Parsing Top-down

- Goal: construct a leftmost derivation of string while reading in token stream
- Partially derived String

<table>
<thead>
<tr>
<th>Lookahead</th>
<th>Parsed Part</th>
<th>Unparsed Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>(1+2+3+4)+5</td>
<td></td>
</tr>
<tr>
<td>→ E+S</td>
<td></td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (S)+S</td>
<td>1</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (E+S)+S</td>
<td>1</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+S)+S</td>
<td>2</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+E+S)+S</td>
<td>2</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+2+S)+S</td>
<td>(</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+2+E)+S</td>
<td>(</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+2+S)+S</td>
<td>3</td>
<td>(1+2+3+4)+5</td>
</tr>
<tr>
<td>→ (1+2+(E+S))+S</td>
<td>3</td>
<td>(1+2+3+4)+5</td>
</tr>
</tbody>
</table>

Problem

- Want to decide which production to apply based on next symbol

- (1) $S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$
- (1)+2 $S \rightarrow E+S \rightarrow (S) + S \rightarrow (E) + S \rightarrow (1)+E \rightarrow (1)+2$

- Why is this hard?

Making a grammar LL(1)

$S \rightarrow E + S$
$S \rightarrow E$
$E \rightarrow \text{num}$
$E \rightarrow (S)$

- Problem: can't decide which $S$ production to apply until we see symbol after first expression

- Left factoring: Factor common $S$ prefix, add new non-terminal $S'$ at decision point. $S'$ derives $(+E)^*$

Parsing with new grammar

<table>
<thead>
<tr>
<th>S</th>
<th>S'</th>
<th>E</th>
<th>E \rightarrow \text{num}</th>
<th>(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES'</td>
<td>ε</td>
<td>+ S</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(E+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+E+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+E+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+S) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+(E+S)) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+(3+S)) S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(1+2+(3+E) S') S'</td>
<td>(1+2+3+4)+5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar is Problematic

- This grammar cannot be parsed top-down with only a single look-ahead symbol
- Not LL(1)
- Left-to-right-scanning, Left-most derivation, 1 look-ahead symbol
- Is it LL(k) for some k?
- Can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language
Predictive Parsing

- **LL(1) grammar:**
  - for a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - uses predictive parsing
- driven by *predictive parsing table* of non-terminals × input symbols → productions

**Predictive Parse Table**

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(  )</th>
<th>EOF($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES'</td>
<td>+S</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>S'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
</tbody>
</table>

**Recursive-Descend Parser**

```java
void parse_S() {  
  switch (token) {  
    case num: parse_E(); parse_S(); return;  
    case '(': token = input.read(); parse_S(); return;  
    case EOF: return;  
    default: throw new ParseError();  
  }  
}
```

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(  )</th>
<th>EOF($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES'</td>
<td>+S</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>S'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
</tbody>
</table>

**Recursive-Descend Parser**

```java
void parse_E() {  
  switch (token) {  
    case number: token = input.read(); return;  
    case '(': token = input.read(); parse_S();  
    if (token != ')') throw new ParseError();  
    token = input.read(); return;  
    default: throw new ParseError();  
  }  
}
```

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(  )</th>
<th>EOF($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES'</td>
<td>+S</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>S'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>num</td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
</tbody>
</table>
9/9/2013

Call Tree = Parse Tree

(1 + 2 + (3 + 4)) + 5

parse S

parse E parse S'

parse S parse S

parse E parse S'

parse S parse S

parse E parse S'

parse S parse S

CS 4120 Introduction to Compilers

13

How to Construct Parsing Tables

• Needed: algorithm for automatically generating a predictive parse table from a grammar

S → E S' + S

E → num | (S)

Constructing Parse Tables

• Can construct predictive parser if:
  – For every non-terminal, every look-ahead symbol can be handled by at most one production
  • FIRST(γ) for arbitrary string of terminals and non-terms γ is:
    • set of symbols that might begin a fully expanded version of γ
  • FOLLOW(X) for a non-terminal X is:
    • set of symbols that might follow the derivation of X in the input stream

Computing nullable and FIRST

• X is nullable if it can derive the empty string:
  • if it derives ε directly (X → ε)
  • if it has a production X → YZ... where all RHS symbols (Y, Z) are nullable
  • Algorithm: Assume all non-terminals non-nullable, apply rules repeatedly until no change in status

Computing FOLLOW

• FOLLOW(X) for a non-terminal X is:
  • set of symbols that might follow the derivation of X in the input stream

Example

• nullable
  • only S’ is nullable

S → ES' + S
S' → ε | + S
E → num | (S)

First

– FIRST(ES') = { num, ( ) }  
– FIRST(S') = { + }  
– FIRST(num) = { num }  
– FIRST( ( ) ) = { ( ) }  
– FIRST(S) = { + }  

Follow

– FOLLOW(S) = { $, ) }  
– FOLLOW(S') = { $, ) }  
– FOLLOW(E) = { +, ) }
### Parse Table Entries

- Consider a production $X \rightarrow \gamma$
- Add $\gamma$ to the $X$ row for each symbol in FIRST($\gamma$)
- If $\gamma$ can derive $\varepsilon$ ($\gamma$ is nullable), add $\gamma$ for each symbol in FOLLOW($X$)
- Grammar is LL(1) if no conflicting entries

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>num, S + S, S * S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Ambiguous grammars

- Construction of predictive parse table for ambiguous grammar results in conflicts (but converse does not hold)

\[
S \rightarrow S + S \mid S * S \mid \text{num}
\]

FIRST($S + S$) = FIRST($S * S$) = FIRST($\text{num}$) = \{ num \}

### Completing the Parser

- Now we know how to construct a recursive-descent parser for an LL(1) grammar.
- LL(k) generalizes this to k lookahead tokens.
- LL(k) parser generators can be used to automate the process (e.g. ANTLR)
- Can we use recursive descent to build an abstract syntax tree too?

### Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- parse_S, parse_S', parse_E all return an Expr:
  - void parse_E() => Expr parse_E()
  - void parse_S() => Expr parse_S()
  - void parse_S'() => Expr parse_S'()
**AST creation code**

```
Expr parse_E() {
    switch(token) {
        case num: // E → number
            Expr result = Num(token.value);
            token = input.read(); return result;
        case '(': // E → ( S )
            token = input.read();
            Expr result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
        default: throw new ParseError();
    }
}
```

```
S → E S'  
S' → e | + S  
E → num | ( S )
```

**parse_S**

```
Expr parse_S() {
    switch (token) {
        case number:
            int result = token.value;
            token = input.read(); return result;
        case '(':
            token = input.read();
            int result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
            default: throw new ParseError();}
}
```

```
Or…an Interpreter!
```

```
int parse_E() {
    switch(token) {
        case number:
            int result = token.value;
            token = input.read(); return result;
        case '(': // E → ( S )
            token = input.read();
            int result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
        default: throw new ParseError();
    }
}
```

```
int parse_S() {
    switch (token) {
        case number:
            int left = parse_E();
            int right = parse_S();
            if (right == 0) return left;
            else return left + right;
        case '+':
            int left = parse_E();
            int right = parse_S();
            if (right == 0) return left;
            else return left + right;
        default: throw new ParseError();
    }
}
```

**Summary**

- We can build a recursive-descent parser for LL(1) grammars
  - Make parsing table from FIRST, FOLLOW sets
  - Translate to recursive-descent code
  - Instrument with abstract syntax tree creation
- Systematic approach avoids errors, detects ambiguities
- Next time: converting a grammar to LL(1) form, bottom-up parsing