MAU22C00 Lecture 26

John Stalker

Trinity College Dublin

Can we construct a right regular grammar which generates every string which is generated by the tokyo (right regular) grammar or the kansai (right regular) grammar?

Can we construct a right regular grammar which generates every string which is generated by the tokyo (right regular) grammar or the kansai (right regular) grammar?

```
Yes, it's very easy!
```

```
%%
tokyo_or_kansai : atamadaka | nakadaka | odaka | heiban
| heishinshiki | teikishiki ;
```

and then the rules from each grammar, other than the rules for the tokyo or kansai symbols.

Can we construct a right regular grammar which generates every string which is generated by the tokyo (right regular) grammar or the kansai (right regular) grammar?

```
Yes, it's very easy!
```

and then the rules from each grammar, other than the rules for the tokyo or kansai symbols.

I did something slightly dangerous here, but it's okay. Both grammars have an error symbol. The safest procedure is to rename symbols to avoid name clashes.

Can we construct a right regular grammar which generates every string which is generated by the tokyo (right regular) grammar or the kansai (right regular) grammar?

```
Yes, it's very easy!
```

and then the rules from each grammar, other than the rules for the tokyo or kansai symbols.

I did something slightly dangerous here, but it's okay. Both grammars have an error symbol. The safest procedure is to rename symbols to avoid name clashes.

Here it's okay because the rules for those symbols are the same in each grammar.

Can we construct a right regular grammar which generates every string which is generated by the tokyo (right regular) grammar or the kansai (right regular) grammar?

```
Yes, it's very easy!
```

and then the rules from each grammar, other than the rules for the tokyo or kansai symbols.

I did something slightly dangerous here, but it's okay. Both grammars have an error symbol. The safest procedure is to rename symbols to avoid name clashes.

Here it's okay because the rules for those symbols are the same in each grammar.

If you're automating this you'd just rename every symbol automatically, e.g. with a prefix to identify which grammar it came from.

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which recognises those strings which are the concatenation of a string from one language with one from the other?

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which recognises those strings which are the concatenation of a string from one language with one from the other?

Yes. It's trickier than the union of two languages though. See the notes for details.

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which recognises those strings which are the concatenation of a string from one language with one from the other?

Yes. It's trickier than the union of two languages though. See the notes for details.

Can we construct a grammar for the language whose strings are the concatenations of zero or more strings from a language?

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which recognises those strings which are the concatenation of a string from one language with one from the other?

Yes. It's trickier than the union of two languages though. See the notes for details.

Can we construct a grammar for the language whose strings are the concatenations of zero or more strings from a language?

Yes. See the notes for details.

Can we construct a grammar which generates those strings which appear in both grammars? Or in one but not the other?

Yes, but this is very non-obvious. To understand this we'll need a different point of view.

Can we construct a grammar which recognises those strings which are the concatenation of a string from one language with one from the other?

Yes. It's trickier than the union of two languages though. See the notes for details.

Can we construct a grammar for the language whose strings are the concatenations of zero or more strings from a language?

Yes. See the notes for details.

In general, given a language the language of concentations of members of it is called its Kleene star.

Reversal

What about generating the strings in a reversed language?

Reversal

What about generating the strings in a reversed language?

It's easy to get a right regular grammar from a reversed language from a left regular grammar for the original language and vice versa. It's hard to get a right regular grammar from a reversed language from a right regular grammar for the original language or left from left.

Reversal

What about generating the strings in a reversed language?

It's easy to get a right regular grammar from a reversed language from a left regular grammar for the original language and vice versa. It's hard to get a right regular grammar from a reversed language from a right regular grammar for the original language or left from left.

Like intersection or relative complement, this requires a new point of view.

We discussed finite state automata informally a long time ago. Formally a finite state automaton is defined by the following:

• A set A of tokens,

- A set A of tokens,
- A set S of states,

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

We discussed finite state automata informally a long time ago. Formally a finite state automaton is defined by the following:

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

The automaton must start in one of the states in *I*.

We discussed finite state automata informally a long time ago. Formally a finite state automaton is defined by the following:

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

The automaton must start in one of the states in I.

 $(r, a, s) \in T$ if the automaton can jump to the state s on reading an a while in state r.

We discussed finite state automata informally a long time ago. Formally a finite state automaton is defined by the following:

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

The automaton must start in one of the states in I.

 $(r, a, s) \in T$ if the automaton can jump to the state s on reading an a while in state r.

There might be more than one member of I and for a given r and a there might be more than one s with $(r, a, s) \in T$, so this automaton may be non-deterministic.

We discussed finite state automata informally a long time ago. Formally a finite state automaton is defined by the following:

- A set A of tokens,
- A set S of states,
- A subset I of S, the initial states,
- A subset F of S, the accepting states, and
- A subset T of $S \times A \times S$, the transition relation.

The automaton must start in one of the states in I.

 $(r, a, s) \in T$ if the automaton can jump to the state s on reading an a while in state r.

There might be more than one member of I and for a given r and a there might be more than one s with $(r, a, s) \in T$, so this automaton may be non-deterministic.

The automaton is said to accept the input if it *can* end up in a member of F at the end of the input. The set of inputs it accepts is the language it recognises.

Example

Last time we considered a language generated by the following right regular grammar:

```
%%
tokyo : atamadaka | nakadaka | odaka | heiban ;
```

```
atamadaka : H t0 | L error ;
nakadaka : H error | L t1 ;
odaka : H error | L t2 ;
heiban : H error | L t2 ;
t0 : H error | L t3 ;
t1 : H t4 | L error ;
t2 : H t5 | L error ;
t3 : | H error | L t3 ;
t4 : H t4 | L t3 ;
t5 : | H t5 | L error ;
error : H error : L error ;
```

Example

Last time we considered a language generated by the following right regular grammar:

```
%%
tokyo : atamadaka | nakadaka | odaka | heiban ;
```

```
atamadaka : H t0 | L error ;
nakadaka : H error | L t1 ;
odaka : H error | L t2 ;
heiban : H error | L t2 ;
t0 : H error | L t3 ;
t1 : H t4 | L error ;
t2 : H t5 | L error ;
t3 : | H error | L t3 ;
t4 : H t4 | L t3 ;
t5 : | H t5 | L error ;
error : H error : L error ;
```

Can we construct a finite automaton which recognises it?



Figure 1: A finite state automaton for the tokyo language



Figure 2: The same finite state automaton

Let's see how this responds to the input LHLL.



Figure 2: The same finite state automaton

Let's see how this responds to the input LHLL.

It could start in atamadaka. Then it would go to error on the first L and stay there. error is not an accepting state.



Figure 2: The same finite state automaton

Let's see how this responds to the input LHLL.

It could start in atamadaka. Then it would go to error on the first L and stay there. error is not an accepting state.

It could start in nakadaka. Then it would go to t1, t4, t3 and t3, in that order, and would finish in the accepting state t3.



Figure 3: The same finite state automaton, yet again



Figure 3: The same finite state automaton, yet again

It could start in odaka. Then it would go to t2, then t5, and then get stuck in error. The same would happen if we started in heiban.



Figure 3: The same finite state automaton, yet again

It could start in odaka. Then it would go to t2, then t5, and then get stuck in error. The same would happen if we started in heiban.

This is not a democracy! The input is accepted because *some* computational path succeeds, even though the majority don't. The string LHLL is in the language.



Figure 4: Are you sick of this finite state automaton yet?



Figure 4: Are you sick of this finite state automaton yet?

How about the input LLHL?



Figure 4: Are you sick of this finite state automaton yet?

How about the input LLHL?

If it starts in atamadaka then it gets stuck in error immediately. If it starts in nakadaka then it goes to t1 before getting stuck in error. If it starts in odaka or heiban then it goes to t2 before getting stuck in error.



Figure 4: Are you sick of this finite state automaton yet?

How about the input LLHL?

If it starts in atamadaka then it gets stuck in error immediately. If it starts in nakadaka then it goes to t1 before getting stuck in error. If it starts in odaka or heiban then it goes to t2 before getting stuck in error.

It can't finish in an accepting state so the string LLHL is not in the language.

Maybe you noticed a connection between the automaton and the FSA?

Maybe you noticed a connection between the automaton and the FSA?

To construct a finite state automaton from a right regular grammar we do the following:

• Draw in a state for each non-terminal symbol, i.e. write down its name somewhere and circle it.

Maybe you noticed a connection between the automaton and the FSA?

- Draw in a state for each non-terminal symbol, i.e. write down its name somewhere and circle it.
- If the empty list is an alternate for a symbol then it will be an accepting state, so circle it again.

Maybe you noticed a connection between the automaton and the FSA?

- Draw in a state for each non-terminal symbol, i.e. write down its name somewhere and circle it.
- If the empty list is an alternate for a symbol then it will be an accepting state, so circle it again.
- Every other alternate is in the rule for some non-terminal and refers to some non-terminal. Draw an arrow from the first to the second.

Maybe you noticed a connection between the automaton and the FSA?

- Draw in a state for each non-terminal symbol, i.e. write down its name somewhere and circle it.
- If the empty list is an alternate for a symbol then it will be an accepting state, so circle it again.
- Every other alternate is in the rule for some non-terminal and refers to some non-terminal. Draw an arrow from the first to the second.
- If it's not the rule for the start symbol then there's also a non-terminal symbol in that alternate. Label the arrow with it. If there would be multiple arrows between those states with different labels you can amalgamate them.

Maybe you noticed a connection between the automaton and the FSA?

- Draw in a state for each non-terminal symbol, i.e. write down its name somewhere and circle it.
- If the empty list is an alternate for a symbol then it will be an accepting state, so circle it again.
- Every other alternate is in the rule for some non-terminal and refers to some non-terminal. Draw an arrow from the first to the second.
- If it's not the rule for the start symbol then there's also a non-terminal symbol in that alternate. Label the arrow with it. If there would be multiple arrows between those states with different labels you can amalgamate them.
- Erase the state for the start symbol, but keep the arrows leading out from it.



Why does this work?

Why does this work?

The label on each state identifies the symbol which generates those strings which are valid (right) continuations of the input seen so far.

Why does this work?

The label on each state identifies the symbol which generates those strings which are valid (right) continuations of the input seen so far.

The start states are valid (right) continuations when you've seen no input.

Why does this work?

The label on each state identifies the symbol which generates those strings which are valid (right) continuations of the input seen so far.

The start states are valid (right) continuations when you've seen no input.

The accepting states are the ones where no further input is needed to get a valid string.

Why does this work?

The label on each state identifies the symbol which generates those strings which are valid (right) continuations of the input seen so far.

The start states are valid (right) continuations when you've seen no input.

The accepting states are the ones where no further input is needed to get a valid string.

Suppose the rule for symbol A has as an alternate cB, where c is a terminal symbol and B is non-terminal symbol. Then when we're in state A all the strings generated by B will be valid continuations after reading a c, so on the input c we can jump from A to B.

Why does this work?

The label on each state identifies the symbol which generates those strings which are valid (right) continuations of the input seen so far.

The start states are valid (right) continuations when you've seen no input.

The accepting states are the ones where no further input is needed to get a valid string.

Suppose the rule for symbol A has as an alternate cB, where c is a terminal symbol and B is non-terminal symbol. Then when we're in state A all the strings generated by B will be valid continuations after reading a c, so on the input c we can jump from A to B.

The restrictions I imposed on the phrase structure grammar in the definition of "right regular" were precisely the ones needed to make this construction work.