Today's Topics

Last Week

- Lexical & syntactic specification of languages,
  context-free grammars, BNF, terminal vs nonterminal symbols,
  relation to S/SL, sentential forms, derivations, parse trees,
  ambiguity, precedence and associativity

This Week

- Context-free parsers -
  bottom up, LR, top down, recursive descent, LL

© 2019 J.R. Cordy

Parsing

- *Parsing* is attempting to find a derivation (parse tree) from a given grammar's start symbol for a given input
- Essentially the problem is, given an input string (e.g., PT Pascal program), discover a derivation from the start symbol of the grammar to input string (e.g., grammatical structure of the program)
- And to discover, if there is no such derivation, where the problem is (i.e., the syntax error)

(the start symbol) *(the input)*

$$S \Rightarrow z_1 \Rightarrow z_2 \Rightarrow z_3 \Rightarrow \ldots \Rightarrow z_{n-1} \Rightarrow z_n \Rightarrow \omega$$

 © 2019 J.R. Cordy

Bottom-up Parsing

- Currently one of the most popular parsing methods is *bottom-up parsing* and in particular *LALR(1)* parsing, the method used by the *yacc* and *bison* parser generators
- Appropriately enough, bottom-up parsing attempts to construct a parse tree from the bottom up - that is, starting at the bottom (the input symbols) of the parse tree and applying grammar productions *in reverse* to build up the derivation backwards
- *(Although this seems like the only possible way to parse, it isn't, as we shall see next time)*
- Each reverse step to the previous sentential form (i.e., reversed derivation step) is called a *reduction*

(the start symbol) *(the input)*

$$S \Rightarrow z_1 \Rightarrow z_2 \Rightarrow z_3 \Rightarrow \ldots \Rightarrow z_{n-1} \Rightarrow z_n \Rightarrow \omega$$

$$\Leftarrow \quad \Leftarrow \quad \Leftarrow \quad \Leftarrow \quad \ldots \Leftarrow \quad \Leftarrow \quad \Leftarrow$$

derivation $\Rightarrow$

reduction $\Leftarrow$

© 2019 J.R. Cordy

Bottom-up Parsing

Grammar

$$S \rightarrow (A) \mid S\ S$$

$$A \rightarrow a\ b \mid b\ a \mid A\ A$$

Input

ω = (a b a b) (a b b a)

© 2019 J.R. Cordy

Bottom-up Parsing

- Definition: a *rightmost derivation*

$$X \Rightarrow^{*}_{rm} Y$$

  is a sequence

$$X \Rightarrow X_1 \Rightarrow X_2 \Rightarrow \ldots \Rightarrow Y$$

  where only the *rightmost* non-terminal in each sentential form is replaced at each step

- Just like a leftmost derivation, only from the right

- A *right sentential form* (RSF) X has the property :

$$S \Rightarrow^{*}_{rm} X$$

- Rightmost derivations are interesting because, if you run them in reverse, they work from the *left!* - and of course we read input from the left

© 2019 J.R. Cordy

Bottom-up Parsing

- Definition: a *handle* is a sequence of symbols Q at a position *i* in a right sentential form *PQR* where :

$$P, Q \in (N \cup T)^*, \quad R \in T^*$$

$$A \rightarrow Q$$

$$S \Rightarrow^*_{rm} PAR \Rightarrow PQR$$

- That is, it is the location where the right hand side of the next production to apply in reverse can be reduced to give the previous sentential form in a rightmost derivation

- Example: Given the grammar

$$S \rightarrow (\,A\,) \mid S\,S \qquad A \rightarrow a\,b \mid b\,a \mid A\,A$$

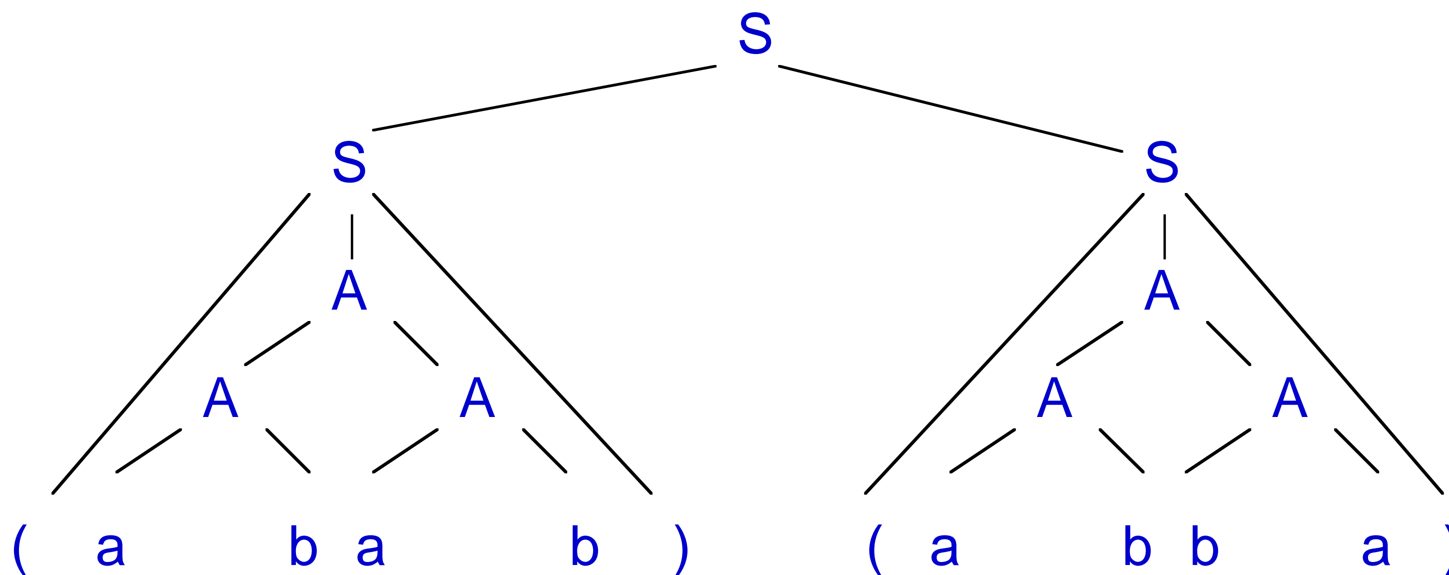
Then RSF S(abba) has handle ab at position 3, and can be reduced using $A \Rightarrow ab$ to give previous RSF S(Aba)

© 2019 J.R. Cordy

Bottom-up Parsing - Example

- *Rightmost* derivation in *reverse* means *leftmost* reduction
- Since we read input from left-to-right when parsing, this is good!

Grammar	Reductions	
S → (A)	(abab)(abba)	S(Aba)
| S S	(Aab)(abba)	S(AA)
A → a b	(AA)(abba)	S(A)
| b a	(A)(abba)	SS
| A A	S(abba)	S



Shift-Reduce Parsing

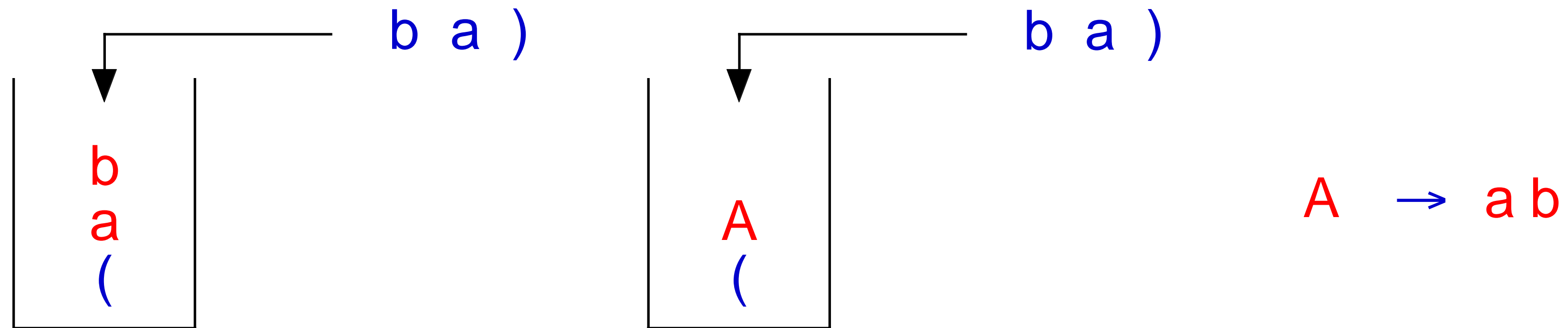
- Shift-reduce is a simple algorithm for implementing bottom-up parsing by reduction of leftmost handles in the input
- Symbols from the input are *shifted* onto a stack until a handle appears on top of the stack
- Example: Given the grammar

$$S \rightarrow (A) \mid S\ S \qquad A \rightarrow a\ b \mid b\ a \mid A\ A$$

and input string (abba)

Shift-Reduce Parsing

- The symbols that are the handle on top of the stack are then replaced by the left hand side of the production (*reduced*)



- If this results in another handle on top of the stack, then another reduction is done, otherwise we go back to shifting
- This combination of shifting input symbols onto the stack and reducing productions when handles appear on the top of the stack continues until all of the input is consumed and the goal symbol is the only thing on the stack - the input is then *accepted*
- If we reach the end of the input and cannot reduce the stack to the goal symbol, the input is *rejected*

© 2019 J.R. Cordy

Shift-Reduce Parsing - Example

```
E → E + E
  | E * E
  | a
```

a + a * a

stack	***input***	***action***
empty	a + a * a	shift
a	+ a * a	reduce E → a
E	+ a * a	shift
E +	a * a	shift
E + a	* a	reduce E → a
E + E	* a	reduce E → E + E
E	* a	shift
E *	a	shift
E * a	*empty*	reduce E → a
E * E	*empty*	reduce E → E * E
E	*empty*	accept

© 2019 J.R. Cordy

Shift-Reduce Parsing - Example 2

```
S  →  (A)          A  →  ab
   |  SS              |  ba
                      |  AA
```

stack	***input***	***action***
empty	(abab)(abba)	shift
(	abab)(abba)	shift
(a	bab)(abba)	shift
(ab	ab)(abba)	reduce A → ab
(A	ab)(abba)	shift
(Aa	b)(abba)	shift
(Aab	)(abba)	reduce A → ab
(AA	)(abba)	reduce A → AA
…	…	…
S(AA	)	reduce A → AA
S(A	)	shift
S(A)	*empty*	reduce S → (A)
SS	*empty*	reduce S → SS
S	*empty*	accept

© 2019 J.R. Cordy

Making Shift-Reduce Efficient - LR Grammars

- In general, finding the handle can be an expensive operation, because we must compare the symbols on top of the stack to the right hand side of every production (often *hundreds* of them)

- This can be addressed by restricting the grammar to those for which it is easy to recognize handles using a small set of states, giving an efficient shift-reduce parser - the LR class of grammars

- Subclasses of the LR class of grammar are :

  LR(k), SLR and LALR(k)

© 2019 J.R. Cordy

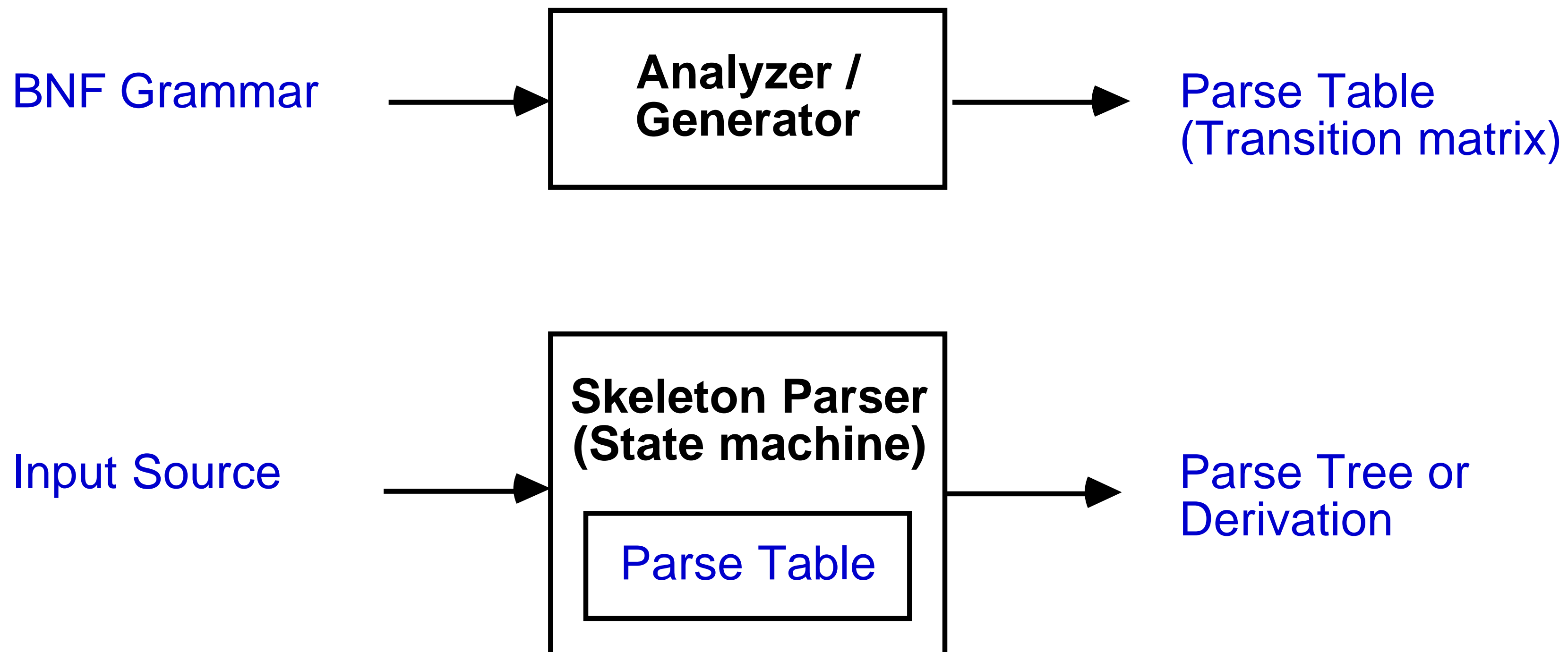
Efficient Shift-Reduce - LR Parsers

- LR stands for Left to right scan of input to find a Rightmost derivation in reverse
- The LR techniques are based on:
  - Bounding the depth to which we must look when finding a handle on the stack, and then
  - Enumerating all the possible combinations of symbols on the stack to that depth
  - Assigning each combination a stack state
- Each of the possible combinations of symbols on the stack are assigned a state in a state machine, and a transition matrix is used to determine the next parser action from the current parse state and the next input symbol
- This yields a highly efficient (linear, O(n)) bottom-up parsing algorithm

© 2019 J.R. Cordy

Efficient Shift-Reduce - LR Parsers

- The transition matrix is merged with a skeleton parsing program which interprets it (much like the generic S/SL walker)
- The generation of the transition matrix and the merge with the skeleton program are automated by parser "generators" such as *yacc* and *bison*



 © 2019 J.R. Cordy

Efficient Shift-Reduce - LR Parsers

- Advantages of LR parsers :
  - Efficient, automated, common
- Disadvantages of LR parsers :
  - LR limitations on the grammar make it awkward to specify a grammar for a language - you often have to work with a much less than desirable grammar
  - Difficult to debug grammar - the algorithm for generating tables is complex, making it difficult to relate conflicts (violations of the LR limitations) to the grammar
  - Non-transparent parsing algorithm - the relation between the grammar and the parsing algorithm is complex, making it difficult to relate syntax errors to problems in the grammar (or the input)

Summary

Bottom-up Parsers

- Bottom-up parsers attempt to build the parse tree for the input by applying grammar productions in reverse - reducing sequences of symbols that match the right hand side to the symbol on the left hand side
- Practical bottom-up parsers use a shift-reduce algorithm, which shifts input symbols on to a stack until a handle (the right hand side of a production) appears on the stack, then reduces (replaces) the symbols with the symbol on the left hand side
- By bounding the depth in the stack the parser needs to search for a handle by placing restrictions on the grammar, we get efficient (linear time) LR parsers

Next Time

- Top-down Parsers