CS 230
Programming Languages

09 / 27 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- Another Bottom up parser example
- Regular Expressions (Regexs)
  - For matching
  - General concepts
  - How the regex engine works (especially backtracking)
  - Java code for regexs
- Last ½ hour you can work on the exercises from yesterday. If you finished, create some invalid syntax programs to test out your error code.

Bottom Up parsers

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - This substring is called the handle
  - This is backwards from the way top down parsing works
  - The most common bottom-up parsing algorithms are in the LR family (L=left-to-right scan of input, R= rightmost derivations)

Bottom Up parsers

- Bottom-up parsers
  - The whole process is: Starting with the input sentence (all terminal symbols) (also this is the last sentential form in a derivation), produce the sequence of sentential forms (up) until all that remains is the start symbol.
  - One step in the process of bottom-up parsing is: given a right sentential form, find the correct RHS to reduce to get the previous right-sentential form in the derivation
  - What are we reducing a RHS to?

Bottom Up parsers

Example:

E -> E + T | T

T -> T * F | F

F -> (E) | id

Bottom Up parsers

Example:

E -> E + T | T

T -> T * F | F

F -> (E) | id

- Note: Sometimes it is not obvious which RHS to reduce in each step. For instance, the right-sentential form E + T * id includes more than one RHS. Which RHSs are included in this sentential form?
- And why can we not choose the others?

Bottom Up parsers

- The correct choice of the RHS to reduce in any given step is called the handle (of that right-sentential form.)
- The text contains intuition on how to find the handle --- we won't cover that here. Just know that the handle of any right-sentential form is unique.

Bottom Up parsers

- Most bottom-up parsers are LR --- what is LR again?
- They use small programs and a parsing table.

Bottom Up parsers

- LR parsers consist of
  - A parse stack
  - An input string (the sentence to be determined if it is syntactically correct)
  - A parse table created from the grammar beforehand. Its rows are states, and its columns are terminal and nonterminal symbols.
  - So, given an input symbol and a state, the program will lookup in the table what to do.

arithmetic expression grammar (same one as on earlier slide)

- 1. E -> E + T
- 2. E -> T
- 3. T -> T * F
- 4. T -> F
- 5. F -> (E)
- 6. F -> id

- For next slide, R means reduce, S means shift. e.g. R4 means *reduce using production 4* above. S6 means *shift the next symbol of input onto the stack and push state 6 onto the stack.*

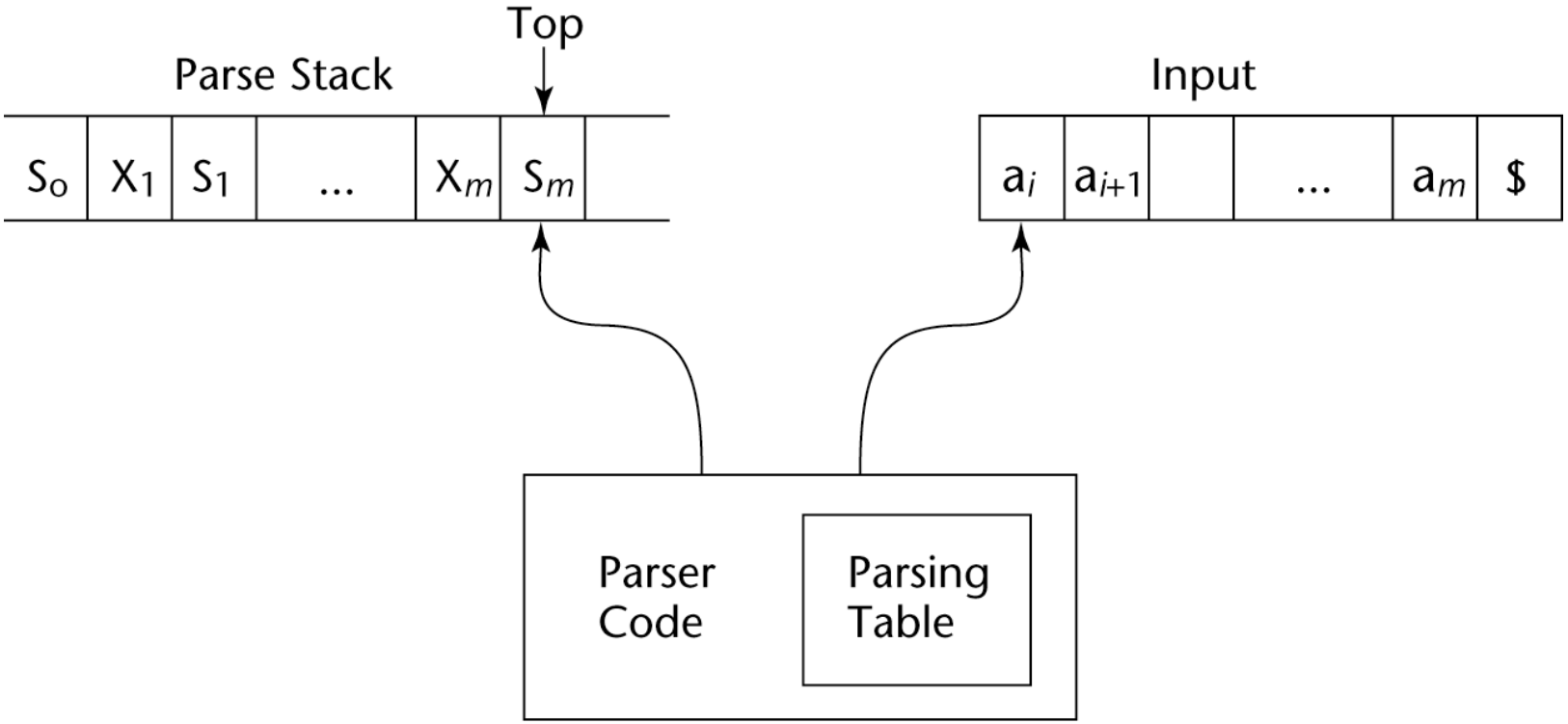
LR parsing table for arithmetic expression grammar the S4 in row: State 0, should be under (not *

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

ACTION & GOTO

- The GOTO portion of the table is used to determine which state to push onto the stack after a reduction.
- The ACTION portion of the table is used to determine whether to shift or reduce based on the current state and next input symbol.
- Blank cells in the table imply syntax errors.
- accept means the sentence is syntactically correct.
- The stack starts with only state 0 on it.
- The input string is the complete sentence followed by a termination symbol, usually a $.

LR Parser structure



- Top of stack is to the right, the next input symbol for the input is the leftmost symbol. S's are State #'s and X's are grammar symbols.

Let's go through an example parse

- Before we go through an example parse a few things should be stated.

- Recall that a shift pushes the next input symbol on the stack and then pushes the specified state on the stack as well. That's pretty straightforward.

Let's go through an example parse

- A reduce is more complex. When a reduce occurs,

- 1. the handle (the whole RHS) is popped off the stack (along with each state per symbol in the handle)
- 2. then the LHS is pushed onto the stack
- 3. followed by another state pushed onto the stack. The state to be pushed is determined by the GOTO portion of the parse table.
  - column is the LHS just pushed
  - row is the state that was on the top of the stack after the handle and its associated states were popped.

Let's go through an example parse

- We already determined if

id + id * id

is syntactically correct for the example grammar a few slides ago.

Let's go through an example parse

- How about we determine if

(id) id + id

is syntactically correct for the example grammar a few slides ago.

We'll do this by viewing the table on screen and I'll show the stack and input and action on the board.

Also, I'll write on the board what happens during a shift and a reduce.

Regex

- Matching
  - If you wanted to test if a string contained exactly some particular value, how would you do it?

Regex

- Matching
  - If you wanted to test if a string contained exactly some particular value, how would you do it?
    - In Java, you might use .equals method of String
      - e.g. if (name.equals(“Mike”))

Regex

- Matching
  - But, what if you wanted to test if a string contained some particular text somewhere within it, how would you do it?
  - If you wanted to test if a string contained 4 digits together somewhere within it, how would you do it?
  - How about determining if a string contained 2 or more consecutive “the ”'s? And if it did, remove all but one of them.

Regex

- Matching
  - But, what if you wanted to test if a string contained some particular text somewhere within it, how would you do it?
  - If you wanted to test if a string contained 4 digits together somewhere within it, how would you do it?
  - How about determining if a string contained 2 or more consecutive “the ”'s? And if it did, remove all but one of them.
  - Regular expressions (aka regexes) can do this, and more.
  - They are powerful!

Regex

- What are regular expressions?
  - Regular expressions consist of a specialized syntax used to describe patterns of text
  - May be used to find and/or extract text from input source (file, network data stream…)

- Why study regular expressions?
  - They are used in a variety of tools
    - Editors, file search utilities, compilers, programming language libraries
  - Useful as a quick way to search and parse data
  - We will use them in several ways including in our lexical analyzer

Regex

- Regular Expression syntax varies among programming languages and tools
- Regexes have been used for many years within computer science and over time variations in the syntax have arisen
  - Different implementations will work with different regex syntax, which can be frustrating when switching between languages
- But, luckily, most of the core regex syntax is common among languages
  - When switching between languages, just need to focus on the differences

Regex

- Languages such as Perl, Python, C, Java all provide regular expression support.
- Terminal based tools such as
  - grep (a search tool that searches files for matching patterns),
  - awk (a programming language for text processing), and
  - vi (a text editor)
  - among many others

  all include support for regular expressions

Regex

- A regular expression is interpreted by a program that compares input text to the regular expression to identify a match, if present
  - The program is a regex engine
  - e.g. a lexical analyzer may use regular expressions that defines the pattern of lexemes/tokens
- The regex engine's
  - Input: is text and the regular expression and
  - Output: is metadata identifying the match
    - e.g. whether or not there was a match and if so where (start and end character positions) in the input text the match was found

Regex

- In the early days of text editing a common syntax for using a regular expression to find or match text in a file was to place the regex pattern between two slashes --- still used in perl
  - Before the first slash the user could include the letter ‘g’ meaning that the search should be global (search the entire file)
  - After the second slash the user could include the letter ‘p’ meaning that each match should be printed on the console
  - This led to the pattern **g/regular_expression/p**
- This was used as the name for a command line text searching tool: grep

Regex

- The simplest form of regular expression is just exact text to look for to try to match.
- e.g. Given the regex “rem” and the input text “Skidmore College is the premier school in the northeast.”
- r is tries to match the S, it doesn’t so r tries to match k and doesn’t, r is compared to i, r is compared to d, r is compared to m, r is compared to o, then r is compared to r and they match so e is compared to e and they match, but since m doesn’t match the space after Skidmore it starts comparing the r to the e, then r to space, then r to C etc. eventually r matches the first r in premier then e matches e and then m matches m in premier and the regular expression successfully matched rem in the input text.
- The result being that “Skidmore College is the p” is the text before the match, and “ier school in the northeast.” is the text after the match.
- When we write the regex we are defining an expected order of characters to match

Regex

- Another example:
- e.g. Given the regex “ledge” and the input text “Skidmore College is the premier school in the northeast.”
- This match attempt will fail to match and the regex engine will indicate in some way that it failed to match.
- Let me explain how it will go about doing it, using backtracking when necessary.

Regex

- A slightly more complex regular expression than previous examples uses a character class. Character classes are put between [ ] square brackets within the pattern --- this means exactly one of the set of characters in the [ ] is able to match.
- e.g. Given the regex "[Cm]o" and the input text "Skidmore College"
  - This will try to match either Co or mo in the input text
- It starts trying to match at the beginning of the text (and so prefers to match as early as possible in the text) so the mo matches the mo in Skidmore and result is "Skid" is before the match, "mo" is the match, "re College" is after the match.
- Notice: because mo appears before Co the mo match happens.

Regex

- Character classes are put between [ ] square brackets within the pattern --- this means any **one single** character in the [ ] is able to match.
    - May use ^ (not) --- to match any one char not part of the character class
    - May use – (hyphen) --- to specify a **range** of characters for the class

input: “Hay is for horses” regex: “[A-Z]ay” –- matches if the input contains any capital letter followed by ay, would match Hay

input: “The boardgame Payday is fun” regex: “[A-Z]ay” –- this would be match too, it matches Pay

input: “The boardgame Payday is fun” regex: “[^A-Z]ay” –- this would be true too, why?

In English, how would you describe, what the regex “[bchr]at” would try to match?

Regex

- Character classes may contain multiple ranges:
- e.g. [a-zA-Z] says match any one letter (lowercase or uppercase)
- [0-9A-F] says to match one Hexadecimal digit
- This next one does not have multiple ranges in one character class, but instead has two character classes one after the next so it will match only substrings of length 2:

  [1-3][0-9] says to match any 2 digit number 10 through 39

Regex

- In Java's String class, several methods take regular expressions:
  - matches – takes a regex and returns true or false if whole String matches the regex or not
  - replaceAll and replaceFirst --- both take a regex and a replacement String as parameters and returns a String with the replacement String inserted where the regex was found
  - split --- splits a string based on regex as delimiters, returns an array of Strings
  - Let's try some of these out on Strings / regexes.

Regex

- Let's see the results of some of those past examples in Java using the Pattern and Matcher classes which provide more functionality than the few methods provided in String.
- Pattern p = Pattern.compile(regex);

  // where regex is a String containing the regular expression
- Matcher m = p.matcher(inputText);

  // where inputText is the text to search for the pattern p

  // associated with the regex
- Methods in Matcher class

  matches --- acts like the matches one in String, requires the whole text to match the regex

  find --- looks for a subset of the input to match the regex

  start --- returns the index of the start of the previous match

  end --- returns the offset after the last character in match

Regex

- There are ways to specify common character classes
  - **\d (any digit)**
  - **\s (any whitespace \ \t\r\n\f)**
  - **\w (any "word" character (a digit, letter or underscore))**
  - **\D (any non-digit)**
  - **\S (any non-whitespace)**
  - **\W (any non-word character)**
  - **. (any character other than newline \n)**
- These can be used within the square brackets or outside of them.

Regex

- How might you make a regular expression to match any even 3 digit number?
- How about a date in either MM/DD/YY or MM-DD-YY format?