CMSC 330: Organization of Programming Languages

Context Free Grammars
A sentence (S) is composed of a noun phrase (NP) and a verb phrase (VP).

A noun phrase may be composed of a determiner (D/DET) and a noun (N).

A noun phrase may also be composed of an adjective (ADJ) and a noun (N).

A verb phrase may be composed of a verb (V) and a noun (N) or noun phrase (NP).
Chomsky Hierarchy

- Categorization of various languages and grammars
- Each is strictly more restrictive than the previous
- First described by Noam Chomsky in 1956

- Type 0: Any formal grammar
  - Turing machines
- Type-1: \( \alpha A \beta \rightarrow \alpha \gamma \beta \)
  - Linear bounded automata
- Type-2: \( A \rightarrow \gamma \)
  - Pushdown automata (PDAs)
- Type-3: Regular expressions
  - Finite state automata (NFAs/DFAs)
Program structure

Syntax
- Source code form
- “What a program looks like”
- In general, syntax is described using grammars.

Semantics
- Execution behavior
- “What a program does”
Motivation

- Programs are just strings of text
  - But they’re strings that have a certain structure
    - 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 either an expression, an if, a goto, etc.
    - An expression may be assignment, addition, subtraction, etc.

- We want to solve two problems
  - We want to describe programming languages precisely
  - We need to describe more than the regular languages
    - Recall that regular expressions, DFAs, and NFAs are limited in their expressiveness
Architecture of Compilers, Interpreters

Source ➔ Parser ➔ Static Analyzer ➔ Intermediate Representation ➔ Back End

Compiler / Interpreter
Front End

- Responsible for turning source code (sequence of symbols) into representation of program structure
- Parser generates parse trees, which static analyzer processes
Parser Architecture

- **Scanner / lexer** converts sequences of symbols into tokens (keywords, variable names, operators, numbers, etc.)
- **Parser** (yes, a parser contains a parser!) converts sequences of tokens into parse tree; implements a *context free grammar*
  - We can’t do everything as a regular expression – not powerful enough
Context Free Grammar (CFG)

- A way of describing sets of strings (= languages)
  - The notation $L(S)$ denotes the language of strings defined by $S$

- Example grammar: $S \rightarrow 0S \mid 1S \mid \varepsilon$

  String $s' \in L(S)$ iff
  - $s' = \varepsilon$, or $\exists s \in L(S)$ such that $s' = 0s$, or $s' = 1s$

- Grammar is same as regular expression $(0|1)^*$
  - Generates / accepts the same set of strings
Context-Free Grammars (CFGs)

- But CFGs can do what regexes cannot
  - $S \rightarrow (S) | \epsilon$ // represents 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: Context-Free Grammar

- A CFG $G$ is a 4-tuple $(\Sigma, N, P, S)$
  - $\Sigma$ – alphabet (finite set of symbols, or terminals)
    - 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: the nonterminal can be replaced by the string of zero or more terminals / nonterminals to the right of the $\rightarrow$
    - Can think of productions as rewriting rules (more later)
  - $S \in N$ – the start symbol
Backus-Naur Form

- Context-free grammar production rules are also called Backus-Naur Form or BNF
  - A production like $A \rightarrow B \cdot c \cdot D$ is written in BNF as $<A> ::= <B> \cdot c \cdot <D>$ (Non-terminals written with angle brackets and ::= instead of →)
  - 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
Generating Strings

- We can think of a grammar as generating strings by rewriting.

Example grammar: $S \rightarrow 0S | 1S | \varepsilon$

- $S \Rightarrow 0S$  // using $S \rightarrow 0S$
- $\Rightarrow 01S$  // using $S \rightarrow 1S$
- $\Rightarrow 011S$  // using $S \rightarrow 1S$
- $\Rightarrow 011$  // using $S \rightarrow \varepsilon$
Accepting Strings (Informally)

- Determining if $s \in L(S)$ is called acceptance: goal is to find a rewriting from $S$ that yields $s$
  - $011 \in L(S)$ according to the previous rewriting
  - A rewriting is some sequence of productions (rewrites) applied starting at the start symbol

- 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**
  - $S \rightarrow 0S \mid 1S \mid \varepsilon$

- **For the string 010**
  - $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010$
  - $S \Rightarrow^+ 010$
  - $S \Rightarrow^* S$
Language Generated by Grammar

- \( L(G) \) the language defined by \( G \) is

\[
L(G) = \{ \omega \in \Sigma^* | S \Rightarrow^+ \omega \}
\]

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

- In other words
  - All strings over \( \Sigma \) that can be derived from the start symbol via one or more productions
Try to make a grammar which accepts

- $0^*|1^*$
- $0^n1^n$ where $n \geq 0$
- $0^n1^m$ where $m \leq n$

$$S \rightarrow A \mid B$$
$$A \rightarrow 0A \mid \epsilon$$
$$B \rightarrow 1B \mid \epsilon$$

Give some example strings from this language

- $S \rightarrow 0 \mid 1S$
  - 0, 10, 110, 110, 11100, ...
- What language is it?
  - $1^*0$
Example: Arithmetic Expressions

- E → a | b | c | E+E | E-E | E*E | (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*c, a-(b*a), c*(b + a), …\}

- Example strings not in the language
  - d, c(a), a+, b**c, etc.
Formal Description of Example

Formally, the grammar we just showed is:

- \( \Sigma = \{ +, -, *, (, ), a, b, c \} \)  
  \hspace{1cm} // terminals
- \( N = \{ E \} \)  
  \hspace{1cm} // nonterminals
- \( P = \{ E \rightarrow a, E \rightarrow b, E \rightarrow c, \)  
  \hspace{1cm} // productions
  
  \hspace{1cm} E \rightarrow E-E, E \rightarrow E+E, \)
  
  \hspace{1cm} E \rightarrow E*E, \)
  
  \hspace{1cm} E \rightarrow (E) \)
- \( S = E \)  
  \hspace{1cm} // start symbol
(Non-)Uniqueness of Grammars

- Different grammars generate the same set of strings (language)

- The following grammar generates the same set of strings as the previous grammar

\[
\begin{align*}
E & \rightarrow E + T \mid E - T \mid T \\
T & \rightarrow T^*P \mid P \\
P & \rightarrow (E) \mid a \mid b \mid c
\end{align*}
\]
Notational Shortcuts

- A production is of the form
  - left-hand side (LHS) → 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 |
- If a production has an empty RHS
  - It means the RHS is $\varepsilon$

\[
\begin{align*}
S & \rightarrow ABC \quad // \ S \text{ is start symbol} \\
A & \rightarrow aA \\
& \quad | \quad b \quad // \ A \rightarrow b \\
& \quad | \quad // \ A \rightarrow \varepsilon
\end{align*}
\]
Parse Trees

- Parse tree shows how a string is produced by a grammar
  - Root node is the start symbol
  - Every internal node is a nonterminal
  - Children of an internal node
    - Are symbols on RHS of production applied to nonterminal
  - Every leaf node is a terminal or $\varepsilon$

- Reading the leaves left to right
  - Shows the string corresponding to the tree
Parse Tree Example

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

\[ S \rightarrow aS \mid T \\
T \rightarrow bT \mid U \\
U \rightarrow cU \mid \varepsilon \]
Parse Trees for Expressions

- A parse tree shows the structure of an expression as it corresponds to a grammar:

  \[ E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E) \]

- Parse trees for expressions:
  - \( a \)
  - \( a*c \)
  - \( c*(b+d) \)

```
  a
  E
   a
```

```
  a*c
  E
   *
    E
     a
     c
```

```
  c*(b+d)
  E
   *
    E
     (E)
```

```
  E + E
  E
   +
    E
     b
     d
```
Practice

\[ E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E) \]

Make a parse tree for…

- \( a*b \)
- \( a+(b-c) \)
- \( d*(d+b)-a \)
- \( (a+b)*(c-d) \)
- \( a+(b-c)*d \)
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: \( aba \)

  **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
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 derivations
  - Equivalent to multiple parse trees
  - Can be hard to determine

1. \[ S \rightarrow aS \mid T \]
   \[ T \rightarrow bT \mid U \]
   \[ U \rightarrow cU \mid \varepsilon \]
   No

2. \[ S \rightarrow T \mid T \]
   \[ T \rightarrow Tx \mid Tx \mid x \mid x \]
   Yes

3. \[ S \rightarrow SS \mid () \mid (S) \]
   ?
Ambiguity (cont.)

Example

• Grammar: $S \rightarrow SS \mid () \mid (S)$  
  String: $()()()$

• 2 distinct (leftmost) derivations (and parse trees)
  
  $S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$

  $S \Rightarrow SS \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$
Recall that our goal is to describe programming languages with CFGs.

We had the following example which describes limited arithmetic expressions:

\[ E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E\cdot E \mid (E) \]

What’s wrong with using this grammar?

- It’s ambiguous!
Example: $a \cdot b \cdot c$

\[
E \Rightarrow E \cdot E \Rightarrow a \cdot E \Rightarrow a \cdot E \cdot E \Rightarrow a \cdot b \cdot E \Rightarrow a \cdot b \cdot c
\]

Corresponds to $a \cdot (b \cdot c)$

\[
E \Rightarrow E \cdot E \Rightarrow E \cdot E \cdot E \Rightarrow a \cdot E \cdot E \Rightarrow a \cdot b \cdot E \Rightarrow a \cdot b \cdot c
\]

Corresponds to $(a \cdot b) \cdot c$
Another Example: If-Then-Else

\[
\begin{align*}
\text{<stmt>} & \rightarrow \text{<assignment>} | \text{<if-stmt>} | ... \\
\text{<if-stmt>} & \rightarrow \text{if (}<\text{expr}>\text{)} \text{<stmt>} | \\
& \quad \text{if (}<\text{expr}>\text{)} \text{<stmt>} \text{ else <stmt>} \\
\end{align*}
\]
(Note < >’s are used to denote nonterminals)

- Consider the following program fragment

  ```
  if (x > y) 
  \quad \text{if (}x < z\text{)}
  \quad a = 1;
  \quad \text{else } a = 2;
  \end{verbatim}
  (Note: Ignore newlines)
Else belongs to inner if
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 \(\text{if (if else)}\) vs. \(\text{if (if) else}\)

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

- Require right operand to not be bare expression
  \[ E \rightarrow E+T \mid E-T \mid E*T \mid T \]
  \[ T \rightarrow a \mid b \mid c \mid (E) \]

- Corresponds to left associativity

- Now only one parse tree for \( a-b-c \)
  - Find derivation
What If We Want Right Associativity?

- **Left-recursive productions**
  - Used for left-associative operators
  - Example
    \[ E \rightarrow E+T \mid E-T \mid E*T \mid T \]
    \[ T \rightarrow a \mid b \mid c \mid (E) \]

- **Right-recursive productions**
  - Used for right-associative operators
  - Example
    \[ E \rightarrow T+E \mid T-E \mid T*E \mid T \]
    \[ T \rightarrow a \mid b \mid c \mid (E) \]
Parse Tree Shape

- The kind of recursion determines the shape of the parse tree

left recursion

right recursion
A Different Problem

- How about the string $a+b*c$?
  
  $E \rightarrow E+T \mid E-T \mid E*T \mid T$
  
  $T \rightarrow a \mid b \mid c \mid (E)$

- Doesn’t have correct precedence for $*$
  - When a nonterminal has productions for several operators, they effectively have the same precedence

- Solution – Introduce new nonterminals
Final Expression Grammar

E → E+T | E-T | T  
T → T*P | P  
P → a | b | c | (E)  

lowest precedence operators  
higher precedence  
highest precedence (parentheses)

Controlling precedence of operators
  • Introduce new nonterminals
  • Precedence increases closer to operands

Controlling associativity of operators
  • Introduce new nonterminals
  • Assign associativity based on production form
    ➢ E → E+T (left associative) vs. E → T+E (right associative)
Tips For Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols
   
   \[ A \rightarrow xA \mid \epsilon \]  // Zero or more x’s
   
   \[ A \rightarrow yA \mid y \]  // One or more y’s

2. Use separate non-terminals to generate disjoint parts of a language, and then combine in a production
   
   \[ \{ a^*b^* \} \]  // a’s followed by b’s
   
   \[ S \rightarrow AB \]
   
   \[ A \rightarrow aA \mid \epsilon \]  // Zero or more a’s
   
   \[ B \rightarrow bB \mid \epsilon \]  // Zero or more b’s
Tips For Designing Grammars (cont.)

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

\{a^n b^n \mid n \geq 0\} \quad // \text{N a’s followed by N b’s}
S \rightarrow aSb \mid \varepsilon

Example derivation: \( \text{S} \Rightarrow \text{aSb} \Rightarrow \text{aaSbb} \Rightarrow \text{aabb} \)

\{a^n b^{2n} \mid n \geq 0\} \quad // \text{N a’s followed by 2N b’s}
S \rightarrow aSbb \mid \varepsilon

Example derivation: \( \text{S} \Rightarrow \text{aSbb} \Rightarrow \text{aaSbbbb} \Rightarrow \text{aabbbb} \)
4. For a language that is the union of other languages, use separate nonterminals for each part of the union and then combine

\[
\{ a^n(b^m|c^m) \mid m > n \geq 0 \}
\]

Can be rewritten as

\[
\{ a^n b^m \mid m > n \geq 0 \} \cup \{ a^n c^m \mid m > n \geq 0 \}
\]

S → T | V
T → aTb | U
U → Ub | b
V → aVc | W
W → Wc | c