Motivation

- Programs are just strings of text
  - But they’re strings that have a certain structure

- Informal description of syntax of a C program
  - A C program is a list of declarations and definitions
  - A function definition contains parameters and a body
  - A function body is a sequence of statements
  - A statement is an expression, if, goto, etc.
  - An expression may be assignment, addition, subtraction, etc.

Motivation (cont’d)

- We want to describe program structure precisely

- Regular expressions are not enough
  - No regular expression for balanced pairs of ( )’s
    - ("0", "0(0)", "0(0)"...) is not a regular language

- Instead, we’ll use context-free grammars
  - These are almost enough for C, C++, Java

Context-Free Grammars (CFGs)

- But CFGs can do a lot more!
  - $S \rightarrow \{ S \} \varepsilon$  // generates balanced pairs of ( )’s

- In fact, CFGs subsume REs, DFAs, NFAs
  - There is a CFG that generates any regular language
  - But REs are a better notation for regular languages

- CFGs can specify programming language syntax
  - CFGs (mostly) describe the parsing process

Formal Definition

- A context-free grammar $G$ is a 4-tuple:
  - $\Sigma$ – a finite set of terminal or alphabet symbols
    - Often written in lowercase
  - $N$ – a finite, nonempty set of nonterminal symbols
    - Often written in uppercase
    - It must be that $N \cap \Sigma = \emptyset$
  - $P$ – a set of productions of the form $N \rightarrow (\Sigma|N)^*$
    - Informally this means that the nonterminal can be replaced by the string of zero or more terminals or nonterminals to the right of the $\rightarrow$
    - Can think of productions as rewriting rules
  - $S \in N$ – the start symbol
Backus-Naur Form

- Context-free grammar production rules are also called Backus-Naur Form or BNF
  - A production like \( A \to B \cdot C \cdot D \) is written in BNF as \( A \ := \ B \cdot C \cdot D \) (Non-terminals written with angle brackets and \( := \) instead of \( \to \))
  - Often used to describe language syntax
- BNF was designed by
  - John Backus
    - Chair of the Algol committee in the early 1960s
  - Peter Naur
    - Secretary of the committee, who used this notation to describe Algol in 1962

Informal Definition of Acceptance

- A string is accepted by a CFG if there is
  - Some sequence of applying productions (rewrites) starting at the start symbol that generates the string
- Example
  - Grammar: \( S \to 0S \mid 1S \mid \epsilon \)
    - Sequence generating the string 010
      \[ S \to 0S \to 01S \to 010 \]
- Terminology
  - Such a sequence of rewrites is a derivation or parse
  - Discovering the derivation is called parsing

Derivations

- Notation
  \( \Rightarrow \) indicates a derivation of one step
  \( \Rightarrow^+ \) indicates a derivation of one or more steps
  \( \Rightarrow^* \) indicates a derivation of zero or more steps
- Example
  \( S \Rightarrow 0S \mid 1S \mid \epsilon \)
- For the string 010
  \( S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010 \)
  \( S \Rightarrow^+ 010 \)
  \( S \Rightarrow^* S \)

Example

\[
\begin{align*}
S & \to aS \mid T \\
T & \to bT \mid U \\
U & \to U \epsilon \\
\end{align*}
\]

- A derivation:
  \( S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac \)
  - Abbreviated as \( S \Rightarrow^+ ac \)
  \( S \Rightarrow T \Rightarrow U \Rightarrow \epsilon \)
- Is there any derivation
  \( S \Rightarrow^+ ccc \) ? \( S \Rightarrow^* Sa \) ?
  \( S \Rightarrow^+ bab \) ? \( S \Rightarrow^* bU \) ?

Practice

- Try to make a grammar which accepts
  \( 0^*1^* \) \( 0^*1^m \) where \( n \geq 0 \)
  \( 0^m1^n \) where \( m \leq n \)
  \( S \to A \mid B \\
A \to 0A \mid \epsilon \\
S \to 0S1 \mid \epsilon \\
S \to 0S1 \mid 0S \epsilon \\
B \to 1B \mid \epsilon \\
\)
- Give some example strings from this language
  \( S \to 0 \mid 1S \)
  \( 0, 10, 110, 1110, 11110, \ldots \)
  - What language is it?
    \( 1^*0 \)

Example (cont’d)

\[
\begin{align*}
S & \to aS \mid T \\
T & \to bT \mid U \\
U & \to U \epsilon \\
\end{align*}
\]

- Generates what language?
- Do other grammars generate this language?
  \( S \to ABC \\
A \to 0A \mid \epsilon \\
B \to bB \mid \epsilon \\
C \to cC \mid \epsilon \)
  - So grammars are not unique
Example: Arithmetic Expressions (Limited)

- $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$
- An expression $E$ is either a letter $a$, $b$, or $c$
- Or an $E$ followed by $+$ followed by an $E$
- etc.

- This describes or generates a set of strings
  - $\{a, b, c, a+b, a+a, a\cdot c, a-(b\cdot a), c\cdot(b+d)\}$

- Example strings not in the language
  - $d, c(a), b^*c$, etc.

Example, Formally

- Formally, the grammar we just showed is
  
  $\Sigma = \{+, -, *, (,), a, b, c\}$ // terminals
  
  $N = \{E\}$ // nonterminals
  
  $P = \{E \rightarrow a, E \rightarrow b, E \rightarrow c, E \rightarrow E+E, E \rightarrow E-E, E \rightarrow E*E, E \rightarrow (E)\}$ // productions
  
  $S = E$ // start symbol

Uniqueness of Grammars

- Grammars are not unique
  - Different grammars can generate same set of strings

- Following grammar generates the same set of strings as the previous grammar:
  
  $E \rightarrow E+E \mid E-E \mid E*E \mid (E)$
  
  $T \rightarrow T*P \mid P$
  
  $P \rightarrow (E) \mid a \mid b \mid c$

Notational Shortcuts

- A production is of the form
  - left-hand side (LHS) $\rightarrow$ right hand side (RHS)

- If not specified
  - Assume LHS of first listed production is the start symbol

- Productions with the same LHS
  - Are usually combined with $\mid$

- If a production has an empty RHS
  - It means the RHS is $\epsilon$

Sentential Forms and Derivations

- A sentential form is a string of terminals and nonterminals produced from that start symbol

- Inductively
  - The start symbol is a sentential form for a grammar
  - If $\alpha\delta$ is a sentential form for a grammar, where $\alpha$ and $\delta \in (\Sigma^* \cup \Gamma^*)$, and $A \rightarrow \gamma$ is a production, then $\alpha\gamma\delta$ is a sentential form for the grammar
    - In this case, we say that $\alpha\delta$ derives $\alpha\gamma\delta$ in one step, which is written as $\alpha\delta \Rightarrow \alpha\gamma\delta$

Sentential Forms Example

- Given grammar
  
  $S \rightarrow 0S \mid 1S \mid \epsilon$

- Possible derivations
  
  $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010$
  
  $S \Rightarrow 1S \Rightarrow 11S \Rightarrow 111S \Rightarrow 111$
  
  $S \Rightarrow \epsilon$

- In other words
  
  - If $S \Rightarrow^* \alpha$, then $\alpha$ is a sentential form
The Language Generated by a CFG

- The language generated by a grammar \( G \) is

\[
L(G) = \{ \omega \mid \omega \in \Sigma^* \text{ and } S \Rightarrow^* \omega \}
\]

- \( S \) is the start symbol of the grammar
- \( \Sigma \) is the alphabet for that grammar

- In other words
  - All sentential forms with only terminals
  - All strings over \( \Sigma \) that can be derived from the start symbol via one or more productions

Parse Trees

- A parse tree shows how a string is produced by a grammar
  - Root node is the start symbol
  - Each interior node is a nonterminal
  - Children of node are symbols on r.h.s of production applied to that nonterminal
  - Leaves are all terminal symbols

- Reading the leaves left-to-right shows the string corresponding to the tree

Example

\[
S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac
\]

Leftmost and Rightmost Derivation

- Leftmost derivation
  - Leftmost nonterminal is replaced in each step
- Rightmost derivation
  - Rightmost nonterminal is replaced in each step

Example

- Grammar
  - \( S \Rightarrow AB, A \Rightarrow a, B \Rightarrow b \)
  - Leftmost derivation for “ab”
    - \( S \Rightarrow AB \Rightarrow aB \Rightarrow ab \)
  - Rightmost derivation for “ab”
    - \( S \Rightarrow AB \Rightarrow Ab \Rightarrow ab \)

Parse Tree For Derivations

- Parse tree may be same for both leftmost & rightmost derivations
  - Example Grammar: \( S \rightarrow a \mid SbS \) String: \( ab \)
    - Leftmost Derivation
      \[
      S \Rightarrow SbS \Rightarrow abS \Rightarrow aba
      \]
    - Rightmost Derivation
      \[
      S \Rightarrow SbS \Rightarrow Sba \Rightarrow aba
      \]
  - Parse trees don’t show order productions are applied
  - Every parse tree has a unique leftmost and a unique rightmost derivation

Parse Tree For Derivations (cont.)

- Not every string has a unique parse tree
  - Example Grammar: \( S \rightarrow a \mid SbS \) String: \( ababa \)
    - Leftmost derivation
      \[
      S \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa
      \]
    - Another leftmost derivation
      \[
      S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa
      \]
Ambiguity

- A grammar is ambiguous if a string may have multiple leftmost (or rightmost) derivations
  - Equivalent to multiple parse trees
  - Can be hard to determine

1. \( S \rightarrow aS | T \)
   \( T \rightarrow bT | U \)
   \( U \rightarrow cU | \epsilon \)
   - No

2. \( S \rightarrow SS | () | (S) \)
   - ?

More on Leftmost/Rightmost Derivations

- Is the following derivation leftmost or rightmost?
  - Both! At most one non-terminal in each sentential form, so there’s no choice which non-terminals to expand

- How about the following derivation?
  - \( S \rightarrow bS \rightarrow bSbS \rightarrow SbabS \rightarrow ababS \rightarrow ababa \)
  - Neither! Selects left, center, left, and rightmost nonterminals

Tips for Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols
   - \( A \rightarrow xA | \epsilon \) Zero or more \( x \)'s
   - \( A \rightarrow yA | y \) One or more \( y \)'s

2. Use separate nonterminals to generate disjoint parts of a language, and then combine in a production
   - \( G = S \rightarrow AB \)
   - \( A \rightarrow aA | \epsilon \)
   - \( B \rightarrow bB | \epsilon \)
   - \( L(G) = a^*b^* \)

Tips for Designing Grammars (cont’d)

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle
   - \( \{ a^n b^n | n \geq 0 \} \) (not a regular language!)
     - \( S \rightarrow aSb | \epsilon \)
   - Example: \( S \rightarrow aSb \rightarrow aSbb \rightarrow abbb \)
     - \( \{ a^n b^n | n \geq 0 \} \)
     - \( S \rightarrow aSbb | \epsilon \)

Tips for Designing Grammars (cont’d)

- \( \{ a^n b^n | m \geq 2n, n \geq 0 \} \)
  - \( S \rightarrow aSbb | B | \epsilon \)
  - \( B \rightarrow bB | b \)

The following grammar also works:
- \( S \rightarrow aSbb | B \)
  - \( B \rightarrow bB | \epsilon \)

How about the following?
- \( S \rightarrow aSbb | BS | \epsilon \)
Tips for Designing Grammars (cont’d)

\{ a^n b^m | n \geq 0, m \geq 0 \}
Rewrite as \( a^n b^n a^m \), which now has matching superscripts (two pairs)

Would this grammar work?
\begin{align*}
S & \rightarrow aSa | B \\
B & \rightarrow bBa | ba
\end{align*}

Corrected:
\begin{align*}
S & \rightarrow aSa | B \quad \text{The outer } a^n a^n \text{ are generated first,} \\
B & \rightarrow bBa | \epsilon \quad \text{then the inner } b^n a^m
\end{align*}

Tips for Designing Grammars (cont’d)

4. For a language that’s the union of other languages, use separate nonterminals for each part of the union and then combine
\[
\{ a^n (b^m | c^m) | m > n \geq 0 \}
\]
Can be rewritten as
\[
\{ a^n b^m | m > n \geq 0 \} \cup \\
\{ a^n c^m | m > n \geq 0 \}
\]

Tips for Designing Grammars (cont’d)

\{ a^n b^m | m > n \geq 0 \} \cup \{ a^n c^m | m > n \geq 0 \}

Will this fix the ambiguity?
\begin{align*}
S & \rightarrow T | U \\
T & \rightarrow aTb | Tb | b \\
U & \rightarrow aUc | Uc | c
\end{align*}

- It’s not ambiguous, but it can generate invalid strings such as \( babb \)

Tips for Designing Grammars (cont’d)

\{ a^n b^m | m > n \geq 0 \} \cup \{ a^n c^m | m > n \geq 0 \}

Unambiguous version
\begin{align*}
S & \rightarrow T | V \\
T & \rightarrow aTb | U \\
U & \rightarrow Ub | b \\
V & \rightarrow aVc | W \\
W & \rightarrow Wc | c
\end{align*}

CFGs for Languages

- Recall that our goal is to describe programming languages with CFGs
  - We had the following example which describes limited arithmetic expressions
    \[
    E \rightarrow a | b | c | E+E | E-E | E^E | (E)
    \]
  - What’s wrong with using this grammar?
    - It’s ambiguous!
### Example: a-b-c

\[
E \rightarrow E \rightarrow a \rightarrow E \rightarrow E \rightarrow \text{a-b} \rightarrow \text{c}
\]

Corresponds to a-(b-c)

\[
E \rightarrow E \rightarrow a \rightarrow E \rightarrow c \rightarrow \text{b}
\]

Corresponds to (a-b-c)

Corresponds to (a-b-c)

Corresponds to (a-b)c

### Another Example: If-Then-Else

\[
\text{<stmt> ::= <assignment> | <if-stmt> | ...}
\]

\[
\text{<if-stmt> ::= if \text{ (<expr>) <stmt> |}
\]

\[
\text{if \text{ (<expr>) <stmt> else <stmt>}
\]

- (Here <>'s are used to denote nonterminals and ::= for productions)

- Consider the following program fragment:
  
  ```
  if (x > y)  
  if (x < z)  
  a = 1;  
  else a = 2;
  ```

- Note: Ignore newlines

### Parse Tree #1

- Else belongs to inner if

### Parse Tree #2

- Else belongs to outer if

### Dealing With Ambiguous Grammars

- Ambiguity is bad
  - Syntax is correct
  - But semantics differ depending on choice
    - Different associativity (a-b)-c vs. a-(b-c)
    - Different precedence (a-b)c vs. a-(b-c)
    - Different control flow if (if else) vs. if (if) else

- Two approaches
  - Rewrite grammar
  - Use special parsing rules
    - Depending on parsing method (learn in CMSC 430)
Fixing the Expression Grammar

- Idea: Require that the right operand of all of the operators not have an operator it in, unless it’s parenthesized
  \[ E \rightarrow E \cdot T \mid E \cdot T \mid E \cdot T \mid T \]
  \[ T \rightarrow a \mid b \mid c \mid (E) \]

- Now only one parse tree for \( a \cdot b \cdot c \)
  - Left associative
  - Exercise: Give a derivation for the string \( a \cdot (b \cdot c) \)

What if We Wanted Right-Associativity?

- Left-recursive productions are used for left-associative operators
- Right-recursive productions are used for right-associative operators
- Left:
  \[ E \rightarrow E \cdot T \mid E \cdot T \mid E \cdot T \mid T \]
  \[ T \rightarrow a \mid b \mid c \mid (E) \]
- Right:
  \[ E \rightarrow T \cdot E \mid T \cdot E \mid T \cdot E \mid T \]
  \[ T \rightarrow a \mid b \mid c \mid (E) \]

Parse Tree Shape

- The kind of recursion/associativity determines the shape of the parse tree
  - Left recursion
  - Right recursion

- Exercise: draw a parse tree for \( a \cdot b \cdot c \) in the prior grammar in which subtraction is right-associative

A Different Problem

- How about the string \( a + b \cdot c \)?
  \[ E \rightarrow E \cdot T \mid E \cdot T \mid E \cdot T \mid T \]
  \[ T \rightarrow a \mid b \mid c \mid (E) \]
- Doesn’t have correct precedence for \( \cdot \)
  - When a nonterminal has productions for several operators, they effectively have the same precedence
- How can we fix this?

Final Expression Grammar

- Exercises:
  - Construct tree and left and right derivations for \( a + b \cdot c \), \( a \cdot b + c \), \( a \cdot b \cdot c \), \( a + b + c \)
  - See what happens if you change the last set of productions to \( P \rightarrow a \mid b \mid c \mid E \mid (E) \)
  - See what happens if you change the first set of productions to \( E \rightarrow E \cdot T \mid E \cdot T \mid T \mid P \)

Regular expressions and CFGs

- Programming languages are neither regular nor context-free
  - Usually almost context-free, with some hacks
**Pushdown Automaton (PDA)**

- A pushdown automaton (PDA) is an abstract machine similar to the DFA
  - Has a finite set of states
  - Also has a pushdown stack
- Moves of the PDA are as follows:
  - An input symbol is read and the top symbol on the stack is read
  - Based on both inputs, the machine
    - Enters a new state, and
    - Writes zero or more symbols onto the pushdown stack
  - String accepted if the stack is empty at end of string

**Power of PDAs**

- PDAs are more powerful than DFAs
  - $a^nb^n$, which cannot be recognized by a DFA, can easily be recognized by the PDA
  - Stack all a symbols and, for each b, pop an a off the stack.
  - If the end of input is reached at the same time that the stack becomes empty, the string is accepted
- As with NFA, we can also have a NDPDA
  - NDPDA are more powerful than DPDA
  - NDPDA can recognize even length palindromes over $\{0,1\}^*$, but a DPDA cannot. Why? (Hint: Consider palindromes over $\{0,1\}^2\{0,1\}^*$)
  - It is true, but less clear, that the languages accepted by NDPDAs are equivalent to the context-free languages

---

**Steps of Compilation**

- Source program → Compiler → Target program

- Lexing → Parsing → Intermediate Code Generation → Optimization

**Parsing**

- There are many efficient techniques for turning strings into parse trees or ASTs
  - They all have strange names, like LL(k), SLR(k), LR(k)...
  - Take CMSC 430 for more details
- We will look at one very simple technique: recursive descent parsing
  - This is a “top-down” parsing algorithm because we’re going to begin at the start symbol and try to produce the string

---

**Recursive Descent Parsing**

- Goal
  - Determine if we can produce the string to be parsed from the grammar’s start symbol
- Approach
  - Recursively replace nonterminal with RHS of production
- At each step, we’ll keep track of two facts
  - What tree node are we trying to match?
  - What is the lookahead (next token of the input string)?
    - Helps guide selection of production used to replace nonterminal

---

**Recursive Descent Parsing (cont.)**

- At each step, 3 possible cases
  - If we’re trying to match a terminal
    - If the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we’re trying to match a nonterminal
    - Pick which production to apply based on the lookahead
    - Otherwise fail with a parsing error
Example

E → id = n | { L }
L → E ; L | ε

• One input might be
  – \{ x = 3 ; \ y = 4 ; \}

Recursive Descent Parsing (cont.)

• Key step
  – Choosing which production should be selected

• Two approaches
  – Backtracking
    • Choose some production
    • If fails, try different production
    • Parse fails if all choices fail
  – Predictive parsing
    • Analyze grammar to find FIRST sets for productions
    • Compare with lookahead to decide which production to select
    • Parse fails if lookahead does not match FIRST

First Sets

• Definition
  – \text{First}(y), for any terminal or nonterminal \( y \), is the set of initial terminals of all strings that \( y \) may expand to
  – We'll use this to decide what production to apply

• Examples
  – Given grammar \( S \to xyz \mid abc \)
    • First(xyz) = \{ x \}, First(abc) = \{ a \}
    • First(S) = First(xyz) \cup First(abc) = \{ x, a \}
  – Given grammar \( S \to A \mid B \quad A \to x \mid y \quad B \to z \)
    • First(x) = \{ x \}, First(y) = \{ y \}, First(A) = \{ x, y \}
    • First(z) = \{ z \}, First(B) = \{ z \}
    • First(S) = \{ x, y, z \}

Example (cont’d)

E → id = n | { L }
L → E ; L | ε

– And we want to turn it into a parse tree

First Sets

• Motivating example
  – The lookahead is \( x \)
  – Given grammar \( S \to xyz \mid abc \)
    • Select \( S \to xyz \) since first terminal in RHS matches \( x \)
  – Given grammar \( S \to A \mid B \quad A \to x \mid y \quad B \to z \)
  – Select \( S \to A \), since \( A \) can derive string beginning with \( x \)

• In general
  – Choose a production that can derive a sentential form beginning with the lookahead
  – Need to know what terminal may be first in any sentential form derived from a nonterminal / production

Calculating First(y)

• For terminal \( a \), \( \text{First}(a) = \{ a \} \)

• For a nonterminal \( N \):
  – If \( N \to \varepsilon \), then add \( \varepsilon \) to \( \text{First}(N) \)
  – If \( N \to \alpha_1 \alpha_2 \ldots \alpha_n \) then (note the \( \alpha_i \) are all the symbols on the right side of one single production):
    • Add \( \text{First}(\alpha_1 \alpha_2 \ldots \alpha_i) \) to \( \text{First}(N) \), where \( \text{First}(\alpha_1 \alpha_2 \ldots \alpha_i) \) is defined as
      – \( \text{First}(\alpha_i) \) if \( \varepsilon \notin \text{First}(\alpha_i) \)
      – Otherwise \( \text{First}(\alpha_i) = \varepsilon \cup \text{First}(\alpha_1 \ldots \alpha_{i-1}) \)
  • If \( \varepsilon \notin \text{First}(\alpha_i) \) for all \( \alpha_i \), then add \( \varepsilon \) to \( \text{First}(N) \)
Examples

<table>
<thead>
<tr>
<th>Syntax</th>
<th>First Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → id = n</td>
<td>{ id }</td>
</tr>
<tr>
<td>L → E : L</td>
<td>{ ε }</td>
</tr>
<tr>
<td>First(id) = { id }</td>
<td>First(id) = { id }</td>
</tr>
<tr>
<td>First(&quot;=&quot;) = { &quot;=&quot; }</td>
<td>First(&quot;=&quot;) = { &quot;=&quot; }</td>
</tr>
<tr>
<td>First(n) = { n }</td>
<td>First(n) = { n }</td>
</tr>
<tr>
<td>First(&quot;{&quot;) = { &quot;{&quot; }</td>
<td>First(&quot;{&quot;) = { &quot;{&quot; }</td>
</tr>
<tr>
<td>First(&quot;{&quot;) = { &quot;{&quot; }</td>
<td>First(&quot;{&quot;) = { &quot;{&quot; }</td>
</tr>
<tr>
<td>First(&quot;;&quot;) = { &quot;;&quot; }</td>
<td>First(&quot;;&quot;) = { &quot;;&quot; }</td>
</tr>
<tr>
<td>First(E) = { id, &quot;{&quot; }</td>
<td>First(E) = { id, &quot;{&quot; }</td>
</tr>
<tr>
<td>First(L) = { id, &quot;,&quot; }</td>
<td>First(L) = { id, &quot;{&quot; }</td>
</tr>
</tbody>
</table>

Recursive Descent Parser Implementation

- For terminals, create function match(a)
  - If lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Otherwise fails with a parse error if lookahead is not a
- In algorithm descriptions, consider parse_a, parse_term(a) to be aliases for match(a)
- For each nonterminal N, create a function parse_N
  - Called when we're trying to parse a part of the input which corresponds to (or can be derived from) N
  - parse_S for the start symbol S begins the parse

Parser Implementation (cont.)

- The body of parse_N for a nonterminal N does the following
  - Let N → β_1 ... | β_k be the productions of N
    - Here β_i is the entire right side of a production's a sequence of terminals and nonterminals
  - Pick the production N → β_j such that the lookahead is in First(β_j)
    - It must be that First(β_i) \cap First(β_j) = ∅ for i ≠ j
    - If there is no such production, but N → ε then return
      - Otherwise fail with a parse error
  - Suppose β_j = α_1 α_2 ... α_p. Then call parse_α_1(); ... ; parse_α_p(); to match the expected right-hand side, and return

Recursive Descent Parser

- Given grammar S → xyz | abc
  - First(xyz) = { x }, First(abc) = { a }:
- Parser
  ```java
  parse_S() {
    if (lookahead == "x") {
      match("x"); match("y"); match("z");  // S → xyz
    } else if (lookahead == "a") {
      match("a"); match("b"); match("c");  // S → abc
    } else error();
  }
  ```

Recursive Descent Parser

- Given grammar S → A | B : A → x | y : B → z
  - First(A) = { x, y }, First(B) = { z }:
- Parse
  ```java
  parse_S() {
    if (lookahead == "x")
      match("x");  // A → x
    else if (lookahead == "y")
      parse_A();  // S → A
    else if (lookahead == "z")
      match("y");  // A → y
        else error();
    parse_B();  // S → B
          else error();
  }
  ```
Example

E → id = n | { L }
L → E ; L | ε

parse_E() {
  if (lookahead == "id") {
    match("id");
    match("=");
    parse_E();
    match("n");
    return E;
  }
  else if (lookahead == ":") {
    match(":");
    parse_L();
    return L;
  }
  else { error();
  }
}

L

Things to Notice

- If you draw the execution trace of the parser
  - You get the parse tree

- Examples
  - Grammar
    • S → xyz
    • S → abc
  - String "xyz"
    • parse_S() {
      match("x");
      match("y");
      match("z");
    }

- Things to Notice (cont.)

  • This is a predictive parser
    - Because the lookahead determines exactly which production to use
  • This parsing strategy may fail on some grammars
    - Possible infinite recursion
    - Production First sets overlap
    - Production First sets contain ε
  • Does not mean grammar is not usable
    - Just means this parsing method not powerful enough
    - May be able to change grammar

- Left Factoring

  • Consider parsing the grammar E → ab | ac
    - First(ab) = a
    - First(ac) = a
    - Parser cannot choose between RHS based on lookahead!
  • Parser fails whenever A → α₁ | α₂ and
    - First(α₁) n First(α₂) ! = ε or ∅
  • Solution
    - Rewrite grammar using left factoring

- Left Factoring Algorithm

  • Given grammar
    - A → x₀ | x₁ | ... | xₙ | β
  • Rewrite grammar as
    - A → x₀L | β
    - L → α₁ | α₂ | ... | αₙ
  • Repeat as necessary
  • Examples
    - S → ab | ac
    - S → abcA | abB | a
      | S → aL | L → b | c
    - L → bcA | bB | ε
      | S → aL | L → bL°C | L→ cA | B

- Left Recursion

  • Consider grammar S → Sa | ε
    - First(Sa) = a, so we're ok as far as which production
    - Try writing parser
      • parse_S() {
        if (lookahead == "a") {
          parse_S();
          match("a"); // S → Sa
        }
        else {
        }
      }
    - Body of parse_S() has an infinite loop
      • if (lookahead == "a") then parse_S()
    - Infinite loop occurs in grammar with left recursion
Right Recursion

- Consider grammar \( S \rightarrow aS | \varepsilon \)
  - Again, \( \text{First}(aS) = a \)
  - Try writing parser
    ```
    
    parse_S() {
      if (lookahead == "a") {
        match("a");
        parse_S();
        // S \rightarrow aS
      } else {
      }
    }
    ```
    - Will \( \text{parse}_S() \) infinite loop?
      - Invoking \( \text{match()} \) will advance lookahead, eventually stop
    - Top down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  ```
  A \rightarrow A_1 | A_2 | \ldots | A_n | \beta
  ```
  - Why must \( \beta \) exist?
- Rewrite grammar as
  ```
  A \rightarrow \beta L
  L \rightarrow a_1 L | a_2 L | \ldots | a_n L | \varepsilon
  ```
  - Replaces left recursion with right recursion
  - Repeat as necessary

Eliminating Left Recursion (cont.)

- Examples
  ```
  \( S \rightarrow S a | \varepsilon \)
  \( S \rightarrow S a | S b | \varepsilon \)
  ```
  - May need more powerful algorithms to eliminate mutual recursion leading to left recursion
    - \( S \rightarrow A a | b \)
    - \( A \rightarrow S b \)

Expr Grammar for Top-Down Parsing

- \( E \rightarrow T E' \)
- \( E' \rightarrow \varepsilon | + E \)
- \( T \rightarrow P T' \)
- \( T' \rightarrow \varepsilon | ^* T \)
- \( P \rightarrow n | ( E ) \)

  - Notice we can always decide what production to choose with only one symbol of lookahead

Tradeoffs with Other Approaches

- Recursive descent parsers are easy to write
  - The formal definition is a little clunky, but if you follow the code then it’s almost what you might have done if you weren’t told about grammars formally
  - They’re unable to handle certain kinds of grammars
- Recursive descent is good for a simple parser
  - Though tools can be fast if you’re familiar with them
- Can implement top-down predictive parsing as a table-driven parser
  - By maintaining an explicit stack to track progress

Tradeoffs with Other Approaches

- More powerful techniques need tool support
  - Can take time to learn tools (lex/flex, yacc/bison)
- Main alternative is bottom-up, shift-reduce parser
  - Replaces RHS of production with LHS (nonterminal)
  - Example grammar
    ```
    S \rightarrow aA A \rightarrow Bc B \rightarrow b
    ```
  - Example parse
    ```
    abc \Rightarrow aBc \Rightarrow aA \Rightarrow S
    ```
  - Derivation happens in reverse
  - Something to look forward to in CMSC 430
What's Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    - Parentheses
    - Extra nonterminals for precedence
  - This extra stuff is needed for parsing

- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts

Abstract Syntax Trees (cont.)

- Intuitively, ASTs correspond to the data structure you'd use to represent strings in the language
  - Note that grammars describe trees
  - So do OCaml datatypes (which we'll see later)
  - \[ E \rightarrow a \mid b \mid c \mid E + E \mid E - E \mid E \cdot E \mid (E) \]

Producing an AST

- To produce an AST, we can modify the parse() functions to construct the AST along the way
  - match(a) returns an AST node (leaf) for a
  - Parse_A returns an AST node for A
    - AST nodes for RHS of production become children of LHS node

Summary

- Learned a little about parsing
  - Recursive descent parser
  - Predictive parsing using FIRST sets
- Rewriting grammars for predicative parsing
  - Left factoring
  - Eliminating left recursion
- Abstract syntax trees (ASTs)