1 3F-1 Bookkeeping

- **0 pts** Correct

**Exercise 3F-2. Regular Expression, Large-Step [10 points].** Regular Expressions are commonly used as abstractions for string matching. Here is an abstract grammar for regular expressions:

| | | | |
|---|---|---|---|
| $e$ | ::= | "x" | singleton — matches the character $\hat{x}$ |
| | \| | empty | skip — matches the empty string |
| | \| | $e_1\ e_2$ | concatenation — matches $e_1$ followed by $e_2$ |
| | \| | $e_1 \mid e_2$ | or — matches $e_1$ or $e_2$ |
| | \| | $e*$ | Kleene star — matches 0 or more occurrence of $e$ |
| | \| | . | matches any single character |
| | \| | ["x" − "y"] | matches any character between $\hat{x}$ and $\hat{y}$ inclusive |
| | \| | $e+$ | matches 1 or more occurrences of $e$ |
| | \| | $e?$ | matches 0 or 1 occurrence of $e$ |

We will call the first five cases the *primary* forms of regular expressions. The last four cases can be defined in terms of the first five. We also give an abstract grammar for strings (modeled as lists of characters):

| | | | |
|---|---|---|---|
| $s$ | ::= | nil | empty string |
| | \| | "x" :: $s$ | string with first character $\hat{x}$ and other characters $s$ |

We write "bye" as shorthand for "b" :: "y" :: "e" :: nil. This exercise requires you to give large-step operational semantics rules of inference related to regular expressions matching strings. We introduce a judgment:

$$\vdash e \text{ matches } s \text{ leaving } s'$$

The interpretation of the judgment is that the regular expression $e$ matches some prefix of the string $s$, leaving the suffix $s'$ unmatched. If $s' = \text{nil}$ then $r$ matched $s$ exactly. Examples:

$$\vdash \text{"h"}(\text{"e"}+) \text{ matches "hello" leaving "llo"}$$

Note that this operational semantics may be considered *non-deterministic* because we expect to be able to derive all three of the following:

$$\begin{array}{l} \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "ello"} \\ \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "hello"} \\ \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "llo"} \end{array}$$

Here are two rules of inference:

$$\frac{s = \text{"x"} :: s'}{\vdash \text{"x" matches } s \text{ leaving } s'} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } s}$$

Give large-step operational semantics rules of inference for the other three primal regular expressions.

**Solution:**

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s' \quad \vdash e_2 \text{ matches } s' \text{ leaving } s''}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } s''} \qquad \frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'} \qquad \frac{}{\vdash e* \text{ matches } s \text{ leaving } s}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } s' \quad \vdash e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''}$$

## 2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

**Exercise 3F-3. Regular Expression and Sets [5 points].** We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x" matches } s \text{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } S'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and $e_1e_2$. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in one or two sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but "wrong" rules of inference and show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

**Solution:** I argue that the task is impossible because sequencing cannot be properly expressed in the given framework. That is, there cannot exist a rule which consists of multiple dependent hypotheses (which are necessary for concatentation and kleene star) because the output of one hypothesis (a set) does not "type match" the input of the other (a string.)

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad S' = \{s' | s' \in \bigcup_{\forall s_i \in S} \vdash e_2 \text{ matches } s_i \text{ leaving } S_i\}}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S'}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \quad S' = \{s' | s' \in \bigcup_{\forall s_i \in S} \vdash e* \text{ matches } s_i \text{ leaving } S_i\}}{\vdash e* \text{ matches } s \text{ leaving } S'}$$

Each of these inference rules are necessarily unsound since they depend on a variable number of hypotheses via the derivation appearing in the set constructors. I assert that this problem is unavoidable for dependent sequenced hypotheses since the cardinality and contents of $S$ are neither fixed nor finite, yet the derivation of the second hypothesis is wholly dependent on both the cardinality and content of $S$! A different framework in which a regular expression matches against a set of strings $s$ may correct this problem.

**Exercise 3F-4. Equivalence [7 points].** In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff $c$ halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge \ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecideable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

## 3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

**Exercise 3F-3. Regular Expression and Sets [5 points].** We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x" matches } s \text{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } S'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and $e_1e_2$. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in one or two sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but "wrong" rules of inference and show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

**Solution:** I argue that the task is impossible because sequencing cannot be properly expressed in the given framework. That is, there cannot exist a rule which consists of multiple dependent hypotheses (which are necessary for concatentation and kleene star) because the output of one hypothesis (a set) does not "type match" the input of the other (a string.)

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad S' = \{s' | s' \in \bigcup_{\forall s_i \in S} \vdash e_2 \text{ matches } s_i \text{ leaving } S_i\}}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S'}$$

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \quad S' = \{s' | s' \in \bigcup_{\forall s_i \in S} \vdash e* \text{ matches } s_i \text{ leaving } S_i\}}{\vdash e* \text{ matches } s \text{ leaving } S'}$$

Each of these inference rules are necessarily unsound since they depend on a variable number of hypotheses via the derivation appearing in the set constructors. I assert that this problem is unavoidable for dependent sequenced hypotheses since the cardinality and contents of $S$ are neither fixed nor finite, yet the derivation of the second hypothesis is wholly dependent on both the cardinality and content of $S$! A different framework in which a regular expression matches against a set of strings $s$ may correct this problem.

**Exercise 3F-4. Equivalence [7 points].** In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff $c$ halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge \ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecideable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

**Solution:**

The problem can be decided by converting the relevant regular expressions to finite automata (via Thompson's algorithm or the like), converting the directed graph representations of these automata to their deterministic and minimized forms $G_1, G_2$, and finally through an instance of the graph isomorphism problem on $G_1$ and $G_2$. I claim that the graphs are isomorphic if and only if $\forall s \in S.\ \vdash e_1$ matches $s$ leaving $S_1\ \wedge\ \vdash e_2$ matches $s$ leaving $S_2 \implies S_1 = S_2$. Observe that the set $S_i$ corresponds to the set of unconsumed symbols resulting from each unique path to an acceptance state in $G_i$ which implies $S_1 = S_2$ when $G_1$ and $G_2$ are isomorphic.

**Exercise 3C. SAT Solving.** Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

**Solution:** Submitted to Gradescope.

**Exercise 3F-5. SAT Solving [6 points].** Why do the last two included tests take such a comparatively long time? Impress me with your knowledge of DPLL(T) — feel free to use information from the assigned reading or related papers, not just from the lecture slides. I am looking for a reasonably detailed answer. Include a discussion of which single module you would rewrite first to improve performance, as well as how you would change that module.

Potential bonus point: The provided code contains at least one fairly egregious defect. Comment.

**Solution:** Both tests 35 and 36 consist of a conjunction of arithmetic expressions. Since the SAT solver converts each arithmetic expressions to a unique temporary literal, the final CNF formula for tests 35 and 36 are conjunctions of unit clauses. Finding satisfying assignments for such CNF formulas is trivial, so it is most likely that the arithmetic contraint solver is responsible for the observed slowdown. Examining `arith.ml`, it is clear that the solver takes an extraordinarily naive approach to constraint satisfaction. The heart of the algorithm, `arith.ml:62-72`, iterates over each arithmetic variable considering every possible combination of numeric values from -127 to 128. This implies an asymptotic runtime complexity of $\mathcal{O}(256^k)$ for $k$ unique variables. Most egregiously, the constraint solver even considers 256 values for variables $x$ belonging to equality expressions, i.e. those of the form $x = l$ for a number literal $l$. Significant runtime savings could be realized if only the arithmetic constraint solver did an inital pass to add equality expressions directly to the model. As a proof-of-concept for this claim, I have made a small modification to `arith.ml` that adds the assignment $z = 10$ to the model initially and removes $z$ from the list of variables that need assignments. The modification involved strategic placement of the following lines:

- `let model = StringMap.add "z" 10 model in`
- `let variables = StringSet.remove "z" variables in`

The results (given the starter version of `arith.ml` first):

```
$ time cat tests/test-35.input | ./solver
```

## 4 3F-4 Equivalence

- **0 pts** Correct

**Solution:**

The problem can be decided by converting the relevant regular expressions to finite automata (via Thompson's algorithm or the like), converting the directed graph representations of these automata to their deterministic and minimized forms $G_1, G_2$, and finally through an instance of the graph isomorphism problem on $G_1$ and $G_2$. I claim that the graphs are isomorphic if and only if $\forall s \in S.\ \vdash e_1$ matches $s$ leaving $S_1\ \wedge\ \vdash e_2$ matches $s$ leaving $S_2 \implies S_1 = S_2$. Observe that the set $S_i$ corresponds to the set of unconsumed symbols resulting from each unique path to an acceptance state in $G_i$ which implies $S_1 = S_2$ when $G_1$ and $G_2$ are isomorphic.

**Exercise 3C. SAT Solving.** Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

**Solution:** Submitted to Gradescope.

**Exercise 3F-5. SAT Solving [6 points].** Why do the last two included tests take such a comparatively long time? Impress me with your knowledge of DPLL(T) — feel free to use information from the assigned reading or related papers, not just from the lecture slides. I am looking for a reasonably detailed answer. Include a discussion of which single module you would rewrite first to improve performance, as well as how you would change that module.

Potential bonus point: The provided code contains at least one fairly egregious defect. Comment.

**Solution:** Both tests 35 and 36 consist of a conjunction of arithmetic expressions. Since the SAT solver converts each arithmetic expressions to a unique temporary literal, the final CNF formula for tests 35 and 36 are conjunctions of unit clauses. Finding satisfying assignments for such CNF formulas is trivial, so it is most likely that the arithmetic contraint solver is responsible for the observed slowdown. Examining `arith.ml`, it is clear that the solver takes an extraordinarily naive approach to constraint satisfaction. The heart of the algorithm, `arith.ml:62-72`, iterates over each arithmetic variable considering every possible combination of numeric values from -127 to 128. This implies an asymptotic runtime complexity of $\mathcal{O}(256^k)$ for $k$ unique variables. Most egregiously, the constraint solver even considers 256 values for variables $x$ belonging to equality expressions, i.e. those of the form $x = l$ for a number literal $l$. Significant runtime savings could be realized if only the arithmetic constraint solver did an inital pass to add equality expressions directly to the model. As a proof-of-concept for this claim, I have made a small modification to `arith.ml` that adds the assignment $z = 10$ to the model initially and removes $z$ from the list of variables that need assignments. The modification involved strategic placement of the following lines:

- `let model = StringMap.add "z" 10 model in`
- `let variables = StringSet.remove "z" variables in`

The results (given the starter version of `arith.ml` first):

```
$ time cat tests/test-35.input | ./solver
```

```
Unsatisfiable!

real    0m5.029s
user    0m4.933s
sys     0m0.027s
$ vim arith.ml  # changes made here
$ make clean && make all
...
$ time cat tests/test-35.input | ./solver

Unsatisfiable!

real    0m0.140s
user    0m0.051s
sys     0m0.023s
```

A 3592.14% real time gain. Of course a general algorithm for this optimization would be much more complex than my case-specific modification, but still far from difficult to implement.

**Submission.** Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.

5 3F-5 SAT Solving

- **0 pts** Correct