# ToC: Homework 5 [150 points]

Due: Saturday, November 13th, by midnight

## Reading

Please read Lectures 26, 27, and 28 of Kozen.

## How to submit

Please put your solution in a folder whose name is your Hawkid. If you worked with a partner please include a file called `partner.txt` that lists the Hawkids for you and your partner. Only one partner should submit their assignment. You will both receive the same grade.

## How to get help

Please post questions to ICON Discussions, if possible (as opposed to emailing us directly), so that our answers can benefit others who likely have similar questions. You may also come to our Zoom office hours, listed on ICON under Pages - Office Hours.

## 1 Cocke-Kasami-Younger Algorithm [50 points]

For this problem, please use the definition of the CYK algorithm that I gave in class Oct. 18th. The definition in Kozen is very similar, but not exactly the same.

The following grammar in Chomsky normal form accepts the same language as the grammar you worked with in Part 2 of hw4, for accepting strings like `(xx(xx))`:

$$
\begin{aligned}
A &\rightarrow \texttt{(} && (1)\\
B &\rightarrow \texttt{)} && (2)\\
E &\rightarrow \texttt{x} && (3)\\
E &\rightarrow A\ F && (4)\\
F &\rightarrow L\ B && (5)\\
L &\rightarrow L\ E && (6)\\
L &\rightarrow E\ E && (7)
\end{aligned}
$$

The start symbol is `E`, and the terminals are `(`, `)`, and `x`. The grammar from Part 2 of hw4 is almost in Chomsky normal form. The change here is to use productions (1), (2), (4), and (5) to implement the single production (from hw4) $E \rightarrow$ `(` $L$ `)`.

We are going to see the execution of the CYK algorithm on `(xxx)`. The following problems ask you to draw tables and fill in entries a step at a time. The tables are not huge, so please just redraw them separately for each step.

1. Since `(xxx)` has 5 characters, we need a table with entries for each $(i, l)$ pair, where $i \in \{1, 2, 3, 4, 5\}$ and $i+(l-1) \leq n$ with $l > 0$. The entry for $(i, l)$ shows the different nonterminals that could generate the substring of length $l$ of the original input string, starting at index 1. (So $(2, 3)$ is for nonterminals that could generate $xxx$, which is the substring starting at position 2 and having length 3.)

   Draw an empty version of this table, where the entry for the pair $(i, l)$ is at row $i$, column $l$. You can draw a triangular table, or else just draw the full $n \times n$ table (where you will then leave about half the entries unfilled) [5 points]

2. Fill in the entries $(i, l)$ of the table where $l = 1$. [10 points]

3. Fill in the rest of the table. [25 points]

## 2 Shift-Reduce Parsing [30 points]

The file `Grammar.info` contains the parsing automaton generated by `happy` for a simple unambiguous grammar for balanced parentheses. The grammar is printed at the top of the file. Most of the rest of the file describing a shift-reduce parsing automaton, which determines whether to shift a symbol from the remaining part of the input string over to the stack, or else to reduce the top of the stack. The determination is made based on the next character of the input string, to initiate a shift or reduce action; or else by the element on the top of the stack, following a reduce action. See the drawing for the week of Oct. 18th for an example (that actually uses the same grammar!).

Show the process of shift-reduce parsing step by step for the string `()(())`. Just as I did in lecture, please write out the stack and the remaining input string, and show the effects, one at a time, of each of the actions that the automaton says to perform. (This is not covered in the book, so please ask in class or office hours if you have trouble figuring out what to do based on my explanation in lecture.)

You can put your answer in a file called `ShiftReduce.txt` (or an image file with similar name if you prefer).

[30 points]

## 3 Understanding TMs in Haskell [25 points]

In `TMExamples.hs` you will find a version of the TM we saw in class, recognizing $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. This is called `tripletm`. It has been simplified following the suggestion of Junnan in class. A transition diagram for this TM is shown in Figure 1, where I am writing $g/g'$ on edges to mean "if $g$ is in the cell where the readhead is currently positioned, write $g'$ to that cell and move the

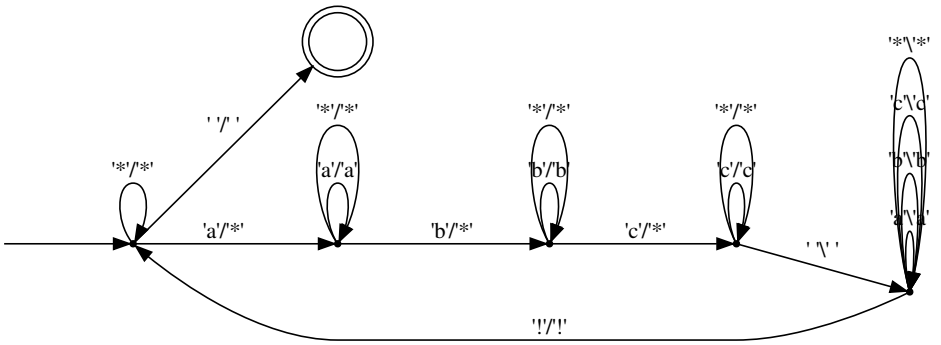


Figure 1: TM accepting $\{a^n b^n c^n \mid n \in \mathbb{N}\}$

readhead one cell to the right". Similarly, with a backward slash means to move the readhead to the left.

Add a definition of a TM called `quadtm` to `TMExamples.hs`. This new TM should be similar to `tripletm`, except that it recognizes $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$. Using the function `writeGraphViz` in `TM.hs`, generate a transition diagram for your TM and call it `quadtm.pdf`. [25 points]

# 4 Writing TMs in Haskell [45 points]

Please add your solutions to the following problems to `TMExamples.hs`.

1. Define a TM called `eqtm`, with input alphabet $\{a, b, c\}$, which recognizes the language $\{w\, b\, w \mid w \in \{a, c\}^*\}$. In other words, `eqtm` is supposed to check that the input has a string (call it $x$) over $\{a, c\}$, then a $b$, and then $x$ again. You will find something similar in Example 29.1 of Kozen, though this problem is easier because you can look for the letter $b$ to find the middle of the string. [30 points]

2. This is a challenge problem, worth just 15 points so you are not penalized too badly if you cannot solve it.

   Define a TM called `bevaltm`, with input alphabet $\{t, f, n, c, \_\}$, that will implement a simple evaluator for boolean expressions, as follows. The idea is that $t$ will mean true, $f$ false, $n$ negation, and $c$ conjunction ("and"); and _ is a space (you can use an actual space character if you want; I am writing an underscore here just for typographical reasons). This _ should also

be the blank symbol for the TM. (The TM does not need to make use of a left endmarker, so you can just use _ for that as well.) Negation and conjunction will be applied postfix. So $t\,f\,c$ means to conjoin true and false; so this would simplify (intuitively) to $f$. (The $c$, for conjunction, is written after its arguments.)

In more detail, your `bevaltm` should recognize the language

$$\{x\,q \mid x \in \{t, f, n, c\}^* \ \wedge \ q \in \{t, f\} \ \wedge \ x \rightsquigarrow^* q\}$$

where $\rightsquigarrow$ is a string-rewriting relation, defined just below, that tries to evaluate the boolean expression. So for example, since $t$ means true and $n$ means negation, we want to reduce $t\,n$ to $f$, which means false. The only small hiccup with this is that we will need to write a space on our tape when we want to transform two symbols, like $t\,n$ to just one. So we will actually turn $t\,n$ into $f\,\_$. The fact that spaces could be sprinkled all through our string then makes the transformation a bit more complicated. We can describe it this way, writing $\_^k$ to mean the string consisting of $k$ spaces:

$$\begin{array}{rcl} t\,\_^k\,n & \rightsquigarrow & f\,\_^{k+1} \\ f\,\_^k\,n & \rightsquigarrow & t\,\_^{k+1} \end{array}$$

$$\begin{array}{rcl} f\,\_^k\,t\,\_^j\,c & \rightsquigarrow & f\,\_^{k+j+2} \\ t\,\_^k\,f\,\_^j\,c & \rightsquigarrow & f\,\_^{k+j+2} \\ f\,\_^k\,f\,\_^j\,c & \rightsquigarrow & f\,\_^{k+j+2} \\ t\,\_^k\,t\,\_^j\,c & \rightsquigarrow & t\,\_^{k+j+2} \end{array}$$

It should actually be much less annoying to implement this as a TM, because you can easily write a loop (as it would appear in the transition diagram format) to skip over spaces, while trying to see if a particular rewrite above is possible. So you won't need to count numbers of spaces (where the above rules do this just to formalize the idea that we have some number of spaces which then grows).

Here is an idea on how to implement this. You can have a state from which you branch out nondeterministically in 6 different directions, one for each of the rewrite rules above. The very first symbol of the input string must always be either $t$ or $f$, even after rewriting. If not, you can reject. To implement a rule, you need to check that all the symbols required occur, possibly with spaces between them. When you reach the last such symbol – which must be either $n$ or $c$ – you can erase that (i.e., change it to _), and then work your way back to the first symbol. In the case of conjunction, you have to erase the middle $t$ or $f$ you saw. Then you can set the value of the first $t$ or $f$ to either $t$ or $f$ as required by that particular rule.

You can nondeterministically try to go from a $t$ past spaces to another $t$ and then a space; or from an $f$ past spaces to another $f$ and then a space. If either of those is possible (nondeterministically), your TM can enter an accept state.

For example, your machine should accept this example string

$$tfncnf$$

since the string $tfncn$ represents $\neg\ (true\ \wedge\ \neg false)$ in a postfix notation. If you proceed as I suggest, you will turn this input string into

$$f\_\_\_f$$

(four spaces) which will then be accepted.

Good luck if you try this one!