JFLAP Activities for Formal Languages and Automata

Peter Linz   Susan H. Rodger

UC Davis   Duke University

February 1, 2011

Contents

Chapter 1

Getting to Know JFLAP

This JFLAP material is intended for students taking a first course in formal languages and automata theory and are using JFLAP in their work. It includes material that will help you to get started with JFLAP, gives hints how to use it, and suggests exercises that show the power and convenience of JFLAP.

1.1   About JFLAP

JFLAP is an interactive visualization and teaching tool for students in a course on formal languages and automata. JFLAP was created by Professor Susan Rodger and her students at Duke University. It is written in Java and runs on most platforms.

JFLAP can be helpful to you in several ways:

- It lets you visualize difficult concepts and makes abstract notions concrete.

- It allows you to follow complicated examples and constructions step by step, thus making clearer what is involved.

- It can save you a great deal of time and remove the tedium of many constructions. Some exercises that could take hours with pencil and paper can be done with JFLAP in a few minutes.

- JFLAP makes extensive testing of examples easy. This helps detect errors and increases the accuracy of your work.

The notation used and the algorithms implemented by JFLAP are closely related to the reference text "An Introduction to Formal Languages and Automata, Fourth Edition" by Peter Linz, Jones and Bartlett, 2006 (subsequently referred to simply as the text). Explanations and motivation for what goes on in JFLAP can be found in that reference and you should work in coordination

Figure 1.1: JFLAP Menu [Courtesy of Sun Microsystems].

with the text. JFLAP can be used with other texts, but adjustments have to be made for possible differences in notation and approach.

You are encouraged to use JFLAP as much as possible. Rather than diluting the rigorous nature of this course, it will increase your appreciation of the material. Once you learn a few basic rules, you will enjoy working with JFLAP.

1.2   How to Use JFLAP

JFLAP can be downloaded without charge from `www.jflap.org`. You need to have Java SE1.4 or a later version installed on your computer to run JFLAP.

When you run JFLAP, the first thing that will appear is the main menu shown in Figure 1.1 that shows all the major options available in JFLAP.

When you select one of the options, a window will be opened. All your work, such as creating structures, modifying them, testing examples, etc., will be done in this window. You can have several windows open at any given time. The appearance of each window and the options in it depend on what type of problem you have selected. Much of JFLAP is intuitive and self-explanatory, but a brief description of each type of window is given in the relevant section of Chapter 2 below. For details, consult the manual *JFLAP An Interactive Formal Languages Package* by Susan H. Rodger and Thomas W. Finley, Jones and Bartlett, 2006. The "Help" option of each window also

provides some guidance.

The main menu includes some options that may be unfamiliar to you, for example, "Mealy Machine". This is an indication that JFLAP has a scope wider than just formal languages and automata.

Chapter 2

Working With JFLAP

In this chapter we present specific information on the various parts of JFLAP, using exercises to illustrate its features. A few of the exercises are simple, designed to make you familiar with the workings of JFLAP. Many others, though, are more complicated and may involve combining several ideas. Such exercises are instructive, but involve a lot of work. When done in the traditional pencil-and-paper way, these exercises can be tedious and time-consuming. With JFLAP, however, solving lengthy problems can become quite simple. The exercises in this chapter are correlated with the chapters in the text as shown in parentheses in each chapter title. Of course, many of the exercises in the text can also benefit from JFLAP, so use JFLAP whenever you see fit.

2.1   A First Look at Grammars (Chapter 1)

Throughout this course you will be asked to construct various objects, such as grammars, finite automata, Turing machines, and so on. To get some confidence in the correctness of your constructions, you need to test the answers with many specific cases. This can be very time-consuming, but JFLAP is a great help. Not only can you test single cases quickly, but often JFLAP allows you to run several examples in a single pass.

To work with a grammar, click on "Grammar" in the main menu. This will open a window in which you can enter the productions of the grammar. JFLAP follows the convention of the text: capital letters stand for variables and lower case letters stand for terminals. The variable on the left of the first production is taken to be the start variable.

Once you have defined a grammar, you can see if a given string is generated by this grammar (the membership question). This can be done in several ways. One way is to click on "CYK Parse". This will let you enter a string and, after you click on "Start", tells you whether or not the string is in the language generated by the grammar. You can do several strings in a single pass by clicking on "Multiple CYK Parse". For each string, JFLAP will tell you if it is in the language of the grammar (accept), or not in the language (reject). The same thing can be accomplished by

selecting "Brute Force Parse". Brute force parsing (also called exhaustive search parsing in the text) is more general than the CYK parser, but can be extremely inefficient. It may fail for simple cases where the CYK parser succeeds easily. For the exercises in this section the CYK parser is adequate and recommended.

Exercises

1. Examine the strings $b, bb, abba, babb, baa, abb$ to determine if they are in the language defined by the grammar in Example 1.12 of the text. (Note: most examples have JFLAP files provided, in this case load the file *Jexample1.12*.)

2. Determine which of the strings $aabb, abab, abba, babab$ can be generated by the grammar

$$S \rightarrow aSb|SS|ab$$

Can you make a conjecture about the language defined by this grammar?

3. The language in Example 1.13 of the text contains the empty string. Modify the given grammar so that it generates $L - \{\lambda\}$.

4. Modify the grammar in Example 1.13 of the text so that it will generate $L \cup \{a^n b^{n+1} : n \geq 0\}$

5. Test the conjecture that the language $L = \{(ab)^k a^{2l+1} b^k a^{2n+1} : k, l, n \geq 1\}$ is generated by the grammar

$$
\begin{aligned}
S & \rightarrow abAbB \\
A & \rightarrow abAb|B \\
B & \rightarrow aaB|a
\end{aligned}
$$

6. Let G be the grammar

$$
\begin{aligned}
S & \rightarrow aAb \\
A & \rightarrow aSb|ab|\lambda
\end{aligned}
$$

Carry out tests to support or contradict the conjecture that

   (a) $L(G) \subseteq \{a^{2n} b^{2n} : n \geq 1\}$,
   (b) $L(G) = \{a^{2n} b^{2n} : n \geq 1\}$.

7. Do you think that the grammar $S \rightarrow aSb|ab|abab|aSbaSb|SSS$ is equivalent to the grammar in Exercise 2 above?

2.2   Finite Automata (Chapter 2)

Chapter 2 of the text deals with finite automata as tools for defining regular languages. In that chapter we look at the formal difference between deterministic and nondeterministic finite automata as well as their essential equivalence. We also examine the state minimization issue. All theoretical results are based on algorithmic constructions. JFLAP implements the constructions described in the text.

In the exercises in this section we explore the construction and reconstruction of finite automata, how to create automata for the reverse and complementation of regular languages, and how to combine automata for the union and concatenation of languages. Note that some of the constructions require that you start with a complete dfa, while others work just as well with an nfa.

In the window for finite automata, there are four tools, represented by the icons at the top of the window. The <u>state creator</u> is used to make new states. By dragging the mouse between states, the <u>transition creator</u> lets you connect states and label transitions. You can use the <u>deleter</u> to remove states and transitions. The <u>attribute editor</u> has several functions: to select initial and final states, to change state names, and to create or change state labels. It can also be used to move states within the window.

Once you have created an automaton, you can check your construction by running different test strings. Click on the "Input" tab to start. There are several options available in this mode. You can run a single case, or you can run multiple cases simultaneously. Processing a single case can be done in a step-by-step manner. In "Step by State" each step takes you from one state to another. "Step with Closure" essentially represents δ^*.

In the "Test" mode you will find a very useful tool, "Compare Equivalences", which allows you to check if two finite automata are equivalent. To use this tool you must have at least two finite automata windows open. There are two additional tools that are occasionally helpful: "Highlight Nondeterminism" and "Highlight λ -transitions".

Under "Convert" are several more useful tools. "Convert to DFA" and "Minimize DFA" implement the constructions in the text. You can also use "Combine Automata". If you have two open automata, this feature will put one of them in the window of the other. This is helpful for constructing the union and concatenation of two regular languages.

JFLAP also helps you to visualize the difficult concept of nondeterminism. When you process an nfa, JFLAP keeps track of all possible configuration and displays them in the window. Since the number of such configurations can become quite large, JFLAP lets you disable some of them temporarily (freeze), or eliminate them altogether (remove). This way you can follow one path, eliminating those you know to be unprofitable. Later, if necessary, you can return to some of the suspended options by "thawing"" them.

Exercises

1. Let L be the language in Example 2.5 of the text. Starting with the dfa in Figure 2.6, build dfa's for

   (a) L^*

   (b) $\overline{L}$

   (c) $L \cup L^3 \cup L^5...$

2. Consider the nfa in Jexercise2.2. Make a conjecture about the language accepted by this nfa. Then prove your conjecture.

3. If M is a dfa for a language L, it may appear that a dfa for L^R can be constructed by simply reversing the arrows in the transition graph of M. There are, however, some additional adjustments to be made. Explore this issue in connection with the dfa in Jexercise2.3

4. Consider the following proposition: "If M is the minimal dfa that accepts a language L, and M' is the minimal dfa for L^R, then M and M' have the same number of states".

   (a) Show that this proposition is false by finding a counterexample. Hint: At least one of the dfa's in this exercise section is suitable.

   (b) Find a simple condition on M for which the proposition is true.

5. Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$, where M_1 and M_2 are the finite automata in Jexercise2.5a and Jexercise2.5b, respectively.

   (a) Find a finite automaton for $L_1 \cup \overline{L_2}$. Give a simple description of this language.

   (b) Use the results of part (a) to find a finite automaton for $\overline{L_1} \cap L_2$.

6. The dfa M in Jexercise2.6 accepts the string $w = a$. Modify M to create a new automaton M', so that $L(M') = L(M) - \{a\}$.

7. Let M be the finite automaton in Jexercise2.7. Find the minimal dfa for $L(M) \cup \{a^{2n}b^k : n \geq 2, k \geq 2\}$.

8. The nondeterministc action of an nfa on an input w can be visualized by means on a <u>configuration tree</u>. In a configuration tree, each node represents (and is labeled with) a state. The root is labeled with the initial state. The children of a node are the successor states for that node, with edges labeled with the input symbol that causes the transition. So, if $\delta(q_0, a) = \{q_0, q_1\}$ and the input string starts with a, the root will have two children labeled q_0 and q_1, connected by edges both labeled a.

The finite automaton in Jexercise2.8 is nondeterministic. Find the configuration tree for input $w = abba$. Use the "Freeze", "Thaw", and "Remove" features to help you with the bookkeeping.

9. For the nfa in Jexercise2.8 use JFLAP to compute $\delta^*(aabbaabb)$. Can you find a string w for which $\delta^*(w) = \emptyset$?

2.3   Connections: Finite Automata, Regular Expressions, and Grammars (Chapters 2 and 3)

The theme of the exercises in this section is the connection between various ways of defining regular languages. The constructions in the text for converting from one representation to another are fully implemented by JFLAP, with some occasional differences that will be discussed below. Using JFLAP can give you insight into the constructions; it can also save a great deal of time in some exercises.

Note that the difficulty of some problems depends on the approach. For example, there is no obvious way of telling if two regular grammars are equivalent. But if you first convert the grammars into finite automata, you can use JFLAP to test for equivalence.

To convert a right-linear grammar to an nfa, select the "Convert" option. You can perform each individual step by inserting a transition into the partial transition graph provided by JFLAP. Alternatively, you can use "Create Selected"; here JFLAP will fill in the transition corresponding to the selected production. Finally, if you select "Show All", JFLAP will give you the complete answer without showing the intermediate steps. You may notice that here JFLAP uses string labels for transitions (See Section 3.1). When the conversion is complete, you can "export" the result. This will put the created nfa in a new window.

To convert an nfa to a right-linear grammar, start with the transition graph of the nfa in its window and select "Convert" and "Convert to Grammar". The grammar can be built step by step by clicking on a state or on an edge. Alternatively, the process can be done in one step by selecting "Show All". When the grammar is exported, the order of the productions may be changed.

To convert an nfa into a regular expression you must start with an automaton that satisfies the conditions required by the procedure nfa-to-rex in the text. The transition graph for this nfa is first converted into a complete generalized transition graph. With the "State Collapser" tool you can then successively remove states until you get a two-state graph. From this, the regular expression is obtained as described in the text. All of this can be done with individual steps or in one single application.

Finally, to convert a regular expression into an nfa, define the regular expression in a "Regular Expression" window, then select "Convert to nfa". Use the generalized transition graph produced

by JFLAP, and click on "Do Step" or "Do All" to convert the generalized transition graph into a normal transition graph. The nfa so produced is usually unnecessarily complicated and should be converted into a minimized dfa before any further work is done with it.

Exercises

1. Show that the finite automaton in Jexercise3.1a accepts the language generated by the grammar in Jexercise3.1b.

2. Find the minimal dfa for $L(((aa + b)^* + aba)^*)$.

3. Find a regular expression for the language generated by the grammar

$$
\begin{aligned}
S &\rightarrow abA \\
A &\rightarrow bA|bB \\
B &\rightarrow bB|a
\end{aligned}
$$

4. Show that the grammars in Jexercise3.4a and Jexercise3.4b are not equivalent. Find a string that can be generated by the first grammar, but not the second.

5. Find a left-linear grammar equivalent to the right-linear grammar in Jexercise3.4a. Simplify your answer as far as possible.

6. Find a regular grammar for the complement of $L(a^*b(a + b)b^*)$.

7. Find a left-linear grammar for the reverse of the language generated by the grammar in Exercise 3 above.

8. Let $L = \{w \in \{a, b\}^* : w \text{ does not contain the substring } aba\}$

   (a) Find a regular grammar for L.
   (b) Find a regular expression for L.

9. Construct a regular expression r so that $L(r)$ is the complement of $L(a(a + bb)^*)$.

2.4   Properties of Regular Languages (Chapter 4)

The exercises at the end of Section 4.1 of the text involve a number of constructions relating to closure results. Many of these exercises are difficult and call for inventiveness and careful testing. Exercises 1 and 2 below are perhaps a little simpler and should be tried first. After you become familiar with the pattern of the approach, you can try some of the more difficult exercises in Section 4.1. Because of the complexity of some of the exercises, mnemonic labeling is highly recommended. Here the JFLAP ability to change state names and labels comes in handy.

To help you understand the ideas behind the pumping lemma, JFLAP has implemented the pumping lemma game described in Section 4.3 of the text. Playing this game can illuminate the logical complexity of the pumping lemma and illustrate the choices that have to be made in carrying out a correct proof. For practical reasons, the pumping lemma game is limited to a few pre-selected games. For regular languages, play the game by clicking on the "Regular Pumping Lemma" option in the main menu.

Exercises

1. The operation *changeleft* replaces the leftmost symbol of any string by all other possible symbols, that is, for all $a \in \Sigma$

$$changeleft(aw) = \{bw : b \in \Sigma - \{a\}\}.$$

Extending this to languages, we define

$$changeleft(L) = \bigcup_{w \in L} changeleft(w).$$

   (a) Let L be the language accepted by the finite automaton on Jexercise4.1. Construct a finite automaton for $changeleft(L)$. Assume that $\Sigma = \{a, b, c\}$.

   (b) Generalize the construction above so that it can be used to prove that the family of regular languages is closed under the *changeleft* operation.

2. Define *even* (w) as the string obtained from w by extracting the symbols at the even numbered positions, that is $even(a_1 a_2 a_3 a_4...) = a_2 a_4...$ This operation can be extended to language by

$$even(L) = \{even(w) : w \in L\}.$$

   (a) Design a general method by which a finite automaton for L can be converted into a finite automaton for $even(L)$. Hint: Replicate the states of the automaton to keep track if you are on an odd or even symbol.

   (b) Apply your method to find a dfa for $even(L(abba^*ba))$.

3. Determine if the language accepted by the finite automaton in Jexercise4.3 contains any string of odd length.

4. Consider the two automata M_1 and M_2 in Jexercise4.4a and Jexercise4.4b, respectively.

   (a) Show that $L(M_1) \subseteq L(M_2)$.

   (b) Show that the inclusion is a proper one.

Does your construction indicate a way to resolve the subset issue in general?

5. Let M_1 and M_2 be the finite automata in Jexercise4.5a and Jexercise4.5b, respectively. Find an nfa for $L(M_1) \cap L(M_2)$, using

   (a) the construction in Theorem 4.1 of the text,

   (b) the identity $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

6. One can draw conclusions from the pumping lemma game only if both sides use a perfect strategy. While every effort has been made to program the computer to play perfectly, in the absence of a formal proof this cannot be guaranteed. Therefore you need to analyze every game you play carefully before drawing any conclusions.

   Play as many games as you can, both positive and negative instances (see Section 3.2 below for more information on how to play). Play each game as prover and as spoiler. For each game write a synopsis that answers the following questions.

   (a) What choices did the computer make and why you think they were made? Are the choices optimal?

   (b) What choices did you make? Why did you make these choices? Are your choices optimal?

   (c) What conclusions can you draw from the outcome of the game?

2.5   Context-Free Grammars (Chapters 5 and 6)

Questions about context-free grammars are generally much harder than the corresponding questions for finite automata and regular languages. There is, for example, no guaranteed way of showing that two context-free grammars are equivalent. When faced with such issues, we have to look at them carefully. For example, suppose that G_1 and G_2 are two context-free grammars. If we can find a string w that is in $L(G_1)$ but not in $L(G_2)$, then clearly the two grammars are not equivalent. On the other hand, if all the strings we test are in both languages, we may suspect that the grammars are equivalent, but this is not a proof. The best we can do is to form a strong presumption of equivalence and this requires extensive testing.

To determine membership of a string in a language defined by a context-free grammar, you can use the CYK algorithm, or the brute force parser, as discussed in Section 2.1. Either one will determine membership, and show you a parse tree or a left-most derivation.

A difficulty also arises in connection with ambiguity. While there is no general algorithm by which one can show that a grammar is unambiguous, a single instance is sufficient to demonstrate that a grammar is ambiguous. With JFLAP's "User Control Parse" feature, you can choose the

order of application of the productions, and in this way perhaps find two different parse trees. This may require a lot of trying, but it is really the only way we can demonstrate ambiguity.

While the general form of a context-free grammar allows a great deal of freedom, restrictions sometimes have to be imposed to avoid unnecessary complications. The major simplification techniques in Chapter 6 of the text are implemented in JFLAP. They are applied in the order described in the text. Start by clicking on "Convert". After several steps, you will end up with a grammar in Chomsky normal form.

Exercises

1. Create context-free grammars for the languages

$$L_1 = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, |w_1| \neq |w_2|\}$$

and

$$L_1 = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, |w_1| = |w_2|, w_1^R \neq w_2\}.$$

2. For $\Sigma = \{a, b, c\}$, use the results of Exercise 1 to construct a grammar for

$$L_1 = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1^R \neq w_2\}.$$

   Test your answers to Exercises 1 and 2 thoroughly.

3. The grammar in Jexercise5.3 was intended to generate the language $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w)\}$, but is not quite correct.

   (a) Find a string in L that cannot be generated by the grammar.

   (b) Correct the grammar.

4. Find a context-free grammar for the language

$$L = \{w \in \{a, b\}^* : n_a(w) = 3n_b(w)\}.$$

   Give reasons why you think that your answer is correct.

5. Brute-force parsing works in principle, but quickly becomes inefficient as the string to be parsed increases in length. Consider the grammar in Jexercise5.5 and use the brute force parser on strings of the form $(ab)^n$, with increasing values of n to see what happens. Compare your observations with those you get with the CYK parser.

6. Show that the grammar $S \rightarrow aSbb|SS|SSS|ab$ is ambiguous.

7. The grammar in Jexercise5.7 claims to accept $L = \{w \in \{a, b\}^* : n_a(w) > 2n_b(w)\}$.

    (a) Is this claim correct? If so, give your reasons for accepting the claim. If not, correct the grammar.

    (b) Use the idea in part(a) to construct a grammar for $L = \{w \in \{a, b\}^* : n_a(w) < 2n_b(w)\}$.

    (c) Combine the two grammars to get a grammar for $L = \{w \in \{a, b\}^* : n_a(w) \neq 2n_b(w)\}$.

8. Transform the grammar in Jexercise5.8 in a sequence of steps:

    (a) Remove λ-productions.

    (b) Remove unit-productions.

    (c) Remove useless productions.

    (d) Transform the grammar into Chomsky normal form.

    Select a set of test examples to check that the grammars are equivalent at each step.

9. <u>Prove</u> that the grammars in Jexercise5.9a and Jexercise5.9b are equivalent.

10. A normal form for linear grammars is defined in Exercise 8, Section 6.2 of the text. Convert the grammar below to this normal form.

$$
\begin{aligned}
S &\rightarrow aSbb \\
S &\rightarrow aAab \\
A &\rightarrow aAb \\
A &\rightarrow bb
\end{aligned}
$$

    Check the two grammars for equivalence.

11. Grammars that have productions of the form $A \rightarrow Aw$ are said to be <u>left-recursive</u>. Exercise 23 in Section 6.1 of the text suggests a way of removing left-recursive productions from the grammar. Apply the suggested method to the grammar

$$
\begin{aligned}
S &\rightarrow SaA|b \\
A &\rightarrow aAb|a
\end{aligned}
$$

    Run a sufficient number of test cases to convince yourself that the two grammars are equivalent and that the method outlined in the text does indeed work.

12. The method suggested in the text for removing λ productions works only if the language does not contain the empty string. But, as was suggested in Exercise 14, Section 6.1 of the text, if we carry out the process anyway, we will get a grammar for $L - \{\lambda\}$. JFLAP allows you to use the process even if the empty string is in the language. Try some examples to see if the claim in the text is plausible.

2.6   Pushdown Automata (Chapter 7)

Constructing pushdown automata can be challenging and occasionally requires a bit of ingenuity. Unlike finite automata, we do not have any way of checking pda's for equivalence, nor do we know any algorithms for state minimization. Consequently, extensive testing is required to gain confidence in the correctness of a solution. JFLAP's multiple run feature greatly simplifies this task.

The algorithm for converting a context-free grammar to a pda is done in JFLAP a little different from the text's method. The JFLAP algorithm is suggested by a comment on pages 189-190 of the text. Read that comment before you use the JFLAP construction.

JFLAP uses the text's method for converting a pda to a context-free grammar.Unfortunately, this method is hard to follow and it is not easy to convince oneself that it works properly. In general, examples have to be small even when JFLAP is used. But JFLAP can simplify the results so they become more transparent. First, when you export the grammar, JFLAP changes from the cumbersome notation of the text to the more conventional form in which variables are represented by capital letters. Next, you can use JFLAP to remove useless productions. This often collapses the grammar significantly so that the final result may be fairly transparent.

As we know, there is a fundamental difference between deterministic and nondeterministic context-free languages. While a solution by an npda does not imply that the language itself is nondeterministic, we have no way of determining the nature of the language. Still, is some cases conversion from an npda to a dpda is possible because of the special nature of the underlying language.

Exercises

1. Let L be the language accepted by the pda in Jexercise6.1.

   (a) Give a simple description of L.

   (b) Modify the pda so that it accepts L^2.

   (c) Modify the pda so that it accepts L^*.

   (d) Modify the pda so that it accepts $\overline{L}$.

2. Construct a pda that accepts $L = \{w \in \{a, b, c\}^* : n_a(w) = n_b(w) + 2n_c(w)\}$.

3. With $\Sigma = \{a, b, c\}$

   (a) Construct a pda for $w_1 c w_2$ with $w_1, w_2 \in \{a, b\}^*$ and $|w_1| \neq |w_2|$.

   (b) Construct a pda for $w_1 c w_2$ with $w_1, w_2 \in \{a, b\}^*$, $w_1 \neq w_2$ and $|w_1| = |w_2|\}$.

   (c) Combine the pda's in parts (a) and (b) to construct a pda for $w_1 c w_2$ with $w_1, w_2 \in \{a, b\}^*$, $w_1 \neq w_2\}$.

4. Let M be the pda in Jexercise6.4.

    (a) Find all the states from which there are nondeterministic transitions.

    (b) Analyze M to find an intuitive description of $L(M)$.

    (c) While M is nondeterministic, the language it accepts is not. Show that $L(M))$ is a deterministic context-free language by constructing a dpda for it.

5. Design a dpda that accepts the language $L = \{(bb)^k a^n b^n : n \geq 0, 1 \leq k \leq 4\}$. If your solution has more than six states, try to reduce the number of states.

6. What language does the pda in Jexercise6.6 accept? Find an equivalent pda with fewer states.

7. The pda in Jexercise6.7 was intended to accept the language

$$L = \{w \in \{a, b\}^* : |n_a(w) - n_b(w)| \bmod 3 \ = 1\}.$$

The answer is incorrect, but can be fixed by changing the label of one transition. Find the error and correct the solution.

8. Convert the context-free grammar $S \rightarrow aaSb|aSS|ab$ to a pda, using the method described in the text. Then use JFLAP to do the conversion. Test the two pda's for equivalence.

9. Convert the pda in Jexercise6.9 to a context-free grammar, then simplify the grammar as far as you can.

10. Test the pda in Jexercise6.9 and the grammar in Exercise 9 above for equivalence.

11. Example 7.8 in the text shows how to convert a pda to a context-free grammar. As you can see, the grammar delivered is extremely complicated. Even after removing some of the more obvious useless productions, the grammar is still larger than necessary.

    Enter the grammar on page 193 of the text, changing variable names to conform to the JFLAP convention. Then simplify and test the original pda and the grammar for equivalence.

2.7 Properties of Context-Free Grammars (Chapter 8)

There are only a few results on closure and decision problems for context-free grammars. These are explored in the exercises in this section. The answers are not always easy to get.

The pumping lemma for context-free grammars is more complicated than the pumping lemma for regular languages, so your analysis of each game will have to be more thorough.

Exercises

1. Let M_1 be the pda in Jexercise7.1a and let M_2 be the finite automaton in Jexercise7.1b. Find a pda for $L(M_1) \cap L(M_2)$, using the construction of Theorem 8.5.

2. Consider the following grammar G.

$$
\begin{aligned}
S &\rightarrow aA|BCD \\
A &\rightarrow aAB \\
B &\rightarrow aC|C \\
C &\rightarrow \lambda \\
D &\rightarrow Ea|EC \\
E &\rightarrow AC|BAD
\end{aligned}
$$

Use all available features of JFLAP to determine if

(a) $\lambda \in L(G)$,

(b) $L(G)$ is finite.

3. While theory tells us that the family of context-free grammars is not closed under complementation, this does not imply that the complement of every context-free grammar is not context-free.

(a) Show that the complement of the language $L = \{a^n b^n : n \geq 1\}$ is context-free by constructing a pda for it.

(b) Show that such a pda cannot be constructed by just taking the deterministic pda for L (Jexercise7.3) and complementing the state set. Explain why this will not work.

4. Find a linear grammar equivalent to the grammar

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aAb|ab \\
B &\rightarrow aaB|a
\end{aligned}
$$

Test your result to illustrate the equivalence (see also Exercise 13 in Section 8.2 of the text).

5. Find a linear grammar that is equivalent to

$$
\begin{aligned}
S &\rightarrow ABC \\
A &\rightarrow aA|a \\
B &\rightarrow aBb|ab \\
C &\rightarrow abC|ab
\end{aligned}
$$

6. It is known that the family of deterministic context-free languages is closed under complementation. In principle therefore, it should be possible to convert a dpda for a context-free language into a dpda for $\overline{L}$. In some cases, for instance when the dpda has no λ-transitions, the conversion is relatively straightforward. Explore this issue by converting the dpda in Jexercise7.6 into a dpda for $\overline{L(M)}$. Assume $\Sigma = \{a, b\}$.

7. Play the pumping lemma game for context-free language, with both positive and negative cases, and as prover as well as spoiler. For each game write a synopsis, analyzing your strategy and that of the computer. Discuss what conclusions you can draw from the outcome of each game.

2.8 Programming Turing Machines (Chapter 9)

As was pointed in the text, the tediousness of Turing machine programming can be alleviated by modularizing the task, selecting a suitable set of macros to implement before combining them into the final program. This approach will make constructions easier to do and the results are more readily checked for correctness.

There are a number of macros that are useful in many different circumstances. Some of these are collected by JFLAP in a building block library (See Section 4.3). Study the contents of this library and use the provided routines whenever suitable in the exercises here.

Exercises

In these exercises the term Turing machine stands for a one-tape Turing machine as implemented by JFLAP (For a discussion, see Section 3.1)

1. Use the building block "interchange" to write a Turing machine program that moves the leftmost symbol of a string to its right end, that is, performs the computation

$$q_0 a w \overset{*}{\vdash} q_1 w a$$

for $w \in \{a, b\}^+$.

2. The monus function is defined for positive integers by

$$
\begin{aligned}
x \overset{.}{-} y &= x - y, \quad x > y, \\
&= 0, \quad x \le y.
\end{aligned}
$$

Using unary notation in which x is represented by $w(x) \in \{1\}^*$, with $|w(x)| = x$, construct a Turing machine for the monus function. Specifically the machine should carry out the computation

$$q_0 w(x) \# w(y) \overset{*}{\vdash} q_1 w(x \overset{.}{-} y).$$

3. Construct a Turing machine for the multiplication of two positive integers in unary notation.

4. Construct a Turing machine for the addition of two positive integers in binary notation.

5. Construct a Turing machine for the function $f(n) = 2^n$, with n a positive number in unary notation.

6. Construct a Turing machine for the function $f(n) = n!$, with n a positive number in unary notation.

7. For x and y in unary notation, construct a Turing machine comparer defined by

$$q_0 w(x) \# w(y) \quad \overset{*}{\vdash} \quad q_1 w(x) \# w(y), \quad x \geq y,$$
$$\overset{*}{\vdash} \quad q_2 w(x) \# w(y), \quad x < y,$$

8. With the usual unary notation, write a Turing machine program for

$$div(x, y) = floor(x/y)$$

and

$$rem(x, y) = x - y * div(x, y).$$

9. Construct a Turing machine that converts positive integers in binary to unary notation.

10. Construct a Turing machine that converts positive integers in unary to binary notation.

11. Let x, y, z be positive integers in unary notation. Construct a Turing machine that ends in state q_1 if $x + y = z$ and in state q_2 if this is not so. In both cases, the tape should be all blanks at the end of the computation.

12. Build a Turing machine that computes

$$f(x, y) = max(x, y),$$

where

(a) x and y are positive integers in unary notation,

(b) x and y are positive integers in binary notation.

2.9 Multitape Turing Machines (Chapter 10)

Many problems are more easily solved if we have several tapes available. JFLAP allows up to five tapes, although one rarely needs more than three. The exercises in this section illustrate this claim.

Exercises

1. Write a two-tape Turing machine program that accepts a string w if and only if it is of the form $w = vv, v \in \{a, b\}^+$.

2. Construct a three-tape Turing machine that accepts the language $L = \{a^n b^n c^{n+1} : n \geq 3\}$.

3. The three-tape Turing machine in Jexercise9.3 was intended to perform addition of positive integers in base three. The program is incomplete in the sense that it works only for a restricted input. Analyze the program to find the error and correct it.

4. Write a Turing machine program that accepts a string $w \in \{1\}^+$ if and only if the value of w in unary notation is prime. Use as many tapes as convenient.

5. Construct a Turing machine for the following computation. Given a string $w \in \{a, b\}^+$ on tape 1, write a value m in unary on tape 2, where $m = |n_a(w) - n_b(w)|$.

6. Construct a two-tape Turing machine that accepts $w \in \{a, b\}^+$ if and only if $w \notin \{a^n b^{n+1} : n \geq 0\}$.

2.10 Unrestricted and Context-Sensitive Grammars (Chapter 11)

It is easy to make mistakes when working with unrestricted grammars and hard to check results or spot errors. Brute-Force parsing works on unrestricted grammars, but its inefficiency limits it to fairly simple grammars and short input strings. One way to debug a context-sensitive grammar is to use "User Control Parsing". With this you can derive various strings and check the idea you used in constructing the grammar.

Derivations with unrestricted grammars cannot be represented as parse trees, but can be visualized as a tree-like structure in which several nodes can be grouped together to have common children. For example, if with the grammar $S \to ABC, AB \to ab$, we apply the first rule, followed by the second rule, we get the partial parse graph in Figure 2.1.
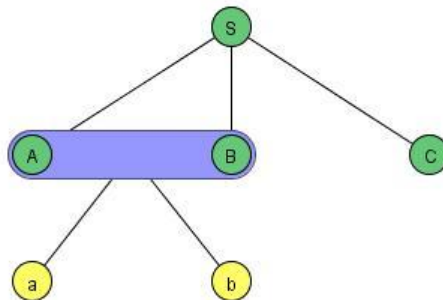
Figure 2.1: Partial Parse Tree for Unrestricted Grammar.

Exercises

1. Find two strings that can be generated by the unrestricted grammar

$$
\begin{aligned}
S &\rightarrow AaB \\
aB &\rightarrow BaC \\
aC &\rightarrow CaC \\
C &\rightarrow a \\
ABbba &\rightarrow AaB
\end{aligned}
$$

   Also, find two strings that are not in the language generated by this grammar.

2. Modify the grammar in JexampleP11.2 so that it accepts $L = \{a^n b^n c^m : m \geq n + 2\}$.

3. Find a context-sensitive grammar for $L = \{a^n b^m c^n : m > n\}$.

4. Analyze the grammar in Jexercise10.4 to make a conjecture about the language generated by this grammar. Test your conjecture.

5. Use the grammar in Jexercise10.4 as a starting point to develop a grammar for $L = \{www : w \in \{a, b\}^+\}$.

6. Use the grammar in Jexercise10.4 as a starting point to develop a grammar for $L = \{ww^R w^R : w \in \{a, b\}^+\}$.

Chapter 3

Special Features of JFLAP

3.1 Differences Between JFLAP and the Text

While JFLAP follows the text to a large extent, there are some differences that you need to be aware of. The differences involve mainly extensions of the definitions of the text. While these extensions are generally not significant theoretically, they can lead to some disagreement between what is stated in the text and what you observe with JFLAP. Use the JFLAP extensions when convenient, but exercise some caution in interpreting the results.

Definition and transition graphs for dfa's. In the text, the transition function for a dfa is assumed to be a complete function. In the transition graph this means that normally there is a non-accepting trap state. Exercise 21, Section 2.2 of the text introduces the idea of an incomplete transition graph in which the non-accepting trap state is implied, but not explicitly shown. JFLAP accepts an incomplete transition graph as a dfa. But keep in mind that some constructions, such as the suggested way of getting a transition graph for the complement of a regular language, do not work properly unless the implied trap state is explicitly shown.

State names and labeling. When creating an automaton, JFLAP uses a generic naming convention, with $q_0, q_1, ...$ as state names. These state names can be changed using the attribute editor tool. Furthermore, you can create additional labels that can be used for more extensive annotation.

Transition labels in finite automata. JFLAP allows you to label the edges of a transition graph with a string, rather than just a single symbol. This can be seen as a shorthand notation, so that a transition from q_i to q_j labeled ab stands for a transition labeled a from q_i to some hidden state q_k, followed by a transition labeled b from q_k to q_j. This option is known to lead to some minor obscure inconsistencies, so use it cautiously.

Regular grammars. JFLAP recognizes only right-linear grammars as regular grammars for proofs. For example, JFLAP will allow you to use the "regular grammar convert to dfa" only if the
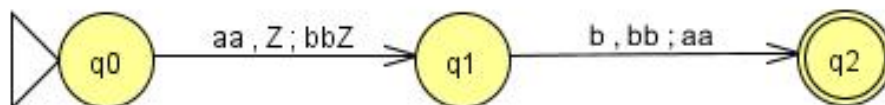
Figure 3.1: JFLAP NPDA.

grammar is right-linear. You can still write and test left-linear grammars, but JFLAP recognizes them as context-free for proofs.

Conversion from a regular expression to an nfa. The text starts with the individual parts of the regular expression and combines them into larger and larger nfa's. JFLAP starts with a generalized transition graph for the entire regular expression, then successively breaks it down into smaller pieces. But the end results are identical, so the two approaches are really the same method executed in different order.

Conversion from a context-free grammar to a pushdown automaton. The conversion procedure in JFLAP does not require the grammar to be in Greibach normal form, but uses the process suggested in the paragraph following Example 7.7 of the text. JFLAP can actually produce two slightly different pda versions, in preparation for certain parsing methods important in compiler construction. Since these issues are not discussed in the text, either version is acceptable for our purposes.

Definition of a pushdown automaton. In the text definition, one step of a pushdown automaton consumes one input symbol and pops off one stack symbol. In JFLAP, each step can consume a string and can pop several stack symbol. The pda in Figure 3.1 accepts *aab*, leaving aaZ on the stack.

Note that JFLAP uses Z as the stack start symbol.

Formally, the transition function of a pda, as implemented by JFLAP is defined by

$$\delta : Q \times \Sigma^* \times \Gamma^* \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

in contrast to Definition 7.1 of the text.

Turing machine definition. The definition of a standard Turing machine in the text allows only two kinds of moves: one cell to the right or one cell to the left. JFLAP also allows the stay option, in which the read-write head remains on the current cell. This is useful in many cases.

JFLAP also uses special symbols as shorthand for several transition labels

- $\sim$ in the input position means: make the transition whatever the current cell symbol is,

- $\sim$ in the output position means: do not change the contents of the current cell,

- $!a$ in the input position means: make the transition as long as the current cell symbol is not an a,

- $!a$ is not permitted in the output position.

JFLAP has one feature that cannot be viewed as shorthand notation for a standard Turing machine, a feature that brings the JFLAP Turing machine a step closer to a simple programming language. JFLAP can assign values to variables! Normally, a transition edge can be labeled with a single input symbol only, but there is a special restricted syntax for variable assignment. In this syntax several input symbols, separated by commas are allowed. These are followed by the special character } and a variable name. The transition is allowed if the current cell content matches one of the input symbols; this matching value is then assigned to the named variable. For example, if a transition is labeled $a, b\}w$, the transition is taken if the current cell contains either a or b. The cell value is assigned to the variable w, available for later use.

3.2 The Pumping Lemma Game

The pumping lemma game, described in Section 4.3 of the text, is a way of visualizing the abstractions in the pumping lemma statements and of exploring the choices that can be made in the proofs of these theorems. The game is played by two players, the <u>prover</u>, whose objective is to get a contradiction of a pumping lemma, and the <u>spoiler</u>, who wants to stop the prover from achieving this goal. At each step of the game, a player has to make a decision that can affect the outcome of the game. Valid conclusions from the pumping lemma game can be drawn only if both players use the best strategy.

In the text it is assumed that you are the prover because the objective there is for you to show that a given language is not in a certain family, say, that it is not regular. If you can win the game, you have shown what you set out to do.

In JFLAP you can choose to play either the prover (the computer goes first), or the spoiler, where you take the first step by choosing m. As a spoiler, you want to make your choices so that the computer cannot win. If, in spite of your best efforts, the computer wins, you can conclude that the pumping lemma is violated.

JFLAP implements the two major pumping lemmas: for regular languages and for context-free languages. The languages available can be grouped into two categories: positive and negative. In the positive cases the prover can always win, so that the language is not in the relevant family. In the negative cases, the language is in the relevant family and the pumping lemma cannot be violated. This means that the spoiler can win with best strategy. But remember, the spoiler can lose if he plays poorly.

Because the logical difficulty in the pumping lemmas is reflected in the pumping lemma game, a complete analysis of all the moves has to be made before any conclusions can be drawn from the outcome of a game.

3.3   Turing Machine Building Blocks

A Turing machine building block (TMBB) is a Turing machine that implements a certain function and is designed as a part of a larger Turing machine. In essence, a TMBB represents a macroinstruction of the type discussed in Section 9.2 of the text. Using TMBB's can greatly simplify the construction of complicated Turing machines.

Creating a TMBB. A building block has the same structure as a normal Turing machine. To construct a TMBB just follow the usual rules for constructing a Turing machine. Store the completed Turing machine in a file. When you want to use a building block, select the "Building Block" tool from the "Edit" menu. Click on the window in which you are constructing a new machine; this will open a file list from which you can select the appropriate block. When you select a TMBB, a rectangular box, labeled with the name of the block, will be placed in the current window. The interior structure of the TMBB will not be shown, but the effect is as if you had placed the Turing machines represented by the block there.

Connecting a TMBB to the machine under construction The TMBB can be connected either to a state of your machine or to another TMBB in several ways.

- Connecting to the TMBB with a normal transition. This will go from the outside state or from another TMBB to the initial state of the TMBB.

- Connecting from the TMBB to an outside state with a normal transition. The connection is

from the final state of the TMBB to the outside state. If the TMBB has several final states, each of them will be connected to the outside state in the same way.

- <u>Transfer to and from a TMBB with a Building Block Transition</u>. By using the "Building Block Connector" tool, you can create a connection with a single symbol. The transition will be taken if the current cell content matches the transition label. No change in the tape contents or the position of the read-write head is made.

Using a TMBB. Once a TMBB has been inserted into a Turing machine, it acts as if it were in place of its representing rectangle. When you process a specific string, the transitions in the TMBB will be executed one step at a time. The TMBB is highlighted, but its internal structure is not shown. What you do see is the tape content and the current internal state of the TMBB.

It is possible to treat the TMBB as a single unit by using the "Step by Building Block" feature. In this mode, all computations in the block appear as one step.

Editing a TMBB. When you create a TMBB in your program, the connection between the building block in your machine and the file in which it is stored is severed. Any change in the original file will not be reflected in your block, nor will any modification that you make in the block affect the stored file.

To make changes in the TMBB from your window, select the attribute editor and click on the block you wish to change. Then select "Edit Block". This will open the block by showing its internal structure in a new window. You can then make changes in the usual way. When you close this window, the changes will be in effect in the block.

In summary, when you insert a TMBB in the machine you are constructing, it acts as if you had inserted the TMBB's program and connected it to the outside states. There is one difference: the scope of the state names is local to the TMBB. So you can have, for example, a state named q_0 in the outside program, and a similarly named state in the block.

Chapter 4

Support Library

This manuscript also provides a library of JFLAP programs designed to help you in studying the material and solving the exercises.

4.1   Text Examples

The structures in many text examples have been implemented in JFLAP. With these JFLAP programs you can follow the text examples step by step, testing them with various strings. This will complement and perhaps clarify the discussions in the text. We have established a uniform naming convention in which *Jexample x.y* refers to the structure in Example *x.y* of the text.

4.2   JFLAP Exercises

In Chapter 2 of this manuscript are exercises that illustrate the use of JFLAP and its power in solving problems. Many of these exercises start with a given structure that you are asked to analyze or modify. The naming convention is that *Jexercisex.y* refers to Exercise *y* in Section *x* of Chapter 2 of this manuscript.

4.3   Turing Machine Building Blocks

The building blocks in this library are all for single-tape machines with input alphabet $\Sigma = \{0, 1, a, b, c, d, \#\}$. In the instantaneous descriptions used below to define the actions of the TMBB's, the following convention is used.

- u, v, w are in Σ^*.

- α and β stand for single symbols from $\Sigma \cup \{\square\}$.

- q_0 marks the read-write head position when the block is entered.

- q_f marks the read-write head position when the block is finished.

Here are TMBB's that are currently supplied for your use.

compare. Compares two fields for equality and writes a 0 or 1 on the tape.

$$uq_0\#v\#w \quad \overset{*}{\vdash} \quad uq_f1\#v\#w \text{ if } v = w$$
$$\overset{*}{\vdash} \quad uq_f0\#v\#w \text{ if } v \neq w$$

copy-field. Makes a copy of part of the tape.

$$q_0\#u\#v \overset{*}{\vdash} q_f\#u\#v\#u$$

   or

$$q_0\#u \overset{*}{\vdash} q_f\#u\#u$$

delete-cell. Deletes a single cell.

$$uq_0\alpha v \overset{*}{\vdash} uq_f v$$

erase-field. Deletes part of the tape.

$$uq_0\#v\#w \overset{*}{\vdash} uq_f\#w$$

insert blank. Inserts a new cell on the tape and writes a blank in it.

$$uq_0v \overset{*}{\vdash} uq_f\square v$$

interchange. Interchanges the contents of two adjacent cells.

$$uq_0\alpha\beta v \overset{*}{\vdash} uq_f\beta\alpha v$$

left-to-blank. Moves the tape head left and stops on the first encountered blank.

left-to α. Moves the tape head left and stops on the first encountered α, where $\alpha \in \Sigma$. If there is no α found, the machine will not terminate.

left-to-α-or-blank. Moves the tape head left and stops on the first encountered α or blank.

right-to-blank. Moves the tape head right and stops on the first encountered blank.

right-to α. Moves the tape head right and stops on the first encountered α, where $\alpha \in \Sigma$. If there is no α found, the machine will not terminate.

right-to-α-or-blank. Moves the tape head right and stops on the first encountered α or blank.