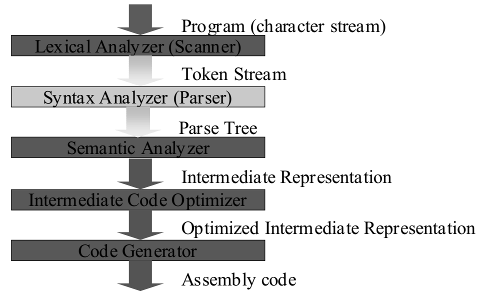
6.035

Fall 2000

Lecture 4: Top Down Parsing

Parsing



Last Lecture

- Focus: Specifying languages
- Generative approach
- Context-Free Grammars
  - Ambiguity
  - Abstract versus Concrete Syntax

This Lecture

- Determining if a program is syntactically correct
- Building a parse tree from a sequence of tokens
- Top-down, recursive descent approach
- Structure of parsing program matches structure of grammar

Basic Approach

- Start with goal 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

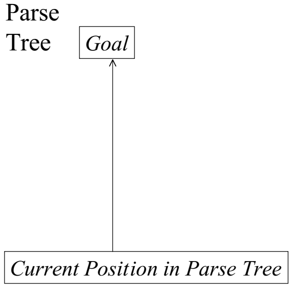
Grammar for Parsing Example

Goal ➔ *Expr*
Expr ➔ *Expr* + *Term*
Expr ➔ *Expr* - *Term*
Expr ➔ *Term*
Term ➔ *Term* * *Num*
Term ➔ *Term* / *Num*
Term ➔ *Num*

Num ➔ 0
Num ➔ 1
Num ➔ 2

Parsing Example

Parse Tree



Remaining Input

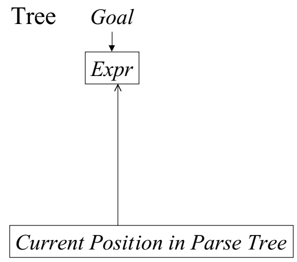
2-2*2

Sentential Form

Goal

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

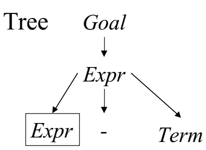
Expr

Applied Production

Goal ➔ *Expr*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

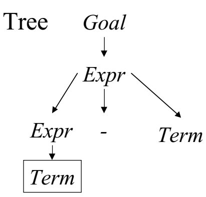
Expr - Term

Applied Production

Expr ➔ *Expr - Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

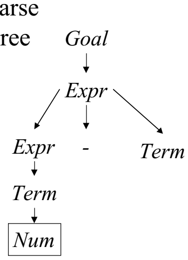
Term - Term

Applied Production

Expr ➔ *Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

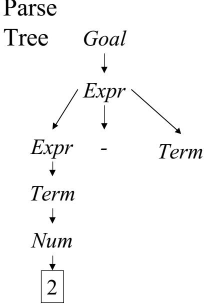
Num - Term

Applied Production

Term ➔ *Num*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

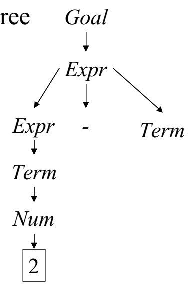
2 - *Term*

Applied Production

Num ➔ 2

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

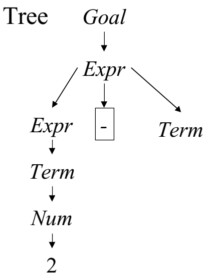
2-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

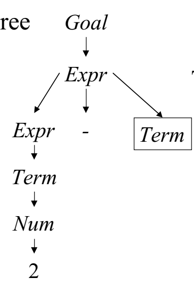
-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

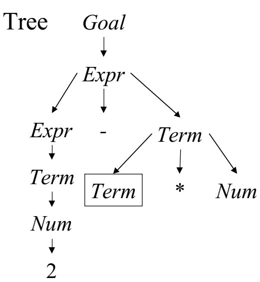
2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Remaining Input

2*2

Sentential Form

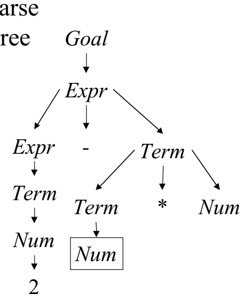
2 - *Term***Num*

Applied Production

Term ➔ *Term* * *Num*

Parsing Example

Parse Tree



Remaining Input

2*2

Sentential Form

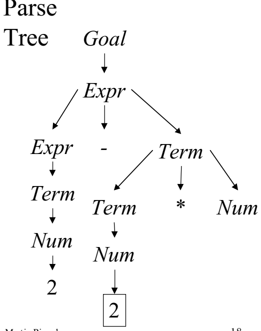
2 - *Num***Num*

Applied Production

Term ➔ *Num*

Parsing Example

Parse Tree



Remaining Input

2*2

Sentential Form

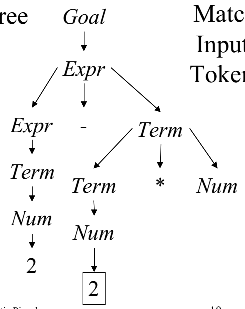
2 - 2**Num*

Applied Production

Num ➔ 2

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

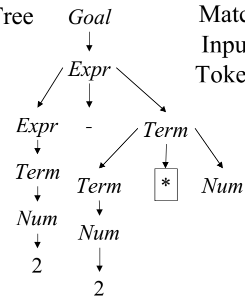
2*2

Sentential Form

2 - 2**Num*

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

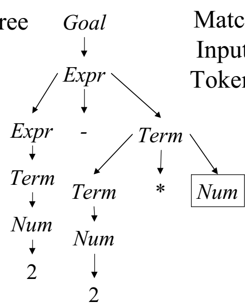
*2

Sentential Form

2 - 2**Num*

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

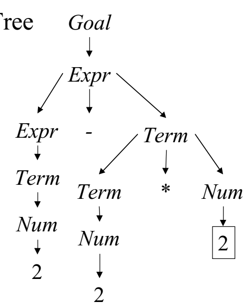
2

Sentential Form

2 - 2**Num*

Parsing Example

Parse Tree



Remaining Input

2

Sentential Form

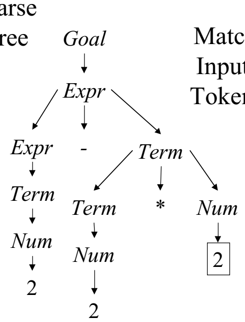
2 - 2*2

Applied Production

Num ➔ 2

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

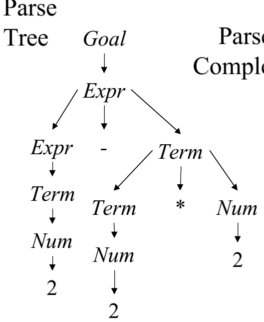
2

Sentential Form

2 - 2*2

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 parsing

Backtracking Example

Parse Tree

Remaining Input

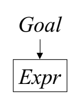
2-2*2

Sentential Form

Goal

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

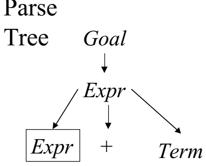
Expr

Applied Production

Goal ➔ *Expr*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

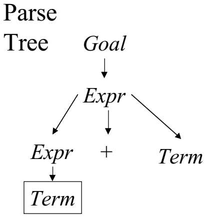
Expr - Term

Applied Production

Expr ➔ *Expr* + *Term*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

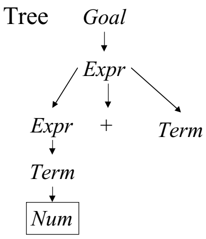
Term - Term

Applied Production

Expr ➔ *Term*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

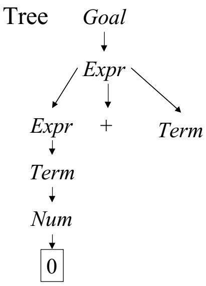
Num - *Term*

Applied Production

Term ➔ *Num*

Backtracking Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

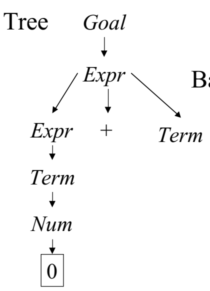
2 - *Term*

Applied Production

Num ➔ 0

Parsing Example

Parse Tree



Match Fails, Backtrack!

Remaining Input

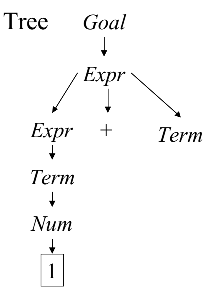
2-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

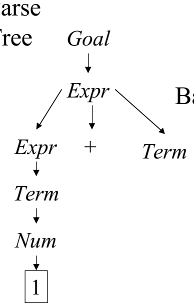
2 - *Term*

Applied Production

Num ➔ 1

Parsing Example

Parse Tree



Match Fails, Backtrack!

Remaining Input

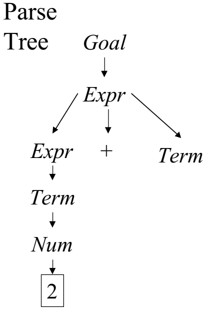
2-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

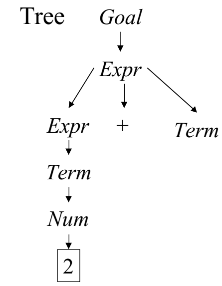
2 - *Term*

Applied Production

Num ➔ 2

Parsing Example

Parse Tree



Match Input Token!

Remaining Input

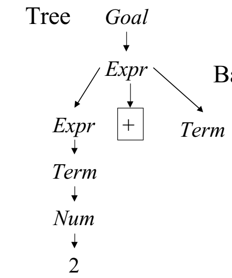
2-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Match Fails, Backtrack!

Remaining Input

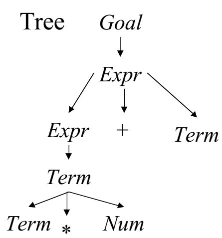
-2*2

Sentential Form

2 - *Term*

Parsing Example

Parse Tree



Remaining Input

2-2*2

Sentential Form

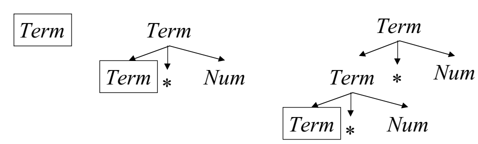
2 - *Term*

Applied Production

Term ➔ *Term***Num*

Left Recursion + Top-Down Parsing = Infinite Loop

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



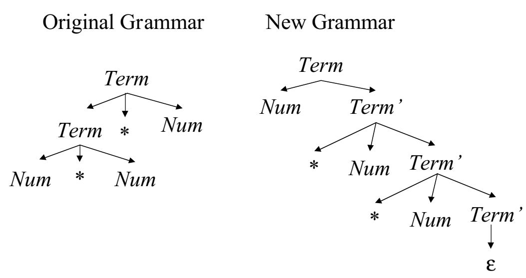
General Problem with Search

- Must ensure that every step makes some sort of meaningful progress
- Handled in various ways in various contexts
- For parsing, hack grammar to remove left recursion

Hacked Grammar

Original Grammar Fragment	New Grammar Fragment
Term ➔ *Term* * *Num*	*Term* ➔ *Num Term'*
Term ➔ *Term* / *Num*	*Term'* ➔ * *Num Term'*
Term ➔ *Num*	*Term'* ➔ / *Num Term'*
	Term' ➔ ε

Parse Tree Comparisons



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

Goal ➔ *Expr*
Expr ➔ *Term Expr'*
Expr' ➔ + *Expr'*
Expr' ➔ - *Expr'*
Expr' ➔ ε

Term ➔ *Num Term'*
Term' ➔ * *Num Term'*
Term' ➔ / *Num Term'*
Term' ➔ ε
Num ➔ 0
Num ➔ 1
Num ➔ 2

Parsing Choice Points

- Assume *Num* is current position in parse tree
  - Must choose one of
    - *Num* ➔ 0
    - *Num* ➔ 1
    - *Num* ➔ 2
- Look at next token and see which one it is
  - If token is 0, choose *Num* ➔ 0
  - If token is 1, choose *Num* ➔ 1
  - If token is 2, choose *Num* ➔ 2

More Choice Points

- Assume *Term'* is current position in parse tree
  - *Term'* ➔ * *Num Term'*
  - *Term'* ➔ / *Num Term'*
  - *Term'* ➔ ε
- If next token is * or /, apply appropriate production
- Otherwise, apply *Term'* ➔ ε

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

First(β)

- $T \in$ First(β) if T can appear as the first symbol in a derivation starting from β
- Start with concept of *NT* deriving ε
- Two rules
  - *NT* ➔ ε implies *NT* derives ε
  - *NT* ➔ NT_1 ... NT_n and for all $1 \le i \le n$ NT_i derives ε implies *NT* derives ε

Fixed Point Algorithm for Derives

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* ➔ NT_1 ... NT_n
    if (for all $1 \le i \le n$ NT_i derives ε)
      set *NT* derives ε to be true

Rules for First

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* ➔ *S* β 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' ➔ * *Num Term'*
Term' ➔ / *Num Term'*
Term' ➔ ε   *Num* ➔ 0
Num ➔ 1   *Num* ➔ 2

Rules

1) $T \in$ First(T)
2) First(S) $\subseteq$ First($S\ \beta$)
3) *NT* derives ε implies First(β) $\subseteq$ First(*NT* β)
4) *NT* ➔ *S* β 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'*) $\subseteq$ First(*Term'*)
First(/ *Num Term'*) $\subseteq$ First(*Term'*)
First(*) $\subseteq$ First(* *Num Term'*)
First(/) $\subseteq$ First(/ *Num Term'*)
* $\in$ First(*)
/ $\in$ 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'*) $\subseteq$ First(*Term'*)
First(/ *Num Term'*) $\subseteq$ First(*Term'*)
First(*) $\subseteq$ First(* *Num Term'*)
First(/) $\subseteq$ First(/ *Num Term'*)
* $\in$ First(*)
/ $\in$ First(/)

Solution

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

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(/) = {/}

General Idea in Computer Science

- Systems of subset inclusion constraints
- Typically, a problem generates a system
- Outline of solution algorithm
  - Initialize all sets to {}
  - Use a constraint propagation algorithm that iteratively update sets by satisfying one constraint at a time
  - Terminate when reach a fixed point.

Predictive Parsing + Hand Coding = Recursive Descent Parser

- One procedure per nonterminal
- That procedure examines the current input symbol
- Recursively calls procedures for RHS of chosen production
- Procedures return true if parse succeeded, false otherwise

Example

```
Term()
   return(Num(), TermPrime())
TermPrime()
   if (token = *) || (token = /)
         token = NextToken();
         return (Num() && TermPrime())
   else return(true)
Num()
   if (token = 1) || (token == 2) || (token == 3)
         token = NextToken(); return(true)
   else return(false)
```


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 A Parse Tree in Example

```
Term()
   return(new TermNode(Num(), TermPrime()))
TermPrime()
   if (token = *) || (token = /)
         t = token; token = NextToken();
         return (new TermPrimeNode(t, Num(), TermPrime()));
   else return(new TermPrimeNode(ε));
Num()
   if (token = 1) || (token == 2) || (token == 3)
         t = token; token = NextToken(); return(new NumNode(t));
   else throw(SyntaxError);
```


Summary

- Top-Down Parsing
- Use Lookahead to Avoid Backtracking
- Parser is Hand-Coded Set of Mutually Recursive Procedures
- General Ideas in Computer Science
  - Backtracking as a Search Mechanism
  - Systems of Subset Inclusion Constraints
  - Fixed-Point Solution Algorithms