Top Down Parsing

- Builds syntax tree as leftmost derivation
  - Like preorder traversal (nodes visited before leaves)
- Two basic methods
  - Predictive parsing uses lookahead to guess structure below a node
  - Backtracking backs up when choice is wrong
    - Expensive – exponential time for general case
- Both methods inherently weak, but predictive recursive descent parsers are relatively easy to hand code

Recursive Descent Parsing

- Grammar is the high level design of the code
- Procedural model – parser is collection of functions
  - Each non-terminal corresponds to a function
    - Function is collection of cases corresponding to rules
    - How to choose among cases?
  - Each rule for the non-terminal corresponds to code:
    - Each terminal in rule is matched (and consumed)
    - Each non-terminal means to call the corresponding function
Example of parser function

Grammar rule:

\[ \text{factor} \rightarrow ( \text{exp} ) \mid \text{number} \]

Code:

```c
void factor() {
    if (token == number)
        match(number);
    else {
        match('(');
        exp();
        match(')');
    }
}
```

Lookahead

- Lookahead was not a problem in this example
  - If next token is expected, consume it
  - Otherwise, error

```c
void match(Token expect){
    if (token == expect) getToken();
    else error(token, expect);
}
```
How to build the syntax tree?

- The parser functions can return a tree (or some representation)

```
Exp factor() {
    Exp tmp;
    if (token == number) {
        tmp = matchNumber();
    } else {
        match('(');
        tmp = exp();
        match(')');
    }
    return tmp;
}
```

Errors During Parsing

- Deep recursion makes errors tricky to handle
  - Need to gracefully exit from nested function calls
  - Best done with exceptions
- Parser should continue
  - Various schemes to synchronize
  - Want to avoid cascading errors
Another rule and parser function

Grammar rule:

\[
exp \rightarrow exp + term \mid term
\]

Code:

```c
void exp() {
    if (token == ???) {
        exp(),
        match(PLUS);
        term();
    } else {
        term();
    }
}
```

How do we choose which rule?

Even worse – this is infinite recursion

Left recursion doesn’t work with recursive descent

Solution: use EBNF

Rewrite

\[
exp \rightarrow exp + term \mid term
\]

With EBNF:

\[
exp \rightarrow \text{term} \{ + \text{term} \}
\]

- \{ \} indicates repetition (zero or more)
  - Applies to general form \( A \rightarrow A \alpha \mid \beta \)
  - Rewrite in EBNF as \( A \rightarrow \beta \{ \alpha \} \)
Write code from EBNF

Grammar rule in EBNF:
\[ exp \rightarrow term \{ + \ term \} \]

Code:
```c
void exp() {
  term();
  while (token == PLUS) {
    match(PLUS);
    term();
  }
}
```

Repetition and EBNF

- EBNF is useful and very similar to code
- Use care to preserve associativity (no longer top down)

```c
Exp exp() {
  tmp = term();
  while (token == PLUS) {
    match(PLUS);
    tmp = PlusExp(tmp, term());
  }
  return tmp;
}
```
**LL(1) Parsing**

- Left to right scan, Leftmost derivation, one symbol of lookahead
- Uses a parsing stack
  - Begins with start symbol on stack
- Two actions
  - Replace nonterminal at top of stack with a grammar rule choice for it (*generate*)
  - Match a token at top of stack with input and pop it (*match*)

---

**LL(1) Parsing Notes**

- $ is used to mark bottom of stack
  - Also used at end of input
- In generate action, right hand side of rule is pushed on stack in reverse order
- In match action, matching token is popped and input token is consumed
- Accept state is reached when stack is empty and input is all consumed
Simple LL(1) Example

Balanced parentheses: \[ S \rightarrow ( S ) S | \epsilon \]

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ S \rightarrow ( S ) S $</td>
<td>( ) $</td>
<td>$ S \rightarrow ( S ) S $</td>
</tr>
<tr>
<td>$ S ) \rightarrow ( ) $</td>
<td>( ) $</td>
<td>match</td>
</tr>
<tr>
<td>$ S ) \rightarrow ) $</td>
<td>) $</td>
<td>$ S \rightarrow \epsilon $</td>
</tr>
<tr>
<td>$ S ) \rightarrow ) $</td>
<td>) $</td>
<td>match</td>
</tr>
<tr>
<td>$ S \rightarrow \epsilon $</td>
<td>$</td>
<td>$ S \rightarrow \epsilon $</td>
</tr>
<tr>
<td>$ \epsilon $</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

LL(1) Parsing Table

- But how do we know which rule to choose?
- Use a table indexed by nonterminals and tokens
  - Token is the lookahead
- Table for balanced parentheses example:

<table>
<thead>
<tr>
<th>M[A,a]</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → ( S ) S</td>
<td>S → \epsilon</td>
<td>S → \epsilon</td>
</tr>
</tbody>
</table>
LL(1) Parsing Table

- Building the table (roughly)
  - If rule produces string with token at beginning, add rule for that token
  - If rule can lead to empty string, add rule to tokens that can appear in a derivation after
- Need computation of First and Follow sets
- Table should not have multiple entries in one slot – this is what defines an LL(1) grammar

First and Follow Sets

- Informally, the First set of a symbol is the terminals that can occur at the beginning of a production of the symbol
- The Follow set is the set of terminals that can occur immediately after the symbol in a production
- Used to build the parsing table
First Sets

- Define First(X) where X is a grammar symbol:
  - If X is a terminal or ε, then First(X) is just { X }
  - If X is a non-terminal, then for each rule
    \[ X \rightarrow X_1 \ X_2 \ldots \ X_n \]
    First(X) contains First(X_1) (but not ε).
    If all of First(X_i) contain ε for i < m, then First(X) contains First(X_m) (but not ε).
    If all First(X_i) contain ε, then so does First(X).

- Algorithm: Iterate over non-terminals to compute First sets until nothing changes

Nullable symbols

- A non-terminal is called **nullable** if there is a derivation of it that leads to ε
- These are the symbols that can “disappear”
- A non-terminal is nullable exactly when its First set contains ε
Example of First set computation

Expression grammar:

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]
\[ \text{addop} \rightarrow + \mid - \]
\[ \text{term} \rightarrow \text{term mulop factor} \mid \text{factor} \]
\[ \text{mulop} \rightarrow * \]
\[ \text{factor} \rightarrow ( \text{exp} ) \mid \text{number} \]

First(set)

- First(exp) = \{ , number \}
- First(addop) = \{ +, - \}
- First(term) = \{ , number \}
- First(mulop) = \{ * \}
- First(factor) = \{ , number \}

Follow Sets

- Define Follow(A) where A is a non-terminal:
  - If A is the start symbol, then $\$ \$ is in Follow(A).
  - If there is a rule \[ \text{B} \rightarrow \alpha \ A \ \beta \]
    then Follow(A) contains First(β) (but not $\epsilon$).
  - If there is a rule \[ \text{B} \rightarrow \alpha \ A \ \beta \quad \text{where } \epsilon \text{ is in First(β)} \]
    then Follow(A) contains Follow(B).
- Algorithm: add to sets until no change
Example of Follow set computation

Expression grammar:

- \( exp \rightarrow exp \ addop \ term \ | \ term \)
- \( addop \rightarrow + \ | \ - \)
- \( term \rightarrow term \ mulop \ factor \ | \ factor \)
- \( mulop \rightarrow * \)
- \( factor \rightarrow ( \ exp ) \ | \ number \)

Follow(\( exp \)) = \{ $, +, -, ) \}
Follow(\( addop \)) = \{ , number \}
Follow(\( term \)) = \{ $, +, -, *, ) \}
Follow(\( mulop \)) = \{ , number \}
Follow(\( factor \)) = \{ $, +, -, *, ) \}

Construction of LL(1) table

- Use the first and follow sets to populate table.
- Algorithm
  - For each rule \( A \rightarrow \sigma \):
    - For each token \( a \) in First(\( \sigma \)), add the rule to \( M[ A, a ] \).
    - If \( \epsilon \) is in First(\( \sigma \)), then for each token \( a \) in Follow(\( A \)), add the rule to \( M[ A, a ] \).
LL(1) Table

```
exp  \rightarrow exp\ addop\ term \mid term
addop \rightarrow + \mid -
term  \rightarrow term\ mulop\ factor \mid factor
mulop \rightarrow *
factor \rightarrow (\ exp\ ) \mid number
```

First(exp) = \{ (, number \}\)
First(addop) = \{ +, - \}
First(term) = \{ (, number \}\)
First(mulop) = \{ * \}
First(factor) = \{ (, number \}\)

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>Two rules!</td>
<td>exp \rightarrow exp\ addop\ term</td>
<td>exp \rightarrow exp\ addop\ term</td>
<td>exp \rightarrow exp\ addop\ term</td>
<td>exp \rightarrow term</td>
<td></td>
</tr>
</tbody>
</table>

Grammar is not LL(1)! But we knew that because of left recursion.

Left Recursion Removal

- Immediate left recursion, e.g.,
  
  \[ exp \rightarrow exp\ +\ term \mid term \]

- General form is
  
  \[ A \rightarrow A\ \alpha \mid \beta \]

- Remove by introducing new non-terminal
  
  \[ A \rightarrow \beta\ A_1 \]
  
  \[ A_1 \rightarrow \alpha\ A_1 \mid \epsilon \]

- E.g.,
  
  \[ exp \rightarrow term\ exp_1 \]
  
  \[ exp_1 \rightarrow +\ term\ exp_1 \mid \epsilon \]
General Left Recursion Removal

- Multiple rules beginning with same non-terminal generalize easily (still immediate recursion)
- Harder case is indirect recursion
  - Use algorithm to remove recursion
  - Requires that there are no \( \epsilon \) productions and no cycles
- Left recursion removal changes grammars and parse trees
  - May affect associativity

Left Factoring

- If two rules begin with same string, LL(1) parsing cannot choose
  \[
  \text{list} \rightarrow \text{item} \text{, list} | \text{item}
  \]
- Solution is to factor out common prefix
  \[
  \text{list} \rightarrow \text{item list}_1 \\
  \text{list}_1 \rightarrow , \text{list} | \epsilon
  \]
- May obscure semantics
LL(1) Example

- Start with the expression grammar
  \[ E \rightarrow E + T \mid T \]
  \[ T \rightarrow T * F \mid F \]
  \[ F \rightarrow (E) \mid \text{num} \]
- Remove left recursion
  \[ E \rightarrow T E_1 \]
  \[ E_1 \rightarrow + T E_1 \mid \varepsilon \]
  \[ T \rightarrow F T_1 \]
  \[ T_1 \rightarrow * F T_1 \mid \varepsilon \]
  \[ F \rightarrow (E) \mid \text{num} \]

LL(1) Example continued

- Compute First sets
  \[ \text{First}(E) = \{ (, \text{num} \} \]
  \[ \text{First}(T) = \{ (, \text{num} \} \]
  \[ \text{First}(F) = \{ (, \text{num} \} \]
  \[ \text{First}(E_1) = \{ +, \varepsilon \} \]
  \[ \text{First}(T_1) = \{ *, \varepsilon \} \]
- Compute Follow sets
  \[ \text{Follow}(E) = \{\$, \varepsilon \} \]
  \[ \text{Follow}(T) = \{+, \$, \varepsilon \} \]
  \[ \text{Follow}(F) = \{ *, +, $, \} \]
  \[ \text{Follow}(E_1) = \{\$, \varepsilon \} \]
  \[ \text{Follow}(T_1) = \{+, \$, \} \]
LL(1) Example continued

- And the table, proving grammar is LL(1)

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E→TE₁</td>
<td></td>
<td></td>
<td>E→TE₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E₁</td>
<td>E₁→+TE₁</td>
<td></td>
<td></td>
<td>E₁→ε</td>
<td>E₁→ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T→FT₁</td>
<td></td>
<td></td>
<td>T→FT₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>T₁→ε</td>
<td>T₁→*FT₁</td>
<td>T₁→ε</td>
<td>T₁→ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F→num</td>
<td></td>
<td></td>
<td>F→(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dangling Else First Sets

Expression grammar:

\[
\begin{align*}
stmt & \rightarrow if-stmt \mid other \\
if-stmt & \rightarrow if \ exp \ stmt \ else-part \\
else-part & \rightarrow else \ stmt \mid \epsilon \\
exp & \rightarrow true \mid false
\end{align*}
\]

First(stmt) = { other, if }  
First(if-stmt) = { if } 
First(else-part) = { else, ε } 
First(exp) = { true, false }
### Dangling Else Follow sets

Expression grammar:

- \( stmt \rightarrow if-stmt \mid other \)
- \( if-stmt \rightarrow if \ exp \ stmt \ else-part \)
- \( else-part \rightarrow else \ stmt \mid \epsilon \)
- \( exp \rightarrow true \mid false \)

<table>
<thead>
<tr>
<th>Follow(stmt)</th>
<th>Follow(if-stmt)</th>
<th>Follow(exp)</th>
<th>Follow(else-part)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ $, else }</td>
<td>{ $, else }</td>
<td>{ other, if }</td>
<td>{ $, else }</td>
</tr>
</tbody>
</table>

### Dangling Else LL(1) Table

<table>
<thead>
<tr>
<th></th>
<th>if</th>
<th>other</th>
<th>else</th>
<th>true</th>
<th>false</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt</td>
<td>stmt → if-stmt</td>
<td>stmt → other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if-stmt</td>
<td>if-stmt → if exp stmt else-part</td>
<td></td>
<td>First(if-stmt) = { if }</td>
<td>Follow(if-stmt) = { $, else }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else-part</td>
<td>else-part → else stmt</td>
<td></td>
<td>First(else-part) = { else, \epsilon }</td>
<td>Follow(else-part) = { $, else }</td>
<td>Follow(exp) = { true, false }</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td></td>
<td>exp → true</td>
<td>exp → false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar is ambiguous, but could always choose first rule to implement most closely nested.
Other concerns

- Can generalize to more lookahead symbols
  - LL(k) grammars and parsers
  - Parsing table much larger, languages uncommon
- Error recovery
  - Panic mode
  - Synchronizing sets