MIT 6.1100
Top-Down Parsing

Martin Rinard
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 +

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)
```

$Term \to$ Int $Term'$
$Term' \to$ * Int $Term'$
$Term' \to$ / Int $Term'$
$Term' \to \varepsilon$

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

$NT_1\ T_1\ T_2\ T_3\ NT_2\ NT_3$

Apply Production Here

Not Here

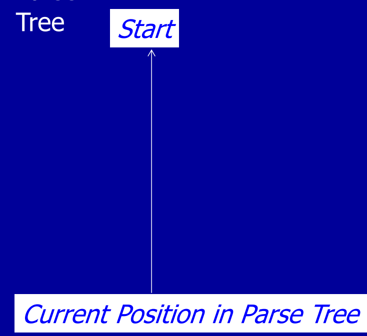
Grammar for Parsing Example

Start → *Expr*
Expr → *Expr* + *Term*
Expr → *Expr* - *Term*
Expr → *Term*
Term → *Term* * Int
Term → *Term* / Int
Term → Int

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

Parsing Example

Parse Tree



Remaining Input

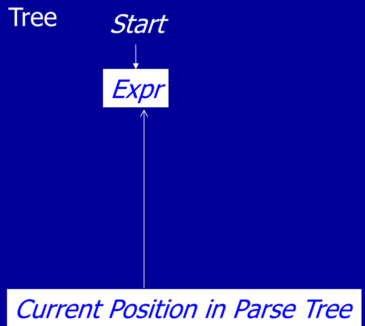
2-2*2

Sentential Form

Start

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

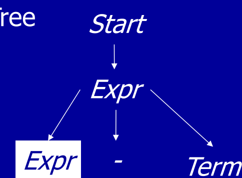
Expr

Applied Production

Start → *Expr*

Parsing Example

Parse Tree



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

Remaining Input

2-2*2

Sentential Form

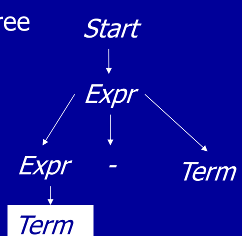
Expr - *Term*

Applied Production

Expr → *Expr* - *Term*

Parsing Example

Parse Tree



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

Remaining Input

2-2*2

Sentential Form

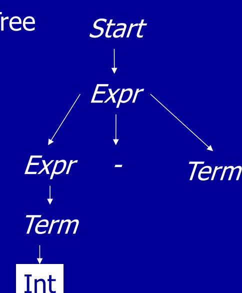
Term - *Term*

Applied Production

Expr → *Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

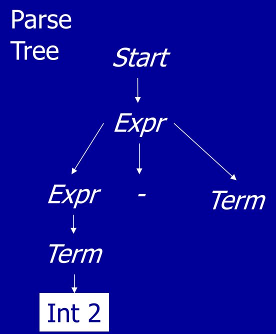
Sentential Form

Int - *Term*

Applied Production

Term → Int

Parsing Example



Match Input Token!

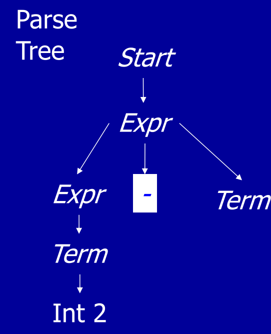
Remaining Input

2-2*2

Sentential Form

2 - *Term*

Parsing Example



Match Input Token!

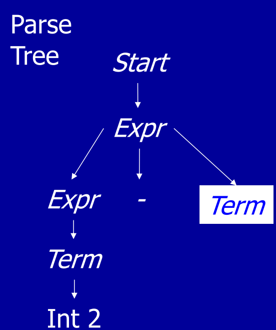
Remaining Input

-2*2

Sentential Form

2 - *Term*

Parsing Example



Match Input Token!

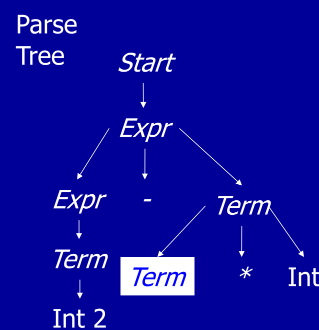
Remaining Input

2*2

Sentential Form

2 - *Term*

Parsing Example



Remaining Input

2*2

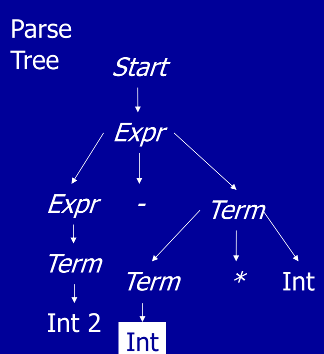
Sentential Form

2 - *Term* * Int

Applied Production

Term → *Term* * Int

Parsing Example



Remaining Input

2*2

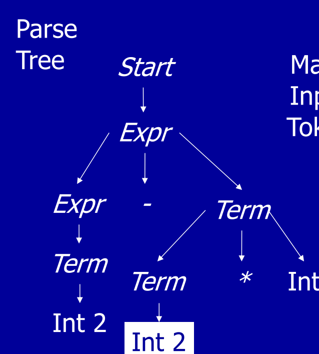
Sentential Form

2 - Int * Int

Applied Production

Term → Int

Parsing Example



Match Input Token!

Remaining Input

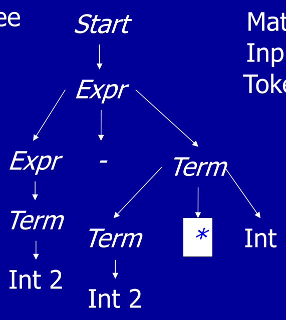
2*2

Sentential Form

2 - 2 * Int

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

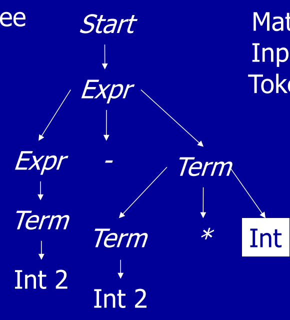
*2

Sentential Form

2 - 2 * Int

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

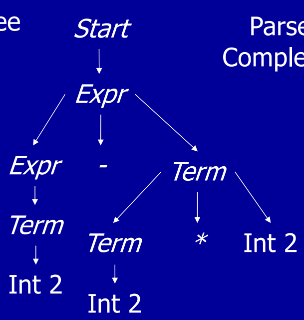
2

Sentential Form

2 - 2 * Int

Parsing Example

Parse Tree



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

Start

Remaining Input

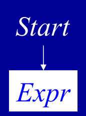
2-2*2

Sentential Form

Start

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

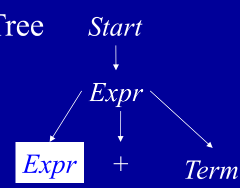
Expr

Applied Production

Start → *Expr*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

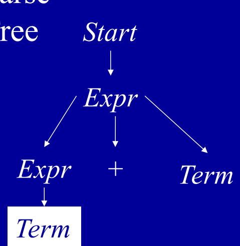
Expr + *Term*

Applied Production

Expr → *Expr* + *Term*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

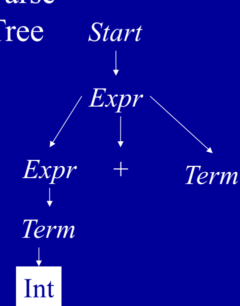
Term + *Term*

Applied Production

Expr → *Term*

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

2-2*2

Sentential Form

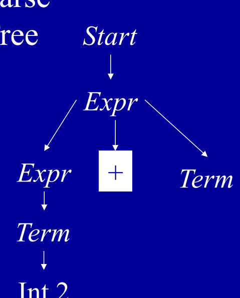
Int + *Term*

Applied Production

Term → Int

Backtracking Example

Parse Tree



Can't Match Input Token!

Remaining Input

-2*2

Sentential Form

2 - *Term*

Applied Production

Term → Int

Backtracking Example

Parse Tree

So Backtrack!

Remaining Input

2-2*2

Sentential Form

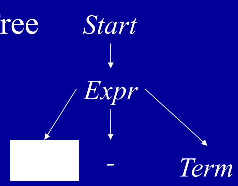
Expr

Applied Production

Start → *Expr*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

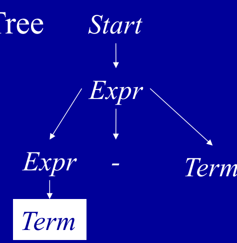
Expr - Term

Applied Production

Expr → *Expr - Term*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

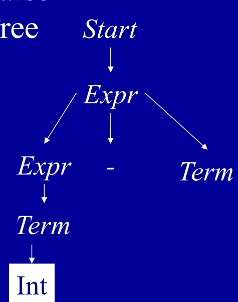
Term - Term

Applied Production

Expr → *Term*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

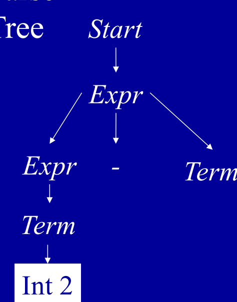
Int - *Term*

Applied Production

Term → Int

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

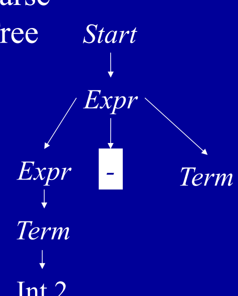
-2*2

Sentential Form

2 - *Term*

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

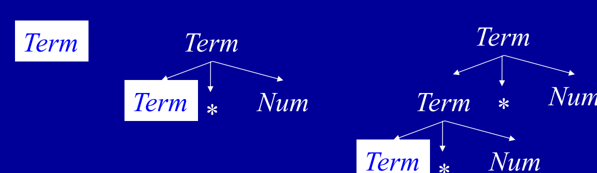
2*2

Sentential Form

2 - *Term*

Left Recursion + Top-Down Parsing = Infinite Loop

- Example Production: *Term* → *Term***Num*
- Potential parsing steps:

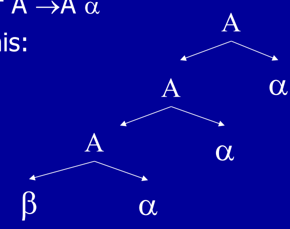


General Search Issues

- 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

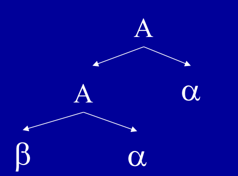
- Start with productions of form
  - $A \rightarrow A\ \alpha$
  - $A \rightarrow \beta$
  - α, β sequences of terminals and nonterminals that do not start with A
- Repeated application of $A \rightarrow A\ \alpha$ builds parse tree like this:



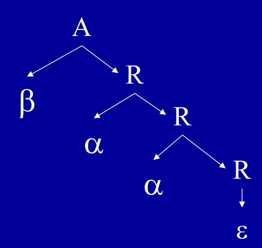
Eliminating Left Recursion

- Replacement productions
  - $A \rightarrow A\ \alpha$ $\quad$ $A \rightarrow \beta\ R$ $\quad$ R is a new nonterminal
  - $A \rightarrow \beta$ $\quad$ $R \rightarrow \alpha\ R$
  - $\quad$ $R \rightarrow \varepsilon$

Old Parse Tree



New Parse Tree



Hacked Grammar

Original Grammar Fragment

$Term \rightarrow Term * \text{Int}$

$Term \rightarrow Term\ /\ \text{Int}$

$Term \rightarrow \text{Int}$

New Grammar Fragment

$Term \rightarrow \text{Int}\ Term'$

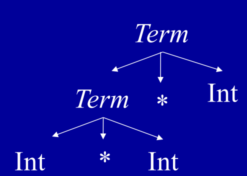
$Term' \rightarrow * \text{Int}\ Term'$

$Term' \rightarrow /\ \text{Int}\ Term'$

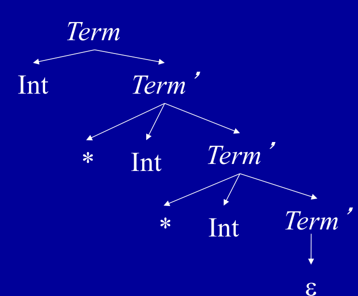
$Term' \rightarrow \varepsilon$

Parse Tree Comparisons

Original Grammar



New Grammar



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

Start → *Expr*
Expr → *Term Expr'*
Expr' → + *Term Expr'*
Expr' → - *Term Expr'*
Expr' → ε

Term → Int *Term'*
Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε

Choice Points

- Assume *Term'* is current position in parse tree
- Have three possible productions to apply
  Term' → * Int *Term'*
  Term' → / Int *Term'*
  Term' → ε
- Use next token to decide
  - If next token is *, apply *Term'* → * Int *Term'*
  - If next token is /, apply *Term'* → / Int *Term'*
  - Otherwise, apply *Term'* → ε

Predictive Parsing + Hand Coding = Recursive Descent Parser

- One procedure per nonterminal *NT*
  - Productions *NT* → β_1 , ..., *NT* → β_n
  - Procedure examines the current input symbol T to determine which production to apply
    - If T∈First(β_k)
    - Apply production k
    - Consume terminals in β_k (check for correct terminal)
    - Recursively call procedures for nonterminals in β_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)
```

Term → Int *Term'*
Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε

Multiple Productions With Same Prefix in RHS

- Example Grammar
  NT → if then
  NT → 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∈First(β_k)
  - if ∈ First(if then)
  - if ∈ First(if then else)

Solution: Left Factor the Grammar

- New Grammar Factors Common Prefix Into Single Production

  $NT \rightarrow$ if then NT'

  $NT' \rightarrow$ 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 ε?
  - Must choose based on α_1 and α_2

NT derives ε

- Two rules
  - $NT \rightarrow \varepsilon$ implies NT derives ε
  - $NT \rightarrow NT_1 \ldots NT_n$ and for all $1 \le i \le n$ NT_i derives ε implies NT derives ε

Fixed Point Algorithm for Derives ε

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

First(β)

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

- Notation
  - T is a terminal, NT is a nonterminal, S is a terminal or nonterminal, and β is a sequence of terminals or nonterminals

Rules + Request Generate System of Subset Inclusion Constraints

Grammar

$Term' \rightarrow$ * Int $Term'$

$Term' \rightarrow$ / Int $Term'$

$Term' \rightarrow \varepsilon$

Rules

1) $T \in$ First(T)
2) First(S) $\subseteq$ First($S\ \beta$)
3) NT derives ε implies First(β) $\subseteq$ First($NT\ \beta$)
4) $NT \rightarrow S\ \beta$ implies First(S β) $\subseteq$ First(NT)

Request: What is First($Term'$)?

Constraints

First(* $Num\ Term'$) $\subseteq$ First($Term'$)

First(/ $Num\ Term'$) $\subseteq$ First($Term'$)

First(*) $\subseteq$ First(* $Num\ Term'$)

First(/) $\subseteq$ First(/ $Num\ Term'$)

* $\in$ 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

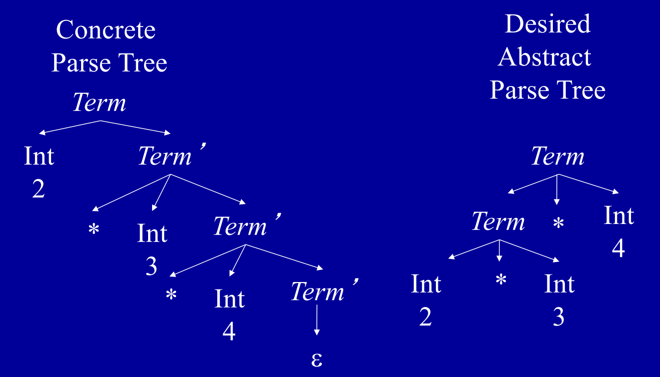
Building A Parse Tree

- 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*3*4



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

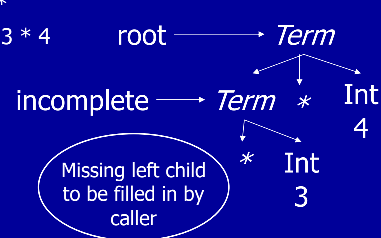
- Recursive descent parser properties
  - Probably more work
  - But less risk of a disaster - you can almost always make a recursive descent parser work
  - May have easier time dealing with resulting code
    - Single language system
    - No need to deal with potentially flaky parser generator
    - 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
- (root, incomplete) = TermPrime()
  - Called with token = *
  - Remaining tokens = 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

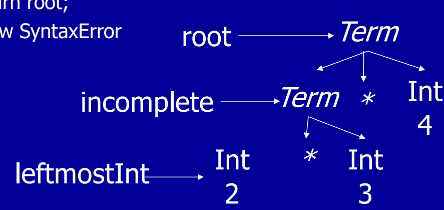
token → Int 2

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

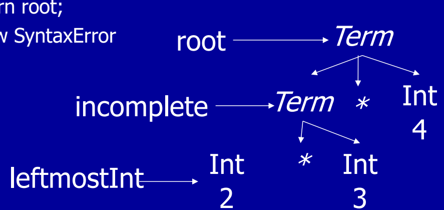


Code for Term

```
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

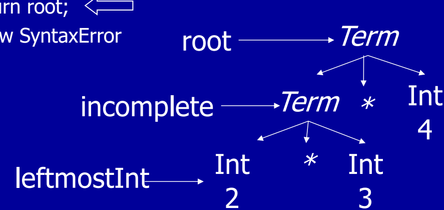


Code for Term

```
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



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