Custom-Ebook

Team 13

Table of contents

Operating_System_200_202

OPTIMIZATION OF DFA-BASED PATTERN MATCHERS 175position of the leaf and also as a position of its symbol. Note that a symbolcan have several positions; for instance, a has positions 1 and 3 in Fig. 3.56.The positions in the syntax tree correspond to the important states of theconstructed NFA.Example 3.32 : Figure 3.57 shows the NFA for the same regular expression asFig. 3.56, with the important states numbered and other states represented byletters. The numbered states in the NFA and the positions in the syntax treecorrespond in a way we shall soon see. 2A B1 C2 DE 3 4 5 6aba b b# ### ####start#FFigure 3.57: NFA constructed by Algorithm 3.23 for (ajb)abb#3.9.2 Functions Computed From the Syntax TreeTo construct a DFA directly from a regular expression, we construct its syntaxtree and then compute four functions: nullable, rstpos, lastpos, and followpos,dened as follows. Each denition refers to the syntax tree for a particularaugmented regular expression (r)#.1. nullable(n) is true for a syntax-tree node n if and only if the subexpressionrepresented by n has in its language. That is, the subexpression can be\made null" or the empty string, even though there may be other stringsit can represent as well.2. rstpos(n) is the set of positions in the subtree rooted at n that corre-spond to the rst symbol of at least one string in the language of thesubexpression rooted at n.3. lastpos(n) is the set of positions in the subtree rooted at n that corre-spond to the last symbol of at least one string in the language of thesubexpression rooted at n.176 CHAPTER 3. LEXICAL ANALYSIS4. followpos(p), for a position p, is the set of positions q in the entire syntaxtree such that there is some string x = a1a2 an in L(r)#such thatfor some i, there is a way to explain the membership of x in L(r)#bymatching ai to position p of the syntax tree and ai+1 to position q.Example 3.33 : Consider the cat-node n in Fig. 3.56 that corresponds to theexpression (ajb)a. We claim nullable(n) is false, since this node generates allstrings of a's and b's ending in an a; it does not generate . On the other hand,the star-node below it is nullable; it generates along with all other strings ofa's and b's.rstpos(n) = f1; 2; 3g. In a typical generated string like aa, the rst positionof the string corresponds to position 1 of the tree, and in a string like ba, therst position of the string comes from position 2 of the tree. However, whenthe string generated by the expression of node n is just a, then this a comesfrom position 3.lastpos(n) = f3g. That is, no matter what string is generated from theexpression of node n, the last position is the a from position 3 of the tree.followpos is trickier to compute, but we shall see the rules for doing soshortly. Here is an example of the reasoning: followpos(1) = f1; 2; 3g. Considera string ac , where the c is either a or b, and the a comes from position 1.That is, this a is one of those generated by the a in expression (ajb). Thisa could be followed by another a or b coming from the same subexpression, inwhich case c comes from position 1 or 2. It is also possible that this a is thelast in the string generated by (ajb), in which case the symbol c must be thea that comes from position 3. Thus, 1, 2, and 3 are exactly the positions thatcan follow position 1. 23.9.3 Computing nullable, rstpos, and lastposWe can compute nullable, rstpos, and lastpos by a straightforward recursionon the height of the tree. The basis and inductive rules for nullable and rstposare summarized in Fig. 3.58. The rules for lastpos are essentially the same asfor rstpos, but the roles of children c1 and c2 must be swapped in the rule fora cat-node.Example 3.34 : Of all the nodes in Fig. 3.56 only the star-node is nullable.We note from the table of Fig. 3.58 that none of the leaves are nullable, becausethey each correspond to non- operands. The or-node is not nullable, becauseneither of its children is. The star-node is nullable, because every star-node isnullable. Finally, each of the cat-nodes, having at least one nonnullable child,is not nullable.The computation of rstpos and lastpos for each of the nodes is shown inFig. 3.59, with rstpos(n) to the left of node n, and lastpos(n) to its right. Eachof the leaves has only itself for rstpos and lastpos, as required by the rule fornon- leaves in Fig. 3.58. For the or-node, we take the union of rstpos at theOPTIMIZATION OF DFA-BASED PATTERN MATCHERS 177NODE n nullable(n) rstpos(n)A leaf labeled true ;A leaf with position i false figAn or-node n = c1jc2 nullable(c1) or rstpos(c1) [rstpos(c2)nullable(c2)A cat-node n = c1c2 nullable(c1) and if (nullable(c1))nullable(c2) rstpos(c1) [rstpos(c2)else rstpos(c1)A star-node n =

c1 true rstpos(c1)Figure 3.58: Rules for computing nullable and rstposchildren and do the same for lastpos. The rule for the star-node says that wetake the value of rstpos or lastpos at the one child of that node.Now, consider the lowest cat-node, which we shall call n. To computerstpos(n), we rst consider whether the left operand is nullable, which it isin this case. Therefore, rstpos for n is the union of rstpos for each of itschildren, that is f1; 2g [f3g = f1; 2; 3g. The rule for lastpos does not ap-pear explicitly in Fig. 3.58, but as we mentioned, the rules are the same asfor rstpos, with the children interchanged. That is, to compute lastpos(n) wemust ask whether its right child (the leaf with position 3) is nullable, which itis not. Therefore, lastpos(n) is the same as lastpos of the right child, or f3g.23.9.4 Computing followposFinally, we need to see how to compute followpos. There are only two waysthat a position of a regular expression can be made to follow another.1. If n is a cat-node with left child c1 and right child c2, then for everyposition i in lastpos(c1), all positions in rstpos(c2) are in followpos(i).2. If n is a star-node, and i is a position in lastpos(n), then all positions inrstpos(n) are in followpos(i).Example 3.35 : Let us continue with our running example; recall that rstposand lastpos were computed in Fig. 3.59. Rule 1 for followpos requires that welook at each cat-node, and put each position in rstpos of its right child infollowpos for each position in lastpos of its left child. For the lowest cat-node inFig. 3.59, that rule says position 3 is in followpos(1) and followpos(2). The nextcat-node above says that 4 is in followpos(3), and the remaining two cat-nodesgive us 5 in followpos(4) and 6 in followpos(5).

Computer_Network_300_303

4.7. MORE POWERFUL LR PARSERS 275SET ITEMLOOKAHEADSINIT PASS 1 PASS 2 PASS 3I0: S0 ! S $ $ $ $I1: S0 ! S $ $ $I2: S ! L = R $ $ $R! L $ $ $I3: S ! R $ $ $I4: L! R = =/$ =/$ =/$I5: L! id = =/$ =/$ =/$I6: S ! L = R $ $I7: L! R = =/$ =/$I8: R! L = =/$ =/$I9: S ! L = R $Figure 4.47: Computation of lookaheads4.7.6 Compaction of LR Parsing TablesA typical programming language grammar with 50 to 100 terminals and 100productions may have an LALR parsing table with several hundred states. Theaction function may easily have 20,000 entries, each requiring at least 8 bitsto encode. On small devices, a more ecient encoding than a two-dimensionalarray may be important. We shall mention briey a few techniques that havebeen used to compress the ACTION and GOTO elds of an LR parsing table.One useful technique for compacting the action eld is to recognize thatusually many rows of the action table are identical. For example, in Fig. 4.42,states 0 and 3 have identical action entries, and so do 2 and 6. We can thereforesave considerable space, at little cost in time, if we create a pointer for eachstate into a one-dimensional array. Pointers for states with the same actionspoint to the same location. To access information from this array, we assigneach terminal a number from zero to one less than the number of terminals,and we use this integer as an oset from the pointer value for each state. Ina given state, the parsing action for the ith terminal will be found i locationspast the pointer value for that state.Further space eciency can be achieved at the expense of a somewhat slowerparser by creating a list for the actions of each state. The list consists of(terminal-symbol, action) pairs. The most frequent action for a state can be276 CHAPTER 4. SYNTAX ANALYSISplaced at the end of the list, and in place of a terminal we may use the notation\any," meaning that if the current input symbol has not been found so far onthe list, we should do that action no matter what the input is. Moreover, errorentries can safely be replaced by reduce actions, for further uniformity along arow. The errors will be detected later, before a shift move.Example 4.65 : Consider the parsing table of Fig. 4.37. First, note that theactions for states 0, 4, 6, and 7 agree. We can represent them all by the listSYMBOL ACTIONid s5(s4any errorState 1 has a similar list:+ s6$ accany errorIn state 2, we can replace the error entries by r2, so reduction by production 2will occur on any input but *. Thus the list for state 2 is s7any r2State 3 has only error and r4 entries. We can replace the former by thelatter, so the list for state 3 consists of only the pair (any, r4). States 5, 10,and 11 can be treated similarly. The list for state 8 is+ s6) s11any errorand for state 9 s7any r12We can also encode the GOTO table by a list, but here it appears moreecient to make a list of pairs for each nonterminal A. Each pair on the listfor A is of the form (currentState; nextState), indicatingGOTO[currentState; A] = nextState4.7.

MORE POWERFUL LR PARSERS 277This technique is useful because there tend to be rather few states in any onecolumn of the GOTO table. The reason is that the GOTO on nonterminal Acan only be a state derivable from a set of items in which some items have Aimmediately to the left of a dot. No set has items with X and Y immediatelyto the left of a dot if X 6= Y . Thus, each state appears in at most one GOTOcolumn.For more space reduction, we note that the error entries in the goto table arenever consulted. We can therefore replace each error entry by the most commonnon-error entry in its column. This entry becomes the default; it is representedin the list for each column by one pair with any in place of currentState.Example 4.66 : Consider Fig. 4.37 again. The column for F has entry 10 forstate 7, and all other entries are either 3 or error. We may replace error by 3and create for column F the listCURRENTSTATE NEXTSTATE7 10any 3Similarly, a suitable list for column T is6 9any 2For column E we may choose either 1 or 8 to be the default; two entries arenecessary in either case. For example, we might create for column E the list4 8any 12This space savings in these small examples may be misleading, because thetotal number of entries in the lists created in this example and the previous onetogether with the pointers from states to action lists and from nonterminalsto next-state lists, result in unimpressive space savings over the matrix imple-mentation of Fig. 4.37. For practical grammars, the space needed for the listrepresentation is typically less than ten percent of that needed for the matrixrepresentation. The table-compression methods for nite automata that werediscussed in Section 3.9.8 can also be used to represent LR parsing tables.4.7.7 Exercises for Section 4.7Exercise 4.7.1 : Construct thea) canonical LR, andb) LALR278 CHAPTER 4. SYNTAX ANALYSISsets of items for the grammar S ! S S + j S S j a of Exercise 4.2.1.Exercise 4.7.2 : Repeat Exercise 4.7.1 for each of the (augmented) grammarsof Exercise 4.2.2(a){(g).! Exercise 4.7.3 : For the grammar of Exercise 4.7.1, use Algorithm 4.63 tocompute the collection of LALR sets of items from the kernels of the LR(0) setsof items.! Exercise 4.7.4 : Show that the following grammarS ! A a j b A c j d c j b d aA ! dis LALR(1) but not SLR(1).! Exercise 4.7.5 : Show that the following grammarS ! A a j b A c j B c j b B aA ! dB ! dis LR(1) but not LALR(1).4.8 Using Ambiguous GrammarsIt is a fact that every ambiguous grammar fails to be LR and thus is not inany of the classes of grammars discussed in the previous two sections. How-ever, certain types of ambiguous grammars are quite useful in the specicationand implementation of languages. For language constructs like expressions, anambiguous grammar provides a shorter, more natural specication than anyequivalent unambiguous grammar. Another use of ambiguous grammars is inisolating commonly occurring syntactic constructs for special-case optimiza-tion. With an ambiguous grammar, we can specify the special-case constructsby carefully adding new productions to the grammar.Although the grammars we use are ambiguous, in all cases we specify dis-ambiguating rules that allow only one parse tree for each sentence. In this way,the overall language specication becomes unambiguous, and sometimes it be-comes possible to design an LR parser that follows the same ambiguity-resolvingchoices. We stress that ambiguous constructs should be used sparingly and ina strictly controlled fashion; otherwise, there can be no guarantee as to whatlanguage is recognized by a parser.

compiler_1_100_103

2.5. A TRANSLATOR FOR SIMPLE EXPRESSIONS 75import java.io.*;class Parser {static int lookahead;public Parser() throws IOException {lookahead = System.in.read();}void expr() throws IOException {term();while(true) {if(lookahead == '+') {match('+'); term(); System.out.write('+');}else if(lookahead == '-') {match('-'); term(); System.out.write('-');}else return;}}void term() throws IOException {if(Character.isDigit((char)lookahead)) {System.out.write((char)lookahead); match(lookahead);}else throw new Error("syntax error");}void match(int t) throws IOException {if(lookahead == t) lookahead = System.in.read();else throw new Error("syntax error");}}public class Postfix {public static void main(String[] args) throws IOException {Parser parse = new Parser();parse.expr(); System.out.write('\n');}}Figure 2.27: Java program to translate inx expressions into postx form76 CHAPTER 2. A SIMPLE SYNTAX-

DIRECTED TRANSLATORA Few Salient Features of JavaThose unfamiliar with Java may nd the following notes on Java helpfulin reading the code in Fig. 2.27: A class in Java consists of a sequence of variable and function de-nitions. Parentheses enclosing function parameter lists are needed even ifthere are no parameters; hence we write expr() and term(). Thesefunctions are actually procedures, because they do not return values,signied by the keyword void before the function name. Functions communicate either by passing parameters \by value"or by accessing shared data. For example, the functions expr()and term() examine the lookahead symbol using the class variablelookahead that they can all access since they all belong to the sameclass Parser. Like C, Java uses = for assignment, == for equality, and != for in-equality. The clause \throws IOException" in the denition of term() de-clares that an exception called IOException can occur. Such anexception occurs if there is no input to be read when the functionmatch uses the routine read. Any function that calls matchmust alsodeclare that an IOException can occur during its own execution.2.6 Lexical AnalysisA lexical analyzer reads characters from the input and groups them into \tokenobjects." Along with a terminal symbol that is used for parsing decisions,a token object carries additional information in the form of attribute values.So far, there has been no need to distinguish between the terms \token" and\terminal," since the parser ignores the attribute values that are carried by atoken. In this section, a token is a terminal along with additional information.A sequence of input characters that comprises a single token is called alexeme. Thus, we can say that the lexical analyzer insulates a parser from thelexeme representation of tokens.The lexical analyzer in this section allows numbers, identiers, and \whitespace" (blanks, tabs, and newlines) to appear within expressions. It can be usedto extend the expression translator of the previous section. Since the expressiongrammar of Fig. 2.21 must be extended to allow numbers and identiers, we2.6. LEXICAL ANALYSIS 77shall take this opportunity to allow multiplication and division as well. Theextended translation scheme appears in Fig. 2.28.expr ! expr + term f print(0+0) gj expr - term f print(0-0) gj termterm ! term * factor f print(0*0) gj term / factor f print(0/0) gj factorfactor ! (expr)j num f print(num:value) gj id f print(id:lexeme) gFigure 2.28: Actions for translating into postx notationIn Fig. 2.28, the terminal num is assumed to have an attribute num.value,which gives the integer value corresponding to this occurrence of num. Termi-nal id has a string-valued attribute written as id.lexeme; we assume this stringis the actual lexeme comprising this instance of the token id.The pseudocode fragments used to illustrate the workings of a lexical ana-lyzer will be assembled into Java code at the end of this section. The approachin this section is suitable for hand-written lexical analyzers. Section 3.5 de-scribes a tool called Lex that generates a lexical analyzer from a specication.Symbol tables or data structures for holding information about identiers areconsidered in Section 2.7.2.6.1 Removal of White Space and CommentsThe expression translator in Section 2.5 sees every character in the input, soextraneous characters, such as blanks, will cause it to fail. Most languagesallow arbitrary amounts of white space to appear between tokens. Commentsare likewise ignored during parsing, so they may also be treated as white space.If white space is eliminated by the lexical analyzer, the parser will neverhave to consider it. The alternative of modifying the grammar to incorporatewhite space into the syntax is not nearly as easy to implement.The pseudocode in Fig. 2.29 skips white space by reading input charactersas long as it sees a blank, a tab, or a newline. Variable peek holds the nextinput character. Line numbers and context are useful within error messages tohelp pinpoint errors; the code uses variable line to count newline characters inthe input.78 CHAPTER 2. A SIMPLE SYNTAX-DIRECTED TRANSLATORfor (; ; peek = next input character) fif (peek is a blank or a tab) do nothing;else if (peek is a newline) line = line+1;else break;gFigure 2.29: Skipping white space2.6.2 Reading AheadA lexical analyzer may need to read ahead some characters before it can decideon the token to be returned to the parser. For example, a lexical analyzer forC or Java must read ahead after it sees the character >. If the next characteris =, then > is part of the character sequence >=, the lexeme for the token forthe \greater than or equal to" operator. Otherwise > itself forms the \greaterthan" operator, and the lexical analyzer has read one character too many.A general approach to reading ahead on the input, is to maintain an inputbuer from which the lexical analyzer can read and push back characters. Inputbuers can be justied on eciency grounds alone, since fetching a block ofcharacters is usually more ecient than fetching one character at a time. Apointer keeps track of the portion of the input that has been analyzed; pushingback a character is implemented by moving back the pointer. Techniques forinput buering are discussed in Section 3.2.One-character read-ahead usually suces, so a simple solution is to use avariable, say peek, to hold the next input character. The lexical analyzer inthis section reads ahead one character while it collects digits for numbers orcharacters for identiers; e.g., it reads past 1 to distinguish between 1 and 10,and it reads past t to distinguish between t and true.The lexical analyzer reads ahead only when it must. An operator like * canbe identied without reading ahead. In such cases, peek is set to a blank, whichwill be skipped when the lexical analyzer is called to nd the next token. Theinvariant assertion in this section is that when the lexical analyzer returns atoken, variable peek either holds the character beyond the lexeme for the currenttoken, or it holds a blank.2.6.3 ConstantsAnytime a single digit appears in a grammar for expressions, it seems reasonableto allow an arbitrary integer constant in its place. Integer constants can beallowed either by creating a terminal symbol, say num, for such constants orby incorporating the syntax of integer constants into the grammar. The jobof collecting characters into integers and computing their collective numericalvalue is generally given to a lexical analyzer, so numbers can be treated as singleunits during parsing and translation.