Today’s Topics

Last Time

• The parsing problem
• Bottom-up parsers - right sentential forms (RSF), handles, the shift-reduce parsing algorithm, LR parsers

This Time

• Top-down parsers - predictive parsing, backtracking, recursive descent, LL parsers, relation to S/SL
Top-down Parsing

• Opposite of bottom-up (obviously)

• Start with the start symbol (at the top of the parse tree) and attempt to find a leftmost derivation of the input string, working from the top down

• The choice of which production to use next is predictive - based on the next input symbol, we must guess which of a set of possible productions might apply
Top-down Parsing

• Top-down parsing tries to predict (guess) which productions will be needed by looking at the next symbol(s) in the input.

• Recall that leftmost derivations have only terminals on the right at each left sentential form (LSF) in the derivation (like RSF’s in reverse, this is consistent with reading input left-to-right).

• A top-down parse does a (forward) leftmost derivation in which at any point in the parse the input symbols that have yet to be read will be in the rightmost part of the LSF.
Top-down Parsing - Example

• Example: Given the grammar

\[
S \Rightarrow (A) \\
| \quad S \; S \\
| \quad A \; A
\]

\[
A \Rightarrow a \; b \\
| \quad b \; a \\
| \quad A \; A
\]

and input string \((abba)\)

• Starting with \(S\), we can predict that we need:

\[
S \Rightarrow (A) \Rightarrow (ab) \Rightarrow (ab)
\]
Top-down Parsing - Example

• Example: Given the grammar

\[
S \Rightarrow (A) \\
| S \ S \\
| A \ A
\]

A \Rightarrow a \ b

and input string (abba)

• Starting with S, we can predict that we need:

\[
S \Rightarrow (A) \Rightarrow (ab) \Rightarrow (ab) \quad \text{Oops! Maybe not ...}
\]
Backtracking

• As we go along, we may discover that things don’t work out - that is, a guess we made must have been *incorrect!*

• If so, we have to *backtrack* to try another guess

• When we backtrack, we must *undo input* as well as *production choices* to “rewind” and try another possibility
Backtracking - Example

• Example:

\[
\begin{align*}
S & \Rightarrow ( A ) & A & \Rightarrow a \ b \\
| & S & S & | & b & a \\
| & A & A & |
\end{align*}
\]

and input string (abba)

• Starting with S, we predicted that we needed:

\[
\begin{align*}
S & \Rightarrow ( A ) \Rightarrow ( ab ) \Rightarrow ( ab ) & \text{Oops! Maybe not ...}
\end{align*}
\]

• But the A \Rightarrow a \ b guess didn’t work, so backtrack to try another

\[
\begin{align*}
S & \Rightarrow ( A ) & \text{Backtrack and try again} \\
S & \Rightarrow ( A ) \Rightarrow ( AA ) & \text{Try } A \Rightarrow AA \\
S & \Rightarrow ( A ) \Rightarrow ( AA ) \Rightarrow ( abA ) \Rightarrow ( abA ) \Rightarrow ( abba ) \\
& \Rightarrow ( abba ) \Rightarrow ( abba )
\end{align*}
\]
Backtracking Problems

- Backtracking may in general require that many production applications be reversed, not just one - sometimes must backtrack all the way to the start symbol and beginning of input.

- As we backtrack, we eventually must try all of the other possible choices at each level of the grammar - a given input symbol may match the beginning of many possible productions, making backtracking exponentially expensive in general.

- Some (recursive) grammars may involve an unbounded number of possible productions for some leading inputs.

- Top down parsing is (of course) not normally used in this general form (although sometimes it is - e.g. in source code transformation systems such as TXL, ANTLR and COLM).
Recursive Descent Parsers

- A simple implementation of top-down parsers involves implementing each nonterminal directly as a recursive boolean function.

\[
\begin{align*}
S & \rightarrow 1 \ B \ 0 \\
& \quad | \ 0 \ B \ 1
\end{align*}
\]

\[
\begin{align*}
B & \rightarrow 10 \\
& \quad | \ 11
\end{align*}
\]

```plaintext
function S : boolean
\begin{align*}
& \text{if (next = "1") then } \% 1B0 \\
& \quad \text{advance} \\
& \quad \text{if } B \text{ then} \\
& \quad \quad \text{if next = "0" then} \\
& \quad \quad \quad \text{advance} \\
& \quad \quad \text{return true} \\
& \quad \text{end if} \\
& \text{end if} \\
& \text{elsif next= "0" then } \% 0B1 \\
& \quad \text{advance} \\
& \quad \text{if } B \text{ then} \\
& \quad \quad \text{if next = "1" then} \\
& \quad \quad \quad \text{advance} \\
& \quad \quad \text{return true} \\
& \quad \text{end if} \\
& \text{end if} \\
& \text{return false}
end S
\end{align*}
```

```plaintext
function B : boolean
\begin{align*}
& \text{const save := pointer} \\
& \text{if next = "1" then } \% 10 \\
& \quad \text{advance} \\
& \quad \text{if next = "0" then} \\
& \quad \quad \text{advance} \\
& \quad \text{return true} \\
& \text{end if} \\
& \text{elsif next= "0" then } \% 0B1 \\
& \quad \text{advance} \\
& \quad \text{if } B \text{ then} \\
& \quad \quad \text{if next = "1" then} \\
& \quad \quad \quad \text{advance} \\
& \quad \quad \text{return true} \\
& \quad \text{end if} \\
& \text{end if} \\
& \quad \text{pointer := save} \quad \% \text{backup} \\
& \quad \text{if next = "1" then } \% 11 \\
& \quad \quad \text{advance} \\
& \quad \quad \text{if next = "1" then} \\
& \quad \quad \quad \text{advance} \\
& \quad \quad \text{return true} \\
& \quad \text{end if} \\
& \quad \text{pointer := save} \quad \% \text{backup} \\
& \text{return false}
end B
\end{align*}
```
Problems with Top-down Parsers

• *Left recursion* in the grammar causes problems for top down parsers

  \[ E \rightarrow E + T \]

  \[ E \rightarrow T \]

• In a recursive descent implementation this would result in the infinite recursion

  \[ \text{function } E : \text{ if } E \text{ then } \ldots \]

• As with shift-reduce LR parsers, this situation can be resolved by changing the grammar to adapt to limitations of the method

  \[ E \rightarrow T \ E' \]

  \[ E' \rightarrow + \ T \ E' \]

  \[ E' \rightarrow \varepsilon \]

• More generally, for any *direct* left recursion, we replace

  \[ A \rightarrow A \ X \]

  \[ A \rightarrow Y \ A' \]

  \[ A' \rightarrow \varepsilon \]

• *Indirect* left recursion has a more complex solution
Avoiding Backtracking

• Besides being **inefficient** in making bad guesses, backtracking also has the practical difficulty that any **output** of the parser must be undone as well as the input - not that easy

• So in general if top-down recursive descent parsing is to be **practical**, we must **avoid backtracking**

• **Deterministic** recursive descent parsing occurs when there is no possibility of backtracking
Avoiding Backtracking

• We achieve this by limiting the grammar

• For each nonterminal $A$, for each legal leading input string $X$ of $A$, there must be a unique $A_i$ in the right hand sides for $A$

$$A \rightarrow A_1$$

$$| \quad A_2 \text{ such that } A_i \Rightarrow^* XY$$

$$| \quad A_3 \text{ where } X \in T^*, \ Y \in (N \cup T)^*$$

$$\ldots$$

$$| \quad A_n$$

• In other words, if we are guessing which production of $A$ to use when the remaining input begins with the string of symbols $X$, there’s only one possibility

• Note that the string $X$ need not be directly in the production, only derived by it
Practical Recursive Descent Parsers

• A practical recursive descent parser that implements grammars with this limitation is called a **deterministic recursive descent** parser

• This is a very common parsing method used in parsers for **scripting language** interpreters and other “lightweight” language implementations

• We can think of **SL** in this way (although as we’ll see the recognition power of **SL** is not limited to this language class)
LL Parsers

• A class of grammars that meets the deterministic top-down limitation is called the LL grammars
  (Left-to-right scan of input, Leftmost derivation)

• If the maximum length of the leading terminal strings $X$ in a grammar meeting the limitation is $k$ symbols, then we have an $LL(k)$ grammar

• If the $X$’s are each a single terminal symbol (i.e. we can decide for certain which production to apply next by looking at only the next input token) then we have an $LL(1)$ grammar

• $LL(k)$ languages are a subset of the $LR(k)$ languages
SL Parsers

- **SL**, the pure parsing subset of S/SL, is a lot like **LL(1)** because each choice can depend only on a **single** next input symbol.

- However, **SL** also has **rule choices**, which **LL(1)** parsers do not.

- This gives **SL** parsers the power to parse languages that are not **LL(k)** languages.

```
AssignmentOrLabel:
  @Variable
    [@ColonOrAssign       ColonOrAssign >> Boolean :
      true:                `:'
        @Expression
      | *:
    ];
```

```
ColonOrAssign >> Boolean :
  `:'
    [   `=:
      true
      | *:
        false
    ];
```
Language Class of SL Parsers

- Rule choices increase the parsing power of SL to handle the same set of languages as LR(k) - that is, more than LL(k), and all of the deterministic context-free languages.

- This does not imply grammar equivalence - in each case, the grammar must be structured to meet the constraints of the parsing method (LR(k), LL(k), SL).

- There is a simple constructive proof (Barnard & Cordy 1988) that SL ↔ LR(k), based on a previous proof that LR(k) ↔ LR(1) and the translation of LR(1) transition matrices to SL programs.
SL Parsers

- Advantages of **SL** parsers:
  - Efficient
  - Easy to modify
  - Transparent parse algorithm
  - Excellent syntax error recovery

- Disadvantages:
  - Not completely automated
  - **BNF** grammars not used directly
Summary

Top-down Parsers

- Top-down parsers attempt to build the parse tree for the input by guessing which production should be applied next based on looking at the next few input symbols.
- May have to backtrack when guess later turns out to be wrong.
- Practical deterministic recursive descent top-down parsers solve this problem by limiting grammars to those where a correct guess can always be made (the LL(k) grammars).
- SL is like LL(1), but like LR(k) can handle all deterministic context-free languages.

Next Time

- Constructing parsers in SL.
- Syntax error recovery and repair.