

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

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:

$$\begin{array}{l}
 S \Rightarrow (A) \\
 | \quad SS \\
 \\
 A \Rightarrow ab \\
 | \quad ba \\
 | \quad AA
 \end{array}$$

and input string (abba)

- Starting with S , we predicted that we needed:

$$S \Rightarrow (A) \Rightarrow (ab) \Rightarrow (ab) \quad \textit{Oops! Maybe not ...}$$

- But the $A \Rightarrow ab$ guess didn't work, so **backtrack** to try another

$$S \Rightarrow (A) \quad \textit{Backtrack and try again}$$

$$S \Rightarrow (A) \Rightarrow (AA) \quad \textit{Try } A \Rightarrow AA$$

$$\begin{aligned}
 S \Rightarrow (A) \Rightarrow (AA) \Rightarrow (abA) \Rightarrow (abA) \Rightarrow (abba) \\
 \Rightarrow (abba) \Rightarrow (abba)
 \end{aligned}$$

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**

$$\begin{array}{l} S \rightarrow 1 B 0 \\ \quad | 0 B 1 \end{array}$$
$$\begin{array}{l} B \rightarrow 10 \\ \quad | 11 \end{array}$$

```
function S : boolean
  if (next = "1") then      % 1B0
    advance
    if B then
      if next = "0" then
        advance
        return true
      end if
    end if
  elseif next = "0" then    % 0B1
    advance
    if B then
      if next = "1" then
        advance
        return true
      end if
    end if
  end if
  return false
end S
```

```
function B : boolean
  const save := pointer
  if next = "1" then      % 10
    advance
    if next = "0" then
      advance
      return true
    end if
  end if
  pointer := save        % backup
  if next = "1" then      % 11
    advance
    if next = "1" then
      advance
      return true
    end if
  end if
  pointer := save        % backup
  return false
end B
```

Problems with Top-down Parsers

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

$$\begin{array}{l} E \rightarrow E + T \\ | T \end{array}$$

- In a recursive descent implementation this would result in the infinite recursion **function E : if E then ...**
- As with shift-reduce LR parsers, this situation can be resolved by changing the **grammar** to adapt to **limitations** of the method

$$\begin{array}{l} E \rightarrow T E' \\ | T E' \end{array} \qquad \begin{array}{l} E' \rightarrow + T E' \\ | \epsilon \end{array}$$

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

$$\begin{array}{l} A \rightarrow A X \\ | Y \end{array} \qquad \text{with} \qquad \begin{array}{l} A \rightarrow Y A' \\ A' \rightarrow X A' \\ | \epsilon \end{array}$$

- *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

$$\begin{array}{l} A \rightarrow A_1 \\ | A_2 \\ | A_3 \\ \dots \\ | A_n \end{array} \quad \begin{array}{l} \text{such that } A_i \Rightarrow^* XY \\ \text{where } X \in T^*, Y \in (N \cup T)^* \end{array}$$

- 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
(**L**eft-to-right scan of input, **L**eftmost 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
```

```
  | 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 $SL \Leftrightarrow LR(k)$, based on a previous proof that $LR(k) \Leftrightarrow LR(1)$ 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