

6.035

Fall 2006

Recursive Decent Parser Construction


Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques

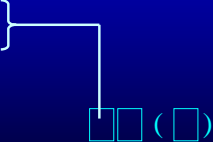
Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques



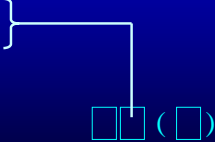
Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques
 - L - parse from left to right
 - R - parse from right to left



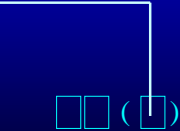
Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques
 - L - leftmost derivation
 - R - rightmost derivation



Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques
 - Number of lookahead characters



Terminology

- Many different parsing techniques
 - Each can handle some set of CFGs
 - Categorization of techniques
 - Examples: LL(0), LR(1)
- Monday: Parse table for a LR(1) parser
 - Automatically generated parser
 - Parser table given in lecture
 - See text if you want to learn about parse table generation
- Today: Building a LL(k) parser
 - Manual construction of a recursive descent parser
 - Code parser as set of mutually recursive procedures
 - Structure of program matches structure of grammar

LR(k)

Source: Adapted from [1], ©MIT Fall 2004

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 +

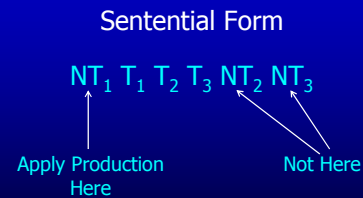
Source: Adapted from [1], ©MIT Fall 2004

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

Source: Adapted from [1], ©MIT Fall 2004

Graphical Illustration of Leftmost Derivation



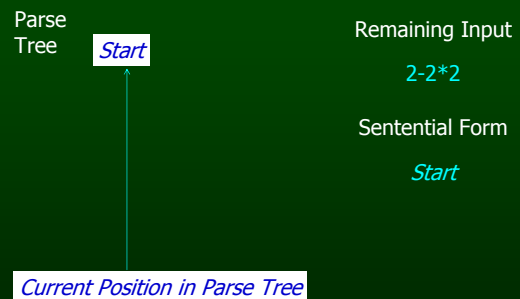
Source: Adapted from [1], ©MIT Fall 2004

Grammar for Parsing Example

- $Start \rightarrow Expr$
 $Expr \rightarrow Expr + Term$
 $Expr \rightarrow Expr - Term$
 $Expr \rightarrow Term$
 $Term \rightarrow Term * Int$
 $Term \rightarrow Term / Int$
 $Term \rightarrow Int$
- Set of tokens is $\{+, -, *, /, Int\}$, where $Int = [0-9][0-9]^*$
 - For convenience, may represent each Int n token by n

Source: Adapted from [1], ©MIT Fall 2004

Parsing Example



Source: Adapted from [1], ©MIT Fall 2004

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr </pre>	<p>Remaining Input</p> <p style="color: cyan;">2-2*2</p> <p>Sentential Form</p> <p style="color: cyan;">Expr</p> <p>Applied Production</p> <p style="color: cyan;">Start → Expr</p>
---	---

Current Position in Parse Tree

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr Expr --> Expr Expr --> Minus["-"] Expr --> Term </pre>	<p>Remaining Input</p> <p style="color: cyan;">2-2*2</p> <p>Sentential Form</p> <p style="color: cyan;">Expr - Term</p> <p>Applied Production</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> <p style="color: cyan; font-size: small;">Expr → Expr + Term</p> <p style="color: cyan; font-size: small;">Expr → Expr - Term</p> <p style="color: cyan; font-size: small;">Expr → Term</p> </div> <p style="color: cyan;">Expr → Expr - Term</p>
--	---

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr Expr --> Expr Expr --> Minus["-"] Expr --> Term Term --> Term </pre>	<p>Remaining Input</p> <p style="color: cyan;">2-2*2</p> <p>Sentential Form</p> <p style="color: cyan;">Term - Term</p> <p>Applied Production</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> <p style="color: cyan; font-size: small;">Expr → Expr + Term</p> <p style="color: cyan; font-size: small;">Expr → Expr - Term</p> <p style="color: cyan; font-size: small;">Expr → Term</p> </div> <p style="color: cyan;">Expr → Term</p>
---	--

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr Expr --> Expr Expr --> Minus["-"] Expr --> Term Term --> Int </pre>	<p>Remaining Input</p> <p style="color: cyan;">2-2*2</p> <p>Sentential Form</p> <p style="color: cyan;">Int - Term</p> <p>Applied Production</p> <p style="color: cyan;">Term → Int</p>
--	---

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr Expr --> Expr Expr --> Minus["-"] Expr --> Term Term --> Int Int --> Int2["2"] </pre>	<p>Remaining Input</p> <p style="color: cyan;">-2*2</p> <p>Sentential Form</p> <p style="color: cyan;">2 - Term</p> <p style="color: red; font-weight: bold;">Match Input Token!</p>
---	--

Parsing Example

<p>Parse Tree</p> <pre> graph TD Start --> Expr Expr --> Expr Expr --> Minus["-"] Expr --> Term Term --> Int Int --> Int2["2"] </pre>	<p>Remaining Input</p> <p style="color: cyan;">2*2</p> <p>Sentential Form</p> <p style="color: cyan;">2 - Term</p> <p style="color: red; font-weight: bold;">Match Input Token!</p>
---	---

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    
```

Remaining Input
2*2

Sentential Form
2 - Term

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    Term1 --> Star[*]
    Term1 --> Int3[Int]
    
```

Remaining Input
2*2

Sentential Form
2 - Term*Int

Applied Production
Term → Term *Int

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    Term1 --> Int3[Int]
    Term1 --> Star[*]
    Term1 --> Int4[Int]
    
```

Remaining Input
2*2

Sentential Form
2 - Int *Int

Applied Production
Term → Int

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    Term1 --> Int3[Int 2]
    Term1 --> Star[*]
    Term1 --> Int4[Int]
    
```

Remaining Input
*2

Sentential Form
2 - 2 *Int

Match Input Token!

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    Term1 --> Term3[Term]
    Term3 --> Int3[Int 2]
    Term1 --> Star[*]
    Term1 --> Int4[Int]
    
```

Remaining Input
2

Sentential Form
2 - 2 *Int

Match Input Token!

Parsing Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int2[Int 2]
    Term1 --> Int3[Int 2]
    Term1 --> Star[*]
    Term1 --> Int4[Int]
    
```

Remaining Input

Sentential Form
2 - 2 *Int

Match Input Token!

Parsing Example

Parse Tree 	Parse Complete!	Remaining Input 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 	Remaining Input $2-2*2$
	Sentential Form $Expr$
	Applied Production $Start \rightarrow Expr$

Backtracking Example

Parse Tree 	Remaining Input $2-2*2$
	Sentential Form $Expr + Term$
	Applied Production $Expr \rightarrow Expr + Term$

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Plus[+]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    
```

Remaining Input
2-2*2

Sentential Form
 $Term + Term$

Applied Production
 $Expr \rightarrow Term$

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Plus[+]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int[Int]
    
```

Remaining Input
2-2*2

Sentential Form
 $Int + Term$

Applied Production
 $Term \rightarrow Int$

Match Input Token!

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Plus[+]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int[Int]
    Int --> Two[2]
    
```

Remaining Input
-2*2

Sentential Form
 $2 - Term$

Applied Production
 $Term \rightarrow Int$

Can't Match Input Token!

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr[Expr]
    
```

Remaining Input
2-2*2

Sentential Form
 $Expr$

Applied Production
 $Start \rightarrow Expr$

So Backtrack!

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    
```

Remaining Input
2-2*2

Sentential Form
 $Expr - Term$

Applied Production
 $Expr \rightarrow Expr - Term$

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Term3[Term]
    
```

Remaining Input
2-2*2

Sentential Form
 $Term - Term$

Applied Production
 $Expr \rightarrow Term$

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int[Int]
  
```

Remaining Input: 2-2*2

Sentential Form: Int - Term

Applied Production: Term → Int

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int[Int]
  
```

Remaining Input: -2*2

Sentential Form: 2 - Term

Match Input Token!

Backtracking Example

Parse Tree

```

graph TD
    Start --> Expr1[Expr]
    Expr1 --> Expr2[Expr]
    Expr1 --> Minus[-]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Term2 --> Int[Int]
  
```

Remaining Input: 2*2

Sentential Form: 2 - Term

Match Input Token!

Left Recursion + Top-Down Parsing = Infinite Loop

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

```

graph TD
    T1[Term] --> T2[Term]
    T1 --> S1[*]
    T1 --> N1[Num]
    T2 --> T3[Term]
    T2 --> S2[*]
    T2 --> N2[Num]
  
```

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:

```

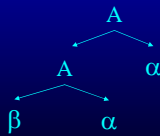
graph TD
    A1[A] --> A2[A]
    A1 --> alpha1[alpha]
    A2 --> A3[A]
    A2 --> alpha2[alpha]
    A3 --> beta[beta]
    A3 --> alpha3[alpha]
  
```

Eliminating Left Recursion

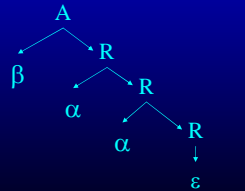
- Replacement productions

- $A \rightarrow A \alpha$ $A \rightarrow \beta R$ R is a new nonterminal
- $A \rightarrow \beta$ $R \rightarrow \alpha R$
- $R \rightarrow \epsilon$

Old Parse Tree



New Parse Tree



Hacked Grammar

Original Grammar
Fragment

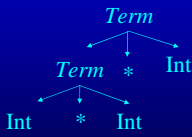
- $Term \rightarrow Term * Int$
- $Term \rightarrow Term / Int$
- $Term \rightarrow Int$

New Grammar
Fragment

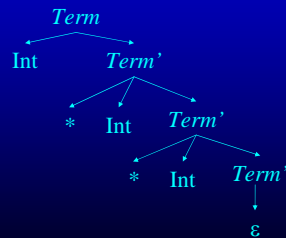
- $Term \rightarrow Int Term'$
- $Term' \rightarrow * Int Term'$
- $Term' \rightarrow / Int Term'$
- $Term' \rightarrow \epsilon$

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

$Start \rightarrow Expr$

$Expr \rightarrow Term Expr'$

$Expr' \rightarrow + Expr'$

$Expr' \rightarrow - Expr'$

$Expr' \rightarrow \epsilon$

$Term \rightarrow Int Term'$

$Term' \rightarrow * Int Term'$

$Term' \rightarrow / Int Term'$

$Term' \rightarrow \epsilon$

Choice Points

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

Source: Adapted from [1]. © MIT Fall 2004.

Predictive Parsing + Hand Coding = Recursive Descent Parser

- One procedure per nonterminal NT
 - Productions $NT \rightarrow \beta_1, \dots, NT \rightarrow \beta_n$
 - 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 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

Source: Adapted from [1]. © MIT Fall 2004.

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)
  
```

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

Source: Adapted from [1]. © MIT Fall 2004.

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 $T \in \text{First}(\beta_k)$
 - $\text{if} \in \text{First}(\text{if then})$
 - $\text{if} \in \text{First}(\text{if then else})$

Source: Adapted from [1]. © MIT Fall 2004.

Solution: Left Factor the Grammar

- New Grammar Factors Common Prefix Into Single Production
 - $NT \rightarrow \text{if then } NT'$
 - $NT' \rightarrow \text{else}$
 - $NT' \rightarrow \epsilon$
- No choice when next token is if !
- All choices have been unified in one production.

Source: Adapted from [1]. © MIT Fall 2004.

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

Source: Adapted from [1]. © MIT Fall 2004.

NT derives ε

- Two rules
 - $NT \rightarrow \epsilon$ implies NT derives ϵ
 - $NT \rightarrow NT_1 \dots NT_n$ and for all $1 \leq i \leq n$ 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 $NT \rightarrow \epsilon$
 set NT derives ϵ to be true
 while (some NT derives ϵ changed in last iteration)
 for all productions of the form $NT \rightarrow NT_1 \dots NT_n$
 if (for all $1 \leq i \leq n$ NT_i derives ϵ)
 set NT derives ϵ to be true

First(β)

- $T \in \text{First}(\beta)$ if T can appear as the first symbol in a derivation starting from β
 - $T \in \text{First}(T)$
 - $\text{First}(S) \subseteq \text{First}(S\beta)$
 - NT derives ϵ implies $\text{First}(\beta) \subseteq \text{First}(NT\beta)$
 - $NT \rightarrow S\beta$ 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 β is a sequence of terminals or nonterminals

Rules + Request Generate System of Subset Inclusion Constraints

Grammar
 $Term' \rightarrow *Int Term'$
 $Term' \rightarrow /Int Term'$
 $Term' \rightarrow \epsilon$

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

Constraints
 $\text{First}(* Num Term') \subseteq \text{First}(Term')$
 $\text{First}(/ Num Term') \subseteq \text{First}(Term')$
 $\text{First}(\ast) \subseteq \text{First}(* Num Term')$
 $\text{First}(/) \subseteq \text{First}(/ Num Term')$
 $\ast \in \text{First}(\ast)$
 $/ \in \text{First}(/)$

Rules

- $T \in \text{First}(T)$
- $\text{First}(S) \subseteq \text{First}(S\beta)$
- NT derives ϵ implies $\text{First}(\beta) \subseteq \text{First}(NT\beta)$
- $NT \rightarrow S\beta$ implies $\text{First}(S\beta) \subseteq \text{First}(NT)$

Constraint Propagation Algorithm

Constraints	Solution
$\text{First}(* Num Term') \subseteq \text{First}(Term')$	$\text{First}(Term') = \{\}$
$\text{First}(/ Num Term') \subseteq \text{First}(Term')$	$\text{First}(* Num Term') = \{\ast\}$
$\text{First}(\ast) \subseteq \text{First}(* Num Term')$	$\text{First}(/ Num Term') = \{/ \}$
$\text{First}(/) \subseteq \text{First}(/ Num Term')$	$\text{First}(\ast) = \{\ast\}$
$\ast \in \text{First}(\ast)$	$\text{First}(/) = \{/ \}$
$/ \in \text{First}(/)$	

Initialize Sets to $\{\}$
 Propagate Constraints Until Fixed Point

Constraint Propagation Algorithm

Constraints	Solution
$\text{First}(* Num Term') \subseteq \text{First}(Term')$	$\text{First}(Term') = \{\ast, /\}$
$\text{First}(/ Num Term') \subseteq \text{First}(Term')$	$\text{First}(* Num Term') = \{\ast\}$
$\text{First}(\ast) \subseteq \text{First}(* Num Term')$	$\text{First}(/ Num Term') = \{/ \}$
$\text{First}(/) \subseteq \text{First}(/ Num Term')$	$\text{First}(\ast) = \{\ast\}$
$\ast \in \text{First}(\ast)$	$\text{First}(/) = \{/ \}$
$/ \in \text{First}(/)$	

Grammar
 $Term' \rightarrow *Int Term'$
 $Term' \rightarrow /Int Term'$
 $Term' \rightarrow \epsilon$

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

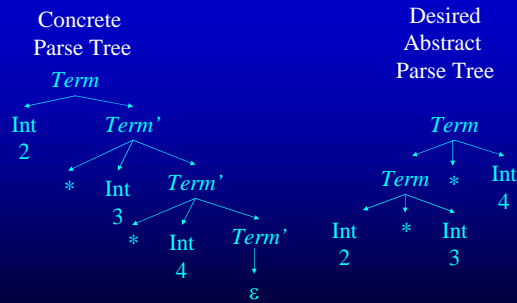
Simon Arnsperth 61 6.035 ©MIT Fall 2005

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)
```

Simon Arnsperth 62 6.035 ©MIT Fall 2005

Parse Tree for 2*3*4



Simon Arnsperth 63 6.035 ©MIT Fall 2005

Why Use Hand-Coded Parser?

- Why not use parser generator?
 - 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
- Can do better error reporting

Simon Arnsperth 64 6.035 ©MIT Fall 2005

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

Simon Arnsperth 65 6.035 ©MIT Fall 2005

Summary

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

Simon Arnsperth 66 6.035 ©MIT Fall 2005

Grammar Example 1

- What is the context free grammar for?
 - $(\text{digit}^+ \text{ dot digit}^*) \mid (\text{digit}^* \text{ dot digit}^+)$
 - Where
 - $\text{digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 - $\text{dot} = .$

6.035 ©MIT Fall 2004

Grammar Example II

- What is the context free grammar for a string with x open wedges followed by $2*x$ close wedges for any $x \geq 0$

6.035 ©MIT Fall 2004

Grammar Example III

- Is this grammar ambiguous?

$E \rightarrow \text{int}$
 $E \rightarrow E B E$
 $B \rightarrow +$
 $B \rightarrow *$

6.035 ©MIT Fall 2004

Grammar Example III

- What is the parse tree for?

$1+2+3$

6.035 ©MIT Fall 2004

Grammar Example III

- What is the parse tree for?

$1+2*3$

6.035 ©MIT Fall 2004

Grammar Example IV

- Hack the grammar to enforce precedence

6.035 ©MIT Fall 2004