MIT 6.035
Top-Down Parsing

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Orientation

- Language specification
  - Lexical structure – regular expressions
  - Syntactic structure – grammar
This Lecture - recursive descent parsers
- Code parser as set of mutually recursive procedures
- Structure of program matches structure of grammar
Starting Point

• Assume lexical analysis has produced a sequence of tokens
  • Each token has a type and value
  • Types correspond to terminals
  • Values to contents of token read in
• Examples
  • Int 549 – integer token with value 549 read in
  • if - if keyword, no need for a value
  • AddOp + - add operator, value +
Basic Approach

• Start with Start symbol
• Build a leftmost derivation
  • If leftmost symbol is nonterminal, choose a production and apply it
  • If leftmost symbol is terminal, match against input
  • If all terminals match, have found a parse!
• Key: find correct productions for nonterminals
Graphical Illustration of Leftmost Derivation

Sentential Form

$\text{NT}_1 \; T_1 \; T_2 \; T_3 \; \text{NT}_2 \; \text{NT}_3$

Apply Production Here

Not Here
 Grammar for Parsing Example

\[
\begin{align*}
\text{Start} & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
\text{Expr} & \rightarrow \text{Expr} - \text{Term} \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Term} \ast \text{Int} \\
\text{Term} & \rightarrow \text{Term} / \text{Int} \\
\text{Term} & \rightarrow \text{Int}
\end{align*}
\]

- Set of tokens is
  \[
  \{ +, -, *, /, \text{Int} \}, \quad \text{where Int} = [0-9][0-9]*
  \]
  For convenience, may represent each Int n token by n
Parsing Example

Parse Tree

Remaining Input
2-2*2

Sentential Form
Start

Current Position in Parse Tree
Parsing Example

Parse Tree

Remaining Input
2-2*2

Sentential Form
Expr

Applied Production
Start → Expr

Current Position in Parse Tree
Parsing Example

Parse Tree

Start

Expr

Expr - Term

Expr → Expr + Term
Expr → Expr - Term
Expr → Term

Remaining Input

2-2*2

Sentential Form

Expr - Term

Applied Production

Expr → Expr - Term
Parsing Example

Parse Tree

Start

Expr

Expr - Term

Term

Expr \rightarrow Expr + Term

Expr \rightarrow Expr - Term

Expr \rightarrow Term

Remaining Input

2 - 2*2

Sentential Form

Term - Term

Applied Production

Expr \rightarrow Term
Parsing Example

Parse Tree

*Start*

```
Expr
  Expr - Term
    Term
      Int
```

Remaining Input

```
2-2*2
```

Sentential Form

```
Int - Term
```

Applied Production

```
Term → Int
```
Parsing Example

Parse Tree

Start

↓

Expr

↓

Expr

- Term

↓

Term

↓

Int 2

Match
Input Token!

Remaining Input

2-2*2

Sentential Form

2 - Term
Parsing Example

Parse Tree

- Start
  - Expr
    - Expr
    - Term
      - Term
      - Int 2

Match Input Token!

Remaining Input

-2*2

Sentential Form

2 - Term
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Int 2

Remaining Input

Match
Input
Token!

2*2

Sentential Form

2 - Term
Parsing Example

Parse Tree

```
Parse Tree

Start

Expr

Expr

Term

Term

Int 2

Remaining Input

2*2

Sentential Form

2 - Term*Int

Applied Production

Term → Term * Int
```
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Term

Int 2

Int

Remaining Input

2\times 2

Sentential Form

2 - \text{Int} \times \text{Int}

Applied Production

\text{Term} \rightarrow \text{Int}
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Term

Int 2

Match
Input
Token!

Remaining Input

2*2

Sentential Form

2 - 2* Int
Parsing Example

Parse Tree

\[
\begin{align*}
\text{Start} & \quad \downarrow \\
\text{Expr} & \quad \downarrow \\
\text{Expr} & \quad - \\
\text{Term} & \quad \downarrow \\
\text{Term} & \quad \downarrow \\
\text{Int 2} & \\
\text{Int 2} & \\
\end{align*}
\]

Match Input Token!

Remainning Input

\[*2\]

Sentential Form

\[2 - 2*\text{Int}\]
Parsing Example

Parse Tree

```
Start
  ↓
Expr
  ↓
Expr - Term
    ↓
Term
    ↓
Term * Int
    ↓
Int
```

Remainig Input

```
2
```

Sentential Form

```
2 - 2* Int
```
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Term

Int 2

Int 2

Int 2

Parse Complete!

Remaining Input

2

Sentential Form

2 - 2*2
Summary

- Three Actions (Mechanisms)
  - Apply production to expand current nonterminal in parse tree
  - Match current terminal (consuming input)
  - Accept the parse as correct
- Parser generates preorder traversal of parse tree
  - visit parents before children
  - visit siblings from left to right
Policy Problem

• Which production to use for each nonterminal?
• Classical Separation of Policy and Mechanism
• One Approach: Backtracking
  • Treat it as a search problem
  • At each choice point, try next alternative
  • If it is clear that current try fails, go back to previous choice and try something different
• General technique for searching
• Used a lot in classical AI and natural language processing (parsing, speech recognition)
Backtracking Example

Parse Tree

Remaining Input
2-2*2

Sentential Form
Start
Backtracking Example

Parse Tree

```
Start

Expr
```
Backtracking Example

Remaining Input
2-2*2

Sentential Form
Expr + Term

Applied Production
Expr → Expr + Term
Backtracking Example

Parse Tree

Remaining Input

2-2*2

Sentential Form

Term + Term

Applied Production

Expr → Term
Backtracking Example

Parse Tree

- Start
  - Expr
    - Expr
      - Term
    - +
  - +
- Term
  - Int

Remaining Input

2 - 2 * 2

Match

Input

Token!

Sentential Form

Int + Term

Applied Production

Term → Int
Backtracking Example

Parse Tree

Start

Expr

Expr

Term

Term

Int 2

Remaining Input

-2*2

Sentential Form

2 - Term

Can’t Match Input Token!

Applied Production

Term → Int
Backtracking Example

Parse Tree

Remaining Input
2-2*2

So
Backtrack!

Sentential Form
Expr

Applied Production
Start → Expr
Backtracking Example

Parse Tree

Start

Expr

- Term

Remaining Input

2-2*2

Sentenceal Form

Expr - Term

Applied Production

Expr → Expr - Term
Backtracking Example

Parse Tree

Remaining Input
2-2*2

Sentential Form
Term - Term

Applied Production
Expr → Term
Backtracking Example

Parse Tree

Remaining Input
2-2*2

Sentential Form
Int - Term

Applied Production
Term → Int
Backtracking Example

Parse Tree

Start

Expr

Expr

Term

Term

Int 2

Match Input Token!

Remaining Input

-2*2

Sentential Form

2 - Term
Backtracking Example

Parse Tree

```
Start
  ↓
Expr
  ↓
Expr
  ↓
Term
  ↓
Int 2
```

Remaining Input

```
2*2
```

Match Input

```
Token!
```

Sentential Form

```
2 - Term
```
Left Recursion + Top-Down Parsing = Infinite Loop

- Example Production: $Term \rightarrow Term * Num$
- Potential parsing steps:
General Search Issues

- Three components
  - Search space (parse trees)
  - Search algorithm (parsing algorithm)
  - Goal to find (parse tree for input program)
- Would like to (but can’t always) ensure that
  - Find goal (hopefully quickly) if it exists
  - Search terminates if it does not
-Handled in various ways in various contexts
  - Finite search space makes it easy
  - Exploration strategies for infinite search space
  - Sometimes one goal more important (model checking)
- For parsing, hack grammar to remove left recursion
Eliminating Left Recursion

• Start with productions of form
  • $A \rightarrow A \alpha$
  • $A \rightarrow \beta$
  • $\alpha, \beta$ sequences of terminals and nonterminals that do not start with $A$

• Repeated application of $A \rightarrow A \alpha$

builds parse tree like this:
Eliminating Left Recursion

- Replacement productions
  - $A \rightarrow A \alpha$  $A \rightarrow \beta R$  $R$ is a new nonterminal
  - $A \rightarrow \beta$  $R \rightarrow \alpha R$
  - $R \rightarrow \varepsilon$  New Parse Tree

Old Parse Tree

```
  A
 /   \
A    A
  /  /  \
\beta \alpha
```

New Parse Tree

```
  A
 /   \\  
β   R
   /  \
  α   R
     / \
 α   ε
```
## Hacked Grammar

<table>
<thead>
<tr>
<th>Original Grammar Fragment</th>
<th>New Grammar Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Term \rightarrow Term \ast Int$</td>
<td>$Term \rightarrow Int \ Term'$</td>
</tr>
<tr>
<td>$Term \rightarrow Term / Int$</td>
<td>$Term' \rightarrow \ast Int \ Term'$</td>
</tr>
<tr>
<td>$Term \rightarrow Int$</td>
<td>$Term' \rightarrow / Int \ Term'$</td>
</tr>
<tr>
<td></td>
<td>$Term' \rightarrow \varepsilon$</td>
</tr>
</tbody>
</table>
Parse Tree Comparisons

Original Grammar

```
Term
 /   \
|    |
Term * Int
 /   \
|    |
Int * Int
```

New Grammar

```
Term
 /   \
|    |
Int Term'
 /   \
|    |
* Int Term'
 /   \
|    |
* Int Term'
 /   \
|    |
ε
```
Eliminating Left Recursion

- Changes search space exploration algorithm
  - Eliminates direct infinite recursion
  - But grammar less intuitive

Sets things up for predictive parsing
Predictive Parsing

- Alternative to backtracking
- Useful for programming languages, which can be designed to make parsing easier
- Basic idea
  - Look ahead in input stream
  - Decide which production to apply based on next tokens in input stream
  - We will use one token of lookahead
Predictive Parsing Example Grammar

\[
\begin{align*}
Start & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Term} \text{Expr}' \\
\text{Expr}' & \rightarrow + \text{Expr}' \\
\text{Expr}' & \rightarrow - \text{Expr}' \\
\text{Expr}' & \rightarrow \varepsilon \\
\text{Term} & \rightarrow \text{Int} \text{Term}' \\
\text{Term}' & \rightarrow * \text{Int} \text{Term}' \\
\text{Term}' & \rightarrow / \text{Int} \text{Term}' \\
\text{Term}' & \rightarrow \varepsilon
\end{align*}
\]
Choice Points

• Assume $Term'$ is current position in parse tree
• Have three possible productions to apply
  
  $Term' \rightarrow * \text{Int} \ Term'$
  
  $Term' \rightarrow / \text{Int} \ Term'$
  
  $Term' \rightarrow \epsilon$

• Use next token to decide
  
  • If next token is $*$, apply $Term' \rightarrow * \text{Int} \ Term'$
  • If next token is $/$, apply $Term' \rightarrow / \text{Int} \ Term'$
  • Otherwise, apply $Term' \rightarrow \epsilon$
Predictive Parsing + Hand Coding = Recursive Descent Parser

• One procedure per nonterminal $NT$
  • Productions $NT \rightarrow \beta_1, \ldots, NT \rightarrow \beta_n$
  • Procedure examines the current input symbol $T$ to determine which production to apply
    • If $T \in \text{First}(\beta_k)$
    • Apply production $k$
    • Consume terminals in $\beta_k$ (check for correct terminal)
      • Recursively call procedures for nonterminals in $\beta_k$
  • Current input symbol stored in global variable token
• Procedures return
  • true if parse succeeds
  • false if parse fails
Example

Boolean Term()
    if (token = Int n) token = NextToken(); return(TermPrime())
    else return(false)

Boolean TermPrime()
    if (token = *)
        token = NextToken();
        if (token = Int n) token = NextToken(); return(TermPrime())
        else return(false)
    else if (token = /)
        token = NextToken();
        if (token = Int n) token = NextToken(); return(TermPrime())
        else return(false)
    else return(true)

\[
\begin{align*}
    \text{Term} & \rightarrow \text{Int} \; \text{Term}' \\
    \text{Term}' & \rightarrow * \; \text{Int} \; \text{Term}' \\
    \text{Term}' & \rightarrow / \; \text{Int} \; \text{Term}' \\
    \text{Term}' & \rightarrow \varepsilon
\end{align*}
\]
Multiple Productions With Same Prefix in RHS

• Example Grammar
  \[ NT \rightarrow \text{if then} \]
  \[ NT \rightarrow \text{if then else} \]

Assume \( NT \) is current position in parse tree, and if is the next token

• Unclear which production to apply
  • Multiple \( k \) such that \( T \in \text{First}(\beta_k) \)
  • \( \text{if} \in \text{First}(\text{if then}) \)
  • \( \text{if} \in \text{First}(\text{if then else}) \)
Solution: Left Factor the Grammar

- New Grammar Factors Common Prefix Into Single Production
  
  \[ NT \rightarrow \text{if then } NT' \]
  
  \[ NT' \rightarrow \text{else} \]
  
  \[ NT' \rightarrow \varepsilon \]

- No choice when next token is if!
- All choices have been unified in one production.
Nonterminals

• What about productions with nonterminals?
  \[ NT \rightarrow NT_1 \alpha_1 \]
  \[ NT \rightarrow NT_2 \alpha_2 \]

• Must choose based on possible first terminals that \( NT_1 \) and \( NT_2 \) can generate

• What if \( NT_1 \) or \( NT_2 \) can generate \( \varepsilon \)?
  • Must choose based on \( \alpha_1 \) and \( \alpha_2 \)
$NT$ derives $\varepsilon$

- Two rules
  - $NT \rightarrow \varepsilon$ implies $NT$ derives $\varepsilon$
  - $NT \rightarrow NT_1 \ldots NT_n$ and for all $1 \leq i \leq n$ $NT_i$ derives $\varepsilon$ implies $NT$ derives $\varepsilon$
Fixed Point Algorithm for Derives $\varepsilon$

for all nonterminals $NT$
  set $NT$ derives $\varepsilon$ to be false
for all productions of the form $NT \rightarrow \varepsilon$
  set $NT$ derives $\varepsilon$ to be true
while (some $NT$ derives $\varepsilon$ changed in last iteration)
  for all productions of the form $NT \rightarrow NT_1 \ldots NT_n$
    if (for all $1 \leq i \leq n$ $NT_i$ derives $\varepsilon$)
      set $NT$ derives $\varepsilon$ to be true
First($\beta$)

- $T \in \text{First}(\beta)$ if $T$ can appear as the first symbol in a derivation starting from $\beta$
  1) $T \in \text{First}(T)$
  2) $\text{First}(S) \subseteq \text{First}(S\beta)$
  3) $NT$ derives $\varepsilon$ implies $\text{First}(\beta) \subseteq \text{First}(NT\beta)$
  4) $NT \rightarrow S\beta$ implies $\text{First}(S\beta) \subseteq \text{First}(NT)$

- Notation
  - $T$ is a terminal, $NT$ is a nonterminal, $S$ is a terminal or nonterminal, and $\beta$ is a sequence of terminals or nonterminals
Rules + Request Generate System of Subset Inclusion Constraints

Grammar

\[ \text{Term}' \rightarrow ^* \text{Int Term}' \]
\[ \text{Term}' \rightarrow / \text{Int Term}' \]
\[ \text{Term}' \rightarrow \varepsilon \]

Request: What is First(\( \text{Term}' \))?

Constraints

First(\( ^* \text{Num Term}' \)) \( \subseteq \) First(\( \text{Term}' \))
First(\( / \text{Num Term}' \)) \( \subseteq \) First(\( \text{Term}' \))
First(\( ^* \)) \( \subseteq \) First(\( ^* \text{Num Term}' \))
First(\( / \)) \( \subseteq \) First(\( / \text{Num Term}' \))
\( ^* \in \) First(\( ^* \))
\( / \in \) First(\( / \))
Constraint Propagation Algorithm

Constraints

First(* Num Term') ⊆ First(Term')
First(/ Num Term') ⊆ First(Term')
First(*) ⊆ First(* Num Term')
First(/) ⊆ First(/ Num Term')
* ∈ First(*)
/ ∈ First(/

Solution

First(Term') = {}
First(* Num Term') = {}
First(/ Num Term') = {}
First(*) = {*
First(/) = {/

Initialize Sets to {}
Propagate Constraints Until Fixed Point
Constraint Propagation Algorithm

Constraints

First( * Num Term’ ) ⊆ First( Term’ )
First( / Num Term’ ) ⊆ First( Term’ )
First( * ) ⊆ First( * Num Term’ )
First( / ) ⊆ First( / Num Term’ )

* ∈ First( * )
/ ∈ First( / )

Solution

First( Term’ ) = {}
First( * Num Term’ ) = {}
First( / Num Term’ ) = {}
First( * ) = { * }
First( / ) = { / }

Grammar

Term’ → * Int Term’
Term’ → / Int Term’
Term’ → ε
Constraint Propagation Algorithm

Constraints

First( * Num Term’) ⊆ First( Term’)
First( / Num Term’) ⊆ First( Term’)
First( * ) ⊆ First( * Num Term’)
First( / ) ⊆ First( / Num Term’)
* ∈ First( * )
/ ∈ First( / )

Solution

First( Term’) = {}
First( * Num Term’) = {* }
First( / Num Term’) = {/ }
First( * ) = {* }
First( / ) = {/ }

Grammar

Term’→ * Int Term’
Term’→ / Int Term’
Term’→ ε
Constraint Propagation Algorithm

Constraints

First( * Num Term’ ) ⊆ First( Term’ )
First( / Num Term’ ) ⊆ First( Term’ )
First( * ) ⊆ First( * Num Term’ )
First( / ) ⊆ First( / Num Term’ )
* ∈ First( * )
/ ∈ First( / )

Solution

First( Term’ ) = {*, / }
First( * Num Term’ ) = { * }
First( / Num Term’ ) = { / }
First( * ) = { * }
First( / ) = { / }

Grammar

Term’ → * Int Term’
Term’ → / Int Term’
Term’ → ε
Constraint Propagation Algorithm

Constraints
First(* Num Term’) ⊆ First(Term’)
First(/ Num Term’) ⊆ First(Term’)
First(*) ⊆ First(* Num Term’)
First(/) ⊆ First(/ Num Term’)
* ∈ First(*)
/ ∈ First(/)

Solution
First(Term’) = {*,/}
First(* Num Term’) = {*}
First(/ Num Term’) = {/}
First(*) = {*}
First(/) = {/}

Grammar
Term’ → * Int Term’
Term’ → / Int Term’
Term’ → ε
Building A Parse Tree

- Have each procedure return the section of the parse tree for the part of the string it parsed
- Use exceptions to make code structure clean
Building Parse Tree In Example

Term()
    if (token = Int n)
        oldToken = token; token = NextToken();
        node = TermPrime();
        if (node == NULL) return oldToken;
    else return(new TermNode(oldToken, node);
    else throw SyntaxError

TermPrime()
    if (token = *) || (token = /)
        first = token; next = NextToken();
        if (next = Int n)
            token = NextToken();
            return(new TermPrimeNode(first, next, TermPrime())
        else throw SyntaxError
    else return(NULL)
Parse Tree for $2 \times 3 \times 4$

Concrete Parse Tree

```
Term
  `-- Term'
    |   `-- Int
    |       |   `-- *
    |       |       `-- Int
    |           |       |   `-- *
    |           |       |       `-- Int
    |           |           |           `-- *
    |           |           |               `-- Int
    |           |           |                   `-- ε
    |           |                   `-- Term'
    `-- Int
      |   `-- *
      |       `-- Int
      `-- Int
```

Desired Abstract Parse Tree

```
Term
  `-- Term
    |   `-- *
    |       `-- Int
    `-- Term
      |   `-- *
      |       `-- Int
      `-- Term
        |   `-- *
        |       `-- Int
        `-- Int
```
Why Use Hand-Coded Parser?

• Why not use parser generator?
• What do you do if your parser doesn’t work?
  • Recursive descent parser – write more code
  • Parser generator
    • Hack grammar
    • But if parser generator doesn’t work, nothing you can do
• If you have complicated grammar
  Increase chance of going outside comfort zone of parser generator
• Your parser may NEVER work
Bottom Line

• Recursive descent parser properties
  Probably more work
• But less risk of a disaster - you can almost always make a recursive descent parser work
• May have easier time dealing with resulting code
  • Single language system
  • No need to deal with potentially flaky parser generator
  • No integration issues with automatically generated code
• If your parser development time is small compared to rest of project, or you have a really complicated language, use hand-coded recursive descent parser
Summary

• Top-Down Parsing
• Use Lookahead to Avoid Backtracking
• Parser is
  Hand-Coded
• Set of Mutually Recursive Procedures
Direct Generation of Abstract Tree

- TermPrime builds an incomplete tree
  - Missing leftmost child
  - Returns root and incomplete node
- \((\text{root}, \text{incomplete}) = \text{TermPrime()}\)
  - Called with token = *
  - Remaining tokens = 3 * 4

Diagram:

```
root ----> Term
|        |        |
|        |        |
incomplete ----> Term * Int
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
| * Int  | 4       |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
|        |        |
```

- Missing Left child to be filled in by caller
Code for Term

Term()
if (token = Int n) ←
  leftmostInt = token; token = NextToken();
  (root, incomplete) = TermPrime();
  if (root == NULL) return leftmostInt;
  incomplete.leftChild = leftmostInt;
  return root;
else throw SyntaxError

Input to parse
2 * 3 * 4
Code for Term

```
Term()
    if (token = Int n)
        leftmostInt = token; token = NextToken(); ←
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    else throw SyntaxError
```

Input to parse

```
2 * 3 * 4
```

```
token → Int
  2
```
Code for Term

Term()
if (token = Int n)
    leftmostInt = token; token = NextToken();
    (root, incomplete) = TermPrime(); ←
    if (root == NULL) return leftmostInt;
    incomplete.leftChild    leftmostInt;
    return root;
else throw SyntaxError

Input to parse

2*3*4

token → Int

2
Code for Term

Term()
if (token = Int n)
    leftmostInt = token; token = NextToken();
    (root, incomplete) = TermPrime();
    if (root == NULL) return leftmostInt;
    incomplete.leftChild = leftmostInt;
    return root;
else throw SyntaxError

Input to parse
2 * 3 * 4

\[
\begin{align*}
\text{leftmostInt} & \rightarrow \text{Int} \\
\text{incomplete} & \rightarrow \text{Term} \ast \text{Int} \\
\text{root} & \rightarrow \text{Term} \\
\end{align*}
\]
Code for Term

Term()
  if (token = Int n)
    leftmostInt nt = token; token = NextToken();
    (root, incomplete) = TermPrime();
    if (root == NULL) return leftmostInt nt;
    incomplete.leftChild leftmostInt nt; ←
    return root;
  else throw SyntaxError

Input to parse
2*3*4

root → Term
  → Term * Int
  → Int 4
incomplete
  → Term * Int
  → Int 2 * Int 3
leftmostInt nt
Code for Term

Term()
   if (token = Int n)
       leftmostInt = token; token = NextToken();
       (root, incomplete) = TermPrime();
       if (root == NULL) return leftmostInt;
       incomplete.leftChild = leftmostInt;
       return root;
   else throw SyntaxError

Input to parse
2*3*4

\[ \text{root} \rightarrow \text{Term} \]
\[ \text{incomplete} \rightarrow \text{Term} \ast \text{Int} \]
\[ \text{leftmostInt} \rightarrow \text{Int} \ast \text{Int} \]
\[ \text{Int} \ast \text{Int} \rightarrow 2 \ast 3 \]
\[ \text{Int} \ast \text{Int} \rightarrow 3 \ast 4 \]
Code for TermPrime

TermPrime()
    if (token == *) || (token == /)
        op = token; next = NextToken();
        if (next == Int n)
            token = NextToken();
            (root, incomplete) = TermPrime();
            if (root == NULL)
                root = new ExprNode(NULL, op, next);
                return (root, root);
            else
                newChild = new ExprNode(NULL, op, next);
                incomplete.leftChild = newChild;
                return(root, newChild);
        else throw SyntaxError
    else return(NULL, NULL)

Missing left child to be filled in by caller
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.