MIT 6.1100
Top-Down Parsing

Martin Rinard
Massachusetts Institute of Technology

1

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

2

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 +

3

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)
```

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

4

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

5

Graphical Illustration of Leftmost Derivation

Sentential Form

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

Apply Production Here

Not Here

6

Grammar for Parsing Example

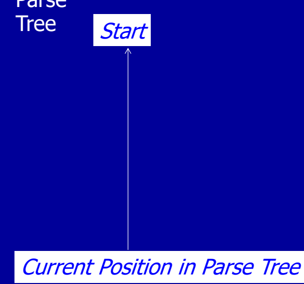
Start → *Expr*
Expr → *Expr + Term*
Expr → *Expr - Term*
Expr → *Term*
Term → *Term ** Int
Term → *Term /* Int
Term → Int

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

7

Parsing Example

Parse Tree



Remaining Input

2-2*2

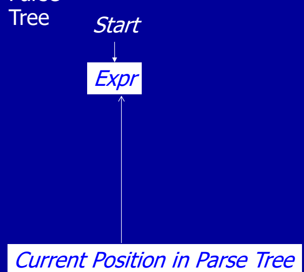
Sentential Form

Start

8

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Expr

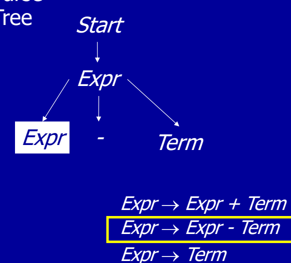
Applied Production

Start → *Expr*

9

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Expr - Term

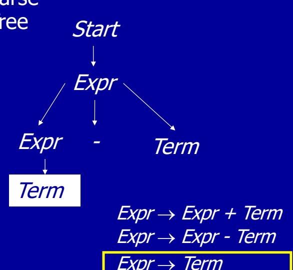
Applied Production

Expr → *Expr - Term*

10

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Term - Term

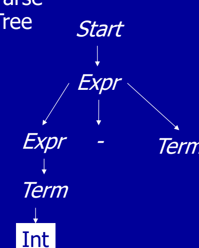
Applied Production

Expr → *Term*

11

Parsing Example

Parse Tree



Remaining Input

2-2*2

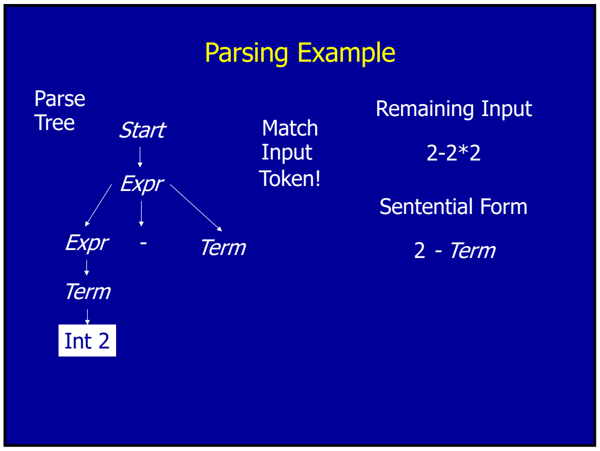
Sentential Form

Int - *Term*

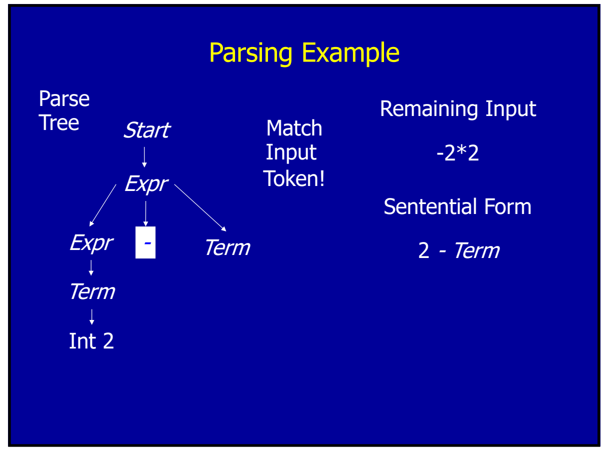
Applied Production

Term → Int

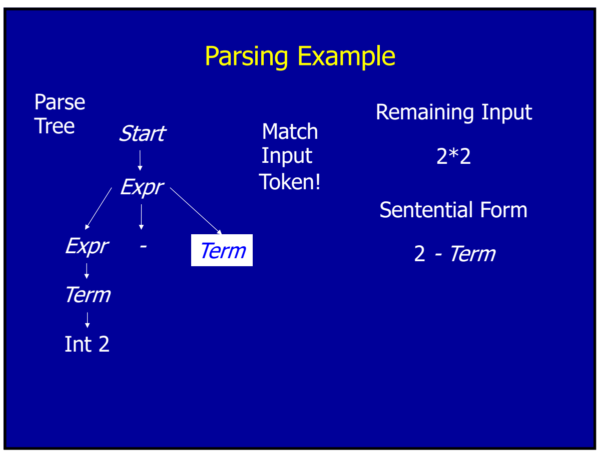
12



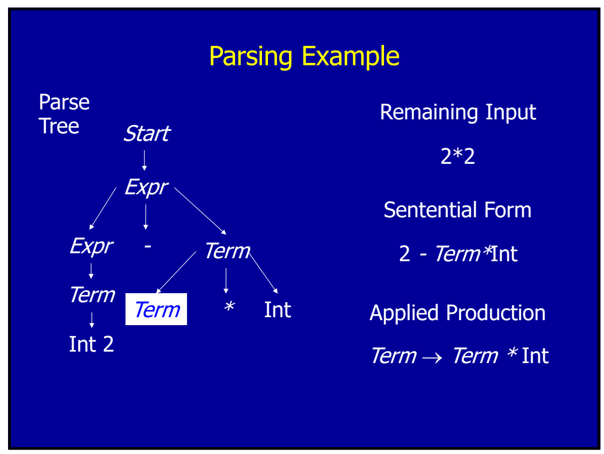
13



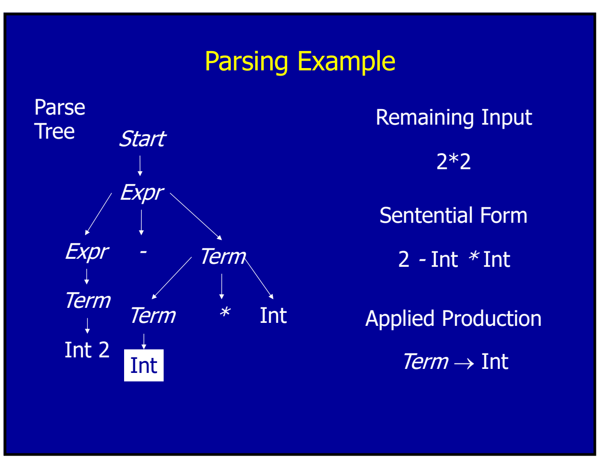
14



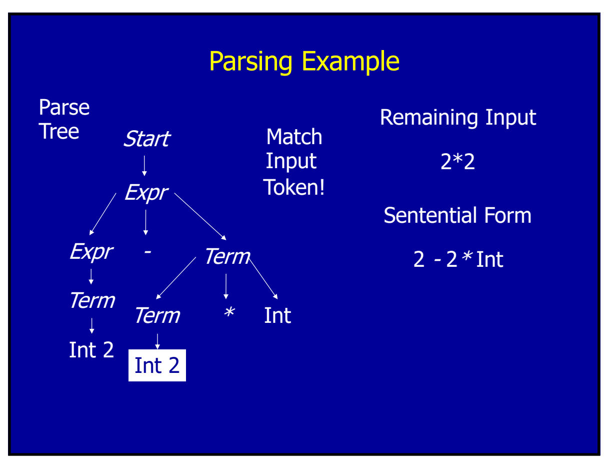
15



16



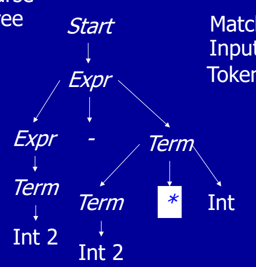
17



18

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

*2

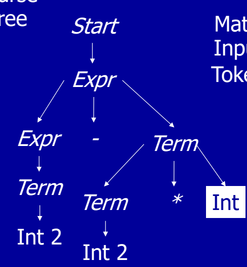
Sentential Form

2 - 2 * Int

19

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

2

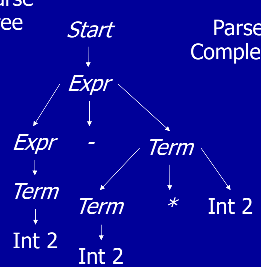
Sentential Form

2 - 2 * Int

20

Parsing Example

Parse Tree



Parse Complete!

Remaining Input

2

Sentential Form

2 - 2 *2

21

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

22

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)

23

Backtracking Example

Parse Tree *Start*

Remaining Input

2-2*2

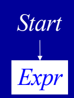
Sentential Form

Start

24

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Expr

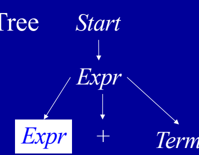
Applied Production

Start → *Expr*

25

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Expr + *Term*

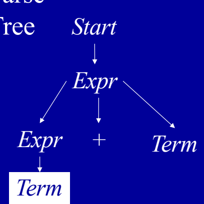
Applied Production

Expr → *Expr* + *Term*

26

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Term + *Term*

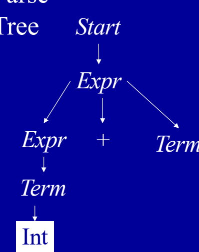
Applied Production

Expr → *Term*

27

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

2-2*2

Sentential Form

Int + *Term*

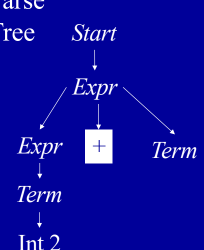
Applied Production

Term → Int

28

Backtracking Example

Parse Tree



Can’t Match Input Token!

Remaining Input

-2*2

Sentential Form

2 - *Term*

Applied Production

Term → Int

29

Backtracking Example

Parse Tree

So Backtrack!

Remaining Input

2-2*2

Sentential Form

Expr

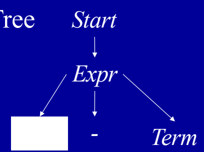
Applied Production

Start → *Expr*

30

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Expr - Term

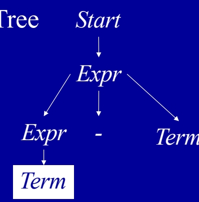
Applied Production

Expr → *Expr - Term*

31

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Term - Term

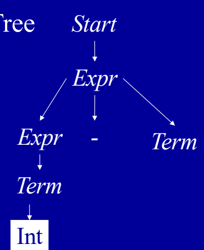
Applied Production

Expr → *Term*

32

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

Int - *Term*

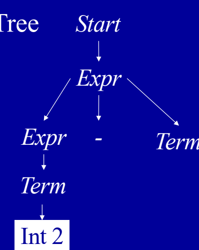
Applied Production

Term → Int

33

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

-2*2

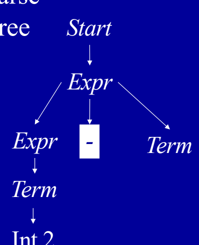
Sentential Form

2 - *Term*

34

Backtracking Example

Parse Tree



Match Input Token!

Remaining Input

2*2

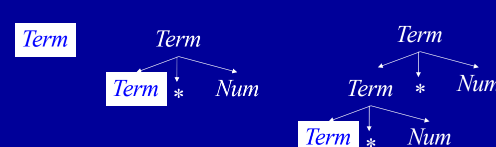
Sentential Form

2 - *Term*

35

Left Recursion + Top-Down Parsing = Infinite Loop

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



36

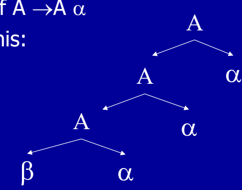
General Search Issues

- 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

37

Eliminating Left Recursion

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



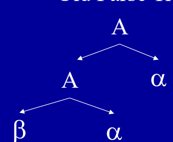
38

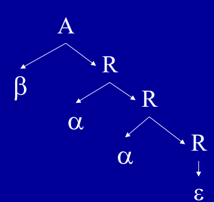
Eliminating Left Recursion

- Replacement productions

– $A \to A\ \alpha$	$A \to \beta\ R$	R is a new nonterminal
– $A \to \beta$	$R \to \alpha\ R$	
–	$R \to \varepsilon$	New Parse Tree

Old Parse Tree





39

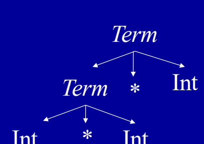
Hacked Grammar

Original Grammar Fragment	New Grammar Fragment
Term → *Term* *Int	*Term* → Int *Term'*
Term → *Term* /Int	*Term'* → *Int *Term'*
Term → Int	*Term'* → /Int *Term'*
	Term' → ε

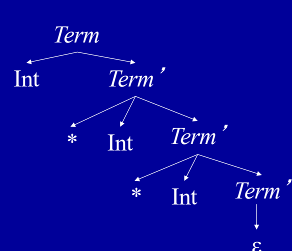
40

Parse Tree Comparisons

Original Grammar



New Grammar



41

Eliminating Left Recursion

- Changes search space exploration algorithm
  - Eliminates direct infinite recursion
  - But grammar less intuitive
- Sets things up for predictive parsing

42

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

43

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

44

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

45

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

46

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

47

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)

48

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.

49

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

50

NT derives ε

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

51

Fixed Point Algorithm for Derives ε

```
for all nonterminals NT
  set NT derives ε to be false
for all productions of the form NT → ε
  set NT derives ε to be true
while (some NT derives ε changed in last iteration)
  for all productions of the form NT → NT1 ... NTn
    if (for all 1≤i≤n NTi derives ε)
      set NT derives ε to be true
```

52

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

53

Rules + Request Generate System of Subset Inclusion Constraints

Grammar

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

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

$Term' \rightarrow \varepsilon$

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

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

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

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

$* \in$ First(*)

$/ \in$ First(/)

54

Constraint Propagation Algorithm

Constraints
First(*Int *Term'*) ⊆ First(*Term'*)
First(/Int *Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(*Int *Term'*)
First(/) ⊆ First(/Int *Term'*)
* ∈First(*)
/ ∈First(/)

Solution
First(*Term'*) = {}
First(*Int *Term'*) = {}
First(/Int *Term'*) = {}
First(*) = {}
First(/) = {}

Initialize Sets to {}

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

55

Constraint Propagation Algorithm

Constraints
First(*Int *Term'*) ⊆ First(*Term'*)
First(/Int *Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(*Int *Term'*)
First(/) ⊆ First(/Int *Term'*)
* ∈First(*)
/ ∈First(/)

Solution
First(*Term'*) = {}
First(*Int *Term'*) = {}
First(/Int *Term'*) = {}
First(*) = {*}
First(/) = {/}

Propagate Constraints Until Fixed Point

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

56

Constraint Propagation Algorithm

Constraints
First(*Int *Term'*) ⊆ First(*Term'*)
First(/Int *Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(*Int *Term'*)
First(/) ⊆ First(/Int *Term'*)
* ∈First(*)
/ ∈First(/)

Solution
First(*Term'*) = {}
First(*Int *Term'*) = {*}
First(/Int *Term'*) = {/}
First(*) = {*}
First(/) = {/}

Propagate Constraints Until Fixed Point

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

57

Constraint Propagation Algorithm

Constraints
First(*Int *Term'*) ⊆ First(*Term'*)
First(/Int *Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(*Int *Term'*)
First(/) ⊆ First(/Int *Term'*)
* ∈First(*)
/ ∈First(/)

Solution
First(*Term'*) = {*,/}
First(*Int *Term'*) = {*}
First(/Int *Term'*) = {/}
First(*) = {*}
First(/) = {/}

Propagate Constraints Until Fixed Point

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

58

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

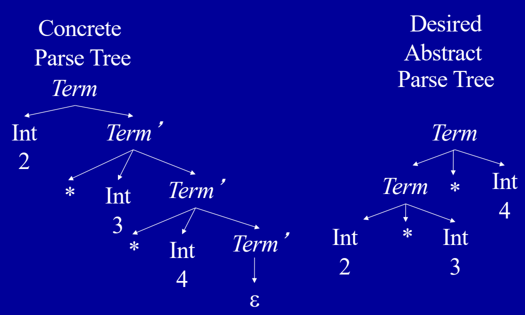
59

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)
```

60

Parse Tree for 2*3*4



61

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

62

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

63

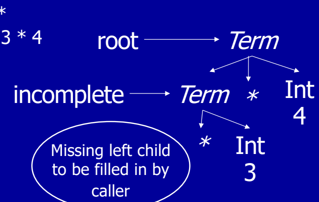
Summary

- Top-Down Parsing
- Use Lookahead to Avoid Backtracking
- Parser is
  - Hand-Coded
  - Set of Mutually Recursive Procedures

64

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



65

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 $\longrightarrow$ Int 2

66

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

67

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

68

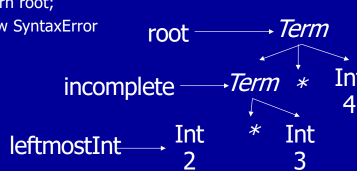
Code for Term

```
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

⇑



69

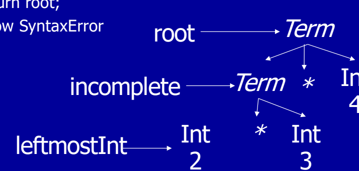
Code for Term

```
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

⇑



70

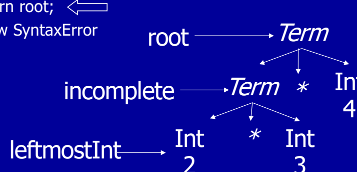
Code for Term

```
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

⇑



71

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

72