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
  - And there are techniques to get around some of the problems
- We will also look at general LL predictive parsing

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:

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

Code:

```c
void factor() {
    if (token == \text{number})
        match(\text{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 subtree or leaf (or some representation)

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

All functions return nodes in tree. OO approach with inheritance is useful.

A syntax tree is built (not a parse tree)

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();
    }
}
```

Spring 2012

CIS 461/561 - Top down and LL Parsing

Solution: use EBNF

Rewrite

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

With EBNF:

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

- \{ \} 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:

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

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;
}
```

Each successive + is a parent tree to the child built so far – expands to left and gives left associativity.
LL(1) Parsing

- Left to right scan, Leftmost derivation, one symbol of lookahead
- Uses an explicit 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: \[ S \rightarrow ( S ) S \mid \epsilon \]

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

Corresponds to leftmost derivation of ()

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>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S \rightarrow ( S ) S )</td>
<td>( S \rightarrow \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 $\varepsilon$, 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 $\varepsilon$).
  - If all of First($X_i$) contain $\varepsilon$ for $i < m$, then First(X) contains First($X_m$) (but not $\varepsilon$).
  - If all First($X_i$) contain $\varepsilon$, then so does First(X).

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

Example of First set computation

Expression grammar:

\[
\begin{align*}
exp & \rightarrow \ exp \ addop \ term \ | \ term \\
addop & \rightarrow + \ | \ - \\
term & \rightarrow \ term \ mulop \ factor \ | \ factor \\
mulop & \rightarrow * \\
factor & \rightarrow ( \ exp ) \ | \ number
\end{align*}
\]

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
    \[ B \rightarrow \alpha A \beta \]
    then Follow(A) contains First(\(\beta\)) (but not \(\varepsilon\)).
  - If there is a rule
    \[ B \rightarrow \alpha A \beta \text{ where } \varepsilon \text{ is in First}(\beta) \]
    then Follow(A) contains Follow(B).
- Algorithm: add to sets until no change

Example of Follow set computation

Expression grammar:

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

\[
\begin{align*}
\text{First}(\exp, \text{ term}, \text{ factor}) &= \{ (, \text{ number} \} \\
\text{First}(\text{ addop}) &= \{ +, - \} \\
\text{First}(\text{ mulop}) &= \{ * \}
\end{align*}
\]

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

Construction of LL(1) table

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

- If there are no conflicts in the table (double entries) then the grammar is LL(1)

---

LL(1) Table

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

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

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>number</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{exp} )</td>
<td></td>
<td></td>
<td></td>
<td>Two rules!</td>
<td></td>
<td>exp ( \rightarrow ) exp addop term</td>
<td>exp ( \rightarrow ) term</td>
</tr>
<tr>
<td>( \text{exp} )</td>
<td>exp ( \rightarrow ) exp addop term</td>
<td>exp ( \rightarrow ) exp addop term</td>
<td>exp ( \rightarrow ) term</td>
<td></td>
<td></td>
<td></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.,
  \[ \text{exp} \rightarrow \text{exp} + \text{term} \mid \text{term} \]
- General form is
  \[ A \rightarrow A \alpha \mid \beta \]
  - Remove by introducing new non-terminal
    \[ A \rightarrow \beta A_1 \]
    \[ A_1 \rightarrow A_1 \alpha \mid \varepsilon \]
  - Applied to expression rule:
    \[ \text{exp} \rightarrow \text{term} \text{exp}_1 \]
    \[ \text{exp}_1 \rightarrow + \text{term} \text{exp}_1 \mid \varepsilon \]
    - Generates \( \beta \alpha^n \)
    - Also generates \( \beta \alpha^n \) (with right recursion)

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 \( \varepsilon \) productions and no cycles
- Left recursion removal changes grammars and parse trees, but not the language
  - May affect associativity
  - Complicates syntax tree construction
**Left Factoring**

- If two rules begin with same string, LL(1) parsing cannot choose
  
  $$\text{list} \rightarrow \text{item, list } \mid \text{item}$$

- Solution is to factor out common prefix
  
  $$\text{list} \rightarrow \text{item list}_1$$
  
  $$\text{list}_1 \rightarrow , \text{list } \mid \varepsilon$$

- May obscure semantics

---

**Nullable symbols**

- A non-terminal is called **nullable** if there is a derivation of it that leads to $\varepsilon$

- These are the symbols that can “disappear”

- A non-terminal is nullable exactly when its First set contains $\varepsilon$
LL(1) Example

• Start with the expression grammar
  \[ E \rightarrow E + T \mid T \]
  \[ T \rightarrow T \ast F \mid F \]
  \[ F \rightarrow ( E ) \mid \text{num} \]

• Remove left recursion
  \[ E \rightarrow TE_1 \]
  \[ E_1 \rightarrow + TE_1 \mid \varepsilon \]
  \[ T \rightarrow FT_1 \]
  \[ T_1 \rightarrow * FT_1 \mid \varepsilon \]
  \[ F \rightarrow ( E ) \mid \text{num} \]

LL(1) Example continued

\[ E \rightarrow TE_1 \]
\[ E_1 \rightarrow + TE_1 \mid \varepsilon \]
\[ T \rightarrow FT_1 \]
\[ T_1 \rightarrow * FT_1 \mid \varepsilon \]
\[ F \rightarrow ( E ) \mid \text{num} \]

• 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) = \{ \$ , \} \}
  \[ \text{Follow}(T) = \{ + , \$ , \} \}
  \[ \text{Follow}(F) = \{ * , + , \$ , \} \}
  \[ \text{Follow}(E_1) = \{ \$ , \} \}
  \[ \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 \rightarrow TE_1$</td>
<td>$E \rightarrow TE_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E_1$</td>
<td>$E_1 \rightarrow +TE_1$</td>
<td>$E_1 \rightarrow \varepsilon$</td>
<td>$E_1 \rightarrow \varepsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>$T \rightarrow FT_1$</td>
<td>$T \rightarrow FT_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_1 \rightarrow \varepsilon$</td>
<td>$T_1 \rightarrow FT_1$</td>
<td>$T_1 \rightarrow \varepsilon$</td>
<td>$T_1 \rightarrow \varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>$F \rightarrow \text{num}$</td>
<td>$F \rightarrow (E)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LL(1) Parsing Example: n+n*n

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>n + n * n $</td>
<td>$E \rightarrow TE_1$</td>
</tr>
<tr>
<td>$E, T$</td>
<td>n + n * n $</td>
<td>$T \rightarrow FT_1$</td>
</tr>
<tr>
<td>$E, T, F$</td>
<td>n + n * n $</td>
<td>$F \rightarrow n$</td>
</tr>
<tr>
<td>$E, T, n$</td>
<td>n + n * n $</td>
<td>match</td>
</tr>
<tr>
<td>$E, T_1$</td>
<td>+ n * n $</td>
<td>$T_1 \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>+ n * n $</td>
<td>$E_1 \rightarrow +TE_1$</td>
</tr>
<tr>
<td>$E, T_+ $</td>
<td>+ n * n $</td>
<td>match</td>
</tr>
<tr>
<td>$E, T$</td>
<td>n * n $</td>
<td>$T \rightarrow FT_1$</td>
</tr>
<tr>
<td>$E, T, F$</td>
<td>n * n $</td>
<td>$F \rightarrow n$</td>
</tr>
</tbody>
</table>
### n+n*n Example continued

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1T_1F$</td>
<td>n * n $</td>
<td>F → n</td>
</tr>
<tr>
<td>$E_1T_1n$</td>
<td>n * n $</td>
<td>match</td>
</tr>
<tr>
<td>$E_1T_1$</td>
<td>* n $</td>
<td>$T_1 \rightarrow *FT_1$</td>
</tr>
<tr>
<td>$E_1T_1F*$</td>
<td>* n $</td>
<td>match</td>
</tr>
<tr>
<td>$E_1T_1F$</td>
<td>n $</td>
<td>F → n</td>
</tr>
<tr>
<td>$E_1T_1n$</td>
<td>n $</td>
<td>match</td>
</tr>
<tr>
<td>$E_1T_1$</td>
<td>$</td>
<td>$T_1 \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>$</td>
<td>$E_1 \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

### LL(1) Theorem

- Another way of stating no conflicts in table
  - For each variable in the grammar, the First sets of all the variable’s rules are pairwise disjoint
  - For each nullable variable, its First and Follow sets are disjoint
Dangling Else First Sets

Expression grammar:

\[ stmt \rightarrow \text{if-stmt} \mid \text{other} \]
\[ \text{if-stmt} \rightarrow \text{if} \ \text{exp} \ \text{stmt} \ \text{else-part} \]
\[ \text{else-part} \rightarrow \text{else} \ \text{stmt} \mid \varepsilon \]
\[ \text{exp} \rightarrow \text{true} \mid \text{false} \]

First(\(stmt\)) = \{ \text{other, if} \}
First(\(\text{if-stmt}\)) = \{ \text{if} \}
First(\(\text{else-part}\)) = \{ \text{else, } \varepsilon \}
First(\(\text{exp}\)) = \{ \text{true, false} \}

Dangling Else Follow sets

Expression grammar:

\[ stmt \rightarrow \text{if-stmt} \mid \text{other} \]
\[ \text{if-stmt} \rightarrow \text{if} \ \text{exp} \ \text{stmt} \ \text{else-part} \]
\[ \text{else-part} \rightarrow \text{else} \ \text{stmt} \mid \varepsilon \]
\[ \text{exp} \rightarrow \text{true} \mid \text{false} \]

Follow(\(stmt\)) = \{ \$ , else \}
Follow(\(\text{if-stmt}\)) = \{ \$ , else \}
Follow(\(\text{exp}\)) = \{ \text{other, if} \}
Follow(\(\text{else-part}\)) = \{ \$ , else \}

First(\(stmt\))={\text{other, if}}
First(\(\text{if-stmt}\))={\text{if}}
First(\(\text{else-part}\))={\text{else, }\varepsilon}
First(\(\text{exp}\))={\text{true, false}}
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 exp stmt else-part</td>
<td>First(if-stmt) = { if }</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else-part → else stmt</td>
<td>First(else-part) = { else, ε }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp → true</td>
<td>First(exp) = {true, false}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp → false</td>
<td>First(exp) = {true, false}</td>
<td></td>
<td></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

- Syntax tree more difficult with LL(1) parsers
  - Parsing stack represents predicted derivation
  - Use another “value” stack for tree construction
- Can generalize to more lookahead symbols
  - LL(k) grammars and parsers
  - Parsing table much larger, languages uncommon
- Error recovery
Parser Error Handling

- Try to determine error as soon as possible
  - Before location of actual error is lost
- Upon error, pick a likely place to resume parse
  - Try to parse as much code as possible
  - Don't stop with first error when more may be found
  - Accumulate sets of symbols during parse that are likely restart points (synchronizing sets)
- Avoid error cascade problem
  - One error results in many uninformative error messages
  - Know when missing token okay but avoid getting stuck in infinite loop because no input consumed