Prelude
- What is the common name of the fruit *Synsepalum dulcificum*?
- “Miracle fruit” – from West Africa
- What is special about miracle fruit?
- Contains a protein called miraculin
- In an acidic environment, binds to “sweet” receptors in taste buds
- Sour becomes sweet
  - Lemons and limes, Tabasco sauce, stinky cheese, dark beer
- “Flavor tripping” – lasts about an hour

Next step
- Parsing: Organize tokens into “sentences”
  - Do tokens conform to language syntax?
  - Good news: token types are just numbers
  - Bad news: language syntax is fundamentally more complex than lexical specification
  - Good news: we can still do it in linear time in most cases

Parsing
- Parser
  - Reads tokens from the scanner
  - Checks organization of tokens against a grammar
  - Constructs a derivation
  - Derivation drives construction of IR

Study of parsing
- Discovering the derivation of a sentence
  - “Diagramming a sentence” in grade school
- Formalization:
  - Mathematical model of syntax – a grammar $G$
  - Algorithm for testing membership in $L(G)$
- Roadmap:
  - Context-free grammars
  - Top-down parsers
    - Ad hoc, often hand-coded, recursive decent parsers
  - Bottom-up parsers
    - Automatically generated LR parsers

Specifying syntax with a grammar
- Can we use regular expressions?
  - For the most part, no
- Limitations of regular expressions
  - Need something more powerful
  - Still want formal specification (for automation)
- Context-free grammar
  - Set of rules for generating sentences
  - Expressed in Backus-Naur Form (BNF)
Context-free grammar

- Example:
  
  ```
  \begin{align*}
  &\text{sheepnoise} \rightarrow \text{sheepnoise baa} \\
  &\text{baa} \rightarrow \text{baa} | \text{baa}
  \end{align*}
  ```

- Formally: context-free grammar is
  
  $G = (s, N, T, P)$

  - $T$: set of terminals (provided by scanner)
  - $N$: set of non-terminals (represent structure)
  - $s \in N$: start or goal symbol
  - $P : N \rightarrow (N \cup T)^*$: set of production rules

Language L(G)

- Language L(G)
  
  $L(G)$ is all sentences generated from start symbol

  Generating sentences
  
  - Use productions as rewrite rules
  - Start with goal (or start) symbol – a non-terminal
  - Choose a non-terminal and "expand" it to the right-hand side of one of its productions
  - Only terminal symbols left → sentence in L(G)

  Intermediate results known as sentential forms

Expressions

- Language of expressions
  
  - Numbers and identifiers
  - Allow different binary operators
  - Arbitrary nesting of expressions

  ```
  \begin{align*}
  \text{expr} &\rightarrow \text{expr} \ op \ \text{expr} \\
  &\mid \text{number} \\
  \text{op} &\rightarrow + | - | * | / 
  \end{align*}
  ```

Language of expressions

- What’s in this language?
  
  We can build the string "x - 2 * y"

  This string is in the language

Derivations

- Using grammars
  
  - A sequence of rewrites is called a derivation
  - Discovering a derivation for a string is parsing

  Different derivations are possible

- Rightmost derivation: always choose right NT

- Leftmost derivation: always choose left NT

  (Other “random” derivations – not of interest)

Left vs right derivations

- Two derivations of "x - 2 * y"

  ```
  \begin{align*}
  &\text{x} \rightarrow \text{expr} - \text{expr} \ op \ \text{expr} \\
  &\mid \text{number} \\
  &\mid \text{expr} \ op \ \text{expr} \\
  &\mid \text{number} \\
  &\mid \text{ident} \\
  &\mid \text{expr} \ - \ \text{number} \ op \ \text{expr} \\
  &\mid \text{number} \ op \ \text{expr} \\
  &\mid \text{ident} \\
  \end{align*}
  ```

  ```
  \begin{align*}
  &\text{x} \rightarrow \text{expr} - \text{expr} \ op \ \text{expr} \\
  &\mid \text{number} \\
  &\mid \text{expr} \ op \ \text{expr} \\
  &\mid \text{number} \\
  &\mid \text{ident} \\
  &\mid \text{expr} \ - \ \text{number} \ op \ \text{expr} \\
  &\mid \text{number} \ op \ \text{expr} \\
  &\mid \text{ident} \\
  \end{align*}
  ```

  Left-most derivation

  Right-most derivation
Derivations and parse trees

- Two different derivations
  - Both are correct
  - Do we care which one we use?
- Represent derivation as a parse tree
  - Leaves are terminal symbols
  - Inner nodes are non-terminals
  - To depict production $\alpha \rightarrow \beta \gamma \delta$
    - show nodes $\beta, \gamma, \delta$ as children of $\alpha$

- Tree is used to build internal representation

Example (I)

- Concrete syntax tree
  - Shows all details of syntactic structure
  - What’s the problem with this tree?

Example (II)

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

- Problem: Evaluates as $(x - 2) \times y$

Example (II) solution

- Solution: evaluates as $x - (2 \times y)$

Derivations and semantics

- Problem:
  - Two different valid derivations
  - One captures “meaning” we want
    - (What specifically are we trying to capture here?)
  - Key idea: shape of tree implies its meaning
- Can we express precedence in grammar?
  - Notice: operations deeper in tree evaluated first
  - Solution: add an intermediate production
    - New production isolates different levels of precedence
    - Force higher precedence “deeper” in the grammar
4

Adding precedence

- Two levels:
  - Level 1: lower precedence – higher in the tree
  - Level 2: higher precedence – deeper in the tree

- Observations:
  - Larger: requires more rewriting to reach terminals
  - But, produces same parse tree under both left and right derivations

| # | 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

In class questions

- What if I want \((x-2) \ast y\)?
- Some common patterns...

Another issue

- Original expression grammar:

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

Another issue

- Multiple leftmost derivations
- Such a grammar is called ambiguous
- Is this a problem?
- Very hard to automate parsing

| Rule | Sentential form
|---|---|
| expr | expr op expr
| expr op expr | expr op expr
| \(<id, x>\) op expr | \(<id, x>\) op expr
| \(<id, x>\) -> expr op expr | \(<id, x>\) -> expr op expr
| \(<id, x>\) + expr | \(<id, x>\) + expr
| \(<id, x>\) * expr | \(<id, x>\) * expr
| \(<id, x>\) -> \(<num, 2>\) * expr | \(<id, x>\) -> \(<num, 2>\) * expr
| \(<id, x>\) -> \(<num, 2>\) + expr | \(<id, x>\) -> \(<num, 2>\) + expr

Right-most derivation

Now right derivation yields \(x - (2 \ast y)\)

Our favorite string: \(x - 2 \ast y\)
Ambiguous grammars

- A grammar is ambiguous iff:
  - There are multiple leftmost or multiple rightmost derivations for a single sentential form
  - Note: leftmost and rightmost derivations may differ, even in an unambiguous grammar

  Intuitively:
  - We can choose different non-terminals to expand
  - But each non-terminal should lead to a unique set of terminal symbols

- What’s a classic example?
  - If-then-else ambiguity

If-then-else

- Grammar:

  ```
  stmt -> if expr then stmt |
  if expr then stmt else stmt |
  ...
  ```

- Problem: nested if-then-else statements
  - Each one may or may not have else
  - How to match each else with if

Removing ambiguity

- Restrict the grammar
  - Choose a rule: “else” matches innermost “if”
  - Codify with new productions

  ```
  stmt -> if expr then stmt |
  if expr then stmt withelse else stmt |
  ...
  ```

  Intuition: when we have an “else”, all preceding nested conditions must have an “else”

Parsing

- What is parsing?
  - Discovering the derivation of a string
    - if it 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 and try to match the input
  - What happens if the parser chooses the wrong one?

- **Bottom-up parsers** LR(1), operator precedence
  - Start at the leaves and grow toward root
  - Issue: might have multiple possible ways to do this
  - Key idea: 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**
  - Left-to-right input
  - Leftmost derivation
  - 1 symbol of look-ahead

- **LR(1) parsers**
  - Left-to-right input
  - Rightmost derivation
  - 1 symbol of look-ahead

  *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 one of A’s productions
  - Add a child node for each symbol on rhs
  - Find the next node to be expanded (a non-terminal)

- Done when:
  - Leaves of parse tree match input string (success)

Example

- **Expression grammar** (with precedence)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential form</th>
<th>Input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expr</td>
<td>x - 2 * y</td>
</tr>
<tr>
<td>2</td>
<td>expr + term</td>
<td>↑</td>
</tr>
<tr>
<td>3</td>
<td>form + term</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>term</td>
<td>x - 2 * y</td>
</tr>
<tr>
<td>5</td>
<td>term * factor</td>
<td>↑</td>
</tr>
<tr>
<td>6</td>
<td>factor + term</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>factor</td>
<td>x - 2 * y</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id&gt; + term</td>
<td></td>
</tr>
</tbody>
</table>

Problem:

- Can’t match next terminal
- We guessed wrong at step 2
- What should we do now?

Backtracking

- If we can’t match next terminal:
  - Rollback productions
  - Choose a different production for expr
  - Continue

Grammars that they can handle are called LL(1) grammars

Grammars that they can handle are called LR(1) grammars

Example

- Input string: x - 2 * y
Retrying

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

Successful parse

- 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

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:

Eliminating left recursion

- Fix this grammar:

  Language is β followed by zero or more α

- Rewrite as:

  These two productions give you zero or more α

  New non-terminal
Back to expressions

- Two cases of left recursion:
  - How do we fix these?

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expr → expr + term</td>
</tr>
<tr>
<td>2</td>
<td>expr → expr - term</td>
</tr>
<tr>
<td>3</td>
<td>expr → term</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>term → term * factor</td>
</tr>
<tr>
<td>5</td>
<td>term → term / factor</td>
</tr>
<tr>
<td>6</td>
<td>term → factor</td>
</tr>
</tbody>
</table>

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

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…

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>2</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → - term expr2</td>
</tr>
<tr>
<td>4</td>
<td>expr2 → ε</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>5</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>6</td>
<td>term2 → / factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → ε</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>factor → number</td>
</tr>
<tr>
<td>10</td>
<td>factor → identifier</td>
</tr>
</tbody>
</table>

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

Right-recursive grammar

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>2</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → - term expr2</td>
</tr>
<tr>
<td>4</td>
<td>expr2 → ε</td>
</tr>
<tr>
<td>5</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>6</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → / factor term2</td>
</tr>
<tr>
<td>8</td>
<td>term2 → ε</td>
</tr>
<tr>
<td>9</td>
<td>factor → number</td>
</tr>
<tr>
<td>10</td>
<td>factor → identifier</td>
</tr>
</tbody>
</table>
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:
  
  $\text{FIRST}(A \rightarrow \alpha) \cap \text{FIRST}(A \rightarrow \beta) = \emptyset$
  
  Parser can make right choice by with one lookahead token
  
  almost...
  
  What are we not handling?

- What about $\varepsilon$ productions?
  
  Complicates the definition of LL(1)
  
  Consider $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\alpha$ may be empty
  
  In this case there is no symbol to identify $\alpha$

- Example:
  
  What is $\text{FIRST}(\#4)$?
  
  $= \{ \varepsilon \}$
  
  What would tells us we are matching production 4?

- If $A$ was empty
  
  What will the next symbol be?
  
  Must be one of the symbols that immediately follows an $A$

- Solution
  
  Build a Follow set for each symbol that could produce $\varepsilon$

- Extra condition for LL:
  
  $\text{FIRST}(A \rightarrow \beta)$ must be disjoint from $\text{FIRST}(A \rightarrow \alpha)$ and $\text{FOLLOW}(A)$

More on First and Follow

- Notice:
  
  FIRST and FOLLOW are sets
  
  FIRST may contain $\varepsilon$ in addition to other symbols

- Question:
  
  What is $\text{FIRST}(\#2)$?
  
  $= \text{FIRST}(B) = \{ \varepsilon, x, y, z \}$

- Question:
  
  When would we care about FOLLOW(A)?

  Answer: if First(C) contains $\varepsilon$

LL(1) property

- Key idea:
  
  Build parse tree top-down
  
  Use look-ahead token to pick next production
  
  Each production must be uniquely identified by the terminal symbols that may appear at the start of strings derived from it.

- Def: $\text{FIRST}(A \rightarrow \alpha)$ as
  
  $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\alpha)$
  
  $\text{FIRST}(\alpha)$, otherwise

- Def: a grammar is LL(1) iff
  
  $A \rightarrow \alpha$ and $A \rightarrow \beta$ and
  
  $\text{FIRST}(A \rightarrow \alpha) \cap \text{FIRST}(A \rightarrow \beta) = \emptyset$
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:
    
    \[
    \text{FIRST}^*(\beta_i) \cap \text{FIRST}^*(\beta_j) = \emptyset \quad \text{for all} \quad i \not= j
    \]

\[
\ast \quad \text{find rule for} \ A \ast \]

if (current token \( \in \text{FIRST}(\beta_i) \))
else if (current token \( \in \text{FIRST}(\beta_j) \))
  select \( A \rightarrow \beta_i \)
else if (current token \( \in \text{FIRST}(\beta_j) \))
  select \( A \rightarrow \beta_j \)
else
  report an error and return false.

Algorithm

- For one production: \( p = A \rightarrow \beta \)

\[
\text{if} \ (\beta \text{ is a terminal } t) \]
\[
\text{FIRST}(p) = \{ t \}
\]
else if (\( \beta = \epsilon \))
\[
\text{FIRST}(p) = \{ \epsilon \}
\]
else (Given \( \beta = B_1 B_2 \ldots B_k \))
\[
\text{FIRST}(p) \leftarrow \text{FIRST}(B_i) - \{ \epsilon \}
\]
\[
\text{while} \ (z \in \text{FIRST}(B_i) \land i < k)
\]
\[
\text{FIRST}(p) \leftarrow \{ z \}
\]

FIRST and FOLLOW sets

**FIRST(\( \alpha \))**

- The right-hand side of a production

**FIRST**

For some \( \alpha \in (T \cup NT)^* \), define \( \text{FIRST}(\alpha) \) as the set of tokens that appear as the first symbol in some string that derives from \( \alpha \).

That is, \( x \in \text{FIRST}(\alpha) \) if \( \alpha \Rightarrow x \gamma \), for some \( \gamma \).

**Note:** may include \( \epsilon \)

**FOLLOW(A)**

For some \( A \in NT \), define \( \text{FOLLOW}(A) \) as the set of symbols that can occur immediately after \( A \) in a valid sentence.

\( \text{FOLLOW}(G) = \{ \text{EOF} \} \), where \( G \) is the start symbol.

Computing FIRST sets

**Idea:**

- Use \( \text{FIRST} \) sets of the right side of production

**Cases:**

- \( \text{FIRST}(A \rightarrow B) = \text{FIRST}(B) \)
  - What does \( \text{FIRST}(B) \) mean?
  - Union of \( \text{FIRST}(B_i) \) for all \( \gamma \)
  - What if \( \epsilon \) in \( \text{FIRST}(B) \)\?
    - \( \text{FIRST}(A \rightarrow B) \cup \text{FIRST}(B) \)
  - What if \( \epsilon \) in \( \text{FIRST}(B) \) for all \( \gamma \)\?
    - \( \text{FIRST}(A \rightarrow B) \cup = \{ \epsilon \} \)

Algorithm

- For one production:
  - Given \( A \rightarrow B_1 B_2 B_3 B_4 \)
  - Compute \( \text{FIRST}(A \rightarrow B) \) using \( \text{FIRST}(B) \)
  - How do we get \( \text{FIRST}(B) \)\?
  - What kind of algorithm does this suggest?
  - Recursive?
  - Like a depth-first search of the productions

**Problem:**

- What about recursion in the grammar?
  - \( A \rightarrow x \ B \ y \) and \( B \rightarrow z \ A \ w \)
Algorithm

- Solution
  - Start with FIRST(B) empty
  - Compute FIRST(A) using empty FIRST(B)
  - Now go back and compute FIRST(B)
  - What if it's no longer empty?
    - Then we recompute FIRST(A)
    - What if new FIRST(A) is different from old FIRST(A)?
    - Then we recompute FIRST(B) again...
- When do we stop?
  - When no more changes occur - we reach convergence
  - FIRST(A) and FIRST(B) both satisfy equations
- This is another fixpoint algorithm

Example

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal → expr</td>
</tr>
<tr>
<td>2</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>factor → number</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

FOLLOW(goal) = { EOF }
FOLLOW(expr) = FOLLOW(goal) = { EOF }
FOLLOW(expr2) = FOLLOW(expr) = { EOF }
FOLLOW(term) = ?
FOLLOW(term2) = FIRST(expr2)
  *= { +, - , * } 
  / = { +, - , FOLLOW(expr) } 
  += { +, - , EOF } 
FOLLOW(number) = { +, - , * } 
FOLLOW(identifier) = { +, - , * } 

Algorithm

- Using fixpoints:
  - forall p  FIRST(p) = {}  
  while (FIRST sets are changing)  
    pick a random p  
    compute FIRST(p)  

- Can we be smarter?
  - Yes, visit in special order
  - Reverse post-order depth first search
  - Visit all children (all right-hand sides) before visiting the left-hand side, whenever possible

Example

Computing FOLLOW sets

- Idea:
  Push FOLLOW sets down, use FIRST where needed
  \[ A \rightarrow B_1 \mid B_2 \mid B_3 \mid B_4 \mid \ldots \mid B_n \]
- Cases:
  - What is FOLLOW(B_j)?
    - FOLLOW(B_j) = FIRST(B_j)
  - In general: FOLLOW(B_j) = FIRST(B_{j+1})
  - What about FOLLOW(B_j)?
    - FOLLOW(B_j) = FOLLOW(A)
  - What if \( \varepsilon \in \text{FIRST}(B_j)? \)
    - \( \Rightarrow \) FOLLOW(B_j) \( \cup \) FOLLOW(A)  extends to \( k-2 \), etc.

Example

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal → expr</td>
</tr>
<tr>
<td>2</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>factor → number</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal → expr</td>
</tr>
<tr>
<td>2</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>factor → number</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal → expr</td>
</tr>
<tr>
<td>2</td>
<td>expr → term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 → + term expr2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>term → factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 → * factor term2</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>factor → number</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Example
Computing FOLLOW Sets

FOLLOW(A) = {EOF}
for each A ∈ NT, FOLLOW(A) = Ø
while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A → α β...
        FOLLOW(β_k) = FOLLOW(β_k) ∪ FOLLOW(A)
        if ε ∈ FIRST(β_i) then
            FOLLOW(β_i-1) = FOLLOW(β_i-1) ∪ {FIRST(β_i) – {ε}} ∪ TRAILER
        else
            FOLLOW(β_i-1) = FOLLOW(β_i-1) ∪ FIRST(β_i)
        TRAILER = Ø

LL(1) property

Def: a grammar is LL(1) iff
A → α and A → β and
FIRST(A → α) ∩ FIRST(A → β) = Ø

Problem
What if my grammar is not LL(1)?
May be able to fix it, with transformations

Example:

Expression example

After left factoring:

Left factoring

Graphically

No basis for choice

Next word determines choice

Question
Using left factoring and left recursion elimination, can we turn an arbitrary CFG to a form where it meets the LL(1) condition?

Answer
Given a CFG that does not meet LL(1) condition, it is undecidable whether or not an LL(1) grammar exists

Example
(a^n b^n | n ≥ 1) ∪ (a^n b^m | n ≠ m) has no LL(1) grammar

aaa0bbb
aaa1bbbbbb
Limits of LL(1)

- No LL(1) grammar for this language:
  \( \{a^n b^n \mid n \geq 1 \} \cup \{a^n b^{2n} \mid n \geq 1 \} \) has no LL(1) grammar

### Production rule

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( G \rightarrow aAb )</td>
</tr>
<tr>
<td>2</td>
<td>( G \rightarrow aBbb )</td>
</tr>
<tr>
<td>3</td>
<td>( A \rightarrow aA )</td>
</tr>
<tr>
<td>4</td>
<td>( A \rightarrow \epsilon )</td>
</tr>
<tr>
<td>5</td>
<td>( B \rightarrow aB )</td>
</tr>
<tr>
<td>6</td>
<td>( B \rightarrow \epsilon )</td>
</tr>
</tbody>
</table>

Problem: need an unbounded number of \( a \) characters before you can determine whether you are in the A group or the B group.

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

- 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.

### Production rule

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{goal} \rightarrow \text{expr} )</td>
</tr>
<tr>
<td>2</td>
<td>( \text{expr} \rightarrow \text{term} \text{expr2} )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{expr2} \rightarrow + \text{term} \text{expr2} )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{expr2} \rightarrow - \text{term} \text{expr2} )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{term} \rightarrow \text{factor} \text{term2} )</td>
</tr>
<tr>
<td>6</td>
<td>( \text{term} \rightarrow \epsilon )</td>
</tr>
<tr>
<td>7</td>
<td>( \text{term2} \rightarrow \text{factor} \text{term2} )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{term2} \rightarrow \epsilon )</td>
</tr>
<tr>
<td>9</td>
<td>( \text{factor} \rightarrow \text{number} )</td>
</tr>
<tr>
<td>10</td>
<td>( \text{factor} \rightarrow \text{identifier} )</td>
</tr>
<tr>
<td>11</td>
<td>( \text{factor} \rightarrow )</td>
</tr>
<tr>
<td>12</td>
<td>( \text{factor} \rightarrow \text{expr} )</td>
</tr>
</tbody>
</table>

Example code

- **Goal symbol:**
  ```c
  main()
  /* Match goal --> expr */
  tok = nextToken();
  if (expr() && tok == EOF)
    then proceed to next step;
    else return false;
  
  expr()
  /* Match expr --> term expr2 */
  if (term() && expr2())
    return true;
  else return false;
  
  expr2()
  /* Match expr2 --> + term expr2 */
  /* Match expr2 --> - term expr2 */
  if (tok == '+' or tok == '-')
    tok = nextToken();
  if (term() && expr2())
    return true;
  else return false;
  
  /* Match expr2 --> empty */
  return true;
  
  factor()
  /* Match factor --> ( expr ) */
  /* Match factor --> id */
  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;
  else return false;
  ```

---

*Tufts University Computer Science*
Top-down parsing

- So far:
  - Gives us a yes or no answer
  - Is that all we want?
  - 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

- 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

```c
expr() {
  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;
}
```

Generating a top-down parser

- Second piece
  - Keep track of progress
  - Like a depth-first search
  - Use a stack
- Idea:
  - Push Goal on stack
  - Pop stack:
    - Match terminal symbol, or
    - Apply NT mapping, push RHS on stack

Generating a top-down parser

<table>
<thead>
<tr>
<th>Production rule</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal -&gt; expr</td>
<td>1</td>
</tr>
<tr>
<td>expr -&gt; term expr2</td>
<td>2</td>
</tr>
<tr>
<td>expr2 -&gt; term expr2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>factor term2</td>
</tr>
<tr>
<td></td>
<td>factor term2</td>
</tr>
<tr>
<td>term -&gt; factor term2</td>
<td>6</td>
</tr>
<tr>
<td>term2 -&gt; factor term2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>factor term2</td>
</tr>
<tr>
<td></td>
<td>factor term2</td>
</tr>
<tr>
<td>factor -&gt; number</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>identifier</td>
</tr>
</tbody>
</table>

Table-driven approach

- Encode mapping in a table
  - Row for each non-terminal
  - Column for each terminal symbol
  - Table[NT, symbol] = rule# if symbol ∈ FIRST(NT -> rhs(#))

<table>
<thead>
<tr>
<th></th>
<th>/*</th>
<th>*/</th>
<th>id, num</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr2</td>
<td>error</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>term expr2</td>
<td>error</td>
<td>factor term2</td>
<td>error</td>
</tr>
<tr>
<td>factor term2</td>
<td>error</td>
<td>do nothing!</td>
<td></td>
</tr>
</tbody>
</table>
Next time

- Bottom-up parsers
  - Why?
  - More powerful
  - But, more complex algorithm (cannot write by hand)
  - Widely used – yacc, bison, JavaCUP

Left factoring

- Algorithm:

  ∀ A ∈ NT, find the longest prefix α that occurs in two or more right-hand sides of A

  if α ≠ ε then replace all of the A productions,
  \[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma \]

  with

  \[ A \rightarrow \alpha Z \mid \gamma \]

  \[ Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \]

  where Z is a new element of NT

  Repeat until no common prefixes remain

Code

push the start symbol, G, onto Stack

top ← top of Stack

loop forever

if top = EOF and token = EOF then break & report success

if top is a terminal then

  if top matches token then

    token ← next_token() // recognized top

  else

    push Bk, Bk-1, ..., B1 // in that order

  top ← top of Stack

else

  if TABLE[top, token] is A

    pop Stack // get rid of A

    push Bk, Bk-1, ..., B1 // in that order

  top ← top of Stack

Missing else’s for error conditions