COMP 181

Lecture 6

Top-down Parsing

September 21, 2006

Prelude

- What is the Tufts mascot?
  "Jumbo" the elephant
- Why?
  - P. T. Barnum was an original trustee of Tufts
  - 1884: donated $50,000 for a natural museum on campus
    Barnum Museum, later Barnum Hall
  - "Jumbo": famous circus elephant
  - 1885: Jumbo died, was stuffed, donated to Tufts
  - 1975: Fire destroyed Barnum Hall, Jumbo

Last time

- Finished scanning
  - Produces a stream of tokens
  - Removes things we don't care about, like white space and comments
- Context-free grammars
  - Formal description of language syntax
  - Deriving strings using CFG
  - Depicting derivation as a parse tree

Grammar issues

- Often: more than one way to derive a string
- Why is this a problem?
  - Parsing: is string a member of L(G)?
  - We want more than a yes or no answer
- **Key**:
  - Represent the derivation as a parse tree
  - We want the ***structure*** of the parse tree to capture the ***meaning*** of the sentence

Grammar issues

- Often: more than one way to derive a string
- Why is this a problem?
  - Parsing: is string a member of L(G)?
  - We want more than a yes or no answer
- **Key**:
  - Represent the derivation as a pa
  - We want the ***structure*** of the pa
    meaning of the sentence

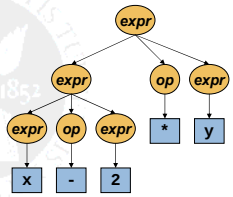
#	Production rule
1	*expr* → *expr op expr*
2	| number
3	| identifier
4	*op* → +
5	| -
6	| *
7	| /

Parse tree: x – 2 * y

Right-most derivation

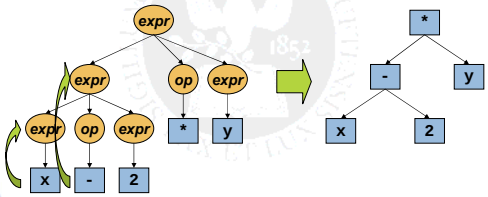
Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Parse tree



Abstract syntax tree

- Parse tree contains extra junk
  - Eliminate intermediate nodes
  - Move operators up to parent nodes
  - Result: ***abstract syntax tree***



Left vs right derivations

- Two derivations of "**x – 2 * y**"

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

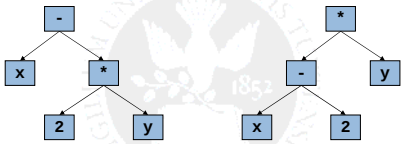
Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Right-most derivation

Derivations

- One captures meaning, the other doesn't



Left-most derivation **Right-most derivation**

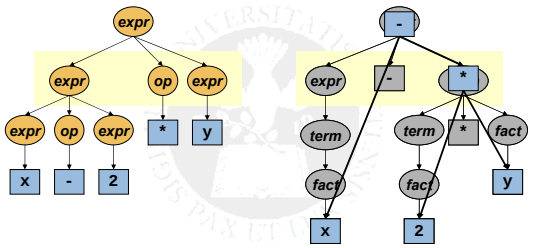
With precedence

- Last time: ways to force the right tree shape
- Add productions to represent precedence

#	Production rule
1	*expr* → *expr op expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → `number`
8	| `identifier`

With precedence



Parsing

- What is parsing?
  - Discovering the derivation of a string
    If one exists
  - Harder than generating strings
    Not surprisingly
- Two major approaches
  - Top-down parsing
  - Bottom-up parsing
- Don't work on all context-free grammars
  - Properties of grammar determine parse-ability
  - **Our goal**: make parsing efficient
  - We may be able to transform a grammar

Two approaches

- Top-down parsers **LL(1), recursive descent**
  - Start at the root of the parse tree and grow toward leaves
  - Pick a production & try to match the input
  - Bad "pick" → may need to backtrack
- Bottom-up parsers **LR(1), operator precedence**
  - Start at the leaves and grow toward root
  - As input is consumed, encode possible parse trees in an internal state *(similar to our NFA → DFA conversion)*
  - Bottom-up parsers handle a large class of grammars

Grammars and parsers

- LL(1) parsers
  - **L**eft-to-right input
  - **L**eftmost derivation
  - **1** symbol of look-ahead

  Grammars that this can handle are called LL(1) grammars
- LR(1) parsers
  - **L**eft-to-right input
  - **R**ightmost derivation
  - **1** symbol of look-ahead

  Grammars that this can handle are called LR(1) grammars
- Also: LL(k), LR(k), SLR, LALR, …

Top-down parsing

- Start with the root of the parse tree
  - Root of the tree: node labeled with the start symbol
- Algorithm:

  Repeat until the fringe of the parse tree matches input string
  - At a node A, select a production for A

    Add a child node for each symbol on rhs
  - If a terminal symbol is added that doesn't match, backtrack
  - Find the next node to be expanded *(a non-terminal)*
- Done when:
  - Leaves of parse tree match input string *(success)*
  - All productions exhausted in backtracking *(failure)*

Example

- Expression grammar *(with precedence)*

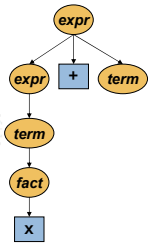
#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → number
8	| identifier

- Input string **x – 2 * y**

Example

Current position in the input stream

Rule	Sentential form	Input string
-	*expr*	↑ x - 2 * y
2	*expr* + *term*	↑ x - 2 * y
3	*term* + *term*	↑ x – 2 * y
6	*factor* + *term*	↑ x – 2 * y
8	<id> + *term*	x ↑ – 2 * y
-	<id,x> + *term*	x ↑ – 2 * y



- Problem:
  - Can't match next terminal
  - We guessed wrong at step 2

Backtracking

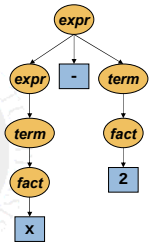
Rule	Sentential form	Input string
-	*expr*	↑ x - 2 * y
2	*expr* + *term*	↑ x - 2 * y
3	*term* + *term*	↑ x – 2 * y
6	*factor* + *term*	↑ x – 2 * y
8	<id> + *term*	x ↑ – 2 * y
?	<id,x> + *term*	x ↑ – 2 * y

Undo all these productions

- Rollback productions
- Choose a different production for *expr*
- *Continue*

Retrying

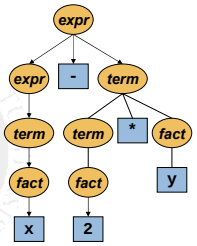
Rule	Sentential form	Input string
-	*expr*	↑ x - 2 * y
2	*expr - term*	↑ x - 2 * y
3	*term - term*	↑ x - 2 * y
6	*factor - term*	↑ x - 2 * y
8	*<id> - term*	x ↑ - 2 * y
-	*<id,x> - term*	x - ↑ 2 * y
3	*<id,x> - factor*	x - ↑ 2 * y
7	*<id,x> - <num>*	x - 2 ↑ * y



- Problem:
  - More input to read
  - Another cause of backtracking

Successful parse

Rule	Sentential form	Input string
-	*expr*	↑ x - 2 * y
2	*expr - term*	↑ x - 2 * y
3	*term - term*	↑ x - 2 * y
6	*factor - term*	↑ x - 2 * y
8	*<id> - term*	x ↑ - 2 * y
-	*<id,x> - term*	x - ↑ 2 * y
4	*<id,x> - term * fact*	x - ↑ 2 * y
6	*<id,x> - fact * fact*	x - ↑ 2 * y
7	*<id,x> - <num> * fact*	x - 2 ↑ * y
-	*<id,x> - <num,2> * fact*	x - 2 * ↑ y
8	*<id,x> - <num,2> * <id>*	x - 2 * y ↑



- All terminals match – we're done

Other possible parses

Rule	Sentential form	Input string
-	*expr*	↑ x - 2 * y
2	*expr + term*	↑ x - 2 * y
2	*expr + term + term*	↑ x - 2 * y
2	*expr + term + term + term*	↑ x - 2 * y
2	*expr + term + term + term + term*	↑ x - 2 * y

- Problem: termination
  - Wrong choice leads to infinite expansion
    (More importantly: without consuming any input!)
  - May not be as obvious as this
  - Our grammar is ***left recursive***

Left recursion

- Formally,
  A grammar is ***left recursive*** if ∃ a non-terminal A such that
  A →* A α ***(for some set of symbols α)***

  What does →* mean?
  A → B x
  B → A y

- Bad news:
  Top-down parsers cannot handle left recursion
- Good news:
  We can systematically eliminate left recursion

Notation

- Non-terminals
  - Capital letter: A, B, C
- Terminals
  - Lowercase, underline: x, y, z
- Some mix of terminals and non-terminals
  - Greek letters: α, β, γ
  - Example:

#	Production rule
1	*A → B + x*
1	*A → B α*

α = + x

Eliminating left recursion

- Consider this grammar:

#	Production rule
1	*foo → foo α*
2	*| β*

Language is β followed by zero or more α

- Rewrite as

#	Production rule
1	*foo → β bar*
2	*bar → α bar*
3	*| ε*

This production gives you one β

These two productions give you zero or more α

New non-terminal

Back to expressions

- Two cases of left recursion:

#	Production rule
1	*expr* → *expr + term*
2	| *expr - term*
3	| *term*

#	Production rule
4	*term* → *term * factor*
5	| *term / factor*
6	| *factor*

- Transform as follows:

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → *+ term expr2*
3	| *- term expr2*
4	| *ε*

#	Production rule
4	*term* → *factor term2*
5	*term2* → ** factor term2*
6	| */ factor term2*
	| *ε*

Eliminating left recursion

- Resulting grammar
  - All right recursive
  - Retain original language and associativity
  - Not as intuitive to read
- Top-down parser
  - Will always terminate
  - May still backtrack

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → *+ term expr2*
3	| *- term expr2*
4	| *ε*
5	*term* → *factor term2*
6	*term2* → ** factor term2*
7	| */ factor term2*
8	| *ε*
9	*factor* → `number`
10	| `identifier`

There's a lovely algorithm to do this automatically, which we will skip

Top-down parsers

- ***Problem***: Left-recursion
- ***Solution***: Technique to remove it
- What about backtracking?
  Current algorithm is brute force
- ***Problem***: how to choose the right production?
  - Idea: use the next input token *(duh)*
  - How? Look at our right-recursive grammar…

Right-recursive grammar

#	Production rule
1	*expr* → *term expr2*
2	*expr2* → *+ term expr2*
3	| *- term expr2*
4	| *ε*
5	*term* → *factor term2*
6	*term2* → ** factor term2*
7	| */ factor term2*
8	| *ε*
9	*factor* → `number`
10	| `identifier`

Two productions with no choice at all

All other productions are uniquely identified by a terminal symbol at the start of RHS

- We can choose the right production by looking at the next input symbol
  - This is called ***lookahead***
  - BUT, this can be tricky…

Lookahead

- Goal: avoid backtracking
  - Look at future input symbols
  - Use extra context to make right choice
- How much lookahead is needed?
  - In general, an arbitrary amount is needed for the full class of context-free grammars
  - Use fancy-dancy algorithm *CYK algorithm, $O(n^3)$*
- Fortunately,
  - Many CFGs can be parsed with limited lookahead
  - Covers most programming languages *not C++ or Perl*

Top-down parsing

- **Goal**:
  Given productions $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α and β
- Trying to match A
  How can the next input token help us decide?
- Solution: *FIRST* sets *(almost a solution)*
  - Informally:
    FIRST(α) is the set of tokens that could appear as the first symbol in a string derived from α
  - ***Def:*** $\underline{x}$ in FIRST(α) iff $\alpha \rightarrow^* \underline{x}\ \gamma$

Top-down parsing

- Building FIRST sets
  We'll look at this algorithm later
- The LL(1) property
  - Given $A \rightarrow \alpha$ and $A \rightarrow \beta$, we would like:
    $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$
  - Parser can make right choice by looking at one lookahead token
  - ..almost..

Top-down parsing

- What about ε productions?
  - Complicates the definition of LL(1)
  - Consider $A \rightarrow \alpha$ and $A \rightarrow \beta$ and α may be empty
  - In this case there is no symbol to identify α
- Example:
  - What is FIRST(3)?
  - $= \{ \varepsilon \}$
  - What lookahead symbol tells us we are matching production 3?

#	Production rule
1	*A* → x B
2	| y C
3	| ε

Top-down parsing

- If A was empty
  - What will the next symbol be?
  - Must be one of the symbols that immediately follow an A
- **Solution**
  - Build a ***FOLLOW*** set for each production with ε
  - Extra condition for LL:
    FIRST(β) must be disjoint from FIRST(α) and FOLLOW(A)

FOLLOW sets

- Example:
  - FIRST(1) = { x }
  - FIRST(2) = { y }
  - FIRST(3) = { ε }

#	Production rule
1	*A* → x B
2	| y C
3	| ε
4	*E* → *A* z

- What can follow A?
  - Look at the context of all uses of A
  - FOLLOW(A) = { z }
  - Now we can uniquely identify each production:
    - If we are trying to match an A and the next token is z, then we matched production 3

More on FIRST and FOLLOW

- Notice:
  - FIRST and FOLLOW may be sets
  - FIRST may contain ε in addition to other symbols
  - Example:
    - FIRST(1) = { x, y, ε }
    - FOLLOW(A) = { z, w }
- **Question**:
  When would we care about FOLLOW(A)?
  Answer: if FIRST(C) contains ε

#	Production rule
1	*A* → B C
2	*B* → x
3	| y
4	| ε
5	*E* → *A* z
6	*F* → *A* w

LL(1) property

- Including ε productions
  - FOLLOW(A) = the set of terminal symbols that can immediately follow A
  - ***Def***: FIRST+($A \rightarrow \alpha$) as
    - FIRST(α) U FOLLOW(A), if $\varepsilon \in$ FIRST(α)
    - FIRST(α), otherwise
- ***Def***: a grammar is LL(1) iff
  $A \rightarrow \alpha$ and $A \rightarrow \beta$ and
  FIRST+($A \rightarrow \alpha$) $\cap$ FIRST+($A \rightarrow \beta$) = $\varnothing$

LL(1) property

- *Question*
  Can there be two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$ in a *LL(1)* grammar such that $\varepsilon \in$ FIRST(α) and $\varepsilon \in$ FIRST(β)?
- *Answer*
  Yes, as long as they have different FOLLOW sets

Parsing LL(1) grammar

- Given an LL(1) grammar
  - Code: simple, fast routine to recognize each production
  - Given $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with
    $\text{FIRST}^+(\beta_i) \cap \text{FIRST}^+(\beta_j) = \varnothing$ *for all i != j*

```
/* find rule for A */
if (current token ∈ FIRST+(β1))
   select A → β1
else if (current token ∈ FIRST+(β2))
   select A → β2
else if (current token ∈ FIRST+(β3))
   select A → β3
else
   report an error and return false
```


Predictive parsing

- ***Predictive parsing***
  - The parser can "predict" the correct expansion
  - Using lookahead and FIRST and FOLLOW sets
- Two kinds of predictive parsers
  - Recursive descent
    Often hand-written
  - Table-driven
    Generate tables from First and Follow sets

Recursive descent

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier
12	| (*expr*)

- This produces a parser with six *mutually recursive* routines:
  - *Goal*
  - *Expr*
  - *Expr2*
  - *Term*
  - *Term2*
  - *Factor*
- Each recognizes one *NT* or *T*
- The term *descent* refers to the direction in which the parse tree is built.

Example code

- Goal symbol:

```
main()
  /* Match goal —> expr */
  tok = nextToken();
  if (expr() && tok == EOF)
    then proceed to next step;
    else return false;
```

- Top-level expression

```
expr()
  /* Match expr —> term expr2 */
  if (term() && expr2());
    return true;
  else return false;
```


Example code

- Match expr2

```
expr2()
  /* Match expr2 —> + term expr2 */
  /* Match expr2 —> - term expr2 */

  if (tok == '+' or tok == '-')
    tok = nextToken();
    if (term())
      then return expr2();
      else return false;

  /* Match expr2 --> empty */
  return true;
```

Check FIRST and FOLLOW sets to distinguish

Example code

```
factor()
  /* Match factor --> ( expr ) */
  if (tok == '(')
    tok = nextToken();
    if (expr() && tok == ')')
      return true;
    else
      syntax error: expecting )
      return false

  /* Match factor --> num */
  if (tok is a num)
    return true

  /* Match factor --> id */
  if (tok is an id)
    return true;
```


Top-down parsing

- So far:
  - Gives us a yes or no answer
  - We want to build the parse tree
  - How?
- Add actions to matching routines
  - Create a node for each production
  - How do we assemble the tree?

Building a parse tree

- Notice:
  - Recursive calls match the shape of the tree

```
main
  expr
    term
      factor
    expr2
      term
```

- Idea: use a stack
  - Each routine:
    - Pops off the children it needs
    - Creates its own node
    - Pushes that node back on the stack

Building a parse tree

- With stack operations

```
expr()
  /* Match expr --> term expr2 */
  if (term() && expr2())
    expr2_node = pop();
    term_node = pop();
    expr_node = new exprNode(term_node,
                             expr2_node)
    push(expr_node);
    return true;
  else return false;
```


Next time…

- Finish top-down parsing
  - Table-driven parsers
  - Building FIRST and FOLLOW sets
- Start bottom-up parsing