MAU22C00 Lecture 30

John Stalker

Trinity College Dublin

Announcements

- Your last assignment of the semester is due on Friday.

Announcements

- Your last assignment of the semester is due on Friday.
- I've posted the last chapter of the lecture notes, on context free languages and pushdown automata. We'll cover that material on Thursday and Friday.

Myhill-Nerode

Last time we met the notions of valid continuations of a list and equivalence of lists. The Myhill-Nerode theorem says that a language is regular if and only if the set of equivalence classes is finite.

Myhill-Nerode

Last time we met the notions of valid continuations of a list and equivalence of lists. The Myhill-Nerode theorem says that a language is regular if and only if the set of equivalence classes is finite.

The proof is somewhat constructive. In the case where the set of equivalence classes is finite it shows how to construct a strongly deterministic finite state automaton with one state for each equivalence class which recognises the language.

Myhill-Nerode

Last time we met the notions of valid continuations of a list and equivalence of lists. The Myhill-Nerode theorem says that a language is regular if and only if the set of equivalence classes is finite.

The proof is somewhat constructive. In the case where the set of equivalence classes is finite it shows how to construct a strongly deterministic finite state automaton with one state for each equivalence class which recognises the language.

It also shows that this is the smallest possible number of states for such an automaton.

Myhill-Nerode

Last time we met the notions of valid continuations of a list and equivalence of lists. The Myhill-Nerode theorem says that a language is regular if and only if the set of equivalence classes is finite.

The proof is somewhat constructive. In the case where the set of equivalence classes is finite it shows how to construct a strongly deterministic finite state automaton with one state for each equivalence class which recognises the language.

It also shows that this is the smallest possible number of states for such an automaton.

You don't need to understand the proof in detail but it's very useful to understand the construction of the automaton. For those who find it easier to understand examples than proofs I'll give two examples.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

The valid continuations of the empty string are described by the regular expression `HLL*|LHH*L*`.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

The valid continuations of the empty string are described by the regular expression `HLL*|LHH*L*`.

The valid continuations of `H` are described by `LL*`.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

The valid continuations of the empty string are described by the regular expression `HLL*|LHH*L*`.

The valid continuations of `H` are described by `LL*`.

The valid continuations of `L` are described by `HH*L*`.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

The valid continuations of the empty string are described by the regular expression `HLL*|LHH*L*`.

The valid continuations of `H` are described by `LL*`.

The valid continuations of `L` are described by `HH*L*`.

There are no valid continuations of `HH` or `LL`.

Example (from a regular expression to an FSA)

As a first example, we can find a finite state automaton for the regular expression `HLL*|LHH*L*`.

We know there must be one. In theory we even know how to find one, but inefficiently. Let's find an optimal one.

The valid continuations of the empty string are described by the regular expression `HLL*|LHH*L*`.

The valid continuations of `H` are described by `LL*`.

The valid continuations of `L` are described by `HH*L*`.

There are no valid continuations of `HH` or `LL`.

The valid continuations of `HL` are described by `L*`. The valid continuations of `LH` are described by `H*L*`.

Example, continued

If we read a string whose valid continuations are `L`$*$ and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L`$*$.

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.
- `LL*`

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.
- `LL*`
- `HH*L*`

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.
- `LL*`
- `HH*L*`
- None

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.
- `LL*`
- `HH*L*`
- None
- `L*`. This includes the empty string.

Example, continued

If we read a string whose valid continuations are `L*` and then read an `H` the there are no valid continuations. If instead we read an `L` then the valid continuations are again `L*`.

If we read a string whose valid continuations are `H*L*` and then an `H` the valid continuations are again `H*L*`. If we read an `L` then they are `L*`.

We've now seen all sets of valid continuations:

- The original language `HLL*|LHH*L*`. This is the set of continuations of the empty string.
- `LL*`
- `HH*L*`
- None
- `L*`. This includes the empty string.
- `H*L*`. This includes the empty string.

A finite state automaton

This is a finite state automaton based on the equivalence classes.

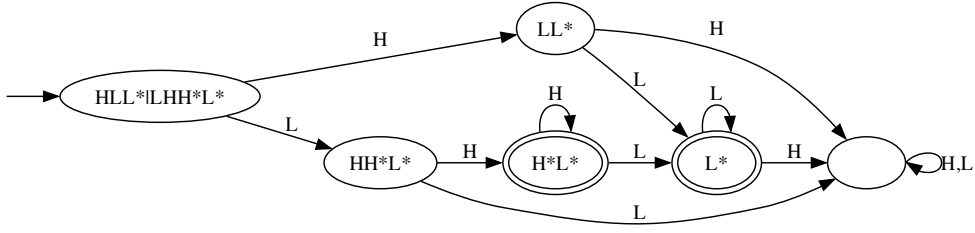

Figure 1: A finite state automaton for the regular expression

The start state is the one for the equivalence class of the empty string. The accepting state is the one for classes where the empty string is a valid continuation. The arrows indicate which equivalence class you get to by appending a character to a string in another equivalence class.

A finite state automaton

This is a finite state automaton based on the equivalence classes.

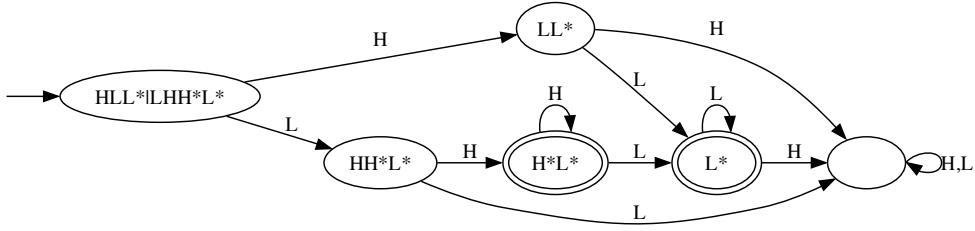

Figure 1: A finite state automaton for the regular expression

The start state is the one for the equivalence class of the empty string. The accepting state is the one for classes where the empty string is a valid continuation. The arrows indicate which equivalence class you get to by appending a character to a string in another equivalence class.

You have seen this finite state automaton before.

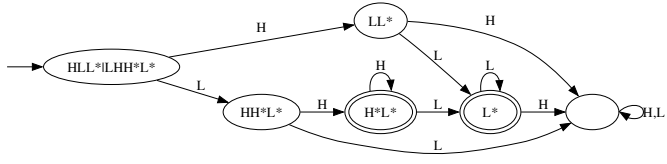

Figure 2: The same finite state automaton for the regular expression

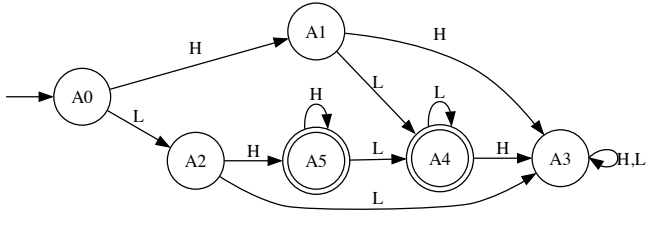

Figure 3: A finite state machine for the Tokyo language

Another example (integers)

As our next example, consider the language of (normalised) integers.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The only valid continuation of `0` is the empty string.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The only valid continuation of `0` is the empty string.

The valid continuations of `-` are described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The only valid continuation of `0` is the empty string.

The valid continuations of `-` are described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The valid continuations of `1`, `2`, ..., `9` are described by `(0|1|2|3|4|5|6|7|8|9)*`.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The only valid continuation of `0` is the empty string.

The valid continuations of `-` are described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The valid continuations of `1`, `2`, …, `9` are described by `(0|1|2|3|4|5|6|7|8|9)*`.

Those are also the valid continuations of `-1`, `-2`, …, `-9`, or really of any non-zero integer.

Another example (integers)

As our next example, consider the language of (normalised) integers.

We found a regular expression for the integers, namely `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Let's turn this into a finite state automaton.

The valid continuations of the empty string are described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The only valid continuation of `0` is the empty string.

The valid continuations of `-` are described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

The valid continuations of `1`, `2`, …, `9` are described by `(0|1|2|3|4|5|6|7|8|9)*`.

Those are also the valid continuations of `-1`, `-2`, …, `-9`, or really of any non-zero integer.

Finally, there are the strings with no valid continuation.

Example, continued

Here are the states we need, their corresponding continuation sets, and whether they're starting or accepting states.

- A0: the set described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. This is the set of continuations of the empty set and so is the start state.

Example, continued

Here are the states we need, their corresponding continuation sets, and whether they're starting or accepting states.

- A0: the set described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. This is the set of continuations of the empty set and so is the start state.
- A1: the empty string, and only the empty string. Since this class contains the empty string it is an accepting state.

Example, continued

Here are the states we need, their corresponding continuation sets, and whether they're starting or accepting states.

- A0: the set described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. This is the set of continuations of the empty set and so is the start state.
- A1: the empty string, and only the empty string. Since this class contains the empty string it is an accepting state.
- A2: the set described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.

Example, continued

Here are the states we need, their corresponding continuation sets, and whether they're starting or accepting states.

- A0: the set described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. This is the set of continuations of the empty set and so is the start state.
- A1: the empty string, and only the empty string. Since this class contains the empty string it is an accepting state.
- A2: the set described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.
- A3: the set described by `(0|1|2|3|4|5|6|7|8|9)*`. This class contains the empty string and so is an accepting state.

Example, continued

Here are the states we need, their corresponding continuation sets, and whether they're starting or accepting states.

- A0: the set described by `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. This is the set of continuations of the empty set and so is the start state.
- A1: the empty string, and only the empty string. Since this class contains the empty string it is an accepting state.
- A2: the set described by `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`.
- A3: the set described by `(0|1|2|3|4|5|6|7|8|9)*`. This class contains the empty string and so is an accepting state.
- A4: the empty set.

A finite state automaton

The diagram for the finite state automaton we've just found is

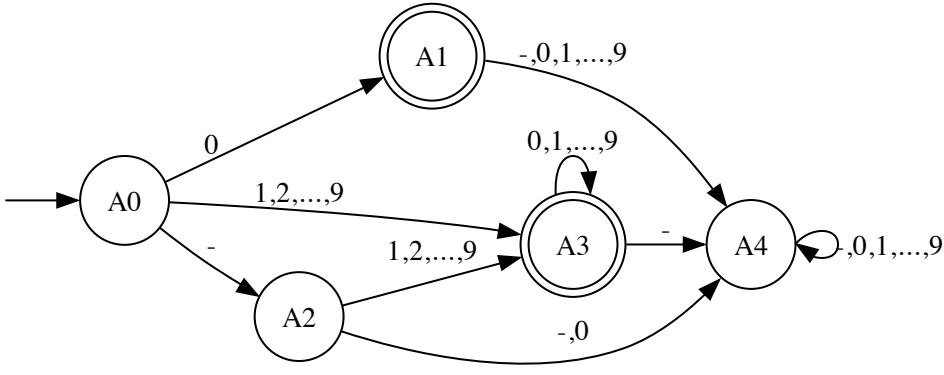

Figure 4: A finite state automaton for the integers

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

If w is not in E then w has at least as many ('s as)'s. Let S_n be the set of w not in E for which there are n more ('s than)'s.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

If w is not in E then w has at least as many ('s as)'s. Let S_n be the set of w not in E for which there are n more ('s than)'s.

Every string is in exactly one of $E, S_0, S_1, \ldots$.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

If w is not in E then w has at least as many ('s as)'s. Let S_n be the set of w not in E for which there are n more ('s than)'s.

Every string is in exactly one of $E, S_0, S_1, \ldots$.

Within any of those sets, each string has the same set of valid continuations.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

If w is not in E then w has at least as many ('s as)'s. Let S_n be the set of w not in E for which there are n more ('s than)'s.

Every string is in exactly one of $E, S_0, S_1, \ldots$.

Within any of those sets, each string has the same set of valid continuations.

So these are the equivalence classes. There are infinitely many of them, so the language is not regular.

Example (language of balanced parentheses)

How about the language of balanced parentheses?

Recall, to figure out whether a string belongs to this language we track how many more ('s we've seen than)'s. If this number never goes negative and is zero at the end of the input the string is in the language.

Let E be the set of strings where the difference goes negative at some point. These have no valid continuation.

If w is not in E then w has at least as many ('s as)'s. Let S_n be the set of w not in E for which there are n more ('s than)'s.

Every string is in exactly one of $E, S_0, S_1, \ldots$.

Within any of those sets, each string has the same set of valid continuations.

So these are the equivalence classes. There are infinitely many of them, so the language is not regular.

The nice thing about Myhill-Nerode is that you don't need to know whether your language is regular in advance.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

We'll say that two lists are bidirectionally equivalent if their sets of such continuations are equal. This is an equivalence relation on the set of lists.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

We'll say that two lists are bidirectionally equivalent if their sets of such continuations are equal. This is an equivalence relation on the set of lists.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. Suppose also that (v, x) is a continuation of qs, i.e. that $vqsx$ is in L.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

We'll say that two lists are bidirectionally equivalent if their sets of such continuations are equal. This is an equivalence relation on the set of lists.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. Suppose also that (v, x) is a continuation of qs, i.e. that $vqsx$ is in L.

Then (v, sx) is a valid continuation of q. q is equivalent to r, so (v, sx) is also a valid continuation of r. In other words $vrsx$ is in L.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

We'll say that two lists are bidirectionally equivalent if their sets of such continuations are equal. This is an equivalence relation on the set of lists.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. Suppose also that (v, x) is a continuation of qs, i.e. that $vqsx$ is in L.

Then (v, sx) is a valid continuation of q. q is equivalent to r, so (v, sx) is also a valid continuation of r. In other words $vrsx$ is in L.

Then (vr, x) is a valid continuation of s. t is equivalent to s, so (vr, x) is also a valid continuation of t. In other words $vrtx$ is in L.

Bidirectional equivalence

Instead of considering one-sided continuations we can consider two sided ones. Given a language L and a list w we can consider the set of pairs of lists (v, x) such that vwx is in L.

We'll say that two lists are bidirectionally equivalent if their sets of such continuations are equal. This is an equivalence relation on the set of lists.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. Suppose also that (v, x) is a continuation of qs, i.e. that $vqsx$ is in L.

Then (v, sx) is a valid continuation of q. q is equivalent to r, so (v, sx) is also a valid continuation of r. In other words $vrsx$ is in L.

Then (vr, x) is a valid continuation of s. t is equivalent to s, so (vr, x) is also a valid continuation of t. In other words $vrtx$ is in L.

This means that (v, x) is a valid continuation of rt.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

The same argument, but swapping q and r and s and t, shows that if (v, x) is a continuation of rt then it's a continuation of qs.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

The same argument, but swapping q and r and s and t, shows that if (v, x) is a continuation of rt then it's a continuation of qs.

In other words, qs and rt have the same set of continuations, so they are bidirectionally equivalent.

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

The same argument, but swapping q and r and s and t, shows that if (v, x) is a continuation of rt then it's a continuation of qs.

In other words, qs and rt have the same set of continuations, so they are bidirectionally equivalent.

If q and r are bidirectionally equivalent and s and t are bidirectionally equivalent then qs and rt are bidirectionally equivalent.

The syntactic monoid

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

The same argument, but swapping q and r and s and t, shows that if (v, x) is a continuation of rt then it's a continuation of qs.

In other words, qs and rt have the same set of continuations, so they are bidirectionally equivalent.

If q and r are bidirectionally equivalent and s and t are bidirectionally equivalent then qs and rt are bidirectionally equivalent.

So bidirectional equivalence is compatible with the monoid structure on lists given by concatenation.

The syntactic monoid

Suppose q and r are bidirectionally equivalent and s and t are bidirectionally equivalent. We've just shown that if (v, x) is a continuation of qs then it's a continuation of rt.

The same argument, but swapping q and r and s and t, shows that if (v, x) is a continuation of rt then it's a continuation of qs.

In other words, qs and rt have the same set of continuations, so they are bidirectionally equivalent.

If q and r are bidirectionally equivalent and s and t are bidirectionally equivalent then qs and rt are bidirectionally equivalent.

So bidirectional equivalence is compatible with the monoid structure on lists given by concatenation.

We can therefore apply the quotient construction to get a monoid. This monoid is called the syntactic monoid.

Uses of the syntactic monoid

There's a version of the Myhill-Nerode Theorem which says that the language is regular if and only if its set of bidirectional equivalence classes is finite, i.e. if and only if its syntactic monoid is finite.

Uses of the syntactic monoid

There's a version of the Myhill-Nerode Theorem which says that the language is regular if and only if its set of bidirectional equivalence classes is finite, i.e. if and only if its syntactic monoid is finite.

As with the earlier version, this one effectively constructs a finite state automaton in the regular case. This automaton isn't minimal, but has some nice properties.

Uses of the syntactic monoid

There's a version of the Myhill-Nerode Theorem which says that the language is regular if and only if its set of bidirectional equivalence classes is finite, i.e. if and only if its syntactic monoid is finite.

As with the earlier version, this one effectively constructs a finite state automaton in the regular case. This automaton isn't minimal, but has some nice properties.

Defining properties of a language in terms of its syntactic monoid rather than its grammar, finite state automaton or regular expression means you don't have to show that your definition is independent of which grammar, finite state automaton or regular expression is chosen.