CS351
Context Free Grammars

As we have shown previously with the pumping lemma, there are several languages that we can describe that are not regular. There are several classes of languages larger than that of regular languages. One of these are the "context-free" languages. These languages can be described by context-free grammars (CFG). The name context-free comes from the property that we will be able to substitute strings for variables regardless of the context (there is a more powerful class of grammars called context-sensitive that we'll see briefly later).

Context-free grammars are useful in many applications today that require parsing. Examples of parsing and the application of CFG's include natural language processing, creating compilers that recognize a computer program, or defining the structure of web documents via DTD's and XML.

The context-free grammar is somewhat like the regular expression is for regular languages. Just as we could build a DFA for a regular expression, we'll see that we can build a machine that is equivalent to the context-free grammar. This machine is called the pushdown automaton.

Context Free Grammar Example

One example of a context free grammar is the language of palindromes. We can easily show using the pumping lemma that the language $L = \{ w \mid w = w^R \}$ is not regular. However, we can describe this language by the following context-free grammar over the alphabet {0,1}:

$P \rightarrow \varepsilon$
$P \rightarrow 0$
$P \rightarrow 1$
$P \rightarrow 0P0$
$P \rightarrow 1P1$

In this grammar, P specifies the palindrome that we can create inductively. As the basis we can have palindromes of empty, 0, or 1. Then based on something that is already a palindrome, we can construct a larger palindrome by surrounding it by 0's or 1's.

Sometimes this grammar is written more compactly as:

$P \rightarrow \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$

That is, we combine all the productions for a single variable together.

Definition of CFG's

Formally, a CFG is described as:

1. There is a finite set of symbols that form the strings, e.g. there is a finite alphabet. In the lingo of context-free grammars, the alphabet symbols are called **terminals** because when we create the parse tree, the symbols will end up at the leaves of the tree and terminate any branches.
2. There is a finite set of **variables**, sometimes called non-terminals or syntactic categories. Each variable represents a language (i.e. a set of strings). In the palindrome example, the only variable is P.
3. One of the variables is the **start symbol**. Other variables may exist to help define the language.
4. There is a finite set of **productions** or production rules that represent the recursive definition of the language. Each production is defined:
    a. Has a single variable that is being defined to the left of the production
    b. Has the production symbol →
    c. Has a string of zero or more terminals or variables, called the body of the production. To form strings we can substitute each variable's production in for the body where it appears.

A CFG G may then be represented by these four components, denoted G=(V,T,P,S) where V is the set of variables, T is the set of terminals, P is the set of productions, and S is the start symbol.

Example CFG

Consider a typical programming language where we want to define what an arithmetic expression is. We can do this through a CFG. Here is a subset of what might go into a CFG grammar for representing arithmetic expressions and identifiers:

1. E→I // Expression is an identifier
2. E→E+E // Add two expressions
3. E→E*E // Multiply two expressions
4. E→(E) // Add parenthesis
5. I→ L // Identifier is a Letter
6. I→ ID // Identifier + Digit
7. I→ IL // Identifier + Letter
8. D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 // Digits
9. L → a | b | c | … A | B | … Z // Letters

Note that identifiers here are actually regular. An identifier could be described as: (letter)(letter + digit)* . That is, we must start an identifier with a letter and then can follow that by any sequence of letters and digits.

Production 1 is the basis rule that says an expression is an identifier. Productions 2-4 inductively build bigger expressions from the basis.

Production 5 is the basis for an identifier. Productions 6-7 then inductively create larger identifier.

Productions 8-9 define the terminals of this problem. We have letters defined by production 8, and digits defined by production 9.

Recursive Inference and Derivations

The process of coming up with strings that satisfy individual productions and then concatenating them together according to more general rules is called *recursive inference*. For example, using Rule 8 let's say that we know D → 5 and using Rule 9 we know that L=r. From rule 5, I→L, we have that r is an identifier. We can then apply recursive inference using rule 6 for I→ID and get that I → rD. We can use the D of 5 to get I→r5. Finally, we know from rule 1 that E→I, so r5 is also an expression.

Exercise: Show the recursive inference for arriving at (x+int1)*10 is an expression.

With recursive inference we are able to show that the resulting string we get after processing the rules leads us to a new string that is also in the language. Another approach is to use the productions from head to body by expanding the start symbol first and working our way down. This process is called *derivation*.

For example, given : a*(a+b1) we can derive this by:

$E \Rightarrow E*E \Rightarrow I*E \Rightarrow L*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(L+E) \Rightarrow a*(a+E)$
$\Rightarrow a*(a+I) \Rightarrow a*(a+ID) \Rightarrow a*(a+LD) \Rightarrow a*(a+bD) \Rightarrow a*(a+b1)$

Note that at each step of the productions we could have chosen any one of the variables to replace with a more specific rule.

More formally, we can describe the process of deriving strings inductively. First we need to introduce some new terminology:

The process of deriving a string by applying a production from head to body is denoted by $\Rightarrow$. If α and β are strings consisting of terminals and variables, and A is a variable, then let $A \rightarrow \gamma$ be a production of grammar G. We can then say $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$.

Often we will assume we are working with grammar G, and leave it off: $\alpha A \beta \Rightarrow \alpha\gamma\beta$.

Additionally, just as we defined δ^, the extended transition function that accepts a string, we can also define a similar notion for the derivation $\Rightarrow$. If we process multiple derivation steps, we use a * to indicate "zero or more steps" as follows inductively:

Basis: For any string α of terminals and variables, we can say $\alpha \Rightarrow^* \alpha$. That is, any string derives itself.

Induction: If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$. That is, if alpha can become beta in zero or more steps, then we can take one more step to gamma meaning alpha derives gamma. The proof is straightforward.

We already saw an example of $\Rightarrow$ in deriving a*(a+b1).
We could have used $\Rightarrow^*$ to condense the derivation. For example, we could just straight to $E \Rightarrow^* E*(E+E)$ or even straight to the final step of $E \Rightarrow^* a*(a+b1)$.

Leftmost and Rightmost Derivations

In the previous example we used a derivation called a *leftmost derivation*. We can specifically denote a leftmost derivation using the subscript "lm", as in:

$\Rightarrow_{lm}$ or $\Rightarrow^*_{lm}$

Similarly, we will also have a rightmost derivation which we can specifically denote via:

$\Rightarrow_{rm}$ or $\Rightarrow^*_{rm}$

A leftmost derivation is simply one in which we replace the leftmost variable in a production body by one of its production bodies first, and then work our way from left to right. A rightmost derivation is one in which we replace the rightmost variable by one of its production bodies first, and then work our way from right to left.

Example: : a*(a+b1) was already shown previously using a leftmost derivation. We can also come up with a rightmost derivation, but we must make replacements in different order:

$E \Rightarrow_{rm} E*E \Rightarrow_{rm} E * (E) \Rightarrow_{rm} E*(E+E) \Rightarrow_{rm} E*(E+I) \Rightarrow_{rm} E*(E+ID) \Rightarrow_{rm} E*(E+I1) \Rightarrow_{rm} E*(E+L1) \Rightarrow_{rm} E*(E+b1) \Rightarrow_{rm} E*(I+b1) \Rightarrow_{rm} E*(L+b1) \Rightarrow_{rm} E*(a+b1) \Rightarrow_{rm} I*(a+b1) \Rightarrow_{rm} L*(a+b1) \Rightarrow_{rm} a*(a+b1)$

Any derivation has an equivalent leftmost and rightmost derivation. That is, $A \Rightarrow^* \alpha$. iff $A \Rightarrow^*_{lm} \alpha$ and $A \Rightarrow^*_{rm} \alpha$.

Language of a CFG

The language that is represented by a CFG G(V,T,P,S) may be denoted by L(G), is a Context Free Language (CFL) and consists of terminal strings that have derivations from the start symbol:

$$L(G) = \{ w \text{ in } T \mid S \Rightarrow^*_G w \}$$

Sentential Forms

A sentential form is a special name given to derivations from the start symbol. If we have a string α that consists entirely of terminals or variables, then $S \Rightarrow^* \alpha$ where S is the start symbol is a sentential form.

Note that we can have leftmost or rightmost sentential forms based on which type of derivation we are using. Also note that the CFL L(G) consists solely of terminals from G.

Exercises:

Give a CFG for the CFL: $\{0^n1^n \mid n \geq 1 \}$

Give a CFG for the CFL: $\{a^ib^jc^k \mid i \neq j \text{ or } j \neq k \}$

Parse Trees

A parse tree is a representation of a derivation that can be quite useful in practice. The parse tree is a common data structure that is used in implementing any type of parser for a grammar, e.g., a compiler. If we can generate multiple parse trees then that means that there is ambiguity in the language – this is often undesirable, for example, in a programming language we would not like the computer to interpret a line of code in a way different than what the programmer intends. However, sometimes an unambiguous language is difficult or impossible to avoid.

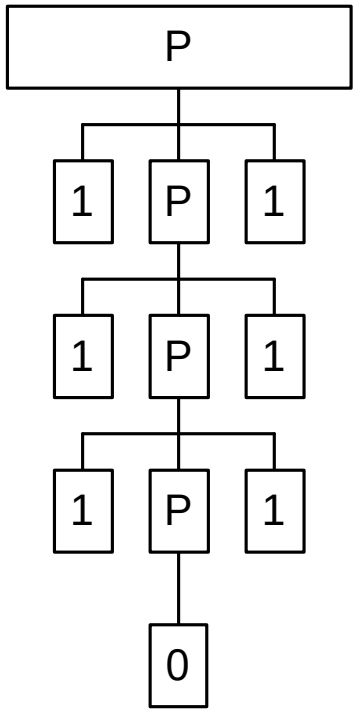
A parse tree for a CFG is constructed according to the following rules:

1. Each interior node is labeled by a variable in V
2. Each leaf node is labeled by a variable, terminal, or ε. If ε, then the parent of this node must have no other children. If we expand each variable, then ultimately leaf nodes must be terminals.
3. If an interior node is labeled P, then the children are labeled $X_1X_2X_3\ldots X_n$ where we have the production:

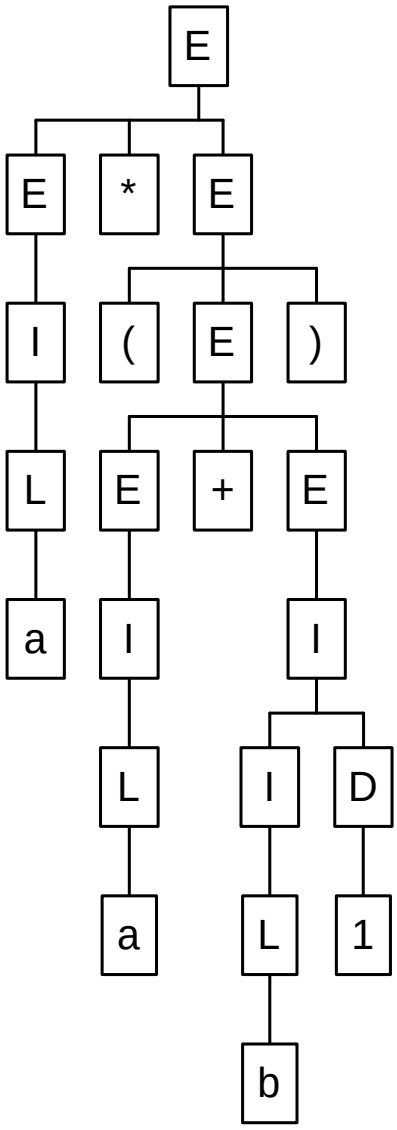
$$P \rightarrow X_1X_2X_3\ldots X_n$$

Here is a sample parse tree for the palindrome CFG for 1110111:

$P \rightarrow \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$



Here is a leftmost derivation of the parse tree for a*(a+b1) :



Is the rightmost parse tree any different from the leftmost parse tree?

Later we'll see examples of some CFG's that have multiple leftmost parse trees – these will be ambiguous languages.

The **yield** of the parse tree is the string that results when we concatenate the leaves from left to right (e.g., doing a leftmost depth first search). The yield is always a string that is derived from the root and is guaranteed to be a string in the language L.

Inference, Derivations, and Parse Trees

So far we have used the following forms to describe the processing of context-free grammars to describe whether or not a string s is in the language given a CFG with start symbol A:

1. The recursive inference procedure run on s can determine that s is in the language
2. $A \Rightarrow^* s$
3. $A \Rightarrow^*_{lm} s$
4. $A \Rightarrow^*_{rm} s$
5. The parse tree rooted at A contains s as its yield

All of these forms are equivalent for strings consisting of terminal symbols.
All of these forms except for #1 are equivalent for strings consisting of terminals or variables (this is because we only defined recursive inference for terminal symbols). However, derivations and parse trees are equivalent even including variables. This means that if we can create a parse tree of some sort, we can create a corresponding derivation, either leftmost, rightmost, or mixed, that expresses the same behavior as the parse tree.

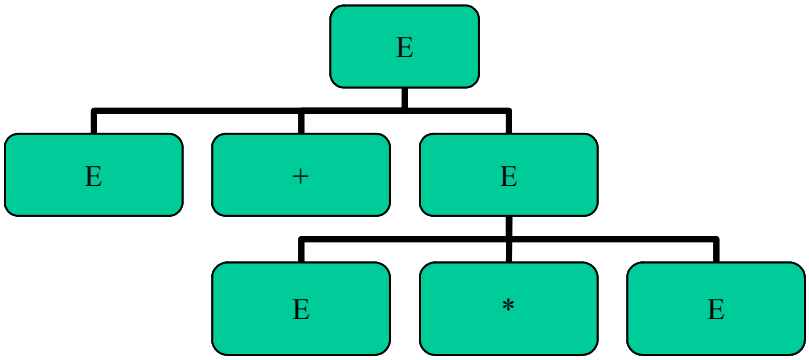
These equivalences are proven in the textbook and we will skip them here. Essentially we show the following equivalences:

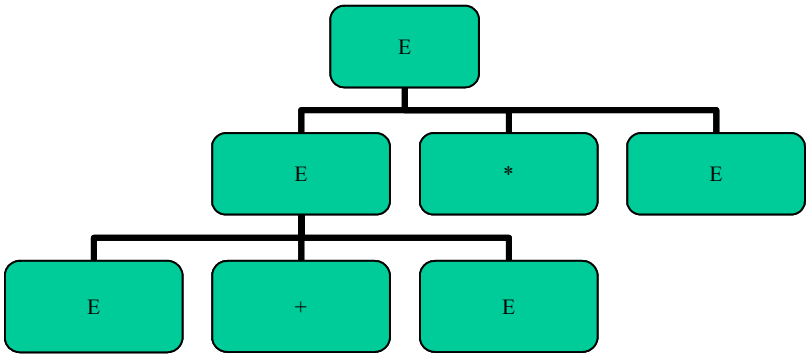
> Recursive Inferences → Parse Tree → (Left | Right derivation) → derivation → Recursive Inference

The loop back to recursive inferences completes the equivalence. To go from recursive inferences to parse trees, we create a child/parent relationship each time we make a recursive inference. The parse tree can generate a leftmost derivation by following leftmost children in the tree first, while the rightmost derivation examines rightmost children in the tree first. Finally a derivation to recursive inference is done by showing that individual productions of the form $A \rightarrow w$ can be built into $A \Rightarrow^* w$.

Ambiguous Grammars

- A CFG is ambiguous if one or more terminal strings have multiple leftmost derivations from the start symbol.
  - Equivalently: multiple rightmost derivations, or multiple parse trees.
- Examples
  - $E \rightarrow E+E \mid E*E$
  - E+E*E can be parsed as
    - $E \Rightarrow E+E \Rightarrow E+E*E$
    - $E \Rightarrow E*E \Rightarrow E+E*E$





Exercise: Is the following grammar ambiguous?
S→AS | ε
A→A1 | 0A1 | 01

Try deriving for 00111:

Removing Ambiguity

- No algorithm can tell us if an **arbitrary** CFG is ambiguous in the first place
  - Halting / Post Correspondence Problem
- Why care?
  - Ambiguity can be a problem in things like programming languages where we want agreement between the programmer and compiler over what happens
- Solutions
  - Apply precedence
  - e.g. Instead of: E→ E+E | E*E
  - Use: E→ T | E + T, T→ F | T * F
    - This rule says we apply + rule before the * rule (which means we multiply first before adding)