p$ where $p$ is a proposition. Then $p$ is the only non-empty subexpression and it is a subformula of itself.

Inductive Step: There are two cases to consider.

Case 1: $\alpha = (\neg \varphi) = \gamma \beta \delta$ for non-empty $\gamma$ and $\delta$. This can be split into two subcases:

1. $(\neg \varphi) = (\beta \delta$. Then $\beta = \neg \gamma'$ which is not possible since $\beta$ is a formula and its first symbol can only be a proposition or $\epsilon$. Contradiction.
2. $(\neg \varphi) = (\neg \gamma' \beta \delta$ where $\gamma'$ can be possibly empty. Then $\beta$ is a subexpression of $\varphi$ and by the induction hypothesis and the definition of subformula, it is a subformula of $\varphi$ and in turn a subformula of $\alpha$.

Case 2: $\alpha = (\theta \circ \psi) = \gamma \beta \delta$ for non-empty $\gamma$ and $\delta$. Since $\beta$ is a formula, $\circ$ cannot be the first or last symbol of $\beta$. Thus the only possible cases are:

1. The first symbol of $\beta$ lies in $\theta$ and the last symbol of $\beta$ lies in $\psi$. That is $\beta = \mu \circ \nu$ where $\theta = \mu' \mu$, $\psi = \nu \nu'$ and $\mu \neq \epsilon$, $\nu \neq \epsilon$. Since $\mu$ is a strict prefix of $\beta$, it cannot be a formula. Thus $\mu$ is a strict suffix of $\theta$. Then $\mu'$ is a strict non-empty prefix of $\theta$ and thus $l(\mu') > r(\mu')$. This in turn implies $l(\mu) < r(\mu)$. But because $\mu$ is a strict prefix of $\beta$, if $\beta$ is a formula, then $l(\mu) > r(\mu)$. Thus we have a contradiction.
2. $\beta$ lies entirely within either $\theta$ or $\psi$. In this case, by the induction hypothesis, $\beta$ is a subformula of either $\theta$ or $\psi$. Thus by the definition of subformula, $\beta$ is a subformula of $\alpha$.

Describe an algorithm that checks whether a given expression is a formula. Analyze the time and space requirements of your algorithm. Implement it and run it over the expressions given.

Checking whether a given expression is a formula: we define a recursive routine “parse” that accepts an expression and parses the first immediate formula at the start of the expression, and returns the rest of the expression. If the expression does not start with a valid formula, it raises an exception, which is caught from the calling function “well-formed?”.

How “(parse expr)” works: it checks whether (car expr) is an atomic proposition; if so, it returns (cdr expr). Otherwise, if (car expr) happens to be the opening parentheses, then it goes a step ahead to see if the next token is a $\neg$, in which case it parses the following formula, and verifies that the string right after that starts with a closing parentheses. For the binary connective case, it recursively parses the stream of tokens immediately following the open
parentheses, and checks for the presence of some binary connective in whatever
remaining expression that the parse returns, and if one is found, parses the rest.

Whenever anything goes amiss, it raises an exception, so that further parsing
is halted and detailed error messages printed. When it does (car expr), it
ought to be checking for the case when expr is the empty list; instead it just
relies on generating a type exception, also caught outside to print error messages
to the effect of abrupt input termination.

Possible output of (parse formula):
  • #t
  • #f, abrupt expression end
  • #f, missing close paren
  • #f, no binary connective
  • #f, something wrong with expression
  • #f, trailing junk

Time requirement: It does one linear pass through the string and covers each
token exactly once, excepting the small number of lookahead required to test for
the atomic proposition and the “not” case, which are smaller than the number
of formulas within. One fact of crucial importance is that “parse” is always re-
cursively called with some cddd..drr of the original expression (effectively); this
is internally implemented by simply passing an offset into the input expression
list; hence we do not incur the cost of copying a linear-length hunk of memory
into every level of recursion. Hence time is linear in the length of the expression.

Space requirement: Because of its recursive nature, this program could,
in the worst case, store as many return addresses as the depth of the formula
tree. This implicitly includes the bit-per-depth that remembers whether the
subformula being parsed is the left or the right subformula of a given binary
formula. Also, the input argument (expression) to the “parse” routine would
be implemented internally by passing a constant-sized offset into the original
expression, to every level of recursion. Hence this algorithm would consume
space proportional to the largest depth, which in the worst case, could be linear
in the length of the expression.

Sample output of program

empty-----------------------------------------------
() : abrupt expression end#f
three-basic-cases-------------------------------------
(A) : #t
(< ~ A >) : #t
(< A & B >) : #t
abruptly-terminating-¬A-------------------------------
(\langle \rangle) : abrupt expression end
(\langle ~ \rangle) : abrupt expression end
(\langle ~ A \rangle) : abrupt expression end
(\langle ~ A >) : #t

abruptly-terminating-A&B-----------------------------------
(\langle \rangle) : abrupt expression end
(\langle A \rangle) : abrupt expression end
(\langle A & \rangle) : abrupt expression end
(\langle A & B \rangle) : abrupt expression end
(\langle A & B >) : #t
given-examples----------------------------------------
(\langle < ~ A ! B > & C >) : #t
(\langle A & B > ! C \rangle) : trailing junk
(A - < B & C >) : trailing junk
(\langle < A = B > - < ~ A > >) : #t
(\langle < ~ A > - B ! C >) : missing close paren
(\langle < < C ! B & A > = D > >) : missing close paren
(\langle < < ! A > & < ~ B > >) : something wrong with expression
(\langle A & B & C > >) : trailing junk

Code (in mzscheme)

;; notation: we're using
;; < for (,
;; > for )
;; ~ for \neg
;; ! for \vee
;; & for \wedge
;; - for \rightarrow
;; = for \leftrightarrow
;;-

(require-library "core.ss")
;; (define (display x) #t) ;; to turn off verbose error reporting

;; whether input symbol is a circle
(define (circle? token)
  (or (eq? token '!) (eq? token '&) (eq? token '-) (eq? token '=')))

;; whether input symbol is anything but parens and connectives
(define (atomic-prop? token)
  (not (or (eq? token '<)
(eq? token '>)
(eq? token ')
(circle? token)))

;; receive the final ' >' and return the rest-of-string
(define (parse> expr)
  (if (eq? (car expr) '>
  (cdr expr)
  (error "missing close paren")))

;; parses the leading formula in expression and returns the rest
;; errors generated: user errors (when we detect something going wrong)
;;               type errors (when we do (car empty) on abrupt end)
(define (parse expr)
  (let ((head (car expr)))
    (cond ((atomic-prop? head) (cdr expr)) ;; atomic prop
      ((eq? head '<)
      (if (eq? (cadr expr) '~)
      (parse> (parse (cddr expr))) ;; not
      (let ((tail (parse (cdr expr)))) ;; circle
      (if (circle? (car tail))
      (parse> (parse (cdr tail)))
      (error "no binary connective")))
      (else (error "something wrong with expression"))))))

;; calls parse, interpreting those generated errors as not-formula.
;; returns true if expr is a well-formed formula.
(define (well-formed? expr)
  (printf ~a : ~ expr)
  ;; check that its been well-formed so far:
  (with-handlers ((exn:user? (lambda (x) (display (exn-message x)) #f))
  (exn:application:type? (lambda (x) (display "abrupt expression end") #f)))
  (if (empty? (parse expr)) ;; check nothing left
  #t
  (begin (display "trailing junk") #f)))))

;; tests
'empty-----------------------------------------------
(well-formed? '())

'three-basic-cases-------------------------------------
(well-formed? '(A))
(well-formed? '(< ~ A >))
(well-formed? '(< A & B >>))

9
(well-formed? '(<) )
(well-formed? '(< ~ ) )
(well-formed? '(< ~ A ) )
(well-formed? '(< ~ A >) )

(well-formed? '(<) )
(well-formed? '(< A ) )
(well-formed? '(< A & ) )
(well-formed? '(< A & B ) )
(well-formed? '(< A & B >) )

(well-formed? '(< < ~ A ! B > & C >) )
(well-formed? '(< A & B > ! C ) )
(well-formed? '(A < < A ! B > > & C >) )
(well-formed? '(< < A = B > - < ~ A > ) )
(well-formed? '(< < ~ A > - B ! C > ) )
(well-formed? '(< < C ! B & A > = D > ) )
(well-formed? '(< < ! A > & < ~ B > ) )
(well-formed? '(< A & < B & C > > ) )