HW3 : due 9 Nov, Wed

Solve the 7 problems circled and numbered I, II, ..., VII.

On the other hand, the language $L = \{a^n b^n \mid n \geq 1\}$ with an equal number of a's and b's is a prototypical example of a language that can be described by a grammar but not by a regular expression. To see why, suppose L were the language defined by some regular expression. We could construct a DFA D with a finite number of states, say k, to accept L. Since D has only k states, for an input beginning with more than k a's, D must enter some state twice, say s_i, as in Fig. 4.6. Suppose that the path from s_i back to itself is labeled with a sequence a^{j-i}. Since $a^i b^i$ is in the language, there must be a path labeled b^i from s_i to an accepting state f. But, then there is also a path from the initial state s_0 through s_i to f labeled $a^j b^i$, as shown in Fig. 4.6. Thus, D also accepts $a^j b^i$, which is not in the language, contradicting the assumption that L is the language accepted by D.

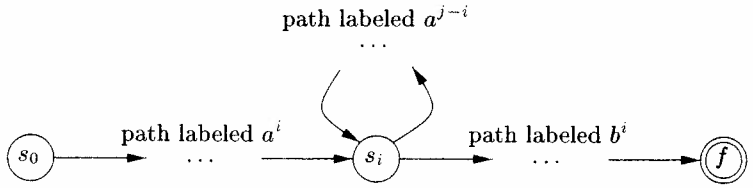

Figure 4.6: DFA D accepting both $a^i b^i$ and $a^j b^i$.

Colloquially, we say that "finite automata cannot count," meaning that a finite automaton cannot accept a language like $\{a^n b^n \mid n \geq 1\}$ that would require it to keep count of the number of a's before it sees the b's. Likewise, "a grammar can count two items but not three," as we shall see when we consider non-context-free language constructs in Section 4.3.5.

4.2.8 Exercises for Section 4.2

Exercise 4.2.1: Consider the context-free grammar:

$$S \rightarrow S\,S\,+ \mid S\,S\,* \mid a$$

and the string $aa+a*$.

a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string.

! d) Is the grammar ambiguous or unambiguous? Justify your answer.

! e) Describe the language generated by this grammar.

Exercise 4.2.2: Repeat Exercise 4.2.1 for each of the following grammars and strings:

a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$ with string 000111.

b) $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ with string $+ * aaa$.

! c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ with string $(()())$.

! d) $S \rightarrow S\ +\ S \mid S\ S \mid (\ S\ ) \mid S\ * \mid a$ with string $(a + a) * a$.

! e) $S \rightarrow (\ L\ ) \mid a$ and $L \rightarrow L\ ,\ S \mid S$ with string $((a, a), a, (a))$.

!! f) $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ with string $aabbab$.

! g) The following grammar for boolean expressions:

$$
\begin{array}{lcl}
\textit{bexpr} & \rightarrow & \textit{bexpr}\ \textbf{or}\ \textit{bterm} \mid \textit{bterm} \\
\textit{bterm} & \rightarrow & \textit{bterm}\ \textbf{and}\ \textit{bfactor} \mid \textit{bfactor} \\
\textit{bfactor} & \rightarrow & \textbf{not}\ \textit{bfactor} \mid (\ \textit{bexpr}\ ) \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

Exercise 4.2.3: Design grammars for the following languages:

a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.

! b) The set of all strings of 0s and 1s that are *palindromes*; that is, the string reads the same backward as forward.

! c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.

!! d) The set of all strings of 0s and 1s with an unequal number of 0s and 1s.

! e) The set of all strings of 0s and 1s in which 011 does not appear as a substring.

!! f) The set of all strings of 0s and 1s of the form xy, where $x \neq y$ and x and y are of the same length.

! **Exercise 4.2.4:** There is an extended grammar notation in common use. In this notation, square and curly braces in production bodies are metasymbols (like $\rightarrow$ or |) with the following meanings:

i) Square braces around a grammar symbol or symbols denotes that these constructs are optional. Thus, production $A \rightarrow X\ [Y]\ Z$ has the same effect as the two productions $A \rightarrow X\ Y\ Z$ and $A \rightarrow X\ Z$.

ii) Curly braces around a grammar symbol or symbols says that these symbols may be repeated any number of times, including zero times. Thus, $A \rightarrow X\ \{Y\ Z\}$ has the same effect as the infinite sequence of productions $A \rightarrow X$, $A \rightarrow X\ Y\ Z$, $A \rightarrow X\ Y\ Z\ Y\ Z$, and so on.

Phrase-level Recovery

Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack. Alteration of stack symbols or the pushing of new symbols onto the stack is questionable for several reasons. First, the steps carried out by the parser might then not correspond to the derivation of any word in the language at all. Second, we must ensure that there is no possibility of an infinite loop. Checking that any recovery action eventually results in an input symbol being consumed (or the stack being shortened if the end of the input has been reached) is a good way to protect against such loops.

4.4.6 Exercises for Section 4.4

Exercise 4.4.1 For each of the following grammars, devise predictive parsers and show the parsing tables. You may left-factor and/or eliminate left-recursion from your grammars first.

a) The grammar of Exercise 4.2.2(a).

b) The grammar of Exercise 4.2.2(b).

c) The grammar of Exercise 4.2.2(c).

d) The grammar of Exercise 4.2.2(d).

e) The grammar of Exercise 4.2.2(e).

f) The grammar of Exercise 4.2.2(g).

!! **Exercise 4.4.2:** Is it possible, by modifying the grammar in any way, to construct a predictive parser for the language of Exercise 4.2.1 (postfix expressions with operand a)?

Exercise 4.4.3 Compute FIRST and FOLLOW for the grammar of Exercise 4.2.1.

Exercise 4.4.4: Compute FIRST and FOLLOW for each of the grammars of Exercise 4.2.2.

Exercise 4.4.5: The grammar $S \rightarrow a\ S\ a \mid a\ a$ generates all even-length strings of a's. We can devise a recursive-descent parser with backtrack for this grammar. If we choose to expand by production $S \rightarrow a\ a$ first, then we shall only recognize the string aa. Thus, any reasonable recursive-descent parser will try $S \rightarrow a\ S\ a$ first.

a) Show that this recursive-descent parser recognizes inputs aa, $aaaa$, and $aaaaaaaa$, but not $aaaaaa$.

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

Figure 4.30: Productions involving procedure calls and array references

STACK	INPUT
··· **id** (**id**	, **id**) ···

It is evident that the **id** on top of the stack must be reduced, but by which production? The correct choice is production (5) if `p` is a procedure, but production (7) if `p` is an array. The stack does not tell which; information in the symbol table obtained from the declaration of `p` must be used.

One solution is to change the token **id** in production (1) to **procid** and to use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure. Doing so would require the lexical analyzer to consult the symbol table before returning a token.

If we made this modification, then on processing `p(i,j)` the parser would be either in the configuration

STACK	INPUT
··· **procid** (**id**	, **id**) ···

or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse. □

4.5.5 Exercises for Section 4.5

Exercise 4.5.1: For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ of Exercise 4.2.2(a), indicate the handle in each of the following right-sentential forms:

a) 000111.

b) 00S11.

Exercise 4.5.2: Repeat Exercise 4.5.1 for the grammar $S \rightarrow S\ S + \mid S\ S * \mid a$ of Exercise 4.2.1 and the following right-sentential forms:

a) $SSS + a * +$.

b) $SS + a * a +$.

c) $aaa * a + +$.

Exercise 4.5.3 Give bottom-up parses for the following input strings and grammars:

a) The input 000111 according to the grammar of Exercise 4.5.1.

b) The input $aaa * a + +$ according to the grammar of Exercise 4.5.2.

4.6 Introduction to LR Parsing: Simple LR

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. When (k) is omitted, k is assumed to be 1.

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift-reduce parsers, called "simple LR" (or SLR, for short). Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator. We begin with "items" and "parser states;" the diagnostic output from an LR parser generator typically includes parser states, which can be used to isolate the sources of parsing conflicts.

Section 4.7 introduces two, more complex methods — canonical-LR and LALR — that are used in the majority of LR parsers.

4.6.1 Why LR Parsers?

LR parsers are table-driven, much like the nonrecursive LL parsers of Section 4.4.4. A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an *LR grammar*. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

! b) The grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ of Exercise 4.2.1.

! c) The grammar $S \rightarrow S\ (\ S\ )\ |\ \epsilon$ of Exercise 4.2.2(c).

VI

Exercise 4.6.2: Construct the SLR sets of items for the (augmented) grammar of Exercise 4.2.1. Compute the GOTO function for these sets of items. Show the parsing table for this grammar. Is the grammar SLR?

Exercise 4.6.3: Show the actions of your parsing table from Exercise 4.6.2 on the input $aa*a+$.

VII

Exercise 4.6.4: For each of the (augmented) grammars of Exercise 4.2.2(a)–(g): (a), (c), (e)

a) Construct the SLR sets of items and their GOTO function.

b) Indicate any action conflicts in your sets of items.

c) Construct the SLR-parsing table, if one exists.

Exercise 4.6.5: Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & A\ a\ A\ b\ |\ B\ b\ B\ a \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

is LL(1) but not SLR(1).

Exercise 4.6.6: Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & S\ A\ |\ A \\ A & \rightarrow & a \end{array}$$

is SLR(1) but not LL(1).

!! **Exercise 4.6.7:** Consider the family of grammars G_n defined by:

$$\begin{array}{lcll} S & \rightarrow & A_i\ b_i & \text{for } 1 \leq i \leq n \\ A_i & \rightarrow & a_j\ A_i\ |\ a_j & \text{for } 1 \leq i, j \leq n \text{ and } i \neq j \end{array}$$

Show that:

a) G_n has $2n^2 - n$ productions.

b) G_n has $2^n + n^2 + n$ sets of LR(0) items.

c) G_n is SLR(1).

What does this analysis say about how large LR parsers can get?