CSC444

October 26, 2014

Contents

1 Parse trees

Let G be a context-free grammar. We've already seen that there may be strings $w \in \mathcal{L}(G)$ that do not have a unique derivation.

Example 1.1. Let G be given by $V = \{S, (,)\}$, $\Sigma = \{(,)\}$, $R = \{S \to \varepsilon, S \to SS, S \to (S)\}$.
Sample derivations:

- $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$
- $S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ()()$

In some sense, these derivations are "the same".

- The *rules* used are the same.
- The rules are applied at the same *places* in the intermediate strings.

Only the *order* in which the rules are applied is *different*.

We would like a way to express that although the rules are applied in a different order, these *two derivations are in essence the same*.

One way to do this is by considering the derivation pictorially using what are called *parse trees*.

Example 1.2. The parse tree for both derivations given above is the following:

```
      S
    /   \
   S     S
  /|\   /|\
 ( S ) ( S )
   |     |
   e     e
```


The *nodes* in the tree are labeled by symbols in V. The topmost node is called the *root*, and the nodes along the bottom are the *leaves*. All leaves are labeled by terminals or ε.

By concatenating the labels of the leaves, from left to right, we obtain the *derived string* of terminals, called the yield of the parse tree.

1.1 Definitions

Let $G = (V, \Sigma, R, S)$ be an arbitrary context-free grammar. Then the *parse tree* for G is defined as follows:

- There is a parse tree for each for each $a \in \Sigma$. It has a single node that is both the root and a leaf. The yield of the parse tree is a.
- If $A \to \varepsilon$ is a rule in R, then $A \to \varepsilon$ is a parse tree. Its root is the node labeled A and its leaf is the node labeled ε. The yield of the parse tree is ε.
- If $A_1 \to (T_1)y_1, ..., A_n \to (T_n)y_n$ are parse trees, where $n \geq 1$, with roots labeled $A_1, ..., A_n$, and with yields $y_1, ..., y_n$, and $A \to A_1...A_n$ is a rule in R, then $A \to A_1, ..., A_n$ is a parse tree. It's root is the new node A, it's leaves are the leaves of subtrees, and its yield is $y_1...y_n$.
- Nothing else is a parse tree.

Example 1.3. Consider the two derivations of the string $()(())$ from the grammar for the language of balanced parentheses:

- $S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow ()(())$
- $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()(())$

```
      S
    /   \
   S     S
  /|\   /|\
 ( S ) ( S )
   |    /|\
```



```
   e   ( S )
         |
         e
```

1.2 Equivalence classes of derivations

Intuitively, parse trees are ways of representing derivations of strings in $\mathcal{L}(G)$ so that the superficial differences between derivations due to a different ordering of the application of rules are suppressed.

In fact, parse trees represent *equivalence classes* of derivations.

More formally, let $G = (V, \Sigma, R, S)$ be a context-free grammar, and let $D = x_1 \Rightarrow x_2 \Rightarrow ... \Rightarrow x_n$ and $D' = x'_1 \Rightarrow x'_2 \Rightarrow ... \Rightarrow x'_n$ be two derivations in G, where

- $x_i, x'_i \in V^*$ for $i = 1, ..., n$,
- $x_1, x'_1 \in V - \Sigma$, and
- $x_n, x'_n \in \Sigma^*$.

Note. This means that both are derivations of terminal strings from a single non-terminal.

We say that $D precedes D'$, written $D \prec D'$, if $n > 2$ and there is an integer k, $1 < k < n$ such that:

- For all $i \neq k$ we have $x_i = x'_i$;
- $x_{k-1} = x'_{k-1} = uAvBw$ where $u, v, w \in V^*$, and $A, B \in V - \Sigma$;
- $x_k = uyvBw$ where $A \to y \in R$;
- $x'_k = uAvzw$ where $B \to z \in R$;
- $x_{k+1} = x'_{k+1} = uyvzw$.

This means that the two derivations are identical except for two consecutive steps during which the same two non-terminals are replaced with the same two strings, but in *opposite orders* in the two derivations.

The derivation in which the *leftmost* of the two non-terminals is replaced first is said to *precede the other*.

Example 1.4. Consider the following three derivations D_1, D_2, and D_3 in the grammar for the language of balanced parentheses:

$D_1 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

$D_2 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

$D_3 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$

We have that $D_1 \prec D_2$ and $D_2 \prec D_3$.

It is not the case, however, that $D_1 \prec D_3$ since D_1 and D_3 differ in more than one intermediate string.

Note that all three have the *same parse tree*, given below:

```
      S
    /   \
   S     S
  /|\   /|\
 ( S ) ( S )
  /|\    |
 ( S )   e
   |
   e
```

Definition 1.5. We say that two derivations D and D' are *similar* if the pair (D, D') belongs in the reflexive, symmetric, transitive closure of $\prec$.

Note. By definition, similarity is an *equivalence relation*.

Informally, two derivations are similar if they can be transformed one into the other via *a sequence of "switchings"* in the order in which the rules are applied.

A "switching" replaces a derivation either by one that precedes it or by one that it precedes.

Example 1.6. The derivations D_1, D_2 and D_3 are all similar since $D_1 \prec D_2$ and $D_2 \prec D_3$ so that D_1 is similar to D_3.

There are also many other similar derivations.

All of these derivations have the same parse tree given above.

The derivations are similar because, informally, they represent applications of the same rules at the same positions in the strings, only differing in the relative order of the applications.

1.2.1 An inequivalent derivation

There are, however, other derivations of the string $(())()$ that are *not similar* to the ones mentioned above.

Example 1.7. $S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(())S \Rightarrow S(())(S) \Rightarrow S(())() \Rightarrow (())()$

This derivation has the following parse tree:

```
    S
  /   \
 e     S
     /   \
    S     S
   /|\   /|\
  ( S ) ( S )
   /|\    |
  ( S )   e
    |
    e
```

1.2.2 Leftmost and rightmost derivations

Each equivalence class of derivations under similarity, that is, each parse tree, contains a derivation that is *maximal under* $\prec$.

This derivation cannot, by definition, be preceded by any other derivation in the tree. It is called the *leftmost derivation*.

A leftmost derivation exists in every tree, and can be found as follows:

- Start at the root of the tree.
- Replace the leftmost non-terminal in the string according to the rule suggested by the parse tree.
- Repeat until the string consists of only terminals.

Example 1.8. Consider a grammar for the language of balanced parentheses.

$G = (V, \Sigma, R, S)$ where $V = \{S, (,)\}$, $\Sigma = \{(,)\}$, $R = \{S \to \varepsilon, S \to SS, S \to (S)\}$.

Consider the following two derivations D_1 and D_2 in this grammar:

$D_1 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

$D_2 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

Note that both have the *same parse tree*, given below:

```
      S
    /   \
   S     S
  /|\   /|\
 ( S ) ( S )
  /|\    |
 ( S )   e
   |
   e
```

The leftmost derivation is $D_1 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

If we look at the chart relating the ten derivations in the equivalence class for this tree we can see that no derivation precedes D_1.

Similarly, a *rightmost derivation* is one that does not precede any other derivation.

We obtain a rightmost derivation by replacing the rightmost non-terminal at each step.

Example 1.9. Again consider the parse tree given in Example 1.

The leftmost derivation for this tree is $D_{10} = S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$

Fact 1.10. *Each parse tree has exactly one leftmost and one rightmost derivation.*

Reason 1.11. There is only one choice at each step for which non-terminal to expand, namely either the leftmost or the rightmost.

1.3 Notation

We write $x \overset{L}{\Rightarrow} y$ if and only if $x = wA\beta$, $y = w\alpha\beta$, where $w \in \Sigma^*$, $\alpha, \beta \in V^*$, $A \in V - \Sigma$, and $A \to \alpha$ is a rule of G.

If $x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow ... \Rightarrow x_n$ is a leftmost derivation, then $x_1 \overset{L}{\Rightarrow} x_2 \overset{L}{\Rightarrow} ... \overset{L}{\Rightarrow} x_n$.

We can define $x \overset{R}{\Rightarrow} y$ similarly for rightmost derivations.

1.4 Summary of results

We can summarize the results for parse trees as follows:

Theorem 1.12. *Let $G = (V, \Sigma, R, S)$ be a context-free grammar, and let $A \in V - \Sigma$, and $w \in \Sigma^*$. Then the following statements are equivalent:*

- $A \Rightarrow^* w$
- *There is a parse tree with root A and yield w.*
- *There is a leftmost derivation $A \overset{L^*}{\Rightarrow} w$.*
- *There is a rightmost derivation $A \overset{R^*}{\Rightarrow} w$.*

1.5 Ambiguity

Definition 1.13. A grammar G that has at least one string with two or more distinct parse trees is called *ambiguous*.

Example 1.14. The grammar G presented before for the language of balanced parentheses is ambiguous.

The string $(())()$ has two distinct parse trees.

Why does this matter?

The process of assigning a parse tree to a string, also known *as parsing the string*, is an important first step toward understanding the structure of a string since it enables us to see why it is that the string is in the language.

This is vitally important when dealing with a programming language.

Ambiguous grammars are of no help in parsing since they do not assign a unique parse tree, that is, a unique meaning, to each string in the language.

It is possible to produce a grammar for the language of balanced parentheses that is unambiguous.

Unfortunately, there are context-free languages with the property that all context-free grammars that generate them are ambiguous.

Such languages are called *inherently ambiguous*.

Example 1.15. The language $L = \{ a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \} \cup \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \}$ is inherently ambiguous.

This is proved by showing that infinitely many strings of the form $a^n b^n c^n d^n$ for $n \geq 1$ must have two distinct leftmost derivations.

The proof is long and tedious, so it is omitted.

Fortunately, programming languages are never inherently ambiguous.

2 PDA review

2.1 Examples

Example 2.1. $L = \{ wcw^R \mid w \in \{a, b\}^* \}$

$w_1 = ababab cbababa \in L$

$w_2 = abcab \notin L$

$w_3 = abba \notin L$

How will the automaton work?

- It will have two states that correspond to "have not seen the c" and "have seen the c". The former will be the starting state, and the latter will be the final state.
- When in state "have not seen the c", it will push the symbols that it reads onto the stack.
- When it encounters the c it switches states without changing the stack.
- In the state "have seen the c", it compares the current input symbol to the symbol on the top of the stack and advances past both if they match.
- Only valid strings, that is, ones that have matching a's and b's and contain a c in the middle will cause acceptance. Any other string will reach a situation where there is no transition to take.

Let $M = (K, \Sigma, \Gamma, \Delta, s, F)$ where $K = \{s, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, and Δ contains the following five transitions:

- $((s, a, \varepsilon), (s, a))$
- $((s, b, \varepsilon), (s, b))$
- $((s, c, \varepsilon), (f, \varepsilon))$
- $((f, a, a), (f, \varepsilon))$

- $((f, b, b), (f, \varepsilon))$

Sample accepting computation: $w = abacaba$

State	Unread input	Stack	Transition
s	$abacaba$	ε	—
s	$bacaba$	a	1
s	$acaba$	ba	2
s	$caba$	aba	1
f	aba	aba	3
f	ba	ba	4
f	a	a	5
f	ε	ε	4

Sample rejecting computation: $w = aaaa$

State	Unread input	Stack	Transition
s	$aaaa$	ε	—
s	aaa	a	1
s	aa	aa	1
s	a	aaa	1
s	ε	$aaaa$	1

Example 2.2. Now consider the language $L = \{ ww^R \mid w \in \{a, b\}^* \}$.

This example differs Example 1 in that there is no center marker c to tell us when to switch from the state that pushes input onto the stack into the state that reads input while popping characters off the stack.

We will have to use nondeterminism to "guess" when to make the switch.

Let $M = (K, \Sigma, \Gamma, \Delta, s, F)$ where $K = \{s, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, and Δ contains the following five transitions:

- $((s, a, \varepsilon), (s, a))$
- $((s, b, \varepsilon), (s, b))$

- $((s, \varepsilon, \varepsilon), (f, \varepsilon))$
- $((f, a, a), (f, \varepsilon))$
- $((f, b, b), (f, \varepsilon))$

Sample accepting computation: $w = abba$

State	Unread input	Stack	Transition
s	$abba$	ε	—
s	bba	a	1
s	ba	ba	2
f	ba	ba	3
f	a	a	5
f	ε	ε	4

If there is no way to "guess" correctly, then the string will not be accepted, for example with $w = babaa$

Example 2.3. Let $L = \{ w \in \{a, b\}^* \mid w$ has an equal number of a's and b's $\}$.

How will the automaton work?

- The automaton will keep either a string of a's or a string of b's on its stack.
- A string of a's indicates that M has seen more a's than b's at the current point in time. The size of the stack is the amount of the excess.
- A string of b's represents an excess of b's.
- In either case, a marker c is used to indicate that the bottom of the stack has been reached.

Let $M = (K, \Sigma, \Gamma, \Delta, s, F)$ where $K = \{s, q, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, c\}$, $F = \{f\}$, and Δ is given below:

- $((s, \varepsilon, \varepsilon), (q, c))$

- $((q, a, c), (q, ac))$
- $((q, a, a), (q, aa))$
- $((q, a, b), (q, \varepsilon))$
- $((q, b, c), (q, bc))$
- $((q, b, b), (q, bb))$
- $((q, b, a), (q, \varepsilon))$
- $((q, \varepsilon, c), (f, \varepsilon))$

The purpose of the transitions is the following:

- Transition 1 initializes the computation. It puts M into state q while placing a c on the bottom of the stack.
- In state q reading a, M either starts up a stack of a's from the bottom using Transition 2, adds an a to an existing stack of a's using Transition 3, or pops a b off the stack using Transition 4.
- In state q reading b, $M3$ either starts up a stack of b's from the bottom using Transition 5, adds to an existing stack of b's using Transition 6, or pops an a off the stack using Transition 7.
- When c is the topmost character on the stack and there are no characters left to read, then we can remove the c using Transition 8 and accept the string since there are no outstanding a's or b's.

Sample accepting computation: $w = aabaaabbbb$

State	Unread input	Stack	Transition	Comment
s	$aabaaabbbb$	ε	—	Initial configuration
q	$aabaaabbbb$	c	1	Bottom marker
q	$abaaabbbb$	ac	2	Start stack of a's
q	$baaabbbb$	aac	3	Continue stack of a's
q	$aaabbbb$	ac	7	Pop off an a
q	$aabbbb$	aac	3	Add another a
q	$abbbb$	$aaac$	3	Add another a
q	$bbbb$	$aaaac$	3	Add another a
q	bbb	$aaac$	7	Pop off an a
q	bb	aac	7	Pop off an a
q	b	ac	7	Pop off an a
q	ε	c	7	Pop off an a
f	ε	ε	8	Accept the string

3 Determinism

The definition we gave for a pushdown automaton was non-deterministic.

Question 3.1.
Can we always find an equivalent deterministic pushdown automaton for a given context-free language?

Answer: Unfortunately not.

There are some context-free languages that cannot be accepted by deterministic pushdown automata.

This is a dire result, especially if we actually want to produce a parser for the context-free language.

Some good news: For most programming languages one can construct deterministic pushdown automata that accept all syntactically correct programs.

3.1 Deterministic context-free languages

We will first introduce the definitions for deterministic pushdown automata and then talk about the negative results we mentioned above.

3.1.1 Definitions

A pushdown automaton is *deterministic* if for each configuration there is at most one configuration that can succeed it in a computation by M.

Note. In the book they make a point to exclude transitions of the form $((s, \varepsilon, \varepsilon), (q, \varepsilon))$ from their definition. Those transitions are implicitly excluded by our definition.

Example 3.2. The pushdown automaton (given below) for the language $\{\, wcw^R \mid w \in \{a, b\}^* \,\}$ is deterministic. For each choice of state and each input symbol, there is only one possible transition.

Let $M = (K, \Sigma, \Gamma, \Delta, s, F\}$ where $K = \{s, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, and Δ contains the following five transitions:

- $((s, a, \varepsilon), (s, a))$
- $((s, b, \varepsilon), (s, b))$
- $((s, c, \varepsilon), (f, \varepsilon))$
- $((f, a, a), (f, \varepsilon))$
- $((f, b, b), (f, \varepsilon))$

Example 3.3. On the other hand, the pushdown automaton for the language $\{\, ww^R \mid w \in \{a, b\}^* \,\}$ was non-deterministic. Either Transition 1 or Transition 2 may be followed by Transition 3. These are the transitions that "guess" the middle of the string, an action that is intuitively non-deterministic.

Let $M = (K, \Sigma, \Gamma, \Delta, s, F\}$ where $K = \{s, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, and Δ contains the following five transitions:

- $((s, a, \varepsilon), (s, a))$

- $((s, b, \varepsilon), (s, b))$
- $((s, \varepsilon, \varepsilon), (f, \varepsilon))$
- $((f, a, a), (f, \varepsilon))$
- $((f, b, b), (f, \varepsilon))$

Deterministic context-free languages are essentially those that are accepted by a deterministic pushdown automaton. However, we need to change the *acceptance condition* slightly so that we don't exclude languages that are intuitively deterministic.

$L \subseteq \Sigma^*$ is a deterministic context-free language if $L\$ = \mathcal{L}(M)$ for some deterministic pushdown automaton.

\$ is some new symbol, not in Σ, which is appended to each input string for the purpose of marking the end.

So a deterministic pushdown automaton has the capability of *sensing the end of the input string*.

Why do we need this additional assumption?

Consider $L = a^* \cup \{\, a^n b^n \mid n \geq 1 \,\}$.

A deterministic machine cannot simultaneously:

- Keep track of how many a's it has seen in order to compare it against any b's it may find.
- Be ready to accept with an empty stack in case no b's do follow.

But $L\$$ is easy to accept deterministically: If \$ is found while still pushing a's, then the string consists of all a's and the automaton can empty its stack and accept.

This additional assumption does not hurt us.

Claim 3.4. *Every deterministic context-free language, as just defined, is a context-free language.*

Reason 3.5. Suppose that a deterministic pushdown automaton M accepts $L\$$. Then a (non-deterministic) pushdown automaton M' can be constructed to accept L.

M' "imagines" a \$ in the input and jumps to a new set of states from which it needs no further input.

3.1.2 (Negative) results

Consider the language $L = \{ a^n b^m c^p \mid m, n, p \geq 0, \text{ and } m \neq n \text{ or } m \neq p \}$.

It would seem that a pushdown automaton could accept this language only by guessing which of the two blocks to compare: either the a's and the b's or the b's and the c's. However, proving that L is not deterministic requires a more indirect approach.

Theorem 3.6. *The class of deterministic context-free languages is closed under complementation.*

We now sketch the proof.

Consider a language $L\$$ accepted by a deterministic pushdown automaton M.

We can assume that M is simple and accepts by empty stack.

Reasons:

- To transform it to one that accepts by empty stack, just do as in the proof of Theorem 3.4.2 and put a bottom of stack marker on the stack as the first move and remove it at the end of every computation.
- Transform the automaton into a simple one using the procedure we described before. It will preserve determinism.

Not-quite-correct idea: Reverse the conditions for acceptance, that is, accept when the stack is not empty and the automaton is in a non-final state.

Sticking point: M may reject because it never finished reading the input.

This can happen in the following two circumstances:

- M reaches a configuration that has no following configuration;
- M enters a configuration from which it can apply an infinite sequence of configurations that do not consume any input.

A configuration C that meets either of these two criteria is called a *dead end*. In such configurations, a deterministic pushdown automaton M can neither complete reading the input nor reduce the length of data in the stack.

More formally, in a simple pushdown automaton M, a triple (s, a, A) is a *dead end* if, from any configuration C, M must apply a transition with left-hand triple (s, a, A) and never reaches either configuration (q, ε, α) [i.e. the end of the input] or a configuration (q, a, ε) [i.e. an empty stack with input remaining to be read].

Construction 3.7. Given M we will produce an automaton M' that accepts all the strings not accepted by M, including those that drive M into a dead state.

The first task is to create M' such that whenever M enters a dead end, M' completes reading the input and empties the stack, thus accepting the string.

Let $\mathbb{D}$ be the set of dead end triples in M. (Note that this proof is not constructive).

Suppose $(s, a, A) \in \mathbb{D}$. There are several steps to the dead-state transformation:

- Remove all transitions that are compatible with (s, a, A).

  Two transitions $((s, a, \alpha), (q, \beta))$ and $((t, b, \sigma), (r, \gamma))$ are *compatible* if $s = t$, $a = b$ or $a = \varepsilon$ or $b = \varepsilon$, and either α is a prefix of σ or σ is a prefix of α.

  By removing all compatible transitions we are ensuring that the automaton M' will not get stuck or enter a loop once it reaches (s, a, A).
- Add the transition $((s, a, A), (q, \varepsilon))$ that reads a and pops A from the stack where q is a new state.
- Add the transitions $((q, b, \varepsilon), (q, \varepsilon))$ for all $b \in \Sigma$.

- Add the transition $((q, \}, \varepsilon), (p, \varepsilon))$ where p is a new state.

  These transitions allow M' to read the remainder the input once it reaches (s, a, A).
- Add the transitions $((p, \varepsilon, B), (p, \varepsilon))$ for all $B \in \Gamma$

  These transitions will allow M' to remove everything from the stack once it reaches state p.

To complete the construction we must:

- Make sure that when M completes reading the input and empties its stack, M' does not clean the stack.
- Make sure that if M completes reading the input but does not accept, M' completes reading the input and empties the stack.

The details of the remainder of the construction are omitted.

How does Theorem 3.7.1 show that L is not deterministic?

Suppose that L is deterministic. Then $\overline{L}$ is deterministic context-free, and thus, context-free.

So $\overline{L} \cap a^*b^*c^*$ would be context-free by Theorem 3.5.2.

But $\overline{L} \cap a^*b^*c^* = \{ a^n b^n c^n \mid n \geq 0 \}$, a language that is not context-free.

Thus, L cannot be deterministic.

Corollary 3.8. *The class of deterministic context-free languages is properly contained in the class of context-free languages.*

End result: For pushdown automata, non-determinism is more powerful than determinism.